

Extracting group relationships within changing software using text analysis

Pamela Dilys Green

School of Computer Science

A thesis submitted to the University of Hertfordshire in partial
fulfilment of the requirements of the degree of Doctor of Philosophy

March 2013

Abstract

This research looks at identifying and classifying changes in evolving software by making simple textual comparisons between groups of source code files. The two areas investigated are software origin analysis and collusion detection. Textual comparison is attractive because it can be used in the same way for many different programming languages.

The research includes the first major study using machine learning techniques in the domain of software origin analysis, which looks at the movement of code in an evolving system. The training set for this study, which focuses on restructured files, is created by analysing 89 software systems. Novel features, which capture abstract patterns in the comparisons between source code files, are used to build models which classify restructured files from unseen systems with a mean accuracy of over 90%. The unseen code is not only in C, the language of the training set, but also in Java and Python, which helps to demonstrate the language independence of the approach.

As well as generating features for the machine learning system, textual comparisons between groups of files are used in other ways throughout the system: in filtering to find potentially restructured files, in ranking the possible destinations of the code moved from the restructured files, and as the basis for a new file comparison tool. This tool helps in the demanding task of manually labelling the training data, is valuable to the end user of the system, and is applicable to other file comparison tasks.

These same techniques are used to create a new text-based visualisation for use in collusion detection, and to generate a measure which focuses on the unusual similarity between submissions. This measure helps to overcome problems in detecting collusion in data where files are of uneven size, where there is high incidental similarity or where more than one programming language is used. The visualisation highlights interesting similarities between files, making the task of inspecting the texts easier for the user.

Acknowledgements

I would like to thank my supervision team: Peter Lane, Austen Rainer and Bodo Scholz. I am extremely grateful to Peter for his guidance, especially during the difficult times. I hope that this work, which would not have been undertaken without his help and advice, reflects well on Peter. I have very much appreciated Austen's conscientious attention to detail, which was invaluable in improving the structure and coherence of this dissertation, and Bodo's help in putting things in perspective.

There are so many in the computer science department to thank for providing help, support, and friendship: Dan, Steve, Min, Frank, Carl, Sue, Bob, Bruce, Mike, Maria, Colin, Rene, Ian, Na, James, and David, to name a few; also to all the friendly astronomers.

On the technical side, thank you to the computer science team, for all your advice, and for fixing the computers when they sulked or died.

No one in the STRI would be able to function without our fantastic administrative team, in particular Lorraine, who deals with every problem calmly, efficiently and sympathetically. Thank you for everything.

My family and friends have been wonderful throughout, though I know they all thought it was a crazy thing to do. Thank you for not saying so. Special thanks to Todd, Joe, and Sam for their continual encouragement. Also to Mum, who would have been so relieved to see me make it to the end.

The one person who deserves more thanks than anybody else is Dave, who has unfailingly supported me in every way throughout the whole process, and through the last 40-odd years – none odder than these last few.

Contents

1	Introduction	1
1.1	Software origin analysis	2
1.1.1	Context and problem	2
1.1.2	Overview of the approach taken in this research	5
1.1.3	Description of the final system	5
1.1.4	Results	10
1.1.5	Contributions	11
1.2	Collusion detection	13
1.2.1	Context and problem	13
1.2.2	Overview of the approach taken in this research	14
1.2.3	Description of the system	15
1.2.4	Results	16
1.2.5	Contributions	16
1.3	Visualisation	18
1.3.1	Visualisation for origin analysis	18
1.3.2	Visualisation for collusion detection	21
1.4	Summary of contributions	22
1.5	Dissertation structure	23
2	Detecting similarity in program code	25
2.1	Clone detection and plagiarism detection	25
2.2	Detecting similarity in code	27
2.3	Approaches to source-code plagiarism detection	31
2.4	Summary	33

3	Origin analysis	39
3.1	Survey	42
3.2	Approaches in detail	47
3.3	Summary	53
4	Machine learning	55
4.1	Why use machine learning?	55
4.2	The machine learning process in outline	56
4.3	Creating new datasets for machine learning	58
4.3.1	Choosing features	58
4.3.2	Creating a labelled dataset from raw data	59
4.4	Classifiers	61
4.5	Machine learning in software engineering	62
4.6	Summary	63
5	Visualisation	65
5.1	Plagiarism detection file comparison	67
5.2	Other code comparison tools	70
5.3	Colour coding features in software evolution	72
5.4	Summary	73
6	Ferret	75
6.1	Background	75
6.2	Program code	76
6.3	Files to illustrate similarity tools	80
6.4	Similarity scores	83
6.5	Ferret XML report	85
6.6	Density analysis	87
6.7	Ferret trigram-to-file index	90
6.8	Summary	91
7	Trigram analysis	93
7.1	Similarity in program code	94
7.1.1	Within-project similarity	94
7.1.2	Across-project similarity	95
7.1.3	Comparing similarity scores in text and code	95

7.1.4	Stopping	97
7.2	Collusion detection	97
7.2.1	Example trigram-file index	98
7.2.2	Proportional trigram-based measures	100
7.2.3	Measures based on counting trigrams	102
7.2.4	Extending unique share counts	104
7.2.5	Making connections	106
7.3	Source or destination of code in origin analysis	107
7.3.1	Split files	108
7.3.2	Disappearing files	110
7.3.3	Trigram-based measures	111
7.4	Summary	112
8	Visualising file relationships	113
8.1	Displaying student assignments	113
8.1.1	Unique trigrams	114
8.1.2	Extending the standard Ferret display	115
8.1.3	Graduated similarity information	116
8.1.4	Groups	117
8.2	Comparing one document with a group of others	117
8.2.1	Multiple blue-red-black displays	118
8.2.2	Displaying multi-file comparisons, 1-to-3	119
8.2.3	Scheme for colouring the text	125
8.2.4	Displaying multi-file comparisons 1-to-many	128
8.3	Summary	128
9	Trigram analysis applied to student assignments	131
9.1	Data	132
9.2	Method	132
9.3	Results	134
9.3.1	Proportional similarity measures	135
9.3.2	Count-based similarity measures	138
9.3.3	Analysis	139
9.3.4	Showing similarity in a group context	143

9.4	Discussion	145
10	Overview of the classification system	147
11	File Comparison Tools	151
11.1	Code Clone Finder (CCFinder)	155
11.2	Simian	158
11.3	Duplo	160
11.4	P-Duplo	162
11.5	Unscrambling clones	164
11.6	Summary	167
12	Data Collection, Preprocessing, Filtering	169
12.1	Source code collection and organisation	169
12.1.1	Selection of projects from SourceForge	169
12.1.2	The selected data - 89 Projects	170
12.1.3	Preprocessing	172
12.2	Terminology	173
12.3	Filtering	175
12.3.1	Information Gathering	175
12.3.2	Selecting Candidate Files	177
12.4	Experimental Data	179
12.4.1	Split file dataset	180
12.4.1.1	Split files	180
12.4.1.2	Non-split files	185
12.4.1.3	Not classified	188
12.4.1.4	Dataset composition	188
12.4.2	Disappearing file dataset	191
12.4.2.1	Renamed, moved or merged files	192
12.4.2.2	Other classes of disappearing files	192
12.4.2.3	Dataset composition	192
12.5	Summary	194
13	Feature Construction	195
13.1	File combinations and comparisons	196
13.1.1	Comparing single files.	197

13.1.2	Comparing the candidate and concatenated files . . .	198
13.2	Features	204
13.2.1	Raw feature sets	204
13.2.1.1	Ferret basics	205
13.2.1.2	Ferret trigrams	205
13.2.1.3	Simian, Code Clone Finder and P-Duplo . . .	208
13.2.2	Feature sets based on blocks	212
13.2.2.1	The candidate difference set	214
13.2.2.2	Measurements	215
13.2.2.3	Block-based features	217
13.2.3	Final feature sets	218
13.3	Summary	221
14	Classifying the split file dataset	223
14.1	Building models to identify split files	224
14.1.1	Machine learning algorithms	224
14.1.2	Classifying split files	226
14.2	Tests on other projects	233
14.2.1	PostgreSQL	234
14.2.2	DNSjava	239
14.2.3	Unseen data classified by trained models	246
14.3	Summary	247
15	Exploring filtering criteria	249
15.1	Similarity measures	250
15.2	Target file selection	252
15.2.1	Similarity score (1a)	252
15.2.2	Containment (1c)	253
15.2.3	Change-based filters (1b, 1e)	253
15.2.4	Combining similarity score conditions (1a, 1b)	254
15.2.5	Stratifying shared trigram conditions (1e)	254
15.2.6	Less direct methods (2a, 2b, 2c)	255
15.2.7	Discussion on selecting target files	255

15.3	Ordering target files	256
15.3.1	Comparing ranking criteria	256
15.3.2	Discussion on ranking target files	259
15.4	Refining file selection	260
15.4.1	Filter conditions summarised	263
15.5	Checking a change log to verify selection	264
15.6	Summary	266
16	Classifying refiltered data	269
16.1	Refiltered split file dataset composition	270
16.2	Classifying the refiltered split files	273
16.3	PostgreSQL & DNSjava split files	276
16.3.1	Refiltered data	276
16.3.2	Classifying unseen refiltered split file candidates	278
16.3.2.1	PostgreSQL	278
16.3.2.2	DNSjava	279
16.3.2.3	Overall	279
16.3.3	Additional unseen data: Struts and PyX	280
16.3.3.1	Struts	280
16.3.3.2	PyX	281
16.4	Disappearing files	283
16.4.1	Data	283
16.4.2	Disappearing files with one target	284
16.4.3	Disappearing files with more than one target file	286
16.5	PostgreSQL and DNSjava disappearing files	289
16.5.1	Candidates with one target file	289
16.5.2	Candidates with two or more target files	290
16.5.2.1	PostgreSQL	293
16.5.2.2	DNSjava	294
16.5.3	Additional unseen data: PyX	295
16.6	Selected models	296
16.7	Selected features	297

16.8 Summary	300
16.8.1 Split files	300
16.8.2 Disappearing files	300
16.8.3 Comparison to other approaches	301
16.8.4 Overview	302
17 Discussion and evaluation	303
17.1 Origin analysis	303
17.1.1 Related work	303
17.1.2 Machine Learning System	307
17.2 Collusion detection	315
17.3 Visualisations	317
17.3.1 Collusion detection	317
17.3.2 File comparison with 3CO	317
17.4 Other tools	319
17.4.1 Density analysis	319
17.4.2 One-to-one matching of clone output	319
17.5 Summary	320
18 Conclusions and further work	321
18.1 Contributions to knowledge	321
18.2 Further work	322
18.2.1 Origin analysis	322
18.2.2 N-gram analysis	323
18.2.3 File comparison with 3CO	325
18.3 Conclusion	326
BIBLIOGRAPHY	327
A Similarity measures	351
B Classifiers	357
B.1 Ensembles	357
B.2 Selected algorithms	359
C Machine learning in software engineering	361
D Additional file comparison visualisations	367
D.1 Clone detection tools	367

D.2	Origin analysis tools	369
E	Ferret: example calculation	371
F	Density tool prototype	373
G	Similarity rankings for Chapter 9	379
H	Details of the 89 projects	381
I	Density test results	387
J	SVM grid search results	389
K	Supplementary results for Chapter 14	393
K.1	Feature sets - further combinations	393
K.2	Heterogenous meta-classifiers	395
L	Less direct methods of filtering	399
L.0.1	Uniquely shared trigrams (2a)	400
L.0.2	Weighted trigram count (2b)	401
L.0.3	Trigrams shared with the candidate difference set (2c)	402
L.0.4	Discussion	403
M	Extracts from the Lifelines change log	405
N	Additional results 1: Chapter 16	413
N.1	PostgreSQL	413
N.2	DNSjava	417
N.3	Reporting	417
O	Additional results 2: Chapter 16	419
O.1	Disappearing files: classifying imbalanced data	419
P	Dig et al.: Struts results	423
Q	PyX: matched and unmatched disappearing files	427
R	Comparison between Moss and 3CO	429
S	Selected Features	431
T	Comparing related feature sets	435

List of Figures

1.1	Group similarity example	4
1.2	Origin analysis system	9
1.3	Collusion detection system	17
1.4	3CO file comparison system	20
3.1	File restructurings	41
4.1	Input to and output from a machine learning algorithm	56
4.2	The classification model labels each unseen instance	57
5.1	Showing file similarities	66
5.2	Comparing the code in two files	67
5.3	Graphical displays of file similarity	69
5.4	Ribler and Abrams’s graphical displays	70
5.5	Windiff, Winmerge and KDiff3	71
5.6	Ball et al.’s line representation	72
5.7	Augur three-feature colour coding of one file	73
6.1	Ferret: ‘different’ sequences, with a similarity of 1.	76
6.2	Plot of trigram frequency across the projects	77
6.3	Ratio of trigrams to words in text or tokens in code	78
6.4	Diagram of the relationship between the three example files	80
6.5	Factorial and power code	80
6.6	Original combination and permutation code	81
6.7	Amended combination and permutation code.	82
6.8	How similarity scores are affected by changes	83

6.9	An example of standard Ferret similarity information . . .	84
6.10	Ferret XML report for a comparison of fact.c and power.c .	86
6.11	Factorial and power code.	88
6.12	Ferret comparison between the files fact.c and fact-2.c. . . .	89
6.13	The impact on similarity of renamed tokens	92
7.1	Venn diagram: standard Ferret similarity score	93
7.2	Two files in the context of a group of files	100
7.3	Excluding trigrams from the similarity measure	101
7.4	A potential drawback of proportional measures	103
7.5	“Uniquely shared” trigrams	103
7.6	Trigrams shared by two in a small group	105
7.7	Illustration for weighted similarity calculation	105
7.8	Graph of weighted similarities in a set of files	106
7.9	Subgraphs of files connected by weighted similarity	107
7.10	An example of a differently connected group	107
7.11	Venn diagram: trigrams in the example split files	108
7.12	Venn diagram: existing target file	108
7.13	Venn diagram: multiway split file	109
7.14	Venn diagram: a renamed or moved file	109
7.15	Venn diagram: a merged file	110
7.16	Venn diagram: a disappearing file which has split	110
7.17	Venn diagrams: 2 candidate split examples	111
7.18	Venn diagrams: 2 more candidate split examples	111
8.1	Trigrams unique within the group highlighted	114
8.2	Amended file similarity display	115
8.3	Graded similarity colour scheme	116
8.4	Problems comparing many files to one with a two-way tool	118
8.5	Three target file colours and their combinations	119
8.6	3CO comparisons between 2 and 3 files	121
8.7	A candidate file compared with its target files	122
8.8	An extract from Figure 8.7, comparing a file to its targets . .	123

8.9	Ferret and compact file comparisons	124
8.10	Tokens coloured to illustrate the scheme	127
8.11	A comparison with seven target files	127
8.12	A comparison with nine target files	129
9.1	File concatenation and comparison	132
9.2	An extract from the feedback based on unique trigrams . . .	134
9.3	Contour maps of weighted similarity counts	136
9.4	Connections by weighted trigram count ≥ 85	137
9.5	Connections by weighted trigram count ≥ 50	139
9.6	Project 19 compared with project 54 and the provided code	140
9.7	Graded colour scheme, repeated	143
9.8	An extract from a comparison between projects 8 and 17 . .	144
9.9	An extract from a comparison between 5 files	146
10.1	Outline of the learning process	148
10.2	Learning system overview	149
10.3	Classification system overview	150
11.1	Combination and permutation code (repeated)	153
11.2	Amended combination and permutation code (repeated) .	154
11.3	CCFinder report on clones between the three example files	157
11.4	Simian parameter hierarchy	158
11.5	Simian report on clones between the three example files . .	159
11.6	Duplo report on clones between the three example files . .	161
11.7	Examples showing potential duplication in clone detection	164
11.8	Steps in unscrambling clones for one-to-one matching . . .	166
12.1	Source code collection and preprocessing	171
12.2	Comment block example from the biew project	173
12.3	Illustration of similarities using the example mini-project .	176
12.4	Information gathering and filtering	178
12.5	Example split files	181
12.6	A simple split file example	182

12.7	An example of a file split three ways	183
12.8	An example of a multiway split file	184
12.9	Example non-split files selected as candidate split files . . .	185
12.10	An example of a non-split file	186
12.11	A more complex example of a non-split file	187
12.12	Number of targets in split file groups	189
12.13	Candidate split files by project, type and class	190
12.14	Classification of uncertain disappearing files	191
12.15	A disappearing file which has been renamed and edited . .	193
12.16	A disappearing file which has merged with an existing file	193
13.1	Pairwise comparisons between the three main files	197
13.2	Example concatenation and comparison	199
13.3	Three target files of similarity 0.3, two scenarios	199
13.4	A file with two sections split out to two existing files	200
13.5	File combination and comparison	202
13.6	Feature extraction	203
13.7	Trigrams shared by two or three files	206
13.8	The files <code>cnp-1.c</code> and <code>cnp-2-edit.c</code>	210
13.9	Blocks shared by the main target file and the difference set .	215
13.10	Composition of the feature sets	219
14.1	Classification by paired feature sets	232
14.2	The PostgreSQL files <code>parser.c</code> , <code>analyze.c</code> , <code>date.c</code>	238
14.3	<code>MXRecord.java</code> : code destination	244
14.4	<code>MXRecord</code> and <code>KXRecord</code> compared	245
15.1	System overview, repeated here for reference	249
15.2	Target file order: by unique share and by shared change . .	257
15.3	Broadening target file selection: one example	260
15.4	Number of target files per group: various filter conditions .	262
15.5	Filter criteria	263
16.1	Comparison between refiltered and original datasets	270

16.2	Refiltered split file dataset by project, type, and class	272
16.3	Classification by paired feature sets	275
16.4	Relationships between original and refiltered split datasets	276
16.5	A disappearing file of indeterminate class	277
16.6	Analysis of disappearing file targets	285
16.7	The merge between CacheResponse and ZoneResponse . .	295
16.8	Features selected by Simple Logistic	299
18.1	Figures 9.4, 9.9, G.2 repeated for reference	324
D.1	Clone tool outputs	368
D.2	Beagle scatter plot and Ecode-wayf class evolution diagram	369
F.1	Density tool input screen	374
F.2	Density tool output screen	375
F.3	Density tool screenshot	376
F.4	Example of Ferret density analysis XML output	377
J.1	SVM grid search results “tris-singles”	390
J.2	SVM grid search results “tris”	391
K.1	The new file MX_KXRecord, referenced in Section 14.2.2 . .	397
N.1	The disappearing PostgreSQL file geqo_paths.c	416
N.2	DNSjava project report on directory changes	417
P.1	Split files from Struts	424
Q.1	Minimal.c matched to simple.py, similarity 0.89	427
R.1	3CO and Moss file comparisons	430
T.1	Comparing related feature sets	436

List of Tables

2.1	Examples of parameterised code	28
2.2	Elements removed from code before plagiarism detection .	33
2.3	Approaches to source code plagiarism detection	37
3.1	Origin analysis and related work	46
3.2	Projects used for origin analysis by various research groups	52
6.1	Matched and unmatched code patterns in different units . .	87
6.2	Dense blocks	89
6.3	Ferret trigram-file index for the three example files	90
6.4	Distribution of trigrams between the three example files . .	90
7.1	Example extracts from a trigram-file index	99
7.2	Different similarity measures based on trigram analysis . .	101
7.3	Shared trigram count report extract	102
8.1	File combinations and the colours associated with them . .	125
8.2	Token colouring by example	126
9.1	Comparing the similarities found by different measures . .	135
11.1	Snippets illustrating parameterised matching	155
11.2	CCFinder representation of tokens for the file fact.c	156
11.3	Duplo matrix showing the result of line-by-line matching .	160
11.4	P-Duplo matrix showing the result of line-by-line matching	163
12.1	Terminology	174

12.2	Size and similarity vectors for the example mini-project . . .	176
12.3	Analysis of file type and classification of candidate split files	188
12.4	File type and classification of “uncertain” disappearing files	194
13.1	Keys and names of the feature sets	204
13.2	Basic Ferret output for the three example files	205
13.3	Trigram-based features for two and three file comparisons .	206
13.4	Raw features taken directly from the tools’ outputs	211
13.5	Clones in cnp-1.c and cnp-2-edit.c	212
13.6	Blocks from different analyses of the Ferret XML report . .	213
13.7	Difference sets and reverse difference sets explained	214
13.8	Features constructed from block sizes	217
13.9	Features built from the example blocks.	218
13.10	Feature sets, their content and names	222
14.1	Algorithms ranked by performance over all feature sets . .	225
14.2	The reduced set of 23 algorithms listed by group	226
14.3	The top 40 results	228
14.4	Performance of each feature set over the 23 algorithms . . .	229
14.5	Keys and names of feature sources, repeated for reference .	229
14.6	Performance of each algorithm over all of the feature sets .	230
14.7	Performance of selected feature sets on multi-target files only	230
14.8	Performance of selected feature sets on .c files only	231
14.9	Split files in PostgreSQL, backend subsystem	235
14.10	Split files in PostgreSQL, other subsystems	236
14.11	Refactorings suggested by Antoniol et al.’s system	239
14.12	Restructured files found by Antoniol et al.	240
14.13	Split files in the DNSjava project, releases 1–39	241
14.14	Split files in the DNSjava project, releases 40–56	242
14.15	Analysis showing why some splits are missed by [6]	243
14.16	Model accuracy and recall on unseen data	246
15.1	Candidate file selection at a range of thresholds	252
15.2	Ordering target files by uniquely shared trigrams	258

15.3	Refiltering target files: varying thresholds	261
15.4	The effect of altering target file selection criteria	261
15.5	Split files for the project Lifelines.	264
15.6	Lifelines change log entry matches	265
16.1	Comparison of refiltered and original dataset composition .	271
16.2	Refiltered candidate split files by project	271
16.3	The top 40 results on refiltered split files	273
16.4	Performance of the feature sets over the 11 algorithms . . .	274
16.5	The difference in mean accuracy over 23 and 11 algorithms .	274
16.6	Performance of each algorithm over all of the feature sets .	275
16.7	Composition of the refiltered unseen data	277
16.8	Model accuracy and recall on unseen data	278
16.9	Split files found by Dig et al., Struts release 1.1 to 1.2.4 . . .	280
16.10	Classification of Struts candidate split files	281
16.11	Classification of PyX candidate split files	282
16.12	Disappearing files, one target, top 40 results	284
16.13	Performance of each algorithm over all feature sets	285
16.14	Performance of each feature set over the 11 algorithms . . .	285
16.15	Disappearing files, one target, top 40 results	286
16.16	Performance of each feature set over the 9 algorithms . . .	287
16.17	Performance of each algorithm over all feature sets	288
16.18	Geometric means for disappearing files	288
16.19	Disappearing PostgreSQL files with one target	289
16.20	Disappearing DNS files with one target	290
16.21	Three models and their classification	291
16.22	Disappearing DNS files with two or more targets	291
16.23	Disappearing PostgreSQL files with more than one target .	293
16.24	Classification of PyX disappearing files (1 target)	296
16.25	Classification of 22 PyX disappearing files (2+ targets) . . .	296
16.26	Summary of classification by selected models	297
16.27	Summary of classification on unseen data	302

17.1	Proportion of files not classified by category.	313
17.2	Steps in classifying a new project with the models.	314
A.1	Similarity measures for sequences	352
A.2	Similarity measures for sets	353
A.3	Similarity measures for vectors.	354
B.1	Methods for introducing diversity in ensemble learners . .	358
C.1	Machine learning and data mining in software evolution . .	365
E.1	Factorial and power code	371
E.2	Fact.c and power.c trigrams	372
G.1	Comparing the similarities found by different measures . .	379
H.1	The open source projects used in this research	383
H.2	Project size information	385
I.1	Density test parameters	387
I.2	Density test results	388
I.3	Sum of ranks for each component of dd-bb-gg parameters	388
K.1	Performance of 3, 4, and 5-way combinations of feature sets	393
K.2	Performance of the full fb set in combination with other sets	394
K.3	Performance of the full “tris” set combined with other sets .	394
K.4	Performance of selected feature sets with selected algorithms	395
K.5	Heterogeneous meta-classifier results	396
L.1	Target files found by uniquely shared trigrams	401
L.2	Weighted trigram count target file selection	402
L.3	Trigrams shared with the candidate difference set	403
N.1	Matched disappearing PostgreSQL files	414
N.2	Unmatched disappearing PostgreSQL files	415
N.3	Unmatched disappearing DNSjava files	417

N.4	Disappearing DNSjava files with a match in the next release	418
O.1	Geometric means for disappearing files	420
O.2	Balancing the dataset with SMOTE	420
O.3	Cost-based classification	422
O.4	Number of minority class examples correct	422
Q.1	Matched and unmatched disappearing files: PyX project. .	428
S.1	fc+tris-singles features selected by Simple Logistic	432
S.2	fl+tris-singles features selected by Simple Logistic	433
T.1	Comparison of classification accuracy of related feature sets	436

Chapter 1

Introduction

The aim of this research is to investigate the application of text analysis techniques to software code, in particular in extracting group relationships between evolving files. These techniques are applied in two areas: source code collusion detection, and software origin analysis, where the history of a software system is examined to trace code which has moved as the system has evolved.

Text analysis is attractive because it is simple to apply and can be used in the same way for many different programming languages. However, files of program code generally have a certain amount of inherent similarity, due to the constraints of the language and to programming idiom. This similarity will tend to increase within a single software system, or in a group of student submissions. Pairwise comparisons between files will therefore find similarity which is not due to the transfer of code from one file to another. Looking at similarities within a group of files can provide context for pairwise comparisons, and clarify file interactions such as code transfer, and is therefore helpful in analysing evolving source code.

The underlying thesis behind this research is that, in spite of the restricted vocabulary in program source code, text analysis techniques can be successfully used to find meaningful relationships within groups of files in evolving software systems.

The main application of these techniques in this research is in software

origin analysis, where a machine learning system for identifying and classifying restructured files is developed. In the other application, source code collusion detection, the similarity between submissions is considered in the context of the whole group. These same techniques are used to generate visualisations which support the first two applications. The three applications, software origin analysis, collusion detection, and visualisation, are discussed in the next three sections of this introduction.

The systems and tools described in this dissertation were coded by the author in Racket¹, with the exception of the third-party tools: Ferret, from the University of Hertfordshire [146], and Weka from the University of Waikato [247]. Three other third-party file comparison tools were explored in building the origin analysis system, but do not form part of the final system.

1.1 Software origin analysis

1.1.1 Context and problem

Software is increasingly becoming a part of everyday life [131] affecting us both directly, in the computer systems we use for work and leisure, and indirectly in the systems which provide us with our services. A software system cannot be static, but must evolve over time in response to the changing needs and wants of its users, to changes in its environment, and to correct faults revealed in its use [17, 168, 237]. Changes made to a system tend to increase its complexity [148]. It is therefore important to restructure software code periodically, to make it simpler, and thus easier to maintain [78].

Typically a software system is not developed by one person, but by a team with fluid membership, and possibly in separate locations. Given that many systems are poorly documented [40, 67], it can be difficult for a developer to trace structural changes at a later stage [108, 245]. Maintainers need to be able to trace the movement of code in a system, to keep track

¹A dialect of Lisp, see <http://racket-lang.org/>

of unresolved bugs, and to update regression tests to target code in its new location. Software evolution researchers also want to ensure continuity in their analysis of code as it moves within a system [6, 22, 68, 128, 231].

Developers, maintainers and researchers therefore need tools to help them to understand structural changes [90, 130, 252]. The branch of software evolution research called origin analysis looks at the history of a system to try to discover where code has moved during its evolution [237, 261]. Origin analysis is based on matching elements of the code in consecutive versions of the software, and for this a suitable matching technique is required [127].

Matching techniques vary in their representations of the code, and in the measures used to determine similarity. Among existing approaches to origin analysis, the majority use techniques which require that the code is parsed prior to matching, making them language dependent. For example, in constructing an abstract syntax tree [77, 179], a control flow graph [8], or in extracting methods or functions from the code [22, 64, 90, 130]. Text-based approaches, which do not require parsing, are by implication more widely applicable, and therefore more attractive. However, challenges remain in successfully implementing a purely textual approach.

Several previous approaches base their comparisons on the source code text, but use a single value to describe similarity in the code, information which lacks depth. For example, Weißgerber and Diehl [245] use a clone detection tool, Rainer et al. [194] use a fingerprinting technique [193] based on n-gram analysis [32], and Antoniol et al. [6] use vector space analysis, to provide a similarity value.

Unless pairs of entities are identical, one of the difficulties faced when using a single similarity measure is finding the threshold which best separates matched and unmatched pairs [6, 22, 64, 127, 262]. A further problem is that the thresholds for one system are not guaranteed to work well with other systems [22, 90, 130]. These problems are magnified when more than one measure is used, as not only does a suitable threshold have to be found for each measure, but also the best combination of measures must be deter-

mined. For example, S.Kim et al. [130],² who combine eight measures from different sources, find thresholds and combinations by exhaustive search, but suggest machine learning as an alternative solution for the future, a challenge which is taken up here.

Matching code is a difficult problem because of its inherent similarity, in part due to the constraints and idioms of the programming language. Added to this, within an evolving software system, files will also share similar code because of the use of a limited range of variables and functions, and because of copy-paste-edit practices. The goal is to find relationships between pairs or groups of files which exist outside of this incidental similarity.

To give an example, in Figure 1.1 there are 12 “files” which are the same size for simplicity. Each file has one or more sections which appear in at least one other file, and these are represented by coloured blocks. The highest similarity, of two blocks, is between each pair of files in the subset a, b, d, e, h and k. The similarity between file f and each of the other files, except file l, is one block. However, because the orange blocks appear only in files f and g, and the green blocks in only f and j, the similarity between these pairs of files is more interesting than the similarity between files sharing the blue or yellow blocks which appear in almost every file in the group.

²Initials are used to differentiate between S.Kim et al. [130] and M.Kim et al. [128]

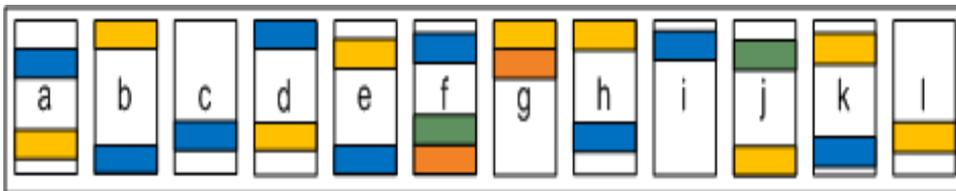


Figure 1.1: This fabricated example shows 12 equally-sized “files” a–l. The colours show parts of a file which occur in other files in the group. File f has one section in common with each of the other files in the group, except l. The interesting similarity is that between the orange and green sections.

1.1.2 Overview of the approach taken in this research

The approach to software origin analysis presented in this dissertation differs from previous approaches in several ways. The main difference is the use of machine learning algorithms to select suitable combinations of features and their weights. This allows a broad range of features not previously considered for this application to be explored. These features, many developed for this research by analysing the outputs from a set of complementary copy-detection tools, are based on simple text analysis. The tools are used to compare not only pairs but also groups of files, which can help to exclude the inherent similarity in code and give context to the pairwise comparisons.

Another difference is that the examples used for training the machine learning models are taken from a range of software projects³ which vary both in application and in development style. This diversity is introduced with the aim of producing models which can generalise across projects. In fact, the resulting models, trained with examples in one language, are able to generalise not only across projects, but also across other languages. This is because the features used to characterise the examples are abstract patterns taken from the simple textual relationships between files. This striking result gives the system a valuable advantage over others.

1.1.3 Description of the final system

The software origin analysis system resulting from this research takes as input the software releases to be analysed, and outputs information about restructured files, and those related to them, in both textual and visual forms. The system is outlined in Figure 1.2 (p.9), where the automated part is enclosed by the bold line, and the original parts of the system are shaded.⁴ Ferret [146], a copy-detection tool developed at the University of Hertfordshire, is used to compare files throughout the system.

³The term *project* is used from here to mean a software system being analysed, to differentiate from a *system* built for this research.

⁴Parts of the system which are original only in their detail are lightly shaded.

The user inputs the software to be analysed, with each release in a separate, consecutively numbered directory. Source code files are selected and each file is stripped of comments and placed in a new set of directories, which have an identical structure to those of the original code.

The next step of the process is to gather information about the files. The system stores the size of each file, and the similarity to each other file in the same release, and in the next release. The information is stored for each file over the lifetime of the project, with special values recorded for releases where the file does not exist. The similarities are computed using Ferret, which tokenises the code, and forms trigrams on which its analysis is based.⁵ Details about Ferret are given in Chapter 6 and the system to this point is explained in Sections 12.1–12.3.

Once information about the files and their similarities is stored, it is initially used to filter the files in the project to find those belonging to two categories. First, and trivially, files which cease to exist during the life of the project, called *disappearing* files. Second, to find potentially *split files*, meaning files from which code may have been moved to another file, rather than files where code has been deleted or edited. This selection is more difficult, and by ensuring that as many split files as possible are found, some non-split files are inevitably also selected. The disappearing files and potential split files are called *candidate* files. Further details can be found in Section 12.4.

The second part of the filtering task is to find *target* files which are possible destinations for the code moved from the candidate file. This part of the filtering task combines several conditions based on the trigrams in the files, and is explained in Chapter 15. These filter conditions aim to include every target for each candidate, however, the cost of such recall is that other files which are not targets may also be selected.

The next part of the process is to rank the target files according to the likelihood that the file is actually a target file, and by the amount of code moved to the file. Previous text-based approaches to ranking targets [22, 245] use

⁵Trigrams are sequences of three consecutive items, e.g. words in text, or tokens in code.

measures of the overall similarity between candidates and targets. In this system, the parts of the code shared by the candidate file and only one member of the target group, called *uniquely shared trigrams*, are also considered in ranking. For more information about group trigram analysis, see Chapter 7, and about experiments in ranking techniques, see Section 15.3.

Once the target files are ranked, candidate groups are formed. For a potential split file, this group consists of the candidate file, its revised version in the next release, and selected target files. Code from disappearing files with no targets is assumed to have been deleted, and files with an exact or very close match among the targets is assumed to have been moved. The remaining disappearing files, whose classification is uncertain, are grouped with the most likely target files. Split file candidates can be classified as split or not. Disappearing files with one target are either matched (that is renamed, moved or merged) or not, those with two or more targets can also be split. Models were trained for each of these classification tasks.

To provide data for training these models, candidate files and their targets were filtered from eighty-nine projects written in C (see Chapter 12). A wide range of features (see Chapter 13) was generated for each candidate group, based on file comparisons made using Ferret and three other copy-detection tools (see Chapter 11). Each tool compares file texts, but has different and complementary methods of determining their similarity. Each candidate file was manually classified to provide labels for the examples. Experiments were run using Weka [247] to explore how different feature sets performed with a range of algorithms in training models (see Chapters 14 and 16). These experiments determined which models are most able to generalise for each task, and these models are used in the final system.

The features selected for the final models are based solely on comparisons made by Ferret, between both pairs and groups of files. Pairwise comparisons are used to create features describing patterns in the blocks of code shared by the two files. Group comparisons generate features based on the interaction between trigrams in the set of files in the candidate group. These two sets of measures provide complementary information.

While trigrams can be scattered throughout the source code text, block-based measures give an indication of their arrangement, and information about the group interactions puts the pairwise comparisons into context. The full set of features explored in building the machine learning models is detailed in Chapter 13, and of those selected for the final models in Section 16.7. The machine learning algorithm which consistently performs well with this data is Rotation Forest [203], as shown by the experiments reported in Chapter 16, with test cases classified with around 90% accuracy.

The output from the system is in two forms: textual and visual. The textual output lists the directly matched and unmatched files, and the automatic classification assigned by the models to the uncertain disappearing files and to the potential split files. The visual output from the system is produced using a tool developed as part of this research, see Section 1.3.1 and Chapter 8. This tool, called 3CO, generates an XML file displaying a comparison between the text of a candidate file and all of its target files, which helps the user to understand where code has moved to. This visual output can also be used directly for manual assessment of the movement of code in the project, especially where few restructured files are found in the project under analysis.

Throughout the system, relationships between groups of files are found using Ferret to analyse their textual similarity:

- in filtering target files, where for every potential split file in release n , and each of its possible target files in release $n+1$, the relationship between four files is analysed, that is each file in both releases;⁶
- in ranking the target files, where all of the files in the candidate group are compared;
- in creating features, where files selected from the candidate group are compared; and
- in generating 3CO visualisations, where all of the target files are compared with the candidate.

⁶Or three files, where the target file does not exist in release n .

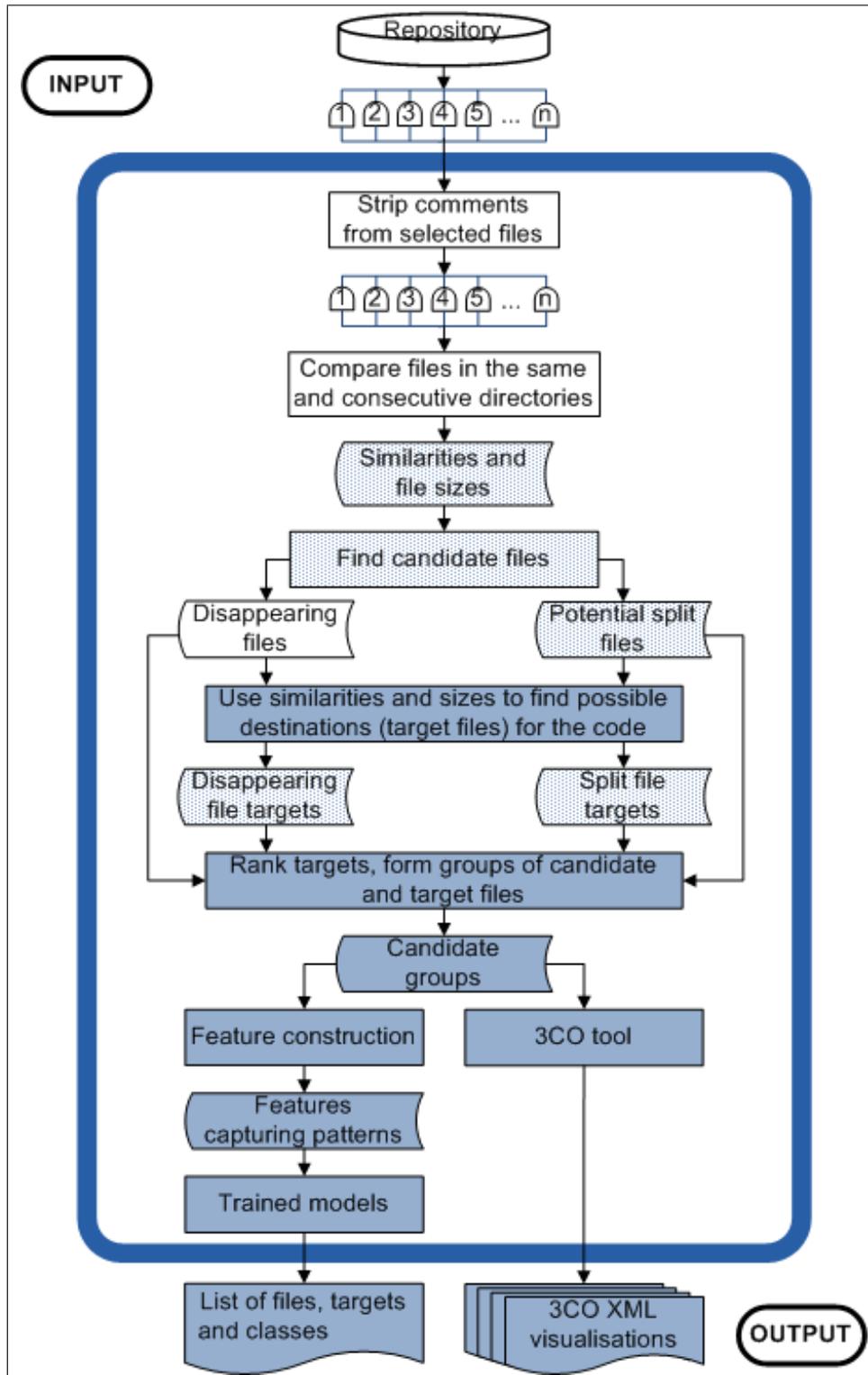


Figure 1.2: Origin analysis system outline. The automated part of the system is enclosed by the bold line, and contributions are shaded. Light shading indicates parts which are original only in their detail.

1.1.4 Results

The final system was used to analyse four software projects not used in training the models, three of which have been investigated by other software origin analysis research groups. The examples extracted from the four projects are also classified with around 90% accuracy. Details can be found in Sections 16.3 and 16.5, and are summarised below.

The PostgreSQL “backend” subsystem written in C, which consists of 12 releases and around four and a half thousand files, is studied by Zou [261]. Zou’s system combines measures of the string similarity of function and parameter names with call analysis and complexity metrics, to find functions which have been renamed, split or merged. All of the file level changes reported by Zou are found by the machine learning system, although two are not reliably classified. Additional restructured files are found by the machine learning system, which may be assumed not to be mentioned by Zou because of the different focus of the two studies.

Forty releases of the DNSJava project, including around three thousand files, are studied by Antoniol et al. [6], who use vector space analysis on identifiers in the code to find classes which have been renamed, split or merged. All of the file level changes found by Antoniol’s system are found by the machine learning system, although one is not classified correctly. Two renamed classes found by Antoniol et al. are not found by the system, because code is not moved from the file and the filename is unchanged. However, seven further split files, one merged file, and one renamed file (and class), are identified by the machine learning system and not reported by Antoniol et al.

Dig et al. [64] study the changes between two releases in the Struts project comprising a total of just under one thousand Java files. Their approach combines the text comparison of method bodies to find candidate refactorings, with call analysis to refine the search. The three file level changes found by Dig et al.’s approach are found and correctly classified, as are two others confirmed by another study of Struts by Wu et al. [250], and two more confirmed by manual inspection.

The last project, Pyx, is a Python project with over one thousand files in twelve releases. Pyx was analysed for this study with the aid of a fairly comprehensive change log, as no studies of this type appear to be available with projects in languages other than C or Java.⁷ These results cannot be compared with those of another research group, but around 88% of the model classifications match the manually assigned classifications.

In summary, it seems reasonable to claim that the machine learning system performs at least as well as the other systems, although because the systems have a different focus, they cannot be compared directly. Nevertheless, it seems that several restructured files found by this system are not mentioned by previous authors. The advantage of this machine learning system is that the trained models apply across the projects and languages tested, and theoretically to many more.

1.1.5 Contributions

The main contribution resulting from this research into using text analysis in the field of origin analysis is the set of trained models for determining the class of candidate restructured files. The individual parts of the system which are original are shaded in Figure 1.2, and are detailed below, starting at the top of the diagram.

The lightly shaded boxes differ from other approaches only in what is stored and used for subsequent analysis. In this case, the information is taken from file comparisons made by Ferret. For each pair of files in one release and in consecutive releases, not only are proportional similarity scores stored, but also the number of trigrams in each file and shared by the files.

Target file filtering takes account of all of the stored information, looking at the changes in each file and its relationship to the changes in the candidate file, using both proportional and discrete measures. The filtering criteria result from experiments undertaken as part of this research.

A novel measure derived from a Ferret comparison of the files in each

⁷Except for one in Smalltalk [60], for which the code is not available.

candidate group is used in ranking the target files. This is the number of trigrams which each target file has uniquely in common with the trigrams which are no longer part of the candidate file.

The features for the final system result from using new ways to analyse the outputs from file comparisons made by Ferret. These features aim to give a fuller picture of the amount and distribution of code shared by the files in a candidate group than is available from a single similarity measure. For details of the analysis of Ferret outputs see Chapters 6 and 7.

The machine learning models are trained with a large set of examples derived from projects written in C. The projects cover a range of application areas and development styles, with the aim of creating models which are able to generalise across projects. As the features are based on patterns in the trigrams shared by two or more files, the models are also able to generalise across projects in the other languages tested, Java and Python, and should work with many other languages and with evolving text files.

During this research, it was necessary to compare each candidate file with its potential target files to understand the interaction between files in the group. No suitable tool could be found to display these comparisons, especially for groups containing a large number of files. The 3CO tool, based on Ferret's trigram analysis, was therefore developed for this task (see Section 1.3.1). The ideas behind the display, which uses primary colours to show which target files contain code from the candidate file, could also be applied to any system which compares a group of files.

1.2 Collusion detection

1.2.1 Context and problem

Plagiarism in universities is reported to be on the increase [119, 177, 222]. A combination of strategies is generally used to deter such inappropriate use of the work of others [174]. First, students are given guidance on plagiarism and how to avoid it. Second, where possible, assignments are designed to minimise the opportunity for copying. Last, it is important for students to know that their work will be checked. The automatic detection of similarities between submissions is crucial in helping tutors in this task.

A group of student programming submissions is also a type of evolving software. Each submission in a group will be based on examples provided during the course, which then evolves to suit each student's work. If students collude, then code will be transferred between them. As in origin analysis, there is inherent similarity in programming submissions; in this case, due to examples used in teaching, to template code, to the common aim of the task, and possibly to use of a development tool, rather than to the limited range of variable and function names present in a single software project. The task in collusion detection is to find sections of code which may have been passed from one student to another.

Unusual similarity, that is parts of the code shared by only two or just a few students, is said to be an indicator of possible collusion [52, 110]. Although a tutor marking a small set of submissions can be expected to spot unusual similarity, when there is a larger number of submissions, or where marking is undertaken by a team, an automated approach is desirable. Filtering by automatic means is one part of the task, the tutor must also assess the similarity found, and for this a suitable visualisation is essential.

Twenty-nine approaches to source code collusion detection were surveyed to provide a background to this work (see Section 2.3). The survey found that few of the approaches take account of unusual similarity [3, 9, 34, 52, 115], only one specifically targets elements shared by few files [199], and none directly measure them. It was also found that many

approaches can handle a variety of languages. However, very few are able to analyse a mix of programming languages. There is an increasing emphasis on teaching web technologies, where several different programming languages can be used in building a single project. It is therefore useful to be able to combine files coded in different languages when measuring the similarity between students' work. Another common feature noted is that similarity measures are generally proportional. While this is suitable for comparing files of around the same size, the similarity of a small file to a larger one will be understated by some proportional measures.

1.2.2 Overview of the approach taken in this research

The approach to collusion detection developed in this research aims to tackle the difficulties outlined above in three ways, and so provide a more suitable system for this type of data.

First, submissions may be written in more than one language. Analysing the source code textually,⁸ without the use of parameterisation means that no language-specific processing is required. Although lack of parameterisation may mean that well-disguised copies will not be detected, there is evidence [120] that many students who copy others do so because they lack either the time or the skill to undertake the task for themselves, and consequently do not use clever disguises.

Second, submissions often have high levels of incidental similarity due to auto-generated code, standard constructs, or examples used in teaching. Measuring similarity which is unusual within the group aims to overcome this problem.

Third, open-ended assignments can result in submissions of uneven sizes. This is tackled by using a count-based measure rather than a proportional measure to find the "amount" of unusual similarity.

⁸Here with a tokeniser for C-type languages, adequate for a range of other languages

1.2.3 Description of the system

The collusion detection system, which is the outcome of the research described in Chapter 9, is outlined in Figure 1.3 (p.17). In the same way as for the origin analysis system, automated parts are enclosed by bold lines, and contributions are shaded.

Input to the system is a set of folders, one for each student. The contents of each folder are concatenated to make one large file of source code, so that there is one file to process for each student. Optionally, code provided by the tutor, in the form of examples or exercises given during the course, or template code given for the assignment, is also input. These concatenated files are passed to Ferret for analysis. This system is based on the Ferret trigram-to-file index, which records every trigram in the set of files along with the numbers of the files in which each trigram appears (for details, see Section 6.7). A number of measures can be determined by analysing this index, for example, unusual similarity and uniqueness.

In Ferret's normal use, the similarity between two files is computed based on the proportion of shared trigrams to the total in the two files. In this system, these trigrams are considered in the context of the rest of the files in the group. Unusual similarity is calculated by counting the trigrams shared by only the two files, and adding an inversely weighted count of those shared by the two files and a few other files. This measure overcomes the problems associated with inflated similarity due to auto-generated code, or to the use of tutor-provided code. As a count-based measure, it will also find sections of code which may not be found by a proportional measure because they make up a small part of a large file. For details, see Sections 7.2 and 9.3.

Uniqueness is a count of the number of trigrams unique to any one file among the group. Identifying unique trigrams has two benefits, first, to show unusual solutions to a problem or interesting extensions to the required task, and second, if there is a suspicion that a student has copied code from an external source, these trigrams provide useful search phrases.

Once similar pairs or groups of files are identified, a visual display of

the comparison between files selected by the tutor can help to understand reasons for the similarity. Two techniques for displaying textual file comparisons are outlined in Section 1.3.

1.2.4 Results

This system was trialled with a set of submissions by students who were developing community websites. It is difficult to draw strong conclusions about the measures tested with only one set of submissions, however, the weighted count-based measure of unusual similarity had advantages over the other measures tested on this data. Details of the experiments can be found in Chapter 9, where there are also examples of the visualisations.

1.2.5 Contributions

Several contributions result from this part of the research. First, the comprehensive survey of work in source-code plagiarism detection. Second, the measure of unusual similarity which helps to identify sections of code appearing in two or only a few submissions. Tied in with this analysis of the trigrams in a group is the ability to find unique trigrams. An important contribution is the colour-coded display (see Section 1.3.2) which highlights both the unusual similarity between files and the unique trigrams. This makes it easier for a tutor to pinpoint areas of interest in a file than in a traditional display, which typically only differentiates between the parts of the files which are shared and the parts which are not.

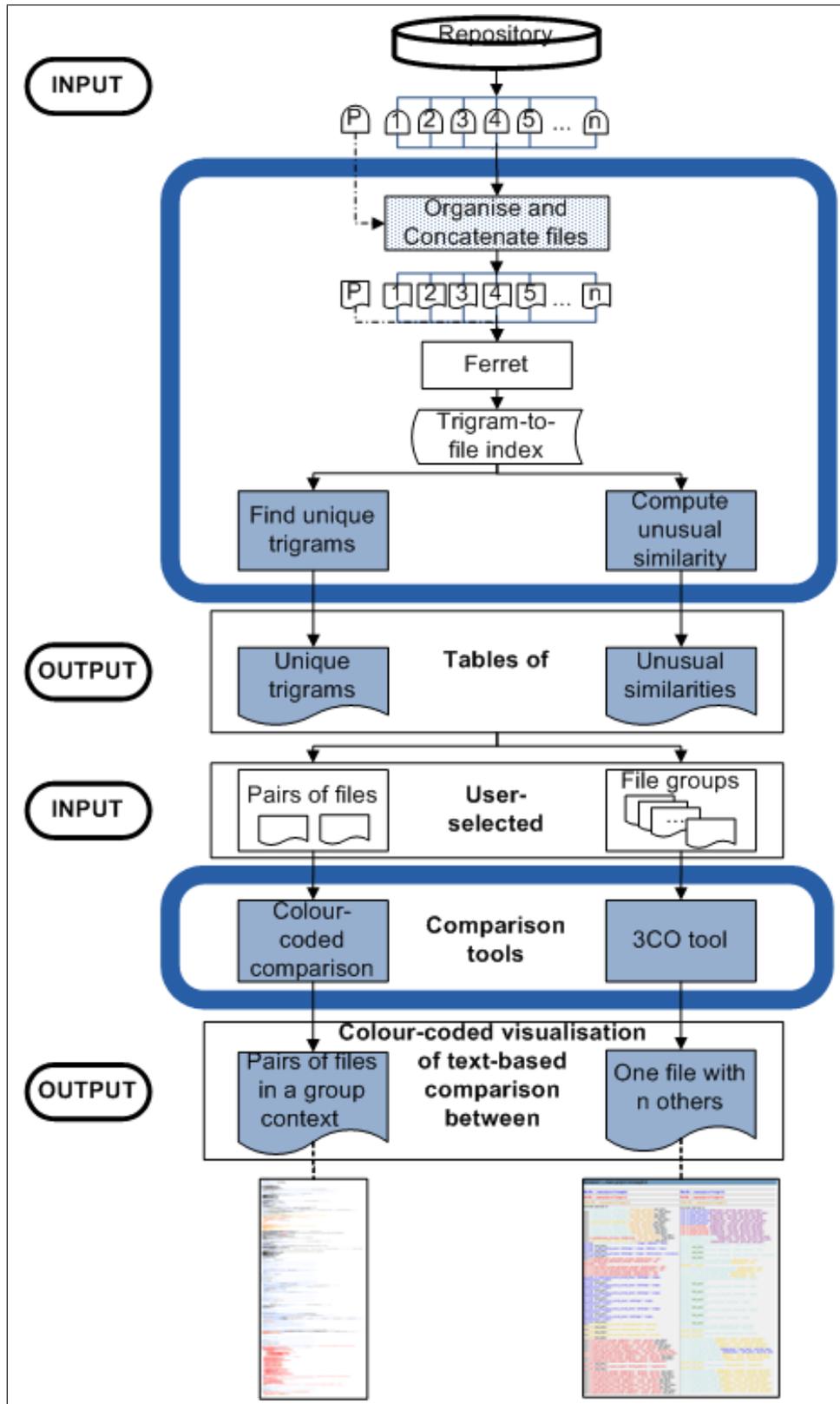


Figure 1.3: Collusion detection system outline. The user inputs folders of student code (1..n) and, optionally, tutor-provided code (P) for analysis. Automated parts are enclosed by bold lines, and contributions are shaded.

1.3 Visualisation

Visualisation of the relationship between files is important in both software origin analysis and collusion detection. In origin analysis, it is invaluable to be able to quickly understand the relationship between files, both in labelling training examples for building the system, and in displaying results for the end user. In collusion detection, a tutor will want to check pairs or groups of submissions which cause concern; having a display where the interesting similarity between two files is highlighted will help in this task. Visualisations developed for these two areas, both based on analysis of the trigrams in a group of files, are described in the next two sections.

1.3.1 Visualisation for origin analysis

The ability to visualise the interaction between files in a project is beneficial in origin analysis. In building the machine learning system for this research, visualisation was important: in deciding whether files belonged to a particular class when labelling the training data, and in deciding about the selection and ranking of target files while exploring the behaviour of the filtering criteria. It is also valuable for looking at the relationship between files in other applications.

While these tasks are possible using existing tools which show comparisons between two, or at most three files, it is difficult when there are more files to compare. What is required is a tool which can show a comparison of one file to a group of other files without the need to align separate displays, or to scroll each display individually. To do this, the 3CO tool was developed, based on analysis of the trigrams in a candidate group.

3CO can be used for other file comparison tasks, such as looking at student files, where a file suspected of containing copied content will be the file for comparison. The suggested applications are for source code files, but as the tool is based on trigram analysis, it is equally suitable for comparing files of natural language.

An outline of the 3CO tool is shown in Figure 1.4. To produce the

visualisations, the user provides a list of the files to be compared. The first file in the list is the *base* file with which all the other files are compared. In origin analysis, this is normally the candidate file, and the remaining files are a ranked list of the target files. In collusion detection, the base file would be a suspicious file, and the other files those which are most similar to it.

The system divides the target files into groups of three, each group is then compared with the base file, using Ferret to provide a trigram-to-file index. The idea is that each of the three files in a group is allocated a primary colour. The text of the base file is coloured depending on which other files the text appears in. For example, code in file 1 and not in files 2 or 3 is coloured blue. The circles to the right of the figure show the colours for each file combination. Each token is a part of three trigrams, each of which can be in none, one, two, or all three of the other files. The colour of each token is determined by a vote between the files in which its trigrams appear. Each group of three (or fewer) files is compared to the base file in this way, with a new column added to the XML output file for each group, until all of the files are included in the output.

The output file allows the user to zoom out to see an overview of the arrangement of the base file contents in the other files, or to zoom in to see details of the text. As the file contains all of the comparisons, no alignment or separate scrolling is necessary. The example in Figure 1.4 shows six files compared to the base file. A more detailed description of the 3CO tool is provided in Section 8.2, with further examples, in detail and in overview.

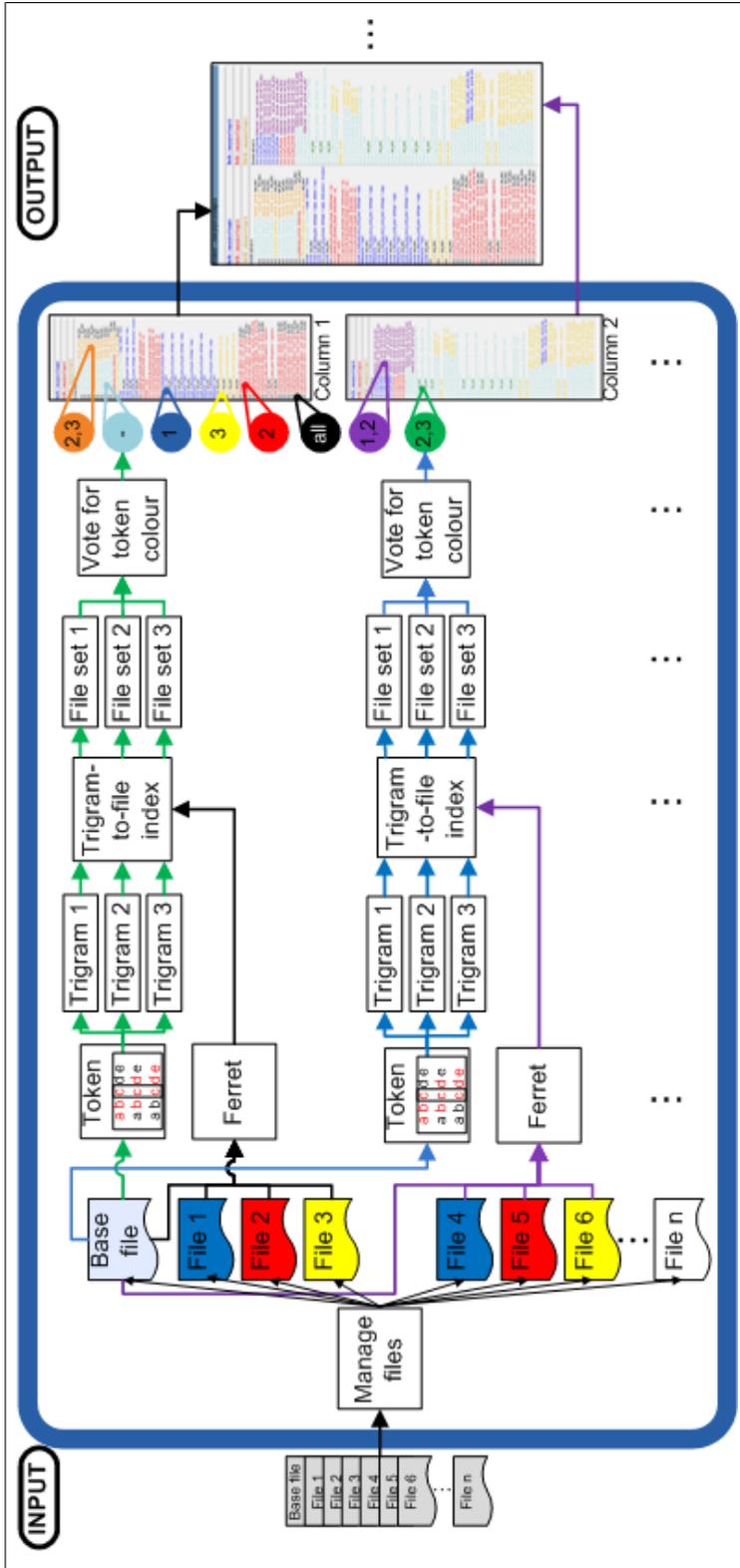


Figure 1.4: 3CO file comparison system. The user inputs a list of files, the first of which is the (base) file to be compared with the remaining files. The output is an XML file showing the text of the base file, which is repeated for every group of three other files, coloured to show which of the three files in the group share trigrams with the base file.

1.3.2 Visualisation for collusion detection

In collusion detection, the standard output of many tools includes a comparison between the source code text of two files, highlighting the parts of the files common to both. While this is useful, it does not always identify interesting similarity between the files. It would be helpful to understand which parts of the file are shared by just the two files, or the two files and a few others. Conversely, it is not very useful to find parts of the files which are shared by not only these two files but by many others, or is code which has been provided by the tutor. Several systems take this last point into account when computing similarity measures, but do not appear to consider it when displaying the text.

In showing a comparison between two files, the colouring system presented in this research distinguishes between:

- trigrams appearing in only one file in the group (unique trigrams),
- trigrams shared by just the two files (uniquely shared trigrams),
- by the two files and up to two others,*
- by the two files and three to five others,*
- by the two files and more than five others,*
- and “greys out” the uninteresting trigrams, that is those in the provided code, or not shared by the two files in question.

* These thresholds can be adapted for different group sizes.

Details are in Chapter 8 with a real-world example in Chapter 9.

The information needed to produce this visualisation is taken from the Ferret trigram-to-file index, in a similar way to the 3CO tool. When two files are compared, the tokens in each file are matched to the trigrams in which they appear. The number of other files that each trigram appears in is noted, as is its presence in the tutor-provided code. Each token is then coloured based on a majority vote among the trigrams in which it appears. Details are given in Section 8.1.

1.4 Summary of contributions

This work investigates the application of text analysis techniques to software code, in particular in extracting group relationships between evolving files. The techniques are used in collusion detection, software origin analysis, and in visualisations to support both applications.

In summary, the contributions made by this research are:

- A major study of the application of text analysis and machine learning techniques to software origin analysis, resulting in:
 - Trained models for identifying restructured files which generalise across projects and a range of programming languages.
 - Development of the 3CO tool for displaying textual file comparisons.
 - Criteria for filtering targets for restructured files.
 - A novel method for ranking these target files.
 - Construction of a set of features for input to the models, using several new techniques for analysing the output of copy-detection tools.
 - Marked-up datasets for studying origin analysis.
- An investigation to find collusion detection measures suitable for sets of submissions of uneven size, with high incidental similarity, and using more than one programming language, resulting in:
 - A comprehensive survey of work in source-code plagiarism detection.
 - Use of text-based group analysis to measure unusual similarity.
 - Development of a colour-coded display to give group context to the similarity between files.
- Design and prototype code for the systems described in Sections 1.2–1.4.

1.5 Dissertation structure

This section gives an overview of the remaining chapters in the dissertation. Chapters 2–6 give background to the research and reviews related studies.

Chapter 2 looks at methods for finding similarity in program code, and surveys previous approaches to detecting collusion in source code.

Chapter 3 provides details of previous methods used for origin analysis.

Chapter 4 gives an overview of machine learning and focuses on the uncertainties encountered in creating a set of features where the space of possible features is not well constrained.

Chapter 5 outlines the use of visualisation in source code comparisons, particularly by the tools reviewed in Chapters 2 and 3, and by tools which use colour to highlight interesting features in software files.

Chapter 6 describes the trigram-based similarity detection tool Ferret, and tools developed in this research to analyse its output to provide a richer set of features than is directly available from Ferret.

Chapters 7 and 8 introduce ideas about finding and visualising relationships between evolving files, based on analysing the trigrams in the source code, which form the basis of the rest of this dissertation.

Chapter 7 explains how analysis of the distribution of trigrams in a set of documents can be applied to collusion detection and to origin analysis.

Chapter 8 shows novel techniques for visualising the interaction between files in a group context, for collusion detection and for origin analysis.

Collusion detection, and the main application, software origin analysis, are covered in Chapters 9, and 10–16, respectively.

Chapter 9 applies the measures and visualisations described theoretically in Chapter 7 and 8 to a set of student assignments.

Chapter 10 gives an overview of the machine learning system for classifying restructured files, which comprises data collection, filtering, feature construction, and the training and testing of classification models.

Chapter 11 introduces the third-party file comparison tools used in feature construction, describes an adaptation to one of the tools, and a method for analysing output from the tools to suit their use in origin analysis.

Chapter 12 describes the collection, preprocessing and filtering of evolving source code from an open source repository.

Chapter 13 details the features constructed for the machine learning system, which are based on the output of the file-comparison tools introduced in Chapters 6 and 11.

Chapter 14 presents the first of the machine learning experiments, where models developed from the training data are used to classify files from two projects studied by other origin analysis researchers [6, 261].

Chapter 15 explores a range of methods based on trigram analysis for improving the filtering techniques described in Chapter 12.

Chapter 16 looks at the effect of the new filtering criteria on data selection, and repeats the machine learning experiments with the refiltered data, testing the new models on the two test projects used in Chapter 14, and on two further projects.

The research is discussed and evaluated in Chapter 17, and Chapter 18 concludes the dissertation and suggests areas for future work.

Chapter 17 relates this research to that of others, evaluates the experimental methodology and results, and discusses issues raised by the research.

Chapter 18 reviews the contributions to knowledge made by this work and suggests ideas for future research.

Chapter 2

Detecting similarity in program code

The theme of this dissertation is finding relationships within groups of evolving source code files based on textual analysis. To accomplish this task, one or more measures of similarity between the contents of the files are required. Two of the main applications where finding similarity in program code is important are plagiarism detection and clone detection. Measures taken from these two areas have previously been used in origin analysis [22, 130, 245].

In this chapter, first clone detection and plagiarism detection are compared, and their application to origin analysis is explained. Next, the detection of similarity in source code is outlined in general terms, to give context to the survey presented in Section 2.2. The survey analyses twenty-nine approaches to plagiarism detection as background to the collusion detection application described in Chapter 9.

2.1 Clone detection and plagiarism detection

Clone detection and source code plagiarism detection are both concerned with finding similarity in program code. In clone detection, the aim is to find sections of similar code within a project or across a group of related projects

so that they can be abstracted, or be uniformly maintained. Four types of code clone are generally recognised (see, for example, Roy et al. [204]). The first three of the four types form a hierarchy: the simplest clones, known as type-1, are textually identical, with the possible exception of white-space and layout; type-2 allows for differences in identifier names, in typing and in literal values; changes introduced by editing, such as additions, deletions and altered statements, are acceptable for type-3 clones, which are sometimes known as “gapped clones” because of the breaks between fragments of similar code [19, 238]. Type-4 are semantic clones, which have the same function, but use different structure, syntax, or both [87]. Changes to code in origin analysis are more likely to be of types-1–3 than type-4.

The usual aim in plagiarism detection is to find similarity between student programming assignments or between assignments and internet-sourced code.¹ Plagiarists sometimes attempt to disguise their plagiarism, and the level at which this is done depends to some extent on their programming ability. Joy and Luck [120] define two main types of plagiarism disguise: lexical changes and structural changes. Faidhi and Robinson [71] define six levels of disguise instead, three of which fall into each of Joy and Luck’s categories [47]. Lexical changes, those to comments, white-space and identifier names are simple. Structural changes require some knowledge of the programming language used, and include changes such as replacing one control structure with an equivalent one, reordering statements, combining functions, or rewriting logical operators.

These disguises are similar to the clone types, lexical changes are like the changes accepted by type-1 and type-2 clones. Simple structural changes are those which would be found by tools which identify type-3 clones, with the more complex structural changes being like type-4 clones.

Unlike clone detection, the focus in plagiarism detection is on inappropriate copying between authors, therefore within-project copying is not of interest. Another difference between the approaches is the focus in the re-

¹Similarity between two sources does not prove collusion or plagiarism, but indicates areas for further investigation [52, pp.22–23]. However, “plagiarism detection” is used in this section to describe this application of similarity detection.

porting of detected similarities. In general, the primary aim of plagiarism detection tools is to provide a measure of similarity between each pair of files compared, to highlight suspicious likeness, with the location of the shared code of secondary importance. In contrast, clone detection tools provide information about the location of matched sections of code, both within and between files, but do not always provide a similarity measure.

Both of these approaches are useful in origin analysis. The more general similarity measure can select files to, or from, which code may have moved, while pinpointing similar sections of code helps to determine whether this is the case.

2.2 Detecting similarity in code

The steps taken when looking for similarity in program code are broadly the same whatever the application, and are outlined in this section, with examples taken from the fields of both clone and plagiarism detection. The steps generally consist of:

- preprocessing the code,
- transforming the processed code,
- changing the new representation into elements suitable for matching,
- matching these elements,
- post-processing the results, and
- displaying the results.

Preprocessing: In preprocessing, parts of the code considered unimportant in matching, such as white-space or headers, are removed. Comments are normally removed in clone detection, whereas in plagiarism detection, duplicated comments are sometimes considered useful indicators of copying, and therefore retained [31, 82]. Some clone detection tools ignore repetitive structures, such as table initialisations or preprocessing directives, because they are considered as unsuitable candidates for abstraction [125].

Code transformation: The simplest first transformation is tokenisation [114, 146, 189]. Other ways to represent code include abstract syntax trees [16, 77, 229], parse trees [89, 116, 221], call graphs [42] or program dependency graphs [134, 151]. In plagiarism detection, where cross-language copying is also considered, intermediate code, such as Register Transfer Language (RTL) [9] or Common Interface Language (CIL) [122], is an alternative. Meta-data, such as metrics, were often used in earlier similarity detection tools when computing power was limited [7, 136, 165, 170].

After initial transformation, further changes may be made. For example, parameterising identifiers, to allow code of similar structure to be matched. In clone detection, the matched sections are candidates for abstraction. In plagiarism detection, structurally similar code may have been copied and disguised. Methods of parameterising include standardising synonymous keywords [246]; replacing syntax tree nodes by their types [18]; or ignoring differences in user-selected items, such as literals or modifiers [105]. Table 2.1 shows a selection of other approaches to parameterising. The code snippet transformed here is: `“for (int i = 0; i < max; i ++)”`.

Method	Ref.	Original code: <code>for (int i = 0; i <max ; i ++)</code>
ignore keywords	[52]	<code>i 0 i max i</code>
unify identifiers	[199]	<code>for (int \$ = \$; \$ <\$; \$ ++)</code>
unify identifiers	[112]	<code>for (int IDName=IDName; IDName<IDName; IDName++)</code>
unify identifiers, remove types	[42]	<code>FOR LPAR identifier ASSIGN SEMI identifier LT ...</code>
one-to-one or p-matching	[12]	<code>for (int p1 = p2 ; p1 <p3 ; p1 ++)</code>
just keywords, numbered	[173]	<code>for1, int1</code>
keyword=0, separator=6 identifier=2	[114]	<code>6 0 2 0 2 6 2 0 2 6 2 0 2 6</code>
replace literals with L, hex-H (N and I are assumed for number & identifier)	[99]	<code>for (int I = N ; I <I ; I ++)</code>
unify all alphanumeric strings	[31]	<code>t (t t = t ; t <t ; t ++)</code>
55 parameters eg R = for, A = (, N = alphanumeric	[34]	<code>R A S N K N N J N N D D B</code>

Table 2.1: Examples of parameterising `“for (int i = 0 ; i < max ; i ++)”`

Another advantage to parameterising is the reduction of complexity in matching. A disadvantage is that some interesting features of the code may be lost. Some methods use relaxed parameters for filtering, and perform more computationally expensive fine-grained matching only on those elements which match under the less constrained conditions [18, 34].

Elements for matching: The new representations of the code are usually broken into smaller elements for matching. Exceptions are metric vectors, which can be compared directly, and file sequences, when similarity is computed using either an alignment or a compression algorithm.

Text and tokens may be divided into physical units such as lines, logical units, such as statements, or sequential units, such as n-grams, which are n adjacent characters or tokens. The units may be further transformed before comparison. For example, by hashing [66, 199, 246] ; by fingerprinting [3, 160], reducing information by sampling; or by counting features to give frequency vectors and therefore losing their order [81]. The transformed code can be compared directly, such as line-by-line [5, 120], or by grouping into bags [114], sets [146, 173, 253] or sequences [89]. Bags lose information about order, as do sets, which also lose information about frequency.

Subtrees are either compared directly [16], or after further processing. For example, Tairas and Gray [229] flatten parameterised nodes into a sequence; Jiang et al. [116] take a similar approach, forming a sequence before matching with a suffix tree [239]; Belkhouche et al. [18] do the same with a structure chart; and Noh et al. [181] and Wahler et al. [241] extract frequency vectors of node types from an XML representation of the tree.

Liu et al. [151] compare the procedure level subgraphs of the program dependence graph. Chilowicz et al. [42] create a new version of the call graph where common sub-functions are extracted from the code, which helps to match code which has been in- or out-lined to disguise copying.

Weighting techniques taken from information theory, and often used in text retrieval, are used with vectors, to reflect the overall frequency of terms within the files under scrutiny. For example, Cosma [52] uses

latent semantic analysis (LSA) in plagiarism detection, as do Marcus and Maletic [163] in finding concept clones. Ji et al. [115] use weights in the adaptive scoring applied in their local alignment of keyword sequences.

Matching non-identical elements requires some measure of their similarity to be computed. The similarity measures referred to in this dissertation, especially in Tables 2.3 (p.37) and 3.1 (p.46), are explained in Appendix A.

Post-processing is dictated by the application, the elements matched, and the method by which they are matched. None is required when an immediate measure of similarity is produced. For example, the similarity between two sets calculated with the Jaccard [113] or Dice [61] coefficient, or when the distance between two attribute vectors is calculated.

In plagiarism detection, methods which match sections of code within files, such as string tiling, alignment methods, or clone-based methods, normally require post-processing to find a measure of similarity between the files. Less post-processing is required in clone detection, but is used in filtering clone classes [66, 125, 229], removing subsumed or insignificant clones, or joining small clones to create larger gapped ones [11, 137].

Results: Roy et al. [204] categorise the output of clone detection tools into three broad groups: a listing of the start and end points of each clone, a graphical output, or a combination of the two.

Plagiarism detection tools typically provide ranked pairwise similarity scores [31, 34, 41, 42, 115, 146, 175, 181], often with additional displays mapping the similarities between a pair of files to the source code texts [41, 120, 146, 189]. Others provide graphical displays [52, 81, 114, 120, 173, 199, 253]. Examples of each type of output can be found either in Chapter 5 or in Appendix D.

2.3 Approaches to source-code plagiarism detection

In 2009, Roy et al. [204] surveyed clone detection tools available at that time. Their survey contains detailed analysis of the methods used by around 50 clone detection tools. The reader is referred to this comprehensive survey for background in clone detection.

Groups of plagiarism detection tools have been analysed previously. For example, Hage et al. [99] compare the performance of Moss, JPlag, Sim and Plaggie, in addition to Marble, developed by one of the authors. Larger studies exist, such as a review of 11 tools by Lancaster [143] in his 2003 dissertation, and of 14 tools by Cosma [52] in her 2008 dissertation. However, there appears not to be a recent large-scale survey of approaches to plagiarism detection such as that by Roy et al. of clone detection tools.

In Table 2.3 (p.37), twenty-nine approaches to source-code plagiarism detection are listed, in date order, from 1996 to 2011 [97]. The tools are specifically aimed at source code plagiarism detection, and include only two, GPlag [151] and Sim [89], of those surveyed by Roy et al.

In the first column, the authors and, where relevant, the names of the tools (in bold text), are listed. During preprocessing, different parts of the code are excluded by different tools, either explicitly, or as a consequence of the transformation process, such as white-space (W) in tokenising, or comments (C) in graph construction. Keys to what is excluded by each tool are in column 4, with the meanings of these keys in Table 2.2, on page 33.

The column headed "TI" notes how the code is transformed initially. Tk means the code is tokenised, the most popular method here, with 21 of the 29 approaches choosing this method; IL that an intermediate language is generated; Mt that metrics are calculated; Gr that the code is represented by a graph; and Tr, a tree.

Brief descriptions of further transformation of the code are provided in the sixth column. This often involves parameterising in one of the ways already discussed in Section 2.2. Similarity measures are noted in column 8, these vary, depending to some extent on what is suitable for the elements to be matched, which are shown in column 7. In some cases the researchers

compare several measures, and the one found to be most effective is listed here. The next column shows how the results are reported and also, to fit the page, remarks, which are in square brackets.

The column headed "Exc." indicates whether the tool is able to exclude template code (✓) [2, 3, 189, 199] or whether common code is inversely weighted in the calculation of similarity between two programs (*) [9, 34, 52, 115], making template code less important in the similarity calculations.

The last column shows whether files which share unusual similarities are of particular interest. This last feature will not highlight plagiarism where clever disguises are employed, however, it is of interest where the plagiarist has too little time or skill to make sufficient alterations to their submission. Many academics have found that unusual features, such as identical spelling mistakes, peculiar layout, or errors, in a pair of assignments have triggered their suspicion [52, p.191]. Hoad and Zobel [110, p.2] state that *"In plagiarized assignments, it is common to find that some errors or atypical usages have been copied verbatim."* Cosma supports this view, saying that *"Suspicious files share similar source-code fragments which characterise them as distinct from the rest of the files in the corpus."* [52, p.23].

Of the 29 approaches analysed, only one, by Ribler and Abrams [199], specifically targets unusual similarity between files. Moss [3, 210] also offers the option to exclude code appearing in more than m documents, where m is set by the user, and can exclude template code. However, the example given in Appendix R shows that its matching technique will not pinpoint unusual code in every file type. Full details of Moss' matching techniques are not publicly available, but from this example, the tool appears to exclude header information, and to parameterise, so that while it would match unusual structures, it appears that, in some cases, parts of the code with a commonly occurring pattern are greedily matched. Four other approaches emphasise less common features by weighting input vectors [52, 115], or by using a weighted similarity measure [9, 34].

2.4 Summary

In this chapter, methods for finding similarity in program code are described. Clone detection and plagiarism detection are compared and their use in origin analysis is discussed. The processes involved in comparing source code are explained as a background to the survey of approaches to source code plagiarism detection. In the survey, twenty-nine approaches to source code plagiarism detection are analysed, in particular to find out whether the approaches are language dependent and whether unusual similarity is taken into account. Among these approaches, only Ferret [146] and PlaGate [52] are able to analyse almost any mix of programming languages. This survey finds that although unusual similarity is said to be an indicator of possible collusion, few of the approaches take this into account [3, 9, 34, 52, 115], with only one specifically targeting elements shared by few files [199] and none directly measuring them. The approach to collusion detection reported in Chapter 9 is based on finding these unusual similarities between source code submissions.

Key	Meaning	Key	Meaning	Key	Meaning
C	Comments	I	Identifiers	P	Punctuation
C-1	Comments replaced by single token	K	Macros	S	Single character identifiers
G	Globals	L	Literals	U	Unique terms
H	Headers	M	Imports	W	White-space
		N	Numbers	?	Possibly other, unknown

Table 2.2: Key to the elements removed from code during preprocessing by the plagiarism detection tools analysed in Table 2.3, column 4.

Author	Ref	Date	P.R.	TI	Transformation detail	Elements matched	Similarity measure	Reporting [Remarks]	Exc.	Dis.
Wise YAP3	[246]	96	C, I L	Tk	Synonym unification, functions rearranged by call sequence	Hashed n-grams where n reduces iteratively	Running Karp-Rabin Greedy String Tiling	% Coverage by matched tiles		
Aiken Moss	[3] [210]	97	W, ?	Tk	Token n-grams, hashed	Winnowed hash values to give fingerprints	Matching fingerprints trigger more detailed text matching	Similarity as %, no.lines& tokens	✓	*
Gitchell and Tan SIM	[89]	99	C-1	Tk	Parameterised, e.g. TKN-ID-I, TKN-FOR	Token strings One program against modules of the other	Normalised alignment ids =, 2; #, 0; gap, -2; other tokens =, 1; #, -2	High scoring pairs displayed		
Joy and Luck Sherlock	[120]	99	- C, W C, W	- - Tk	1. Original code 2. Excluding C, W 3. Tokenised	Each set of lines	Runs of equal lines, may be gapped, as prop n of file size	Similar files clustered by Kohonen SOFM [Since updated]		
Ribler&Abrams Categorical patterngram	[199]	00	W	-	Character n-grams (optionally parameterised)	Hashed n-grams	Matching hash-values	Graphical display for results	✓	✓
Jones	[117]	01	C, W	Tk	Counts of lines words, chars, length, vocabulary & volume as YAP3	1. Code attribute vectors 2. Error log att. vectors 3. Execution log att.vecs.	1. Euclidean distance 2. Compilation log dist. 3. Execution log dist.	Pairwise similarities		
Prechelt et al. JPlag	[189]	02	C, I L	Tk				as YAP3 with hashing optimised	✓	
Belkhouche et al. Brass	[18]	04	C, W	Tr	Partitioned structure chart and data dictionary	Subgraph type sequ's. Nodes Type frequency Ids	L.C. subsequence Subtrees & token types % matched nodes, types & matched type-id pairs	Pairwise % Matches, near-matches, & nodes [Tiered: 1. subgraphs, 2. nodes, 3. symbols]		

Table 2.3. Approaches to plagiarism detection

Continued on next page

Author	Ref	Date	P.R.	TI	Transformation detail	Elements matched	Similarity measure	Reporting [Remarks]	Exc.	Dis.
Chen et al. SID	[41]	04	-	Tk	Token sequences	Compressed files (Cm)	Compression distance $1 - \frac{Cm(a) - Cm(ab)}{Cm(ab)}$	File pairs ranked by similarity		
Lancaster and Tetlow	[144]	05	-	Tk	1. Words 2. Parameterised 3. Unprocessed	'Word' bigrams LCS Compressed files	$100 \times \text{Jaccard coeff't}$ $200 \times \frac{\text{Tokens in matched sections}}{\text{Total tokens in file}}$ $100 \times \frac{Cm(a)+Cm(b)}{Cm(ab)+Cm(ba)} - 1$	[Test framework to compare 3 measures - finds bigrams best]		
Moussiades & Vakali PDetect	[173]	05	C	Tk	Indexed keywords e.g. {void1, int1, int2, for1, ...}	Keyword set	Jaccard coefficient	Clustered on weighted undirected graph		
Mozgovoy et al. FDPS	[175]	05	W	Tk		Token sequences using suffix array	$\frac{\text{Tokens in matched sections}}{\text{Total Tokens in file}}$	Similarity matrix		
Arwin & Tahaghoghi Xplag	[9]	06	C, W	IL	Converted to RTL optimised, tokens filtered	N-grams of selected tokens	BM25	Relative % score [Inter-lingual]	*	*
Lane et al. Ferret	[146]	06	W	Tk		Token trigrams	Jaccard coefficient	Pairwise similarities		
Liu et al. Gplag	[151]	06	C, I W	Gr	Procedural program dependence graphs (PDGs)	Graph pairs, filtered to exclude small units and unlikely matches	Subgraph isomorphism	Counts of approx. matched procedures		
Merlo CLan	[170]	06	C, W	Mt	Function metrics (branch, parameter, etc. counts)	Metric vectors	Clone clusters based on thresholds	Proportion of files covered by clones	?	?
Noh et al. EXPDec	[181]	06	C, W	Tr	1. XML tree to 6 sets of feature frequencies 2. Control sequences	Feature matrices Control matrix	Comparison depends on the features Levenshtein (weighted)	Pairwise similarities		

Table 2.3. Approaches to plagiarism detection

Continued on next page

Author	Ref	Date	P.R.	TI	Transformation detail	Elements matched	Similarity measure	Reporting [Remarks]	Exc.	Dis.
Burrows et al.	[34]	07	C, W	Tk	Parameterised to 1 of 55 token types, e.g. int S, return g, (A, = K	Token type 4-grams	1. BM25 2. String alignment on best matches match 1, indel -2, mis -3	Pairwise similarities	*	*
Ahtiainen et al. Plagge	[2]	07	C, I L	Tk	as JPlag	-- --	-- --	[as JPlag but open source]	✓	
Freire et al. AC	[82] [81]	07	U	Tk	1.Token type counts 2.Token sequence	1. Token frequ. vectors 2. Compression (Cm)	Cosine distance $\frac{Cm(a b) - \min(Cm(a), Cm(b))}{\max(Cm(a), Cm(b))}$ and variance analysis	Pairwise similarities File cluster graph Relative sim'y histogram		
Ji et al.	[115]	07	C, W	Tk	Static trace "keywords" e.g. IF BLOCKSTART RETURN FUNCCALL	Local alignment with scores weighted on keyword frequencies	Dice coefficient aligned section to file sizes	Similarity matrix	*	*
Cosma PlaGate	[52]	08	C, N P, S, U	Tk	Term frequencies	File level term vectors resulting from latent semantic analysis	Cosine similarity	Similarity matrix/plot	*	*
Jadalla & Elhagar PDE4Java	[114]	08	C, W	Tk	Parameterised, unifying identifiers, separators, keywords .. as digits	Bag of 4-grams of the digits	Jaccard coefficient	Pairwise similarities Clusters similar files using DBSCAN		
Chilowicz et al.	[42]	09	C, I W	Tk	Parameterised, unifying some token types	Token suffix array Global call graph of subfunc'ns inc. shared	Jaccard coefficient & containment based on subfunctions	Pairwise file similarities sim, cont1, cont2		
Xiong et al. BUAA_ AntiPlagiarism	[253]	09	C, W) H, G	Tr	CIL to AST in linear representation, tokenised	Token 4-grams	Jaccard coefficient	Clusters similar files		
Table 2.3. Approaches to plagiarism detection										
										Continued on next page

Author	Ref	Date	P.R.	TI	Transformation detail	Elements matched	Similarity measure	Reporting [Remarks]	Exc.	Dis.
Brixtel et al.	[31]	10		Tk	Alphanumerics unified e.g. $t = t + t$;	Choice e.g. lines, functions	Choice, any suitable measure for sequences	Similarity matrix [1-to-1 section matching by Munkres algorithm]		
Hage et al. Marble	[99]	10	C, M S	Tk	Tokens unified by type Optional function sort	Transformed lines	L.C. Substring (diff) Similarity = $100 - \frac{\text{no. different lines}(f1, f2) \times 100}{\text{length } f1 + \text{length } f2}$	Pairwise similarities above threshold score		
Huang et al.	[112]	10	C, K W, ?	Tk Tr	1. Identifiers unified, concatenated into string 2. 4 AST node type sets	1. Winnowed n-gram hash values = fingerprint 2. Nodes paired on sim. of similarities 1 & 2	1. Dice. $\frac{(2 \times \text{LCS}(f1, f2))}{\text{length}(f1 + f2)}$ 2. $\sum_{i=1}^4 w_i \cdot (\text{mean sim})_i$ weighted combination	Pairwise similarities		
Juričić	[122]	11	C, W	IL	Converted to Common Int. Language, filtered to exclude metadata & locations	Machine instruct'n string e.g. ldc, stloc, add	$1 - \frac{\text{Levenshtein}}{\text{Maximum filesize}}$	Similarity matrix		

Table 2.3: Approaches to source code plagiarism detection. Author and **tool** names are listed in column 1. Parts of the code removed during preprocessing are in column 4, comments (C) and white-space (W) are most common, as these can be a by-product of transformation, other keys are in Table 2.2. The initial transformation of the code is shown in the next column, (TI), where Tk means that the code is tokenised, other keys are: Gr-graph, Tr-tree, Mt-metrics, and IL-intermediate language. Brief descriptions of further transformation, elements matched, and similarity measures are in columns 6-8. Column 9 has report formats and [selected remarks]. The last 2 columns show whether the tool allows **exclusion** of template code, and whether similarity between files which is **dissimilar** to the rest of the group is considered. [\surd - explicit, * - implicit]

Chapter 3

Origin analysis

In this chapter, origin analysis is introduced, along with a survey of related approaches. Origin analysis is the study of the movement of code within an evolving software system and is thus a motivating example for the techniques proposed in this dissertation.

Software systems change over time in response to both internal and external factors [17]. The changes generally result in increased complexity [148], which may lead to restructuring to simplify the system. These changes may occur under conditions such as time or cost constraint, leaving documentation incomplete or absent [40, 67]. The missing documentation and the often large amount of code make it difficult for subsequent developers to track changes in the system. This lack of knowledge can lead to problems such as unexpected side-effects when changes are made to the system [6], duplication of effort, or time taken in comprehension [51, 159].

It is important to be able to discover how code has been restructured, both to avoid these problems, and for other reasons. For example, Antoniol et al. [6] give three reasons why software engineers find it useful to know about changes caused by restructuring. First, those maintaining the software will want to be able to trace the system functionality. Second, those testing the system will want to refocus system tests appropriately. Third, those studying software development will be affected by the apparent discontinuity of program elements and want to understand the changes.

Godfrey and Zou [90, p.1] support this last view by saying that “having an accurate evolutionary history that takes structural changes into account is also of aid to the research community”.

Analysis of the version history of a program, available in CVS or SVN repositories, provides one means of recovering information about such changes. Fowler defines refactoring as making small changes to the internal structure of a program which do not change its external behaviour, and says that a system can be restructured through a series of refactorings [78]. Murphy-Hill et al. [176, p.3] point out that there are four main methods used to find refactorings in a system, each with strengths and weaknesses:

1. Watching programmers at work is expensive in terms of programmer and researcher time, and may be limited in the data available over the period of observation, but can provide detailed information.
2. Recording refactoring tool use is accurate when the tool is used in refactoring, but not informative when the tool is not used.
3. Searching the change log for key words relating to refactoring, helpful when the log is complete and accurate, but this is often not the case [187, 245].
4. Analysing source code history, either manually, which is very time-consuming and not necessarily complete; or by automatic detection, which although faster, limits the type of refactorings detected.

The term “origin analysis” was first used in a software evolution context by Tu [236, 237], to describe the process of matching apparently new program entities in a software system to those apparently disappearing from the previous release. The term was later applied not only to complete entities, but to code which moves as a result of splitting and merging [262].

Apart from the introduction of new functionality, new entities may arise in a system from renaming or moving existing ones, or from splitting, merging or recombining entities. Examples of each of these restructurings (here files) are shown in Figure 3.1, where the original file(s) are shown on

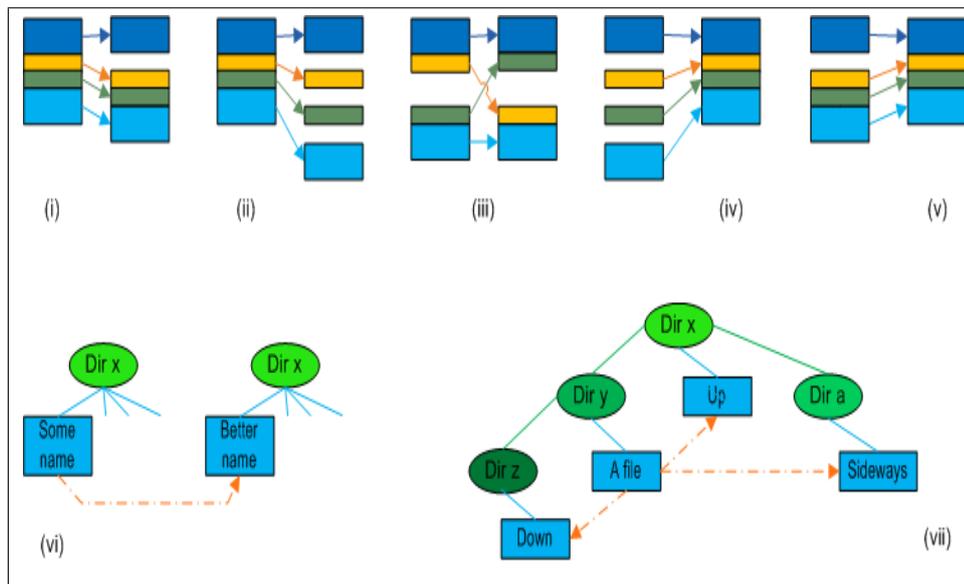


Figure 3.1: File evolution: splits:(i)-(ii), merges:(iv)-(v), split and merge, or recombination:(iii), rename:(vi), moves:(vii)

the left, and the result of restructuring is on the right of diagrams (i) to (v). Diagrams (i) and (ii) show a file split into two or more parts, in diagrams (iv) and (v) the process is reversed as the files merge. In diagram (iii), two files are recombined, one section of each file is moved to the other file; in effect each file has been split and then merged with the other partial file. In Figure 3.1 (vi) a file is renamed, remaining in the same directory. Diagram (vii) shows a file in directory y moving in three ways: up or down one or more levels, e.g. to directory x or z; or into a new directory, e.g. directory a. The processes may be combined. For example, a file may be split, have its name changed, and move directory at the same time.

As noted by Murphy-Hill, and by other researchers (for example, [6, 60, 64]), manually tracing restructured files is very time-consuming, therefore an automatic approach to the detection of these files is desirable.

3.1 Survey

Table 3.1 (p. 46) provides information about selected work related to automatic origin analysis. The first three columns identify the author, paper and date. The next column shows the programming language of the code studied: **Smalltalk**, **C/C++**, or **Java**.

All of these approaches require the code to be parsed before entities, and facts about them, are extracted. In many cases this is lightweight parsing, for example, to extract method headers. The column headed *A/C/M* shows whether full parsing is needed, either to build an abstract syntax tree, a control flow graph or to compute **metrics**, all language specific processes.

The problem of tracing software evolution by comparing consecutive versions of a program is tackled in various ways by the different research groups. Five other variables are noted in the table:

- the granularity of the entities compared,
- the amount of activity between versions,
- the complexity of the matching task,
- the representation of the code,
- and the method used to match these representations.

Granularity of entities: Kim and Notkin [127] state that versions are matched at different granularities depending on the task. For example, file-level matching is useful in program understanding, while more fine-grained units are matched for more precise tasks. The columns headed **Entity** and **Detail** show the entities matched, **Package**, **Class/File** or **Method/Function** and whether detailed differences are found, such as those between signatures, parameters, or variables. When more than one entity is listed, comparisons are usually made in detail at the lower level, and higher levels are computed from an aggregation of information about the lower level entities they contain. For example, the Weißgerber group [22, 91, 245] combine method level changes to find class and package level changes.

Granularity between versions: Murphy-Hill et al. point out that working at coarser intervals between versions, between releases, rather than transactions, can mean that details of refactoring activities go undetected. While this loss of fine detail is a problem if programmer behaviour is of interest, it is of less concern where the aim is to track structural changes to program elements. The column labelled “Int’val” shows the interval between versions: Release or Commit/Transaction level. The majority of those surveyed work at release level. Transaction level analysis has the advantage of reducing the number of possible matches, and of changes between consecutive versions, while the disadvantage is the increase in the volume of changes to be analysed in any time frame.

Matching complexity: There are two main approaches to matching program entities between versions. The simplest looks for renamed or moved entities, matching those whose names have disappeared from one version to those with new names in the following version, e.g. [64, 179, 237, 252]. In this task, if version $n+1$ has t new entities, then each disappearing entity from version n has $t+1$ possible “destinations” (t entities and deletion).

The more complex approach also considers split and merged entities [6, 90], when possibilities for matching increase exponentially. The set of candidate entities is larger, as it will include not only those which have disappeared, but also those whose changes indicate that they may have split or merged. The target set of entities includes not only new entities, but also all existing entities. Each disappearing entity can be split into any (reasonable) number of sections, which can be placed in any of the target entities, making the number of possible combinations of destinations for each candidate 2^t where t is the total number of entities in version $n+1$. In practice, this number would be $\Sigma_1^k \binom{t}{k}$ where k is the maximum number of likely splits.¹ The value of k is further limited to two for approaches which consider only two-way splits or merges, such as that of Antoniol et al. [6].

The column labelled S/C shows whether the approach is simple, match-

¹The maximum found during this research was an 18-way split.

ing whole entities, or complex, matching split and merged entities as well. The difference between the two is blurred, in that if methods are matched one-to-one, this is a simple approach. However, a method moved to a new file is a split at file level, unless all of the methods in a file are moved together, when it is a file renaming.

Code representation: The elements, or representations of the entity, which are matched are listed in the last but one column, with a brief description of the technique or similarity measure in the last column. Most approaches initially match by name, reasoning that an entity with the same full name as one in the previous version is the same (if edited) entity. Once the matched entities are filtered out, matches between the remaining items are often based, at least in part, on string similarity; 9 of the 15 examples [22, 90, 91, 128, 130, 179, 237, 245, 250], match method header elements like this.

Matching is also based on comparing metrics [90, 130, 237], text [6, 22, 77, 130, 245], incoming or outgoing calls [64, 90, 130, 237, 252, 250], graphs [8], or trees [22, 77, 179]. Many approaches combine two or more techniques for comparing code, or use a simple method to filter possible matches and a more precise method to refine the search (e.g. [22, 64, 252]).

Matching technique: As discussed in Chapter 2, the representation of the code guides the matching techniques and similarity measures chosen for the task. Each group's approach to matching is described in Section 3.2.

Authors	Ref	Date	Lang.	Parse	A/C/M	Entity	Detail	Int'val	S/C	Match	Technique / Similarity measure
DeMeyer et al.	[60]	00	S	Y	M	C, M	-	R	S	Metrics	Match to heuristic combinations
Tu & Godfrey	[237]	02	C	Y	M	M	-	R	S	Metrics Name Calls	Euclidean vector distance LCSq (longest common subsequence) Match callers and callees
Antoniol et al.	[6]	04	J	Y	-	C	-	R	C	Body	Cosine between tf-idf vectors of identifiers
Apiwattanapong et al.	[8]	04	J	Y	C	C, M	-	R	S	Extended CFG	Proportion of nodes in hammock graph [73]
Zou & Godfrey	[90] [262] [261]	05	C	Y	M	M	-	R	C	Metrics Parameters Name Calls	Sum of normalised distances Dice coefficient on LCSq Dice coefficient on LCSq Dice coefficient
S.Kim et al.	[130]	05	C	Y	M	M	-	C	S	Body Name Calls Metrics Method header	diff[63], CCFinder[124], Moss[3] LCSC, ISC (intersecting bag of words) LCSC, ISC Normalised distances LCSC, ISC
Görg & Weißgerber	[91]	05	J	Y	-	C, M	Y	C	S	Method header	Equality of names or types of elements
Neamtiu et al.	[179]	05	C	Y	A	M	Y	R	S	Func'n-level AST	One-to-one name and type matching
Weißgerber & Diehl	[245]	06	J	Y	-	P, C, M	Y	C	S	Method header Body	Changes (for detection) CCFinder (for ranking)

Table 3.1. Origin analysis and related work

Continued on next page

Authors	Ref	Date	Lang.	Parse	A/C/M	Entity	Detail	Int'val	S/C	Match	Technique / Similarity measure
Dig et al.	[64]	06	J	Y	A	P, C, M	Y	R	S	Method body Calls	Hashed bigrams (for candidates) [32, 193] Mean of directed similarities
Xing & Stroulia	[251] [252]	06	J	Y	A	P, C, M	Y	R	S	Name Entity contents, calls and uses	Dice coefficient of character bigrams Normalised intersection of equal items
M.Kim et al.	[128]	07	J	Y	-	P, C, M	Y	R	S	Method header	LCSq (names split on caps)
Fluri et al.	[77]	07	J	Y	A	P, C, M	Y	C	S	AST Leaves	Tree edit distance adapted from [38] Dice coefficient of character bigrams
Wu et al.	[250]	10	J	Y	-	P, C, M	Y	R	C	Method header Calls	Lev'n.Dist, LCSq (names split on caps) Ratio of common to total, known matches
Biegel et al.	[22]	11	J	Y	A	P, C, M	Y	C	S	Method header Body	Changes (for detection) CCFX/AST/Bigrams (for ranking)

Table 3.1: Origin analysis and related work

Key. Lang: S-Smalltalk, C-C/C++, J-Java

Entity: P-Package, C-Class/File, M-Method/Function

Interval: R-Release, C-Commit

A/C/M: need to construct AST, CFG or to calculate Metrics

Detail: Y-detailed differences found (e.g. in signatures, variables)

S/C: S-Simple, C-Complex

3.2 Approaches in detail

In this section, each of the approaches listed in Table 3.1 is described in more detail, for later comparison with the approach used in this research.

Tu and Godfrey [237] introduced the tool Beagle, which used three matching algorithms to find renamed and moved functions in a system, and so to reason about files and subsystems. Matching was first based on the Euclidean distance between vectors of five complexity metrics [135]. The top five matches were then ranked based both on textual similarity of the function names and on call dependency analysis.

Zou and Godfrey [90, 261, 262] extended Beagle, both to alter the matching techniques and to add a declaration matcher. Parameter names in a function's declaration are concatenated alphabetically. The Dice coefficient is used to compute similarity between these parameter strings, as well as between function names, and the sets of callers and callees of two functions. However, the user must select a combination of the four matchers and their thresholds to define a match.

Zou and Godfrey also reason about other restructurings. They consider three patterns of two- or multi-way splits or merges: (clone introduction or elimination, service extraction or consolidation, and pipeline expansion or contraction) based on the manual analysis of call relations between functions. For example, in clone extraction, incoming calls to an extracted clone should be the union of the calls to the original functions, while the outgoing calls of the functions and the extracted clone should be much the same.

Basing their ideas on those of Zou and Godfrey, S.Kim et al. [130] look for renamed functions automatically. Their approach has several differences. First, to the similarity metrics, where adapted features based on function names, calls, complexity metrics and signatures are combined with the similarity measures of three tools which match the source code text: the strict line-based matching of diff [63], the more flexible clone detection tool Code Clone Finder (CCFinder) [125], and the plagiarism detection tool, Moss [210]. Second, they create an oracle set of matches agreed on by a minimum of seven of their panel of ten judges, but reinforce the point made

by Walenstein et al. [242] that people's opinions vary. The oracle sets consist of approximately 85% of the Apache and 91% of the Subversion examples. Each metric is weighted depending on its accuracy in predicting whether the oracle pairs match. The best combination of metrics combines diff, names, signatures, and complexity metrics. The dominant factors overall are diff, outgoing calls, and function names, while individually CCFinder and complexity metrics are least significant. In addition, the weights from a subset of the examples in the oracle set from one project are applied to the rest of the examples in the project, and to the whole of the other project's oracle set, with 97% and 86% success respectively. However, 9–15% of the possible pairs are not agreed on by sufficient judges to be included, meaning that the set excludes the examples which are more difficult to categorise.

In work pre-dating that of Tu and Godfrey, DeMeyer et al. [60], also use metrics to find refactorings. Rather than complexity metrics [130, 237, 261], these are count-based metrics, such as the number of class variables, or the depth of the inheritance tree. The metrics are not used directly as a similarity measure, but rather heuristics about combinations of changes to the metrics are used to identify class or method refactoring. For example, split methods are assumed to have reduced numbers of statements, lines and messages sent. To find factored out code, they look for similar reductions in other methods. However, although the search space is reduced, the suggested destinations for the code must be browsed manually to determine the exact location. Of the possible splits or merges they found at class level, 96 of the 101 cases found in their three test systems were false positives, although 19 did belong to other classes of refactoring.

Görg & Weißgerber [91] collect names of classes, and information about methods, before and after a transaction.² From this information, a subset of possible refactorings are found: rename, hide/unhide, or move methods, add or remove parameter, and move class. This approach is limited in that to match two entities only one element can have changed. For example, a renamed method will not be detected if the parameters have also changed.

²A group of commits performed within a set time frame and so presumed to be related.

Weißgerber and Diehl [245] build on the approach described in [91]. Local refactorings are found in a similar way. For structural refactorings, where entities move, or classes are renamed, candidate pairs are found from the whole system. Candidate pairs are compared using CCFinder [125], to find identical bodies, non-identical clones or unmatched code.

Extending this research, Beigel et al. repeat the work in [245], both with the original filter criteria and with a relaxed filter to give more candidate refactorings. They investigate three similarity measures for comparing entity bodies to rank potential targets: text-based - the Jaccard coefficient of (token) bigrams [32], token-based - CCFinder, and AST-based - JCCD [21]. The outputs from CCFinder and JCCD are used to compute Weißgerber's CloneFraction metric [244], which is the proportion of the tokens in the code covered by clones, to the total number of tokens. They found that the three measures overlap in 50-60% of cases, with each having its strengths and weaknesses for the task. Text-based comparison was marginally more effective than the other two methods in ranking the candidates in this study.

Dig et al. [64] also take a two stage approach, first performing a text-based analysis by comparing two entity bodies based on token bigrams. The containment of each entity in the other is used to measure similarity. Second, they perform semantic analysis based on the calls and references between two entities. The similarity between two entities is the mean of their directed similarities, that is the ratio of the common edges to their total.

Control flow graphs (CFGs) are compared by Apiwattanapong et al. [8] to match changed interfaces, classes and methods. They introduce an enhanced CFG to include features found in object-oriented code, such as types. Before matching entities, their graphs are simplified to become a series of minimal hammock nodes [73].³ If one simplified graph is sufficiently similar to another, the graphs are expanded for detailed matching.

Two groups use abstract syntax trees as the basis for matching the entities. Neamtiu et al. [179] aim to find changes to types and variables in

³Hammock nodes are induced subgraphs with a single entry and a single dummy exit.

a system. They match functions by name, then traverse the AST from top to bottom, noting the identifiers and types which have changed between versions. The calls to and from already matched functions are used to infer matches between functions which have changed their names.

A bottom-up approach to matching ASTs is taken by Fluri et al. [77], who categorise changes which occur between commits. The leaves of the tree are matched first, and must have the same label, meaning that they represent the same type of operation. To match, the string similarity of the leaf values, based on character bigrams, must exceed a threshold. Second, the inner nodes of the tree are matched, by comparing the nodes in their subtrees. To match, their Dice coefficient must reach a threshold, which varies depending on the size of the tree.

Xing and Stroulia [252] use their tool UMLDiff [251] to extract a model of the system where nodes are packages, classes, interfaces, fields or methods, and the edges are labelled with their dependencies. Similarity between nodes is assessed in two ways. First, using the Dice coefficient of character bigrams of node names. Identically named nodes are stored to allow a recursive search for entities with similar names and structures. Measurement of structural similarity is based on “relevant facts” for the entity. For example, the classes and interfaces in a package, or the parameters, fields and calls of a method. The structural similarity between two entities is the normalised intersection of identical or previously matched relevant facts.

M.Kim et al. [128] base their work on matching method headers between program versions. Non-identical methods are matched on the longest common subsequences among tokens in the method headers, where names are split on capital letters (assuming camelCase style). Unmatched items are taken to be deletions. General descriptions of sets of matched pairs are formed by replacing parts of their names by wildcards. A rule is formed if the support for a set is above a threshold. Exceptions to the rules are also noted, as they may be changes which have been missed. This approach is reported to be robust to combined rename and move operations, but can fail when method headers have little textual similarity or when unrelated

methods have similar names.

Wu et al. [250] build on the work of M.Kim et al., using call dependencies as well as method headers in matching. Their approach handles one-to-many and many-to-one changes, providing rules which cover groups of changes, along with exceptions to these rules. In the same way as Xing and Stroulia, they start with a set of matched entities, and recursively add new matches based on those previously found. Two complementary matches are performed on the tokenised header sequence: textual difference is calculated using the Levenshtein distance, and textual similarity with the longest common subsequence. Call dependency similarity is the ratio of shared calls to or from already matched entities, to the total of such calls. This value is used to rank the potential matches.

In general, text-based comparisons of the entity bodies are made in conjunction with other similarity measures. S.Kim et al., Weißgerber and Diehl, and Biegel et al. use the similarity between method bodies to rank the target files found by coarser matching. Conversely, Dig et al. find potential matches by comparing method bodies and then refine on call relations. Antoniol et al. are the only group of those listed in Table 3.1, to use text-based analysis on its own, although they suggest that their results may have been improved by adding further matches to their identifier-based system, such as matching class names, or finding clones in the text.

Antoniol et al. [6] use identifier names to investigate class level refactorings, specifically class merge, split, rename, move and recombination. They note that this analysis is applicable at other granularities, such as method level. Their approach uses an information retrieval method known as vector space modelling. Vectors of the weighted frequency (tf-idf [206]) of identifiers in the program code are compared. When a class (A) is split into two parts (A' and B) with no other changes to the text, the sum of the vectors for A' and B will be the same as the vector for A. With editing, the two vectors should still be close in vector space, and the cosine between them be high. Similar reasoning is applied to renaming, merging and recombination.

Rainer et al. [194] proposed that trigram analysis of source code text

Compared by	Author	Ref.	Software projects
M.Kim et al.	S.Kim et al. Weißgerber and Diehl Xing and Stroulia	[130] [245] [251]	jEdit and ArgoUML jEdit and Tomcat JFreeChart
Wu et al.	M.Kim et al. Schafer et al. Dagenais and Robillard	[128] [208] [56, 55]	JHotDraw, jEdit and JFreeChart JHotDraw and Struts eclipse.jdt.debug.ui, Mylyn and JBossIDE

Table 3.2: Projects used by M.Kim et al. and Wu et al. to compare their work with others

could be used to characterise aspects of the evolution of a software system. Their study looks at changes to an individual file through its lifetime, at patterns of change across a release, and at files which split and merge. They studied two (C) projects, measuring similarity between files based on the trigrams present in the code. Apart from the requirement for a lexical analyser, this approach is language-independent. To find split files, they find the similarity between files in consecutive releases. Potential source and target split file pairs are those which have different names, and have a similarity to each other which is greater than the mean plus one standard deviation of all pairs. They find 433 such candidates in nearly 8,000 pairs, and find example split files, but do not go on to categorise all of the pairs, as this requires visual inspection of each candidate pair. They discovered that these criteria also select renamed or moved files, and although they do not report any such examples, merged files would also be selected.

M.Kim and Notkin [127, p.59] noted the absence of benchmarks in matching program code elements. Researchers typically check the changes found by inspecting the code themselves. However, S.Kim et al. [130] use a panel of judges to provide classification, and two of the groups surveyed compare their results to those of previous research groups. These groups are listed in Table 3.2 together with the projects they analysed.

3.3 Summary

Sixteen approaches to origin analysis and closely related work are reviewed in this chapter. Each one, except for that of Rainer et al., requires that the code be parsed before analysis, making them language dependent. The majority look for restructuring at the method or function level, only Antoniol et al. and Rainer et al. study systems solely at class or file level.

There are differences in the granularity of the analyses: four of the approaches analyse changes at commit or transaction level, while the others work at release level. Most of the approaches match only whole entities, in other words, those which are renamed or moved, rather than matching more complex types of restructurings, such as splits.

Four of the methods use one feature of the code in matching, while the rest combine a number of different measures of similarity. These combinations differ in their application: in some cases the features are used together, and in others they are applied in two passes, with a finer-grained match applied to the result of a preliminary coarser-grained match.

Third-party similarity detection tools are used in three of the approaches, each one uses Code Clone Finder [125], with one [130] also using diff [63] and Moss [3], and one also using JCCD [21]. Apart from Biegel et al.'s use of the CloneFraction metric [244], these methods use measures taken directly from the similarity detection tools.

Although many approaches use thorough searches to find threshold values for similarity measures, or a fairly complex set of rules to make decisions about what to match, none of the approaches uses machine learning.

The approach to origin analysis taken in this research is described in Chapters 10, and subsequent chapters. In outline, the approach is based on that of Rainer et al. [194] in the use of n-gram analysis on source code text for comparing files, a method also used by Dig et al. [64]. It also combines elements from other approaches:

- text-based comparisons at file (class) level [6, 194]
- considering splits and merges [6, 90]

- ranking targets based on comparison of source code text [22, 91, 245]
- use of third-party file comparison tools [22, 130, 245]
- combination of similarity measures [90, 130, 237, 250]

Several aspects of this research differ from previous approaches:

- development of a range of descriptive measures based on clone and plagiarism detection tool output
- comparison between varied file groups
- the use of machine learning to find a suitable set of features
- the use of a range of projects with the aim of generalising
- experiments to test applicability across both projects and languages

Note on the use of the terms candidate and target:

Some origin analysis research groups (e.g. [6, 250, 261]), use the term *target* to mean the entity which has been restructured (old), and *candidate* to describe potential matches (new). Other groups (e.g. [128, 130]) use the word candidate to describe possible matched pairs or rules (old → new).

In this dissertation, as in [22, 245], candidate means an entity which may have been restructured, in other words, the source of code which has moved, and target means the destination of the code.

Chapter 4

Machine learning

This chapter gives an overview of machine learning, and explains its use in this research. The uncertainties involved in creating a feature set for novel data are the main focus of the chapter.

4.1 Why use machine learning?

Machine learning has developed from ideas taken from statistics, artificial intelligence, information theory, data management, psychology and neuroscience, among others [104, 164, 171]. Samuel described machine learning as *“a field of study that gives computers the ability to learn without being explicitly programmed.”* [207].

Within the space of computer applications there are some tasks which can be defined as a precise set of instructions for a computer. There is another set of tasks which are too complicated to be explicitly defined, but which can be approximated using machine learning [24, 172]. Broadly, three main groups of tasks fall into this second category [104, 172, 180]:

1. those which are best described by giving labelled examples, such as machine vision tasks,
2. very large datasets containing interesting, but difficult to find, relationships, for example, astronomical or genomic data, and

3. machines which need to adapt their behaviour according to their environment, such as robots in the field.

The task in this research falls into the first category. Finding a set of text-based features to characterise a file belonging to one category of restructured file is very difficult because of the large number of variables. These variables include the quantity of code moved between files, the size of the enclosing files, the multiplicity of target files, the amount of editing, both in the code which has moved and in the files involved in the move, the distribution of the transferred code in the target file(s), the inherent similarity between files in a project, and the variation in style between projects.

4.2 The machine learning process in outline

The input to a machine learning system is a set of examples from the domain of interest. These examples, the training set, take the form of feature vectors. The aim is for the machine learning algorithm to adapt in response to the provided data, to create a useful model (also called the output function, or hypothesis) [24, 171] (see Figure 4.1). Although machine learning algorithms learn by fitting to the examples they are given, they also need to generalise to the population from which the examples are taken, and not overfit to the training data, for example, by rote learning.

Machine learning tasks can be divided into two main categories, supervised and unsupervised learning; other categories include semi-supervised

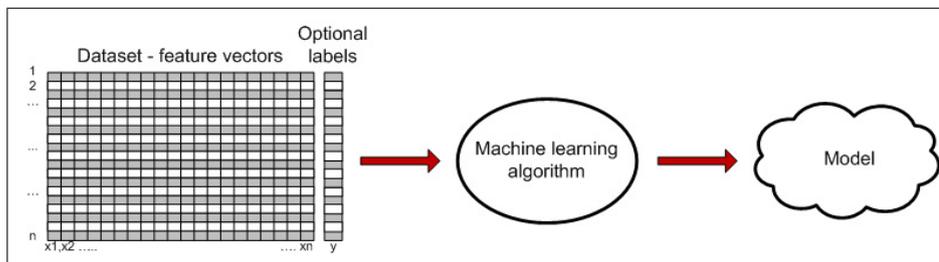


Figure 4.1: Examples (1...n) in the form of feature vectors ($x_1 \dots x_n$), with or without labels [y], are input to the machine learning algorithm, to create a model

learning [256] and reinforcement learning [228]. In unsupervised learning, the aim is to find patterns in the data. For example, by clustering, which means partitioning the data into a number of groups where the members are related to each other under some criteria; or by association mining, which looks for relationships between features, especially those which occur frequently.

Supervised learning, based on labelled data, aims to find links between the input and expected output. There are two main tasks in supervised learning: regression and classification. When the data labels are numeric, and usually continuous, the regression function aims to predict the numeric value of new examples. When the labels are nominal, the task is classification. In this case, the feature values are used to create a model which aims to separate the data into the labelled categories.

A number of steps are recognised in the process of creating a model [27, 72, 161], which, in outline, can be thought of as:

- understanding the domain,
- gathering the data,
- creating, selecting or combining the features,
- cleaning and preparing the data,
- allocating labels, where appropriate,
- choosing learning algorithms, and
- building and refining the model.

In broad terms, the resulting model is either interpreted to discover facts about relationships in the input data, or, as in this research, is used to predict the outcome for previously unseen data (see Figure 4.2).

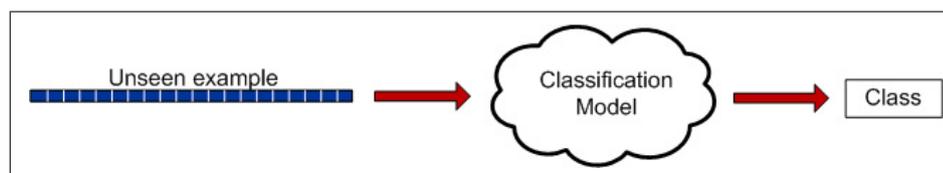


Figure 4.2: The classification model labels each unseen instance

4.3 Creating new datasets for machine learning

There are two parts in this section, the first explores the difficulty of deciding on a set of features when creating a dataset in a new domain, and the second outlines the steps taken in creating a labelled dataset from raw data.

4.3.1 Choosing features

Features describe the characteristics of each instance and therefore the choice of which features to use has a major role in the outcome of the learning task. Datasets collected by a third party and available ‘off-the-shelf’, for example the UCI sets [79], will have a ready-made set of features. Other datasets are made by selecting information from one or more existing databases [27], when the features will be determined by the information available. However, when features are not available, they must be generated. Guyon and Elisseeff [98, p.2] state that *“Finding a good data representation is very domain specific and related to available measurements.”* When no such measurements exist, then information must be collected and analysed to create the dataset.

It is not always obvious what information to collect. Ciesielski et al. say that *“There are few guidelines for choosing feature sets. Generally the only way to determine whether a particular set of features will be useful is by trial and error”* [44, p.1]. Pachet and Roy, in their work on audio analysis, state that *“In machine learning research, it is typically assumed that features naturally arise from the problem definition. However good feature sets may not be directly available, motivating the need for techniques to generate features from the raw representation of objects ...”* [184, p.1]. They go on to explain their use of evolutionary methods to build a set of features taken from representations which exploit domain knowledge.

In some circumstances, there may be limitations to the information that can be recorded. For example, because of the instruments available, or because of practical, financial or moral restrictions [141]. Without these limitations, the range of options can be vast. For example, as shown in

Chapter 2, when matching source code, there is a wide range of tools, of ways in which the code can be represented, of elements to be matched, and of similarity measures. Any combination can be used to provide information, which can then be analysed in various ways. Leather et al. support this view, in talking about creating features for compiler optimisation, they highlight the difficulty of selecting suitable features from a potentially infinite set [147, p.1].

Ludmila Kuncheva recommends a large set of features, stating that *“If the data set is not given, an experiment is planned and a data set is collected. The relevant features have to be nominated and measured. The feature set should be as large as possible, containing even features that may not seem too relevant at this stage. They might be relevant in combination with other features.”* [141, pp.1–2]. This idea is echoed by Guyon, who says that *“Although dimensionality reduction is often summoned when speaking about complex data, it is sometimes better to increase the dimensionality. This happens when the problem is very complex and first order interactions are not enough to derive good results”* [98].

In general, domain knowledge and the available tools will guide the initial measurements taken. Once this set of measurements is collected, additional features can be created, either by further analysing the data manually, or by some method of automatic feature construction. For example, evolutionary methods, such as those surveyed by Espejo et al. [70, pp.5–6], or pattern-mining methods as discussed by Bringmann et al. [30].

4.3.2 Creating a labelled dataset from raw data

In some datasets the items of interest occur infrequently, however, it may be possible to filter to remove those instances which are unlikely to belong to the category of interest. For example, in looking for restructured files, those which are exactly the same from one release to the next have clearly not been restructured. There is normally a trade-off between precision and recall in filtering. With relaxed criteria, all relevant instances should be selected, at the cost of selecting many irrelevant ones. If the criteria are too strict, then irrelevant instances have less chance of being selected, but

relevant instances may be missed.

Preparation is an important step in modelling data, as poor quality input will not lead to good models [104, 191]. Once a dataset is selected and features created, it is cleaned to deal with unusual or erroneous items [27]. Data prepared by others may have missing values which cannot be corrected, these can be treated as a new category where a feature has nominal values, but are more usually estimated, or the example is discarded [152]. Outliers, or unusual values, may be errors in measurement or data entry, when they should be corrected. True outliers are either omitted, which means information loss, or they are handled by a classifier whose outcome will not be unduly biased by their presence (e.g. a decision tree). In classification tasks, duplicate instances are normally removed, while in pattern mining, duplicates can show support for the patterns and be left in the set.

Where labels are not already available for a classification task, manual labelling is required, a task which is generally recognised as very time-consuming [25, 213] and requiring considerable effort, either on the part of the researchers or volunteers. For example, in software engineering research, Hindle et al. [108] manually classify 2000 commits using the Extended Swanson Classification scheme, and Buse and Weimer [35] asked 120 people to categorise 100 code snippets by assessing “readability”.

Once the data is labelled, another concern is the balance of the classes [140]. When the training set is imbalanced, learning algorithms can overlook the minority class. For example, if a dataset is split 99:1, then a prediction of the majority class for every instance gives 99% accuracy without finding any of the probably more interesting minority class. A set in which the classes are balanced is preferable. Two methods for arriving at an artificially balanced dataset, when one class is significantly smaller than another, are under-sampling and over-sampling [46, 107].

Under-sampling means making a selection from the majority class to match the size of the minority class. This strategy will generally be repeated for a number of partitions of the majority class to ensure good coverage of the instances. Otherwise, the minority class can be over-sampled to produce

new examples, using an over-sampling technique such as SMOTE [39]. This involves selecting a random data point and one of its n nearest neighbours, then creating a new instance from a random point between the two. This is repeated until the minority class is the same size as the majority class. As in under-sampling, over-sampling may be repeated with different random seeds. Alternatives to balancing the classes include adjusting the learning algorithm [45, 249], active learning [69], or cost-sensitive learning [215].

4.4 Classifiers

Choosing a suitable classifier is an iterative process. Pyle [191] says that “*Building any model should be a continuous process incorporating several feedback loops and considerable interaction among the components.*” This iteration involves the data, possibly adding more where the problem turns out to be more complex than first thought; the features, refining them by addition or reduction; and the algorithms, in choosing from the large range of possibilities, and tuning the selected algorithm where appropriate.

The choice of algorithm will be influenced by several factors. For example, data which includes nominal features cannot be handled by some classifiers: support vector machines [111] work with numeric values between 0 and 1, whereas a decision tree can deal with a mixture of numeric and nominal values. If transparency is important, then tree-based algorithms, such as C4.5 [192], or rule-based algorithms, such as RIPPER [48], will give results which the user can interpret. If the dataset is very large, either because of the number of instances or the number of features, then the time taken by slower algorithms, such as neural networks [205] or stacked classifiers [248], may be a problem. When such factors are unimportant, one approach is to try a variety of classifiers to find a suitable one. In this case, it is useful to have a tool which supports a range of algorithms, such as the open source machine learning toolkit, Weka [247].

Weka has a large number of tools, to understand the data, to preprocess it, to run experiments, and to compare the results. The preprocessing

filters automate tasks such as representational changes, e.g. discretization or normalisation. Feature selection algorithms are provided, as well as clustering and association algorithms. The large selection of classification algorithms are grouped by type: trees, rules, nearest neighbour, Bayesian, functions, and meta-classifiers.

There are two main sources of error in any classifier, bias and variance. Bias results from the (in)ability of an algorithm to fit a problem, and variance results from the under-representation of a problem by the training set. Bias can be seen as underfitting and variance as overfitting [141]. There is a trade-off between the two. However, by creating an ensemble of classifiers based on an algorithm with low bias, variance may be reduced [247]. A brief explanation of classifier ensembles is provided in Appendix B, along with the algorithms which have proved most useful in this research: Rotation Forest [203], Random Forest [29] and Simple Logistic [145, 227].

4.5 Machine learning in software engineering

Machine learning has not been used in origin analysis before, although, as discussed in Chapter 3, S.Kim et al. [130] use a statistical approach, along with exhaustive search, to combine measures in their system. A brief review of the development of machine learning in the more general area of software engineering can be found in Appendix C. Also included in this appendix is a review of a selection of software engineering applications which use machine learning, in particular those related to software evolution, which aims to ascertain whether there are pointers to suitable features, matching techniques, or machine learning algorithms.

Among the applications surveyed, those with some relevance to this research are two which use a combination of a large number of features, mostly based on analysis of changes to terms in source code text, by Aversano et al. [10] and S.Kim et al. [129]. Shivaji et al. [218] base their work on [129] and explore the effect of feature selection on classification. Also Zimmermann et al. [258] test defect prediction models built on one project

on another project, finding a low success rate of 3.4%.

Although the examples reviewed form a subset of the work in this area, making it difficult to draw conclusions about commonly-used techniques, it is clear that a wide variety of algorithms are applied in classification. In around half of the classification tasks, a number of different algorithms are applied to the data to find that best suited to the task. Also, a large majority of the classification tasks use a broad range of features taken from more than one source.

Ideally, a set of features should be easy to extract, while providing good discrimination between classes. However, a more complex set of features may improve performance, at the cost of the time and effort in construction.

4.6 Summary

This chapter has introduced machine learning and outlined the steps involved in categorising data. It focused on some of the questions which have to be addressed when creating a new dataset for which there appears to be limited guidance in the literature. The main message is that where there is uncertainty about the features, one strategy is to create a large number from which a good subset may be found.

In the absence of previous use of machine learning in origin analysis, a survey of a selection of machine learning applications to software engineering classification problems was undertaken, aiming to find pointers to suitable algorithms or features. Although no clear direction is found, this exercise has shown that one strategy is to use relevant features from a range of sources, and a selection of machine learning algorithms, and to test the alternatives experimentally.

Chapter 5

Visualisation

This chapter gives a brief introduction to the ways that source code comparison tools display information. It is not intended as a full overview of information visualisation, nor does it explore techniques used in the wider field of software engineering. Instead it focuses on two aspects relevant to this research: the use of colour in providing information to the user, and methods for showing interactions between files in a group.

The benefits of visualisations in imparting information are well understood. Stephen Few states that *“One of the great strengths of data visualization is our ability to process visual information much more rapidly than verbal information. Data visualization is effective because it shifts the balance between perception and cognition to take fuller advantage of the brain’s abilities”* [74, Sect.6.11]. Colour is usually an important component of visualisation because it helps to provide separation between different elements (see for example, Figure 5.5a), and because colour-coding can convey extra layers of information [224] (see for example, Figure 5.3d).

Visualisation is of interest in two areas of this research. First, because of the need for an “at-a-glance” comparison of a group of files in origin analysis. This comparison is used to label candidate restructured files for input to a machine learning system, a task which proved both difficult and time-consuming with other methods tried. Second, to express the different “types” of similarity between files in collusion detection. For example, to

show whether the code identified as common to two files also occurs in many other files, in a few files, or in just the two. Both of these applications display the similarity between files by colour coding the text; first, of one file to a group of others, and second, of two files in the context of a group.

Examples from a variety of code comparison tools are shown in this chapter to give context to the visualisations developed in this research. The majority of these comparison tools are from the field of plagiarism detection. The other group of tools sampled add colour to text to provide the user with additional information (see Section 5.3). The illustrations are taken from the publications referenced, unless stated otherwise.

Individual histograms	Graph + Population histogram	Table
Distance	One	The other
0.055928413	p6e67	p6e67_v2
0.059943583	p6d09_2	p6d09
0.072289154	p6a04_v2	p6a04
0.094527364	p6f70	p6f70v2
0.18985507	p6a09	p6e09v2
0.2137883	p6f01	p6d66
0.22214854	p6f03v2	p6f03
0.29664722	p6a64	p6f65
0.37528604	p6d68	p6d08
0.37769517	p6f09	p6d68
0.38217053	p6d04	p6d08
0.38289964	p6e65	p6f09
0.38432267	p6d69	p6d08
0.3860947	p6c05	p6f09
0.38682172	p6d08	p6d05
0.3888476	p6f09	p6d04
0.38934734	p6e70	p6d08
0.3897788	p6e65	p6d68

(a) AC's sorted pairwise similarities [82]

(b) Ferret's sorted pairwise similarities [146]

	5	2	7	1	13	15	3	8	14	11	10	6	12	9	4
5	0.00	0.00	0.62	0.62	0.62	0.90	0.95	0.90	0.76	0.88	0.92	0.86	0.94	0.96	0.95
2	0.00	0.00	0.62	0.62	0.62	0.90	0.95	0.90	0.76	0.88	0.92	0.86	0.94	0.96	0.95
7	0.62	0.62	0.01	0.01	0.01	0.87	0.87	0.91	0.69	0.84	0.85	0.95	0.96	0.98	0.97
1	0.62	0.62	0.01	0.01	0.01	0.87	0.87	0.91	0.69	0.84	0.85	0.95	0.96	0.98	0.97
13	0.62	0.62	0.01	0.01	0.01	0.87	0.87	0.91	0.69	0.84	0.85	0.95	0.96	0.98	0.97
15	0.90	0.90	0.87	0.87	0.87	0.01	0.60	0.61	0.66	0.81	0.74	0.86	0.94	0.97	0.97
3	0.95	0.95	0.87	0.87	0.87	0.60	0.00	0.75	0.69	0.83	0.82	0.92	0.91	0.96	0.96
8	0.90	0.90	0.91	0.91	0.91	0.61	0.75	0.01	0.62	0.86	0.82	0.86	0.95	0.98	0.96
14	0.76	0.76	0.69	0.69	0.69	0.66	0.69	0.62	0.02	0.62	0.64	0.82	0.95	0.96	0.95
11	0.88	0.88	0.84	0.84	0.84	0.81	0.83	0.86	0.62	0.01	0.81	0.96	0.95	0.96	0.96
10	0.92	0.92	0.85	0.85	0.85	0.74	0.82	0.82	0.64	0.81	0.01	0.92	0.95	0.97	0.97
6	0.86	0.86	0.95	0.95	0.95	0.86	0.92	0.86	0.82	0.96	0.92	0.01	0.95	0.96	0.96
12	0.94	0.94	0.96	0.96	0.96	0.94	0.91	0.95	0.95	0.95	0.95	0.95	0.04	0.92	0.95
9	0.96	0.96	0.98	0.98	0.98	0.97	0.96	0.98	0.96	0.96	0.96	0.97	0.96	0.92	0.01
4	0.95	0.95	0.97	0.97	0.97	0.97	0.96	0.96	0.95	0.96	0.97	0.96	0.95	0.96	0.01

(c) Brixtel et al.'s similarity heatmap [31]

	S1	S2	S3	S4	S5	S6	S7	S8	S9
S1	-	69	67	87	31	62	22	93	63
S2	69	-	66	72	30	60	25	73	61
S3	67	66	-	92	32	72	23	68	84
S4	87	72	92	-	33	62	22	76	63
S5	31	30	32	33	-	36	25	31	31
S6	62	60	72	62	36	-	25	65	88
S7	22	25	23	22	25	25	-	24	24
S8	93	73	68	76	31	65	24	-	66
S9	63	61	84	63	31	88	24	66	-

(d) Juricic's similarity matrix [122]

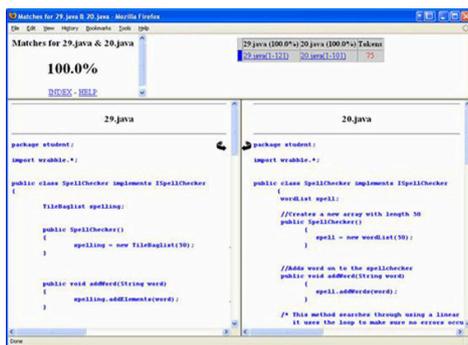
Figure 5.1: Showing file similarities

5.1 Plagiarism detection file comparison

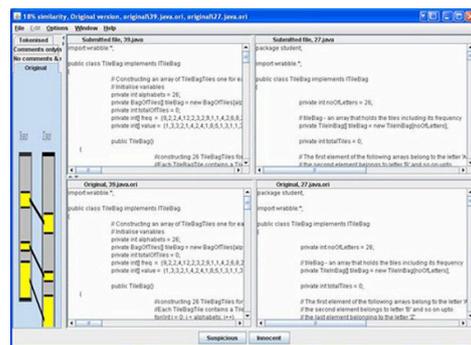
The most common way to provide initial information about the similarity between files in plagiarism detection is as a list or matrix of file similarities. Two examples of sorted pairwise similarity lists are shown in Figures 5.1a, for AC [82] and 5.1b, for Ferret [146].¹

Two matrices are those of Juricic [122] in Figure 5.1d (p.66), and of Brixtel et al. [31] in Figure 5.1c (p.66) which has heatmap colouring (red - similar, green - dissimilar). The colour in the left-hand matrix allows the user to quickly focus on the areas of interest, which are in red and orange, whereas in the right-hand diagram, the user must scan the numbers to find the range before locating the higher values.

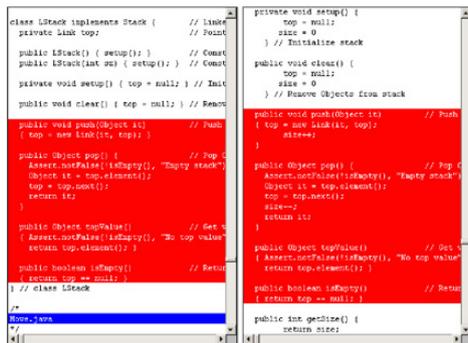
¹Locally produced screenshot



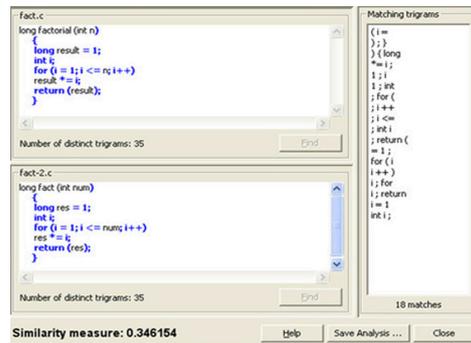
(a) JPlag [189]



(b) Sherlock [120]



(c) SID [41]



(d) Ferret [146]

Figure 5.2: Comparing code, side-by-side (a–c), or one above the other (d)

Many of the tools provide adjacent views of the code in the pairs of similar files, and these are normally accessed by selecting a pair from the similarity report. Four examples are shown in Figure 5.2. Each of these tools uses colour to convey information to the user. JPlag [189]² and SID [41] colour match sections of code, JPlag giving the text a different colour for each section, and SID changing the background colour. Sherlock [120]³ has two views of the code, original code (at the top), or a tokenised version (at the bottom), and colours the matching sections in a profile view to the left of the text windows. Ferret highlights the matching trigrams in blue and allows the user to select trigrams from the right-hand window to move the texts to the first occurrence of the trigram.⁴

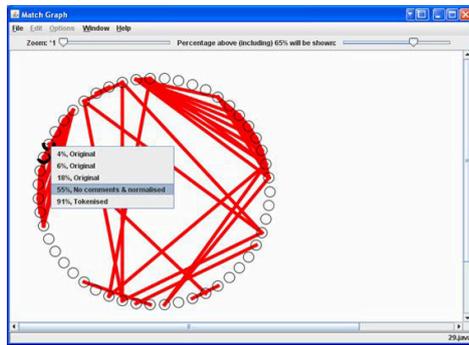
Graphical displays giving an overview of similarity between pairs of files in the group are provided by some of the tools surveyed, examples of which are in Figure 5.3. Sherlock uses a circular network graph, see Figure 5.3a.⁵ The nodes around the circle represent the files and the edges show files with a similarity above the selected threshold. Plagate [52] shows the similarities between files as a box-and-whisker plot. Each vertical line shows the similarities between a file (on the x-axis) and the rest of the group. The black line is the median value and the box covers the inter-quartile range (IQR). Outliers between 1.5–3.0 times the IQR are shown by a circle and those outside of this range by an asterisk, both are labelled with the file number. AC displays clusters, also with a user-selected threshold, see Figure 5.3c. It also shows, for each file, the distribution of similarities to the other files in the set, using a compressed form of histogram which relies on colour coding instead of area. Purple means that few files are at this distance from the file, red that many are, with frequencies between the two extremes following the rainbow colour sequence. An example of the histograms are in Figure 5.3d, where one line (p6d66) has been selected for expansion into a traditional histogram.

²<http://www.ics.heacademy.ac.uk/resources/assessment/plagiarism/demo.jplag.html>

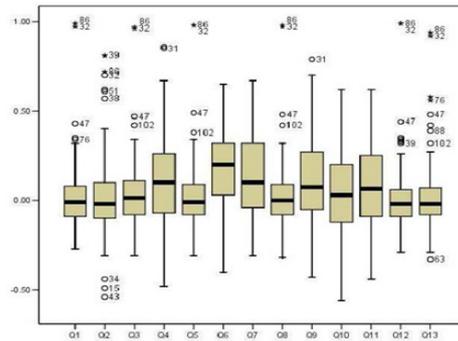
³<http://www.ics.heacademy.ac.uk/resources/assessment/plagiarism/demo.sherlock.html>

⁴Locally produced screenshot

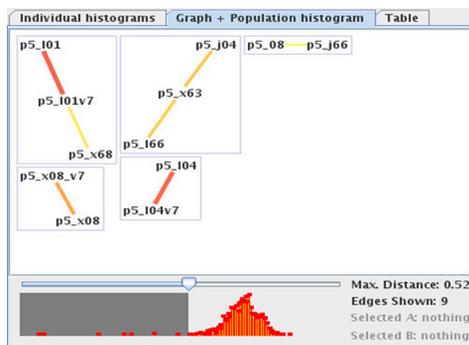
⁵<http://www.ics.heacademy.ac.uk/resources/assessment/plagiarism/demo.sherlock.html>



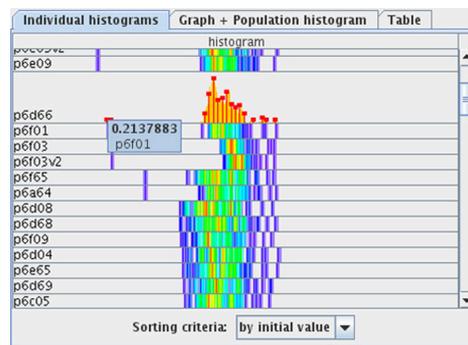
(a) Sherlock's graph [120]



(b) Cosma's box and whisker plot [52]



(c) AC's clusters [82]



(d) AC's histograms [82]

Figure 5.3: Graphical displays of file similarity

Ribler and Abrams take a different approach, which gives a detailed look at the similarity of one file in the context of the group. The categorical patterngram in Figure 5.4a has the sequence of n-grams in one file along the x-axis. The number of other files in the group which contain each n-gram are plotted in red for the values 1–9. If the n-gram is unique to the file being analysed, it is plotted at $y = 1$ and shown in blue. If in 10 or more files, it is considered uninteresting, plotted at $y = 10$, and shown in green. Figure 5.4a shows a file which shares a suspicious sequence of code with one other file, indicated by the dense sequence of red lines at $n = 2$.

To show which files contain the n-grams, another graph, the composite categorical patterngram, is constructed, see Figure 5.4b. This graph is based on one file, with the n-grams it contains plotted for each of the files in the comparison group. This time, the y-axis is labelled with the file numbers

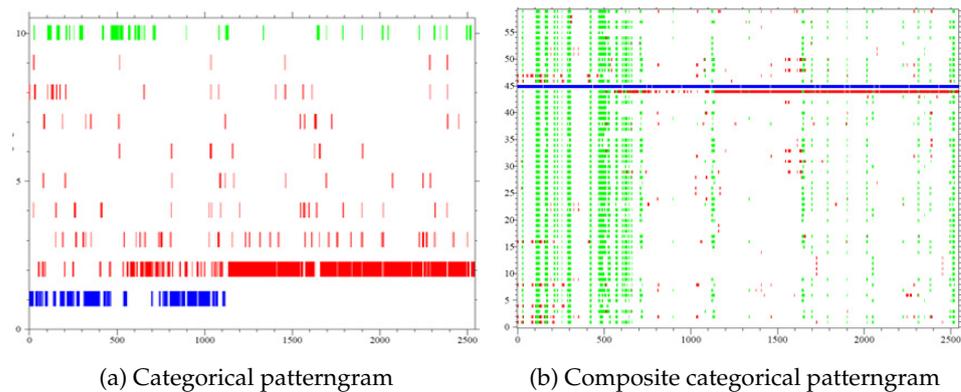


Figure 5.4: Ribler and Abrams's graphical displays [199]

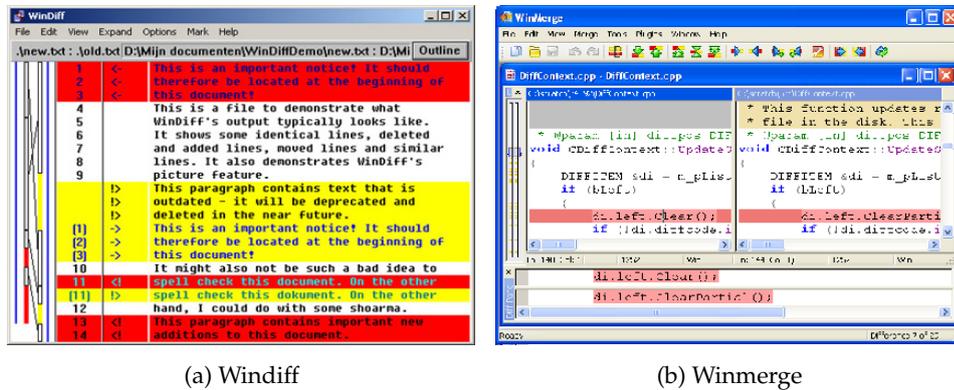
and the presence of the n-gram in a file marked by a coloured point. The point is green if the n-gram is in 10 or more files and red if in fewer. The composite patterngram highlights with which files the base file shares significant sections. Figure 5.4b is based on the same file, number 45, as Figure 5.4a, and shows that most of the code shared by few is in file 46, where the large amount of common code indicates collusion.

Ribler and Abrams' tool does not appear to give a measure of similarity between files but gives a pointer to areas in the files which deserve further attention. Their method is the closest to that developed in this dissertation. The measure for collusion detection is based on the unusual similarities between files in the context of a group, and the visualisation colour codes the text of the two files under investigation, depending on the number of other files which contain the code they share.

5.2 Other code comparison tools

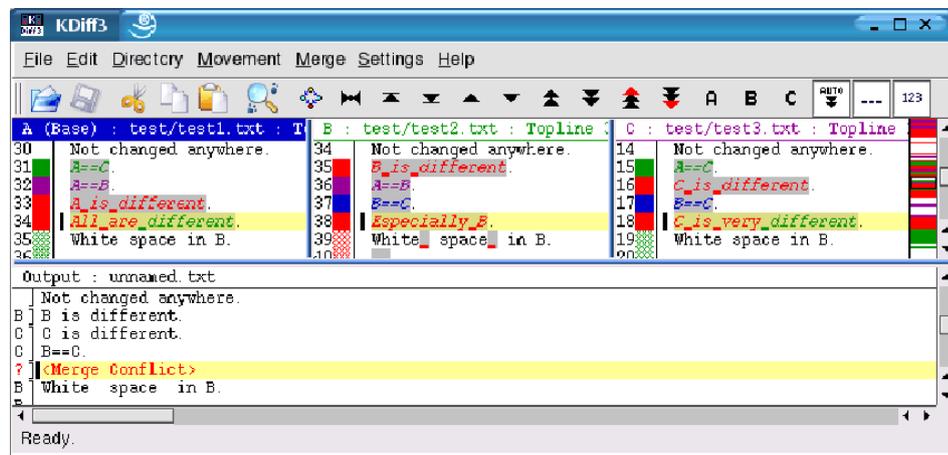
There are a large number of other graphical tools for comparing code. A small selection of relevant tools are described in this section. Windiff,⁶ see Figure 5.5a, compares two files, and shows the lines which appear in both files on a white background. Lines which differ are repeated with

⁶[http://msdn.microsoft.com/en-us/library/aa242739\(v=VS.60\).aspx](http://msdn.microsoft.com/en-us/library/aa242739(v=VS.60).aspx)



(a) Windiff

(b) Winmerge



(c) KDiff3 - merge screen

Figure 5.5: Windiff, Winmerge and KDiff3

the line from one file on a red background and from the other, a yellow background. Winmerge,⁷ see Figure 5.5b, is similar, but presents the two files side-by-side, highlighting differences between them by colouring the background.

KDiff3,⁸ shown in Figure 5.5c is one of the few tools available which compare three files. The upper part of the screen colours the differences between the three files in red. Lines which match in all three files are shown in black text, if in two files they are purple (1, 2), green (1, 3) or blue (2, 3)

⁷<http://winmerge.org/>

⁸<http://kdiff3.sourceforge.net/doc/index.html>

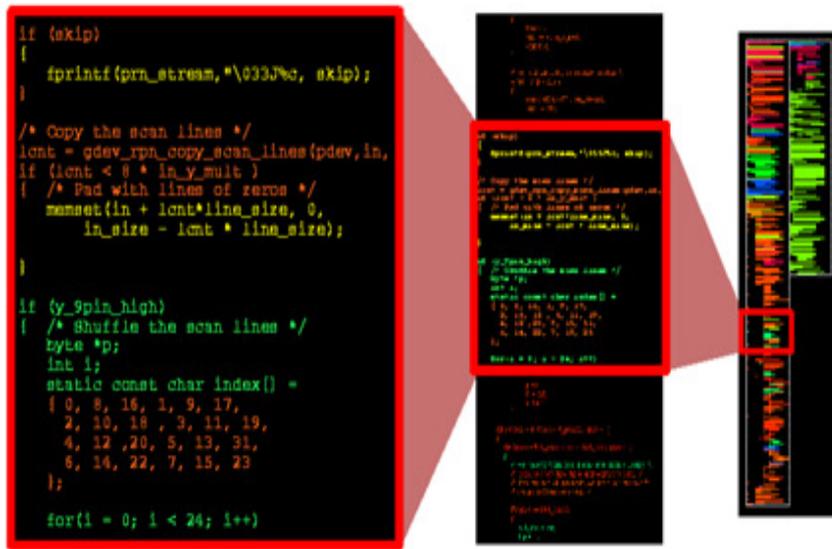


Figure 5.6: Ball et al.'s line representation [14]

depending on which pair has the similar code. The lower part of the screen shows what will happen when the files are merged.

There do not appear to be any tools which make text-based comparisons between more than three files.

5.3 Colour coding features in software evolution

The field of software evolution visualisation is large, and visualisations are often very sophisticated. Starting points for further information are the surveys by Storey et al. [225] and Caserta [37], or Voinea's dissertation [240].

Some tools depict information about an evolving system, such as 3DSoft-Vis [200]. Other tools give information about one version of a system. Two examples of single version tools are shown here. These tools are chosen because they colour the source-code text to display facts about the system, which is how the visualisations developed in this research provide information to the user.

Figure 5.6 [14] shows Ball et al.'s "line representation" where the code in a file is coloured to represent one aspect of the file's development. For example, the developers, the version, or as here, the age (where green means

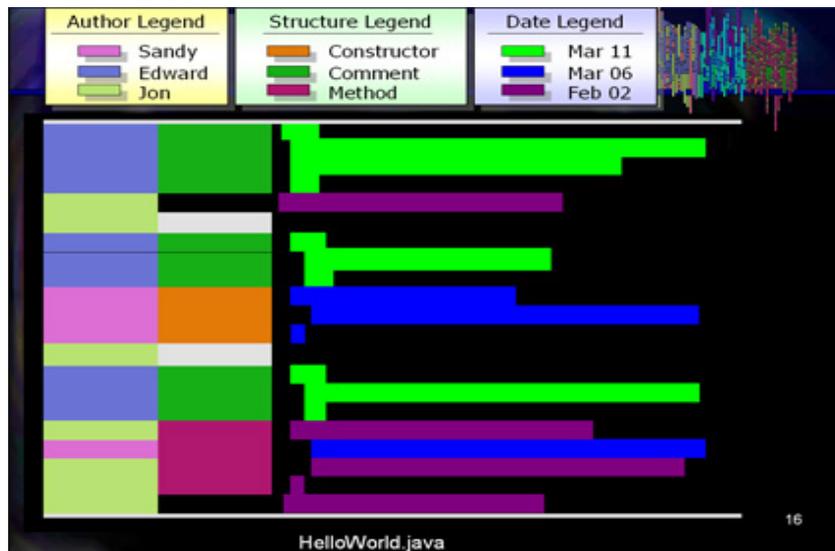


Figure 5.7: Augur three-feature colour coding of one file [86]

old code and red new code).

Augur, a tool developed by Froelich and Dourish [86], also colours features of the code. This tool uses different sets of colours to show different sets of features. In the example in Figure 5.7, three features are highlighted. The code is coloured to show its age, there are also two other colour sets applied on the left margin, the left-hand set showing the author and the middle set showing elements of the code, such as comments and methods.

Displays of similarity between files, taken from three clone detection tools, are available in Appendix D. This appendix also shows graphical displays from two tools from the field of origin analysis, where colour is used to add information about the movement of code in a system.

5.4 Summary

This chapter has reviewed tools which are relevant to the visualisations developed in Chapter 8 to support the applications described in Chapters 9 and 10. It looked at the output of source code comparison tools, in particular, comparisons between pairs of files, between groups of files, and of pairs of files in a group context. The use of colour was also consid-

ered, both for highlighting important parts of a display, and for providing additional information to the user when applied to text.

The majority of the plagiarism detection tools surveyed provide information about the similarity between files in one of three forms: pairwise similarity based on a single similarity measure (see Figure 5.1), group interaction based on a single similarity measure (see Figure 5.3), or pairwise text comparisons (see Figure 5.2). Ribler and Abrams provide information about the interaction between the files in a group at n-gram level, but do not appear to map this to the text of the file. In general, file comparison tools do not show comparisons between the text of more than three files.

Chapter 6

Ferret

The primary similarity detection tool used in this research is Ferret, developed at the University of Hertfordshire [155]. Ferret is an obvious first choice for the file comparison tasks, both because it is efficient and because it is an in-house tool. Ferret detects matching token trigrams wherever they occur, and is therefore useful for finding similarity between files even when copied sections are subject to replacement or rearrangement [157, 158].

In this chapter, Ferret is introduced, and its use with program code is discussed. Example files are then provided for illustrating Ferret and the other similarity detection tools used in this research. Next, the outputs from Ferret are described. During this research, features for machine learning were created by finding new ways to analyse these outputs. The analyses are described in the last three sections of this chapter.

6.1 Background

Ferret was originally developed by the UH Plagiarism Detection Group to measure similarity between text files [155, 157].¹ The similarity measure is based on trigrams – sets of three consecutive words. For example, the phrase “sequences of three consecutive words” produces the three trigrams: “sequences of three”, “of three consecutive” and “three consecutive words”.

¹<http://homepages.stca.herts.ac.uk/~comqpc1/ferret.html>

Ferret accepts as input a list of files for comparison. Processing efficiency is achieved by making a single pass through each file, during which an inverted index is constructed. The file identifier is recorded once against each trigram it contains, using a table to which new trigrams are added as they are encountered. The resulting trigram to file index is used to calculate the similarity between two documents using the Jaccard coefficient, also known as the resemblance measure [162, p.299].

$$\text{Jaccard coefficient} = \frac{|\text{Intersection of the distinct trigrams}|}{|\text{Union of the distinct trigrams}|}$$

This measure has a value between 0 and 1; 0 means there are no matching trigrams and therefore no matched text, while a score of 1 shows that all of the trigrams in one text also occur in the other. Although, in theory, this does not necessarily mean that the two texts are identical, because trigram frequency is not measured; in practice, two texts with a similarity score of 1 are unlikely to be different. One way in which exact repetitions of every trigram can occur in two non-identical sequences is illustrated in Figure 6.1. In the sequences x and y, the blocks labelled 'a'-'d' represent words.

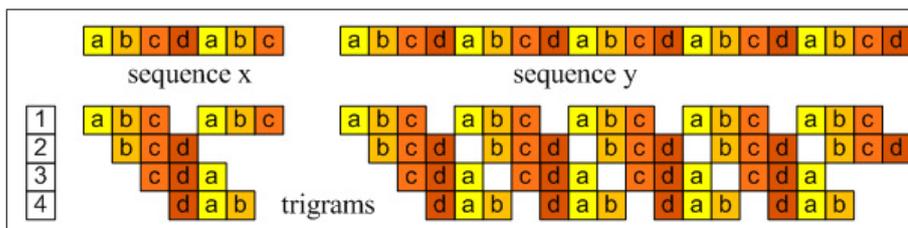


Figure 6.1: Sequences x and y are two sequences with a similarity score of 1. The same 4 trigrams, shown below the sequences, are repeated in the files.

6.2 Program code

Ferret has also been adapted for use with program code, with lexical tokens used in place of words.² Variations in white-space and layout are ignored,

²Currently with a tokeniser for C-type languages

because the code is tokenised. A study by Rainer et al. shows that although some language specific trigrams, such as “) } ; ” or “ i = 1 ”, occur often, the frequency of trigrams in program code follows a similar Zipfian, or negative exponential, distribution to that observed in trigrams of words in a natural language sequence [156]. Rainer et al. obtained this distribution [194, Fig.1] by analysing the files in every snapshot in their code base, taken from the SAC (Single Assignment C)³ project, and repeated here on the left of Figure 6.2.

To look at the trigram distribution over a broader selection of code, the files in the last release of each of the eighty-nine projects studied for this research (see Chapter 12) were analysed. This code base consists of 7,449 files, a similar number to the 7,819 files analysed by Rainer et al. There is a difference in the number of trigrams, the multiple projects have 2,749,150 trigrams, nearly four times as many as the 722,425 in the Rainer et al. study. The Zipfian distribution is based on plotting the frequency of the trigram against its rank, 1 for the most frequently occurring, 2 for the next most frequent, and so on. The right-hand graph in Figure 6.2 shows a distribution of similar shape to that of Rainer et al. There is one difference which reflects the data: analysing one project across a number of releases

³<http://www.sac-home.org/>

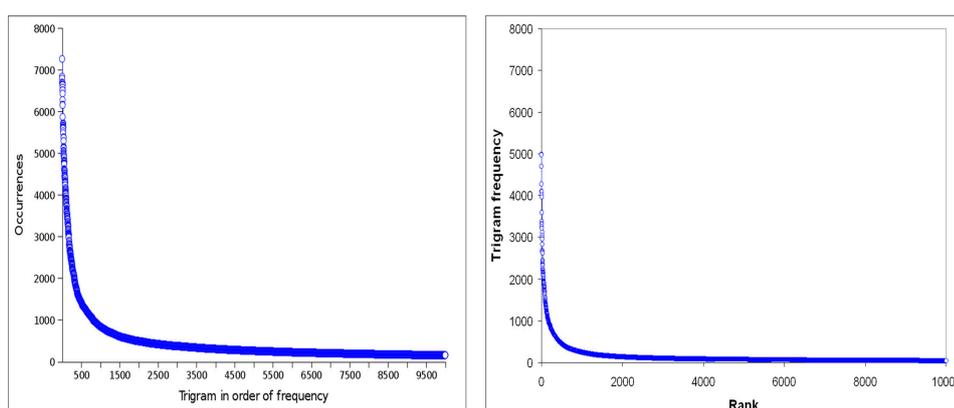


Figure 6.2: On the left, the first 10,000 trigrams of 0.72m trigrams, ranked by frequency, from Rainer et al. [194, Fig.1]. On the right, the first 10,000 of 2.75m trigrams in the 89 projects. Frequency is plotted against rank.

inevitably produces more frequently occurring trigrams than analysing one release of each of a number of projects.

Rainer et al. state that this type of trigram distribution in code indicates that Ferret should be as effective in matching program texts as it is in matching document texts because the majority of the trigrams appear in few files [194]. However, the trigram distribution is only part of the story. What is also interesting is the ratio of tokens to trigrams in code against the ratio of words to trigrams in text.

To compare the two, the code in one release of each of the projects used in this research was analysed, along with fifty books from the Gutenberg project. In addition, figures were taken from an analysis of corpora previously undertaken by Lyon et al. [157].

The books from the Gutenberg project comprise twenty books by Charles Dickens, and thirty other books, each with different authors. Each of the books had the preamble and the licence at the end removed. The books were analysed to find the number of trigrams to the number of words in

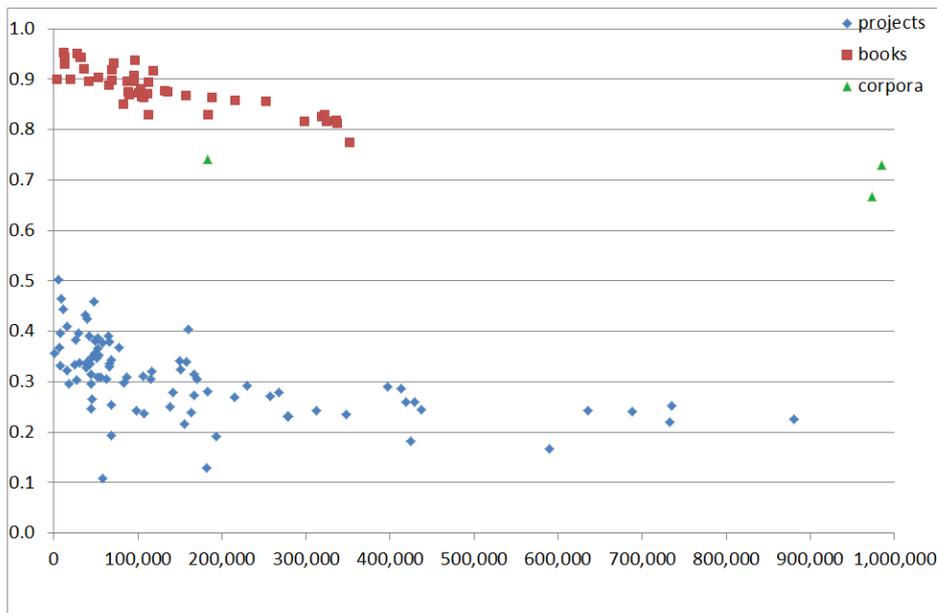


Figure 6.3: The ratio of trigrams to words/tokens in the books, corpora or projects against the number of words (books and corpora) or tokens (projects).

each one. The source code was also analysed to find the trigram to token ratio in one release in each project.

These ratios are plotted in Figure 6.3 together with data on three of the five corpora analysed by Lyon et al.: TV News corpus (335 documents), the Federalist Papers (81 documents) and one of the three from the Wall Street Journal corpora (whose size is not reported).⁴

Unsurprisingly, the graph shows that, for each group, there is a slight trend for the ratio of trigrams to words/tokens to decrease as the number of words/tokens in the documents increases. The distinction between text and code is clear. For example, at 100,000 words or tokens, there are about 90% as many trigrams as words in the books, around 75% as many trigrams as words in the corpora, but, on average, in the project code the ratio of trigrams to tokens is just 25%.

'Singleton trigram' is the name used by Lyon et al. to describe a trigram which occurs in only one document in a set. There is less variation between text and code in the ratio of singleton trigrams to the total trigrams in the sets. Lyon et al. report ratios of 85–87% on the three corpora plotted, and for the larger corpora 82% and 77%, while the mean ratio of singletons to the total trigrams in the project code is just over 80%.

However, because of the small proportion of trigrams to tokens compared to the trigrams to words, the ratio of singleton trigrams varies between text and code. Over the three corpora, the ratio of singleton trigrams to the total number of words in a document is around 61%. This figure is close to the 66% of singleton trigrams to words in the set of 20 Dickens books (2.2m singleton trigrams to 3.3m words). In contrast, in the projects, the mean ratio of singleton trigrams to total tokens is just under 20%, with a range of 2–45%.

These figures indicate that although in general there are sufficient singleton trigrams to make it possible to match a file in one release to the correct one in another release, the task will be more difficult for source code files

⁴The fourth and fifth sets are not shown because they have 4.5m and 38.5m words, which squashes the rest of the data points on the graph. Their ratios are 0.54 and 0.37 respectively.

than for files of text, and especially difficult in projects with few singleton trigrams. The problem may be compounded when the section of code to be matched is small, has been edited, or both. Similarly, it is harder to detect duplication or copying between files of code than between files of text.

6.3 Files to illustrate similarity tools

Four example files are used to illustrate both Ferret and the other similarity detection tools applied in this research (see Chapter 11). The first file, `cnp-1.c`, listed in Figure 6.6, is for calculating the number of combinations and permutations which can be made from a subset of items taken from a set of a given size. Two other files are the amended version of `cnp-1.c`, called `cnp-2.c`, and a new file, `fact.c`, formed when the first file is split by moving the factorial function, see Figure 6.4. These two files are shown in Figure 6.7. A small file, `power.c`, on the right of Figure 6.5, is used for illustration where larger files would produce unwieldy output.

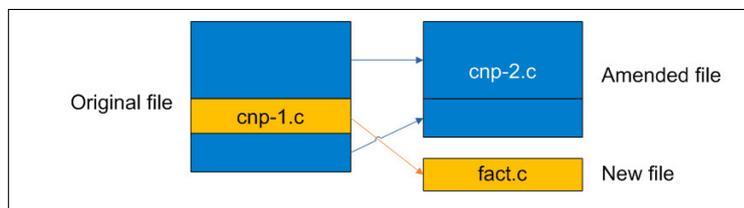


Figure 6.4: The relationship between the 3 example C files, `cnp-1`, `cnp-2` and `fact.c`

<pre>//fact.c long factorial (int n) { long result = 1; int i; for (i = 1; i ≤ n; i++) result *= i; return (result); }</pre>	<pre>// power.c long power (int base, int n) { long result = 1; int i; for (i = 1; i ≤ n; i++) result *= base; return (result); }</pre>
--	---

Figure 6.5: Code for `fact.c` and `power.c`.

```
// cnp-1.c
#include <stdio.h>

long factorial (int n);
long combinations (int n, int k;
long permutations (int n, int k);

int main()
{
    int setsize, subsetsize;
    printf ("Set size? ");
    scanf ("%d", &setsize);
    printf ("Subset size? ");
    scanf ("%d", &subsetsize);
    if (setsize < 0 || subsetsize < 0 || setsize < subsetsize)
    {
        printf ("Mission impossible\n");
        return (1);
    }
    printf ("%ld combinations and %ld permutations of %d items
            taken from %d \n", combinations (setsize, subsetsize),
            permutations (setsize, subsetsize), subsetsize, setsize);
    return (0);
}

long factorial (int n)
{
    long result = 1;
    int i;
    for (i = 1; i ≤ n; i++)
        result *= i;
    return (result);
}

long combinations (int n, int k)
{
    return (factorial(n) / (factorial(k) * factorial(n-k)));
}

long permutations (int n, int k)
{
    return (factorial(n) / factorial(n-k));
}
```

Figure 6.6: Original code for finding combinations and permutations of a subset of items, cnp-1.c. This file, and the two files which result from splitting it, cnp-2.c and fact.c, shown in Figure 6.7 are used as a running example.

```
// cnp-2.c
#include <stdio.h>
#include "fact.c"

long combinations (int n, int k);
long permutations (int n, int k);

int main()
{
    int setsize, subsetsize;
    printf ("Set size? ");
    scanf ("%d", &setsize);
    printf ("Subset size? ");
    scanf ("%d", &subsetsize);
    if (setsize < 0 || subsetsize < 0 || setsize < subsetsize)
    {
        printf ("Mission impossible\n");
        return (1);
    }
    printf ("%ld combinations and %ld permutations of %d items
            taken from %d\n", combinations (setsize, subsetsize),
            permutations (setsize, subsetsize), subsetsize, setsize);
    return (0);
}

long combinations (int n, int k)
{
    return (factorial(n) / (factorial(k) * factorial(n-k)));
}

long permutations (int n, int k)
{
    return (factorial(n) / factorial(n-k));
}

// fact.c

long factorial (int n)
{
    long result = 1;
    int i;
    for (i = 1; i ≤ n; i++)
        result *= i;
    return (result);
}
```

Figure 6.7: Amended code for finding combinations and permutations of a subset of items, split into 2 files - cnp-2.c (top) and fact.c (below).

6.4 Similarity scores

As already explained, Ferret provides the Jaccard coefficient, or similarity score, for two files.⁵ Broadly, five factors affect this similarity score:

- the amount of matching code,
- the size of each document,
- repetitions of trigrams in the documents and whether they are in the

⁵An example similarity score calculation can be found in Appendix E.

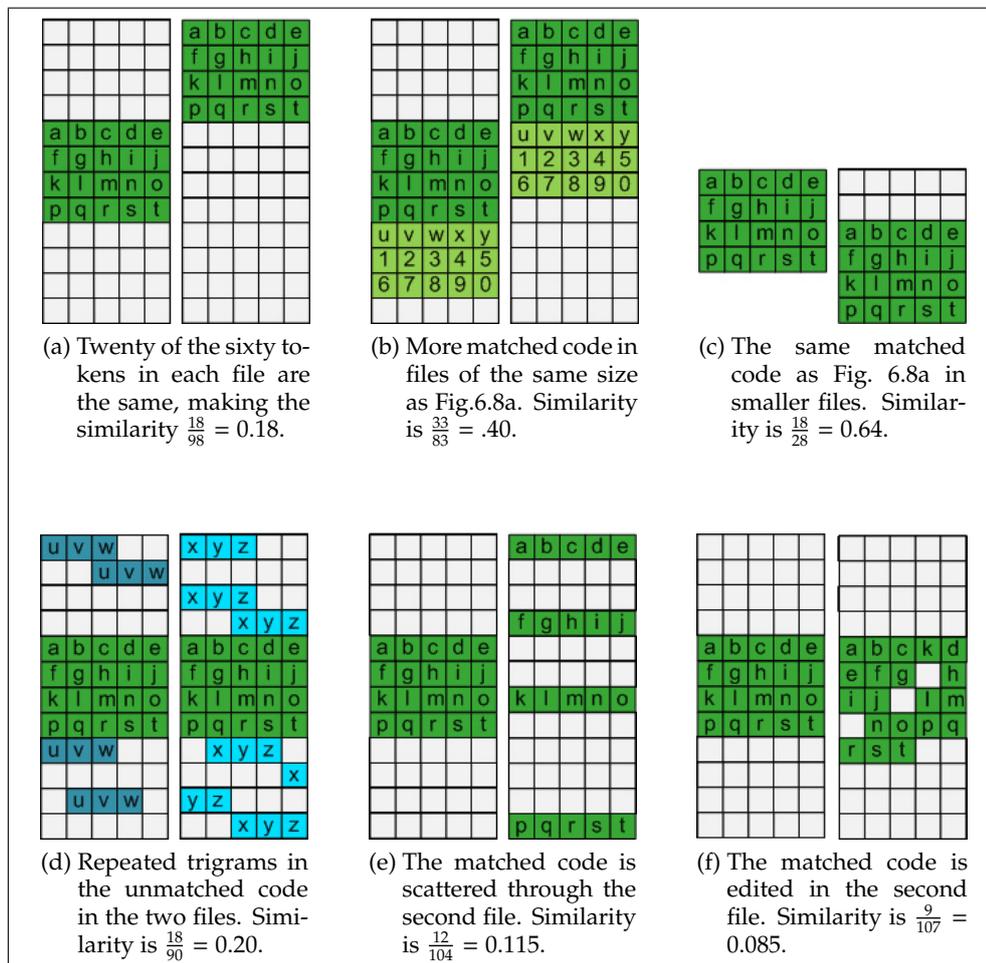


Figure 6.8: The effect of changes to the amount of matched code, to file size, to trigram repetitions and to layout in the matched code. Darker coloured blocks with letters show matched code, and blank pale blocks represent tokens in unmatched and unique trigrams.

matched portions of code,

- whether the matched code is contiguous or scattered,
- and whether there is editing within the “matched” sections of code.

The effect on similarity scores of each of these factors is illustrated in Table 6.8. In the diagrams, matched code is shown in colour with a letter or number to identify the token. Unmarked ‘tokens’ are assumed to be unique. Figure 6.8a is the example against which the others are compared. It is clear that either more matched code, Figure 6.8b, or smaller containing files, Figure 6.8c, increase the similarity score. Figures 6.8d, 6.8e and 6.8f show files of the same size as those in Figure 6.8a with the same number of matching tokens. Figure 6.8d depicts files with repeated trigrams in the unmatched code, thus reducing the total number of trigrams and increasing the similarity score. Likewise, if the matched code contains repeated trigrams, the similarity score is reduced. In Figure 6.8e the code in the second file is scattered and in Figure 6.8f the code is edited; in each case the number of shared trigrams, and therefore the similarity score, is reduced.

The basic output from Ferret is a set of comparisons, one for each pair of files, listing the name of file 1, the name of file 2, the number of distinct trigrams common to the two files, the number of distinct trigrams in file 1, in file 2, the similarity measurement and containment of each file in the other. The containment of file A in file B is calculated by dividing the number of trigrams appearing in both files A and B by the number in file A ($\frac{|A \cap B|}{|A|}$). An example of this output is given in Figure 6.9.

file 1	file 2	common trigrams	file 1 trigrams	file 2 trigrams	similarity	cont't 1 in 2	cont't 2 in 1
../rel-n/filea	../rel-n+1/filea	1335	1453	1512	0.8190	0.9188	0.8829
...
../rel-n/filea	../rel-n+1/filez	46	1453	173	0.0291	0.0317	0.2659
../rel-n/fileb	../rel-n+1/filea	182	290	1512	0.1125	0.6276	0.1204
...
../rel-n/fileb	../rel-n+1/filez	0	290	173	0	0	0
...

Figure 6.9: Ferret output for files in consecutive releases of a project

Similarity score and containment are rapidly calculated measures between two files. There is a trade-off between the speed of calculation and the amount of detail provided about the similarity between files.

Other outputs are available from Ferret: a report of the trigram to file (trigram-file) index (see Table 6.3, p.90) listing the trigrams and the files in which they occur; reports for each comparison between a pair of files highlighting the duplicated trigrams in the files, either in PDF or XML format (see Figure 6.10, p.86).

The rest of this chapter describes methods developed in this research to analyse the XML output and the trigram-file index to obtain further information about the distribution of matched and unmatched trigrams and the relationship between the files in the group presented for comparison.

6.5 Ferret XML report

The XML report produced by Ferret shows matched and unmatched trigram sequences related to the source code (or text). The example in Figure 6.10 is the report for a comparison between `fact.c` and `power.c`. In the header section, the number of matched trigrams and the similarity score for the pair of files is shown. For each document, the number of trigrams and the containment of the file in the other file are given, along with the source text, which is separated into sequences of tokens (or words) for which the trigrams appear in the other file (tagged “copied”), and the rest (tagged “normal”).

More information about the similarity between files can be recovered by analysing the XML file to find patterns in the alternating sequences of matched and unmatched trigrams in the code. The sizes of contiguous blocks of code containing shared trigrams can be found, giving an idea of the presence or absence of interesting similarity.

The sizes of these blocks of code can be expressed in terms of tokens, characters, or lines of code, which are approximated here by deducting one from the total number of new line characters in a sequence. The patterns

taken from the example XML file in Figure 6.10 are shown in Table 6.1. Looking at the last copied section in the file fact.c, which is:

```

;
return (result);
}
```

there are seven tokens which are: `;` `return` `(` `result` `)` `;` `}`

These tokens consist of $1 + 6 + 1 + 6 + 1 + 1 + 1 = 17$ characters.

There are 3 newline characters, so the estimated number of lines is $3-1=2$,

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="uhferret.xsl" ?>
<uhferret>
<common-trigrams>29</common-trigrams>
<similarity>0.659091</similarity>
<document> <source>C:\...\power.c</source>
<num-trigrams>38</num-trigrams>
<containment>0.828571</containment>
<text>
<block text="normal"><![CDATA[long power (int base, )]]></block>
<block text="copied"><![CDATA[int n)
{
    long result = 1;
    int i;
    for (i = 1; i <= n; i++)
        result *= i]]></block>
<block text="normal"><![CDATA[ base]]></block>
<block text="copied"><![CDATA[;
    return (result);
}]]></block>
</text>
</document>

<document> <source>C:\...\fact.c</source>
<num-trigrams>35</num-trigrams>
<containment>0.763158</containment>
<text>
<block text="normal"><![CDATA[long factorial ( )]]></block>
<block text="copied"><![CDATA[int n)
{
    long result = 1;
    int i;
    for (i = 1; i <= n; i++)
        result *= i]]></block>
<block text="normal"><![CDATA[ i]]></block>
<block text="copied"><![CDATA[;
    return (result);
}]]></block>
</text>
</document>
</uhferret>
```

Figure 6.10: Ferret XML report for a comparison between fact.c and power.c.

which is a reasonable approximation, as the first semi-colon in the sequence is only a small part of a line.

	power.c				fact.c			
	n	c	n	c	n	c	n	c
Tokens	6	27	1	7	3	27	1	7
Characters	18	49	4	17	14	49	1	17
Lines	0	4	0	2	0	4	0	2

Table 6.1: The pattern of matched (c) and unmatched (n) code between power.c and fact.c, shown in three different units: tokens, lines and characters.

6.6 Density analysis

Analysing the Ferret XML report as described in Section 6.5 finds contiguous blocks of code, or identical copies in the code, like type-1 clones. When code has been edited between releases, for example, by replacing identifiers, or by a copy-paste-edit sequence, then the blocks of matched code will be “gappy”, like those found in type-2 or type-3 clones [204, 238]. The idea behind density analysis is to find these gapped blocks by matching blocks of code which are nearly contiguous.

The alternating matched and unmatched blocks in the Ferret XML report can be analysed to discover these larger blocks of densely matched tokens, where most, but not all, of the tokens match. To illustrate, a contrived example is shown in Figure 6.11. The original factorial file, fact.c, is amended to become fact-2.c, in which the function name is changed from “factorial” to “fact”, “n” to “num”, and “result” to “res”. These are the type of changes which may be made in a simple attempt to disguise plagiarism, or in the renaming of identifiers to suit a new location. The result of comparing the two files with Ferret is shown in Figure 6.12, where matched trigrams are highlighted in bold blue text.

The matched and unmatched tokens can be represented in shorthand as a pattern of blocks of copied and non-copied tokens as follows:

(n 5)(c 3)(n 1)(c 14)(n 1)(c 4)(n 1)(c 5)(n 1)(c 3).

This means that 5 unmatched tokens are followed by 3 matched tokens, then 1 unmatched, 14 matched, and so on.

The method developed for density analysis uses a top-down algorithm, which is described in detail in a technical report [96, pp.8-14], or briefly at http://homepages.stca.herts.ac.uk/~gp2ag/density_analysis_overview.html, with diagrams. The copy patterns used in this explanation are based on tokens but can also be expressed in lines or characters.

In brief, the tool finds dense blocks based on three main parameters: minimum density, minimum block size and maximum gap size. Density is calculated by dividing the number of matched tokens by the total number of tokens in the sequence. In the example above, the first unmatched block of 5 tokens would be excluded from the pattern, leaving 33 tokens, of which 29 are matched, a density of $\frac{29}{33} = 0.88$. To avoid selecting blocks with few tokens, which are likely to be incidentally similar rather than the product of moved or copied code, a minimum block size is specified by the number of matched tokens it contains. Maximum gap size is the maximum number of consecutive unmatched tokens permitted within a dense block. For example, a sequence such as ((c 200)(n 100)(c 200)) has a density of 0.8, however, this is likely to be considered as two separate copied blocks rather than one edited block. The maximum gap ensures that sequences of this type are split into two blocks. There are two other parameters used in this analysis. The first allows a choice of unit: tokens (or words in text), characters or lines. The second allows a choice of block selection criterion, which can be based on the highest density, on the most copied tokens, or on

<pre>//fact.c long factorial (int n) { long result = 1; int i; for (i = 1; i ≤ num; i++) result *= i; return (result); }</pre>	<pre>// fact-2.c long fact (int num) { long res = 1; int i; for (i = 1; i ≤ n; i++) res *= i; return (res); }</pre>
--	---

Figure 6.11: Code for fact.c and an amended version fact-2.c.

Block Name	Copied/Non-copied Pattern	Copied tokens	Total tokens	Density	Will be selected because most ...
a	((c 9)(n 1)(c 10)(n 1)(c 9)	27	30	0.90	dense
b	((90)(20)(80)(20)(90))	260	300	0.87	copied tokens
c	((80)(20)(50)(20)(30)(10)(60)(10)(30))	250	310	0.81	tokens overall

Table 6.2: Three dense blocks illustrate the selection criteria.

the largest total number of tokens, in a block. For example, given the three blocks, a, b and c, in Table 6.2, the most dense is block a, the block with the most copied tokens is block b, and block c has the highest total number of tokens.

This method of analysis is used in two ways, first, in finding dense blocks of similar code between files, which are analysed to provide features for the machine learning experiments (see Chapter 14). Second, the idea has been developed to provide a prototype stand-alone visualisation tool which

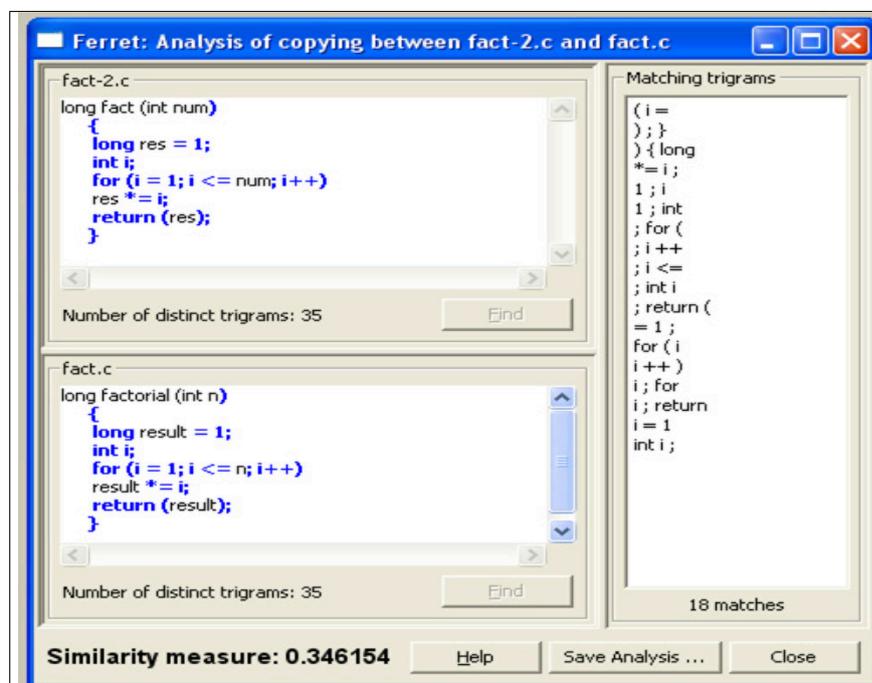


Figure 6.12: Ferret comparison between the files fact.c and fact-2.c.

...
stdio . h	FILES:[0 1]	; int main	FILES:[0 1]	% ld combs	FILES:[0 1]
. h >	FILES:[0 1]	; int i	FILES:[0 2]	% ld perms	FILES:[0 1]
. c "	FILES:[1]	; printf (FILES:[0 1]	d " ,	FILES:[0 1]
h > #	FILES:[1]	; scanf (FILES:[0 1]	d \n "	FILES:[0 1]
h > long	FILES:[0]	; if (FILES:[0 1]	d items taken	FILES:[0 1]
> # include	FILES:[1]	; return (FILES:[0 1 2]	& setsize)	FILES:[0 1]
> long fact	FILES:[0]	; } long	FILES:[0 1]	& subsize)	FILES:[0 1]
...
int n)	FILES:[0 1 2]	main ()	FILES:[0 1]	ld perms of	FILES:[0 1]
int n ,	FILES:[0 1]	{ long result	FILES:[0 2]	and % ld	FILES:[0 1]
...

Table 6.3: Extract from the Ferret trigram-file index for `cnp-1.c` [0], `cnp-2.c` [1] and `fact.c` [2]. The trigrams are listed with the files in which they appear. For example, `"h > #"` is only in file 1, while `"int n)"` is in all 3 files.

can be used to find and display dense blocks from any Ferret XML file [92]. It highlights areas where there are gapped matches in the text, which may be the result of obscured plagiarism or a copy-paste-edit operation on the code. It is especially useful where the files being compared are large, providing easy access to the interesting parts of the file. Details of this prototype can be found in Appendix F, (or at <http://homepages.stca.herts.ac.uk/~gp2ag/density.html>).

6.7 Ferret trigram-to-file index

Apart from the pairwise comparisons, Ferret also provides access to its trigram-file index. Table 6.3 shows an extract from the trigram-file index output by Ferret when the three files `cnp-1.c` (file 0), `cnp-2.c` (file 1) and `fact.c` (file 2) are compared. The trigrams are shown together with the files

Key	Present in	No.of trigrams
[0]	cnp-1.c only	3
[1]	cnp-2.c only	9
[2]	fact.c only	0
[0 1]	cnp-1.c and cnp-2.c	141
[0 2]	cnp-1.c and fact.c	29
[1 2]	cnp-2.c and fact.c	0
[0 1 2]	all 3 files	6

Table 6.4: Trigrams shared by different combinations of the 3 example files

in which they appear. For example, the trigram “ h > # ” appears only in the file `cnp-2.c`, shown in the report as “FILES:[1]”, whereas “ int n) ” is in all three files, “FILES:[0 1 2]”. These examples are highlighted in the table.

The trigram-file index can be analysed to find out how many trigrams are shared by different combinations of files. The distribution of trigrams between the files can give an indication of their relationship. For example, when a file is split without further edits, such as when `cnp-1.c` is split to become `cnp-2.c` and `fact.c`, the expectation is that most of the trigrams in `cnp-1.c` will be shared with one or other of `cnp-2.c` and `fact.c`. It is possible that each file may have a few trigrams which occur only in that file as the code is normally disturbed by a split. In a simple split, it is unlikely that there will be many trigrams shared by the amended and new files which are not also shared by the original file.

Analysis of the trigram-file index, shown in Table 6.4, supports this expectation; the majority of the trigrams, 170 out of 188, are shared either by `cnp-1.c` and `cnp-2.c` “FILES:[0 1]” or `cnp-1.c` and `fact.c` “FILES:[0 2]”.

6.8 Summary

The copy detection tool Ferret was introduced in this chapter. Factors which impact on the similarity score between files were identified. A comparison between files of natural language text related by topic or by author, and files of source code text related because they belong to the same project, showed that trigrams in source code occur more frequently in a set of files than trigrams in text. This makes the task of matching files across releases difficult, in that multiple files share the same trigrams and there are fewer singleton trigrams.

All reductions used in comparing code, such as the transformations considered in Section 2.2, result in some loss of the information in the original code. Reducing code to a set of trigrams means that location and frequency information is lost. However, because the matched trigrams are subsequently mapped back to the code, and output as an XML report, this

information can be recovered to some extent.

Ferret is moderately robust to identifier renaming (see Figure 6.13). However, by analysing the density in the sections of code, copied and not copied, in the XML report, blocks of code with replacements can be found. An early prototype tool based on this analysis was also introduced.

Ideas for analysing another output of the Ferret tool, the trigram-file index, were also outlined, and these ideas are developed in the next two chapters.

a	b	c	d	e	a	b	c	x	e
f	g	h	i	j	f	g	h	i	j
k	l	m	n	o	y	i	m	n	o
p	q	r	s	t	p	q	z	s	t

Figure 6.13: The impact on similarity of renamed tokens. In this example three of the twenty tokens are renamed, bringing the similarity from 1 to 0.5. At 0.85 density, the whole block would be identified as a gapped copy.

Chapter 7

Trigram analysis

As described in Chapter 6, Ferret normally uses the trigram-file index to calculate similarity scores for pairs of documents in a set. The similarity measure between two files is the ratio of the number of trigrams they share, to the number they contain in total, $\frac{|A \cap B|}{|A \cup B|}$ in Figure 7.1.

This chapter considers alternative ways to analyse the trigram-file index to identify relationships between files. For example, between one file and the rest of the set; between pairs of files in the context of the whole set; or between a small group in the context of the whole set of files.

There is inherent similarity in program code because of the constraints of the language and because of idiomatic style [16, 33]. This similarity may be increased when documents are drawn from a related set, such as files from the same project, or from a set of student assignments [52]. When comparing files, this background similarity can obscure more interesting specific similarities.

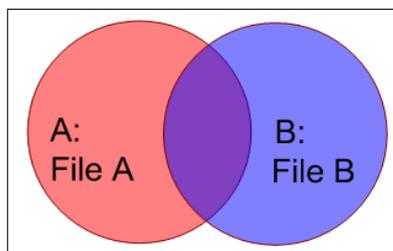


Figure 7.1: The Venn diagram represents the trigrams in the files A and B.

In the next section, similarity in program code is discussed and compared to similarity in natural language text. Measures which discount background similarity and target specific relationships between files can be derived from the trigram-file index. These measures differ depending on the application; those for collusion detection and for software origin analysis are described in Sections 7.2 and 7.3 respectively.

7.1 Similarity in program code

In any group of documents there are elements which occur frequently. For example, articles and prepositions will appear in most English language texts. If the documents are on the same subject, topic-related words may also be repeated [23, 59]. In program code, keywords, idioms such as “for (i=0; i<n; i++)”, and common constructs, such as “#include <stdio.h>” in C, will appear in many files. Other causes of similarity between documents within and across projects are considered in the next two sections.

7.1.1 Within-project similarity

In software origin analysis, the interest is in code which has moved between files. For example, code from files which have disappeared from the system may have moved to one or more files in the next release. Similarity measures offer one way to try to find out where this code has gone. Within a project, background similarity between files can be high because:

- there are common function calls in the project,
- there is an “in-house” style, or
- a copy-paste-edit sequence has been used to create new code [33, 204].

When a file has simply been renamed, the location of the new file will be evident because the two files have a similarity of, or close to, one. However, where the file has been edited, split, or merged with another file, there may be a number of files with similarities of the same order as the true destination file(s).

7.1.2 Across-project similarity

Investigating similar code in programming assignments differs from investigating similar text in essay-based assignments. There is likely to be more duplication in code-based assignments [52, 83, 99, 174], because of:

1. code typically provided by the tutor and therefore likely to be in most submissions, such as code used in exercises and examples during the course, or template code provided as part of the assignment; or
2. the constraints of the task, which may lead to similar code, or the requirement for the students to use a development tool which automatically produces code (e.g. Microsoft Visual Studio^(TM)).

When comparing assignments using the usual proportional measures, similarity between the files may be high in a number of situations where copying has not taken place. This is particularly so when two students have done little independent work. For example, if they have used in-class examples and not developed the ideas, they will share much of their code with other students who do the same. Similarity will also be increased when there is automatically generated code, which will be common to all, or most, of the submissions. This code will have a greater impact on the similarity between small files than between larger ones, if the similarity measure does not take account of frequency of occurrence, or on the larger files when frequency is accounted for.

7.1.3 Comparing similarity scores in text and code

In this section, the similarity between program code files is compared to that of files of text of approximately the same size, to establish whether there is a difference between the two.

Analysis of random selections of 10,000 words from books in the Gutenberg Project by Lyon et al. [154] showed the highest Ferret similarity score between texts belonging to the same book to be 0.03, and between texts from different books, 0.002.

In the projects selected for origin analysis (see Chapter 12), the mean line count per file is 260, and the mean token count per line is 7, making mean file size around 1820 tokens. To find a measure of similarity in natural language text to compare to within-project similarity and inter-project similarity, 21 Dickens books, and 84 books¹, by different authors were selected, also from the Gutenberg project. The books have a mean of nearly 11 words per line, so that 170 lines is approximately equivalent to 1820 words.

Chunks were taken from the books by selecting a random start point and a random number of lines between 10 and 330. The chunks were selected in the following groups:

1. 1 from each of the mixed set of 84 books
2. 1 from each of the 21 Dickens books, and
3. 84 non-overlapping chunks from one book, Bleak House.

The resulting chunks ranged in size between 41 to 3184 words, with a mean of 1816. Mean similarity between the mixed chunks was 0.0013, that between the Dickens chunks was 0.0025, and between the Bleak House chunks, 0.003. The figures for separate books are similar to those of Lyon et al., but those from the same book are very different. The range for pairs of Bleak House chunks is 0.0–0.0135, and for the set of complete Dickens books, which contain a mean of 117,000 words is 0.000004–0.0667. The maximum similarity value for the chunks of 10,000 words of Lyon et al., 0.03, lies between these two maximum values, indicating that document size is likely to be a factor in similarity. It therefore seems reasonable to compare 0.003, the within-book similarity for chunks of comparable size to the files, to the within-project similarity in code.

The mean within-project similarity of the 89 projects collected for this work, based on the most recent release of each project, is 0.034, more than 10 times that for the within-book chunks of a similar size to the project files. It is therefore likely to be more difficult to separate relevant similarity from incidental similarity in files of code than in files of text.

¹The mean number of files per release in the projects is 84.

7.1.4 Stopping

In comparing natural language documents, stop-lists are sometimes used to exclude common words, such as "the" or "a" in English texts, from calculations of similarity between documents [20]. Han and Kamber [103] state that beside these common words, stop words may also be context dependent, so that words which occur frequently in the set of documents to be compared are added to the list of excluded words.

The same principle can be applied to program code. Several collusion detection tools, such as JPlag [189], Moss [3], and Plaggie [2] allow the user to provide template code for exclusion from the similarity calculations.

This idea is extended here, so that all frequently occurring elements within the set of documents are also excluded from the similarity measures, leaving only the less common elements accounted for. Ribler and Abrams' [199] use a similar method, highlighting elements in the code which are unusual within the group, as shown in their graphs in Figures 5.4a and 5.4b (p.70). The application of these measures to collusion detection and to origin analysis is discussed in the next two sections.

7.2 Collusion detection

Joy and Luck [120] highlight two reasons for one student to copy another's work: they are either unable to understand how to do the work, or are unwilling to take the time to do so. In both cases, it might be assumed that the ability or time needed to disguise the copied work is lacking. As discussed in Chapter 2, unusual elements shared by two or more student submissions trigger suspicion of inappropriate collusion.

Finding code which is unusual in the group, but which is present in two, or a small group, of assignments can thus provide clues to inappropriate collusion. The review of 29 approaches to source-code plagiarism detection in Section 2.3 showed that the majority of the approaches focus on overall file similarity, and not on unusual shared elements. In particular, none of these approaches directly measure the unusual similarities between files.

Two reasons for code appearing in many files among a group of student assignments were identified in Section 7.1.2: code provided for the course, and code produced due to the constraints of the task. Either one or both of these elements can be discounted by analysing the trigram-file index. Provided code can be added to the set of documents presented for comparison, allowing the trigrams in the code to be identified, and excluded from similarity measures if required. To exclude commonly occurring code, the number of documents in which each trigram occurs is taken into account.

7.2.1 Example trigram-file index

Table 7.1 gives an excerpt from a made-up trigram-file index to help explain the analysis. Thirty-one files are compared, file [0] is the code provided for the course, and files [1–30] are the student assignments. Trigrams are listed on the left of the table and the files in which the trigrams occur are on the right. Several interesting relationships between files are exemplified here.

The top three lines, section A, show a set of trigrams which are only in one file. These trigrams represent code which is not shared by other students and can be seen as an indication of the student's individual effort.

The next four lines, section B, show trigrams shared by just two students, numbers 12 and 24, indicating possible collusion. The trigrams in section D also appear in the code of students 12 and 24, but are shared by a few other students as well. It is possible that this is the result of collusion among a small group, or that the two students, 12 and 24, have copied from other students. There is evidence, from membership of the small groups of files sharing the trigrams in sections C, D, and E, that student 19 is working with students 12 and 24. Of the thirteen trigrams for student 19, six are in the provided code and therefore unlikely to be cause for concern. The remaining seven are shared with student 24, and five by both 12 and 24.

The six trigrams in section F are in the provided code [file 0], unsurprisingly, more assignments include these trigrams. The three trigrams in section G also appear in a large number of files, although not in the provided code. Sharing these trigrams will not usually indicate collusion.

	Trigram	Files
A. unique_identifier = -1 } unique_identifier = else } unique_identifier FILES:[17] FILES:[17] FILES:[17]
B.	char* id123 , void func234 (; shiftleft (if (xyz	FILES:[12 24] FILES:[12 24] FILES:[12 24] FILES:[12 24]
C.	; struct abc	FILES:[19 24]
D.] ; struct * list) char timestr [nodedata); timestr [50	FILES:[12 19 24] FILES:[12 17 19 24] FILES:[2 12 19 24] FILES:[2 12 19 24] FILES:[1 12 14 24]
E.), frame int a =	FILES:[13 19 20 24] FILES:[3 8 12 16 17 19 20 22 24 25 27 29]
F.	int y = = 1 ; printf (" = (" static void { (i =	FILES:[0 4 6 7 9 13 16 17 19 20 22 24 27 29] FILES:[0 1 3 5 6 7 8 10 12 14 15 17 18 19 20 21 23 24 25 26 27 28 29 30] FILES:[0 1 3 4 6 7 9 10 12 14 15 17 19 20 23 24 25 27 28 29 30] FILES:[0 1 3 5 9 12 13 17 19 24] FILES:[0 2 3 4 5 6 7 8 11 13 14 15 17 19 22 24 25 26 28 30] FILES:[0 1 3 4 5 6 7 8 9 11 12 13 14 15 16 17 18 19 21 22 23 24 26 27 28 29 30]
G.	a = b d > f g == h	FILES:[1 2 4 5 6 7 9 11 12 13 16 17 18 20 22 23 25 26 27 29 30] FILES:[1 2 3 4 5 6 7 9 11 12 13 15 16 17 18 21 23 24 25 27 28 29] FILES:[2 3 4 6 7 8 9 10 13 14 15 17 20 21 22 23 24 25 26 28 30]

Table 7.1: Example extracts from a trigram-file index showing trigrams and the files where they occur. For example, the trigram “if (xyz” is in just two files, numbered 12 and 24, while the trigram “(i =” is in the majority of the 31 files. The letters A–G are not part of the report, but section labels.

This example shows how the trigram-file index can provide useful information about the interaction between the files presented for analysis. In the next three sections, measurements based on the trigram-file index are discussed. First, amendments to the standard proportional similarity measure are considered. Then two count-based measures are described: one counts the trigrams shared by only two files within the group, and the other extends this to count the trigrams shared by small groups of files.

7.2.2 Proportional trigram-based measures

Three variations on proportional measures which exclude commonly occurring trigrams are considered here. Instead of comparing the whole of the two files, elements are excluded from the comparison when calculating their similarity. These elements are:

1. trigrams in code provided as part of the course (P),
2. trigrams in code shared by the majority of other students (O), and
3. both of these sets of trigrams.

In Figure 7.2 the files A and B are represented in red and blue, as in Figure 7.1, also shown are the trigrams in the provided code (P) and in the other files in the group (O). Measure 1, which excludes the trigrams in the provided code, shown in Figure 7.3a, is

$$\frac{|(A \cap B) \setminus P|}{|(A \cup B) \setminus P|}$$

and measure 2

$$\frac{|(A \cap B) \setminus O|}{|(A \cup B) \setminus O|}$$

excludes commonly occurring code, i.e. that which occurs in other students' work, shown in Figure 7.3b. Combining the two previous exclusions gives measure 3

$$\frac{|(A \cap B) \setminus (P \cup O)|}{|(A \cup B) \setminus (P \cup O)|}$$

which will generally be the same as that which excludes other students' code, unless one student has used provided code which others have not,

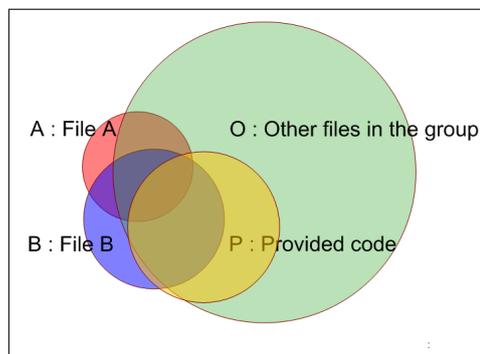


Figure 7.2: Venn diagram representing the trigrams in the files A and B, with those in the provided code (P) and those in other students' files (O).

Files	Ferret similarity score	Provided code excludes	Code shared by others excluded
12 and 24	$\frac{15}{21} = 0.71$	$\frac{11}{15} = 0.73$	$\frac{4}{4} = 1$
17 and 24	$\frac{11}{24} = 0.46$	$\frac{5}{15} = 0.33$	$\frac{0}{3} = 0$

Table 7.2: Different similarity measures based on trigram analysis

for example, by using alternative or advanced ideas introduced during the course which are not essential to the completion of the assignment.

To illustrate the measures, imagine that the trigrams in Table 7.1 are the only trigrams in the set. Three similarity measures, calculated for files 12 and 24, and for files 17 and 24, are shown in Table 7.2. The first of these measures is the standard Ferret score, the second excludes trigrams which are in the provided code, and the third excludes trigrams shared by any file other than the two of interest. As expected, those for files 12 and 24 are high, but as the common trigrams are removed, the similarity between the two files increases. Scores for the files 17 and 24 change as the method of calculation changes, from 0.46, through 0.33 to zero. Although the example is drawn from a very small subset of the trigrams in the set of documents, it shows that by removing the commonly occurring trigrams, the more interesting similarity between documents is clarified. An empirical study of a set of assignments is described in Chapter 9.

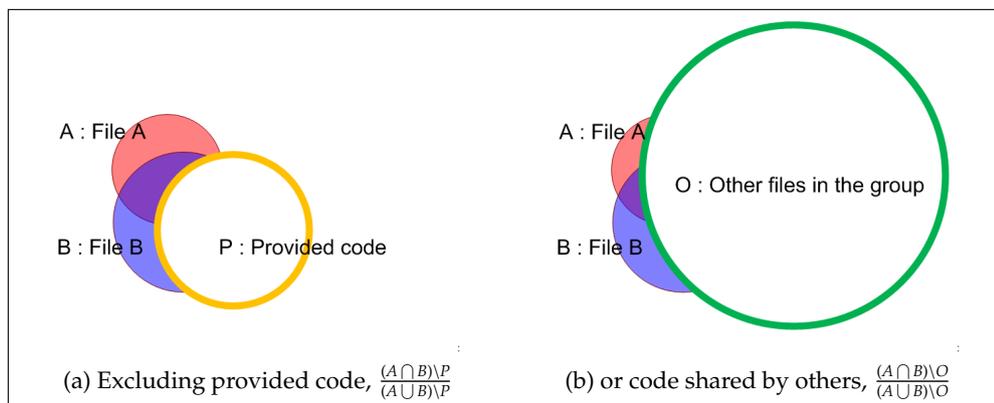


Figure 7.3: Venn diagrams representing the trigrams in the files A and B, with those in the provided code (P) or those in other students' files (O).

Analysis of the trigram-file index gives the number of trigrams shared by different combinations of documents, as in the extract in Table 7.3. For example, at the top left of the table, document 12 has 35 unique trigrams, while document 5 has 7,902. Below this, the uniquely shared trigrams are shown, such as documents 12 and 24 which share 586 trigrams not in any of the other documents in the group. It is from this analysis that measures are calculated.

7.2.3 Measures based on counting trigrams

When a portion of a large document is copied to another document, the proportion of copied trigrams to the total may be small. This means that, although significant, copying may be missed. In Figure 7.4, file B is compared with two other files, A and C. The similarity score between files A and B is $\frac{100}{300} = 0.33$, and between files B and C is $\frac{200}{1000} = 0.2$. File B is more likely to be derived from file C than file A, but the similarity score between files A and B is higher than that between files B and C.

No. of docs	No. of trigrams	Document identifiers	No. of docs	No. of trigrams	Document identifiers
1	35	(12)	4	5	(0 6 10 20)
1	165	(24)	4	10	(2 12 14 24)
1	720	(19)	4	25	(12 21 24 29)
		...	4	37	(12 16 21 24)
1	7407	(17)			...
1	7902	(5)	5	18	(1 12 19 21 24)
		...	5	24	(5 12 19 24 30)
2	50	(12 19)			...
2	154	(19 24)	6	48	(0 6 13 18 23 29)
2	586	(12 24)			...
		...	10	30	(0 1 3 6 7 9 11 13 17 21)
3	13	(6 12 24)			...
3	16	(12 14 24)	15	22	(0 2 4 6 7 9 11 15 17 20 23 26 27 28 30)
3	42	(12 21 24)			...
3	65	(12 16 24)	20	69	(0 2 4 5 7 8 10 12 14 15 17 19 21 23 24 25 27 28 29 30)
3	72	(12 24 30)			(0 1 2 3 6 7 9 11 12 13 14 16 17 18 19 20 22 24 26 28)
3	88	(12 21 24)	20	71	
3	240	(12 19 24)			

Table 7.3: The triples in this extract show the number of trigrams shared by a group of documents. For example, the top left triple shows that document 12 has 35 unique trigrams; and the top right, that 4 documents, 0, 6, 10, and 20, share 5 trigrams which are not in other documents.

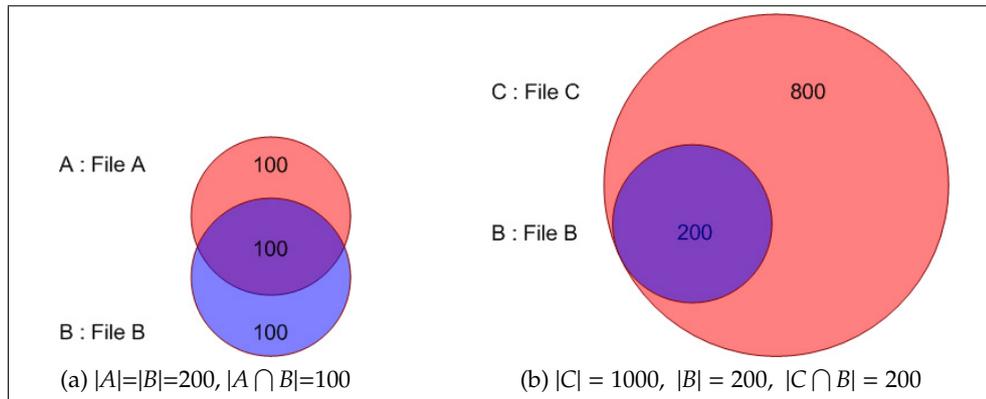


Figure 7.4: File B has 200 trigrams and is compared to two other files, A and C. File A also has 200 trigrams and the 2 files share 100 trigrams. File C has 1000 trigrams and contains file B.

One option which overcomes the potential drawbacks of proportional measures where file sizes differ is to count the number of trigrams uniquely shared by each pair of files, $|(A \cap B) \setminus (P \cup O)|$ in Figure 7.5. This count gives an idea of the amount of code shared by only the two documents.

Counting trigrams is useful in other respects: for giving a measure of individual effort, for measuring engagement with tasks set during a course, and for measuring group co-operation.

Trigrams which occur in only one document give an idea of its “uniqueness”. This may be seen as a measure of individual effort. Alternatively, if

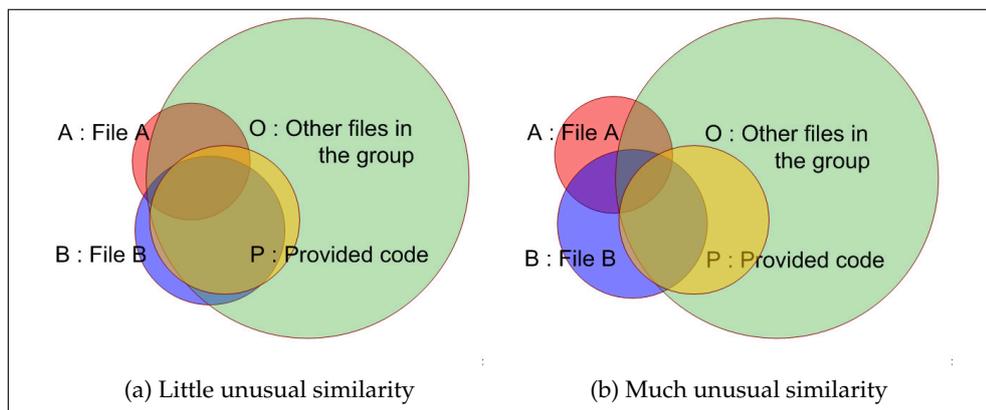


Figure 7.5: Files A and B share the same number of trigrams. On the left, the majority appear in other files in the group, while on the right, there are a large number of “uniquely shared trigrams”, $|(A \cap B) \setminus (P \cup O)|$.

there is suspicion that code has been sourced from the internet, the unique trigrams can be used as search terms.

If the students' work is regularly committed to a repository, then a measure of each student's engagement with exercises set during the course can be found by counting trigrams shared with the provided code. For example, $A \cap P$ or $B \cap P$ in Figure 7.5. It may be difficult to determine the number of trigrams which should be shared, as the provided code will probably need to be edited to complete the exercises. However, assuming some of the students are doing the exercises, a baseline level should be apparent, and students falling behind with the set work can then be identified.

7.2.4 Extending unique share counts

The similarity measure discussed in Section 7.2.3 focuses on uniquely shared trigrams. Collusion is not always between just two people, but may also be among a group. For example, in the excerpt in Table 7.1, students 12 and 24 not only uniquely share 3 trigrams, but also share 5 other trigrams with one or two others (section D). This may be because of group collusion or because the pair working together have had help from others in parts of their work, but is less likely to be due to incidental similarity.

The count of trigrams shared by just two students can be extended to include the trigrams shared by the two and by a small group of others. In Figure 7.6, the area shared by the two files A and B, and by other files except for the provided code, $((A \cap B \cap O) \setminus P)$, is shown split into sections. Some of these trigrams will be shared by files A, B, and just one of the other files. The other file can be any one of the rest of the group. In the top left part of the diagram, three such groups are shown. These groups are formed by A, B, and each of the files labelled i (green), ii (pink) and iii (cyan). In the lower left section, three groups of four files which include files A and B are shown in the same way.

One way to compute this extended count of shared trigrams is to weight the trigrams shared by A, B, and n others according to the number of files containing the trigrams. Illustrated in Figure 7.7, the two files, A and B, uniquely share 100 trigrams, share 60 with any one of the other files, and

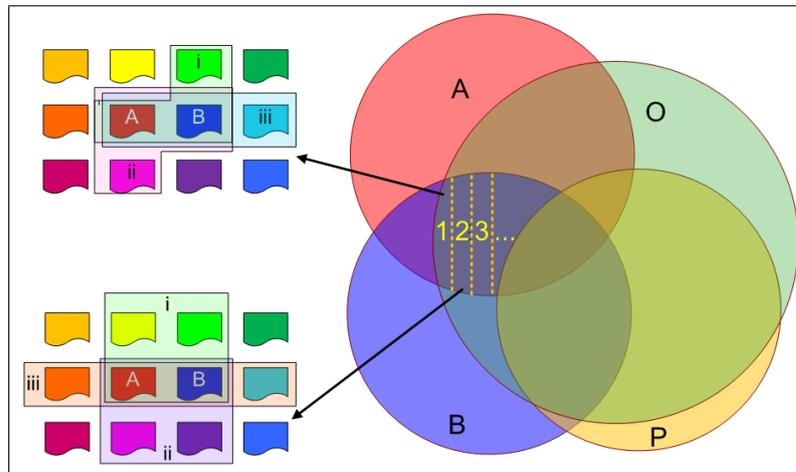


Figure 7.6: Two files may share trigrams uniquely, or may share trigrams which are shared with few other files. This diagram illustrates the files A and B, and examples of the way they may share trigrams with one other file at the top, or two other files at the bottom.

240 with any two others. If the shared trigrams are weighted by $\frac{1}{n+1}$, the weighted similarity count is:

$$\left(100 * \frac{1}{1}\right) + \left(60 * \frac{1}{2}\right) + \left(240 * \frac{1}{3}\right) = 100 + 30 + 80 = 210.$$

In this example, groups of up to four files are taken into account, however, the method allows for calculation of any maximum group size.

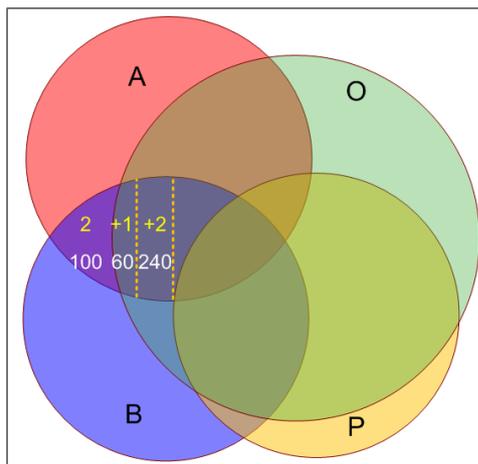


Figure 7.7: Files A and B uniquely share 100 trigrams, share 60 which are also shared with one other member of the group, and 240 with two others.

7.2.5 Making connections

Several of the plagiarism detection tools reviewed in Chapter 2 produce graphs of clusters of similar files, for example, those shown in Figure 5.3 (p.69). In common with other similarity measures, the weighted trigram counts between each pair of documents can be used to construct a similarity graph for a set of files. Figure 7.8 shows a graph for a small example set of six files [1–6]. The lengths of the connections are inversely proportional to the weighted similarity between the file pairs, marked on the graph edges.

By setting a threshold value below which connections are removed, groups within the set can be found. For this example, connections of less than the mean weight, 160, are removed, leaving the three subgraphs shown in Figure 7.9. File 2 is unrelated to the other files in the set, files 1 and 5 are related, and files 3, 4 and 6 form a totally connected group. Relationships between the files indicate possible collusion between the authors.

Different structures will indicate different types of co-operation in a group. For example, the group shown in Figure 7.10, where file x is at the centre of the four files, labelled a–d. Here, file x could result either from a student having had help from a number of others (a, b, c and d), or from student x providing help to the others in the group in different aspects of the work. Connections between the “satellites” may mean that the connected

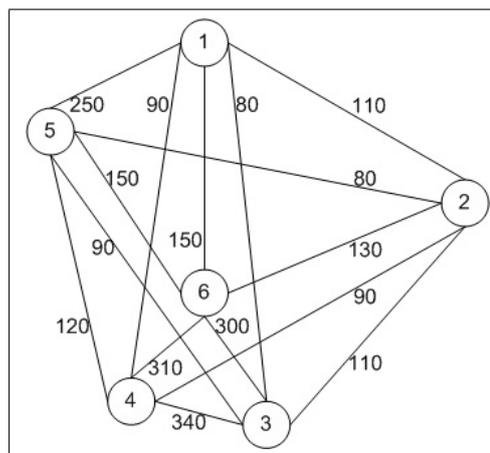


Figure 7.8: An example graph for a set of six files. The connection lengths are inversely proportional to the weighted similarity between the files.

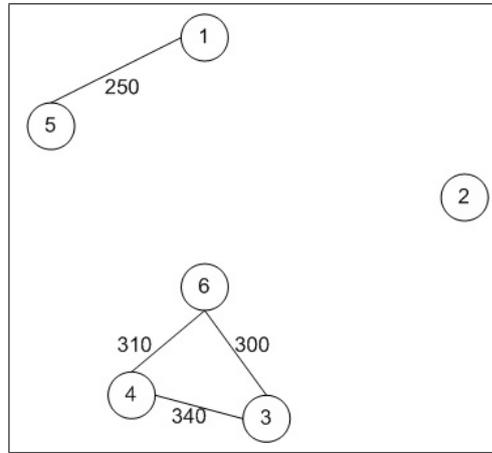


Figure 7.9: Connections \leq the mean weight, 160, are removed from the graph in Fig. 7.8 leaving 3 subgraphs, showing that files 3, 4 and 6, and files 1 and 5 are related. File 2 is not especially similar to other files in the set.

students have given or received help in the same area, or that one student has passed code taken from the source to another student.

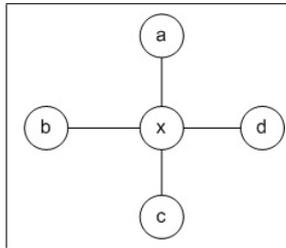


Figure 7.10: A group centred on one document implies that the file at the centre is either the source, or the recipient, of code in the other files.

7.3 The source or destination of code in origin analysis

When tracing the source or destination of code which has moved in a restructured software system, a file which has been subject to change is compared to all files in the next release to find target files from, or to which, code has been moved. This comparison can be efficiently undertaken using Ferret. However, as discussed in Section 7.1, similarity between files does not necessarily mean that code has been moved from one file to the other.

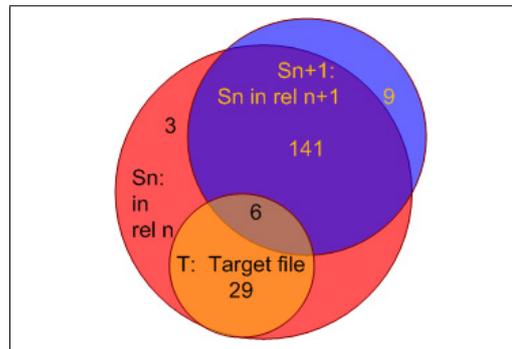


Figure 7.11: A Venn diagram of the trigrams shared by a file which is split, S_n , its amended form, S_{n+1} , and a new target file, T. This diagram corresponds to the split file example *cnp-1.c* (S_n), *cnp-2.c* (S_{n+1}) and *fact.c* (T).

7.3.1 Split files

The Venn diagrams in Figures 7.11–7.13 show the distribution of trigrams in three example file groups, where a file is split into new, existing, or multiple files. The circles in these diagrams are placed to represent the approximate overlap between the files: one in release n , S_n , in red, from which code has been moved, the revised version of this file in release $n+1$, S_{n+1} , in blue, and a number of target files, labelled T[m], also in release $n+1$, selected as possible recipients of the code because of their similarity to the original file.

Figure 7.11 is the simple split file example, *cnp-1.c*, its revised form *cnp-2.c*, and the new file *fact.c* (see pp. 81–82). The trigram analysis is

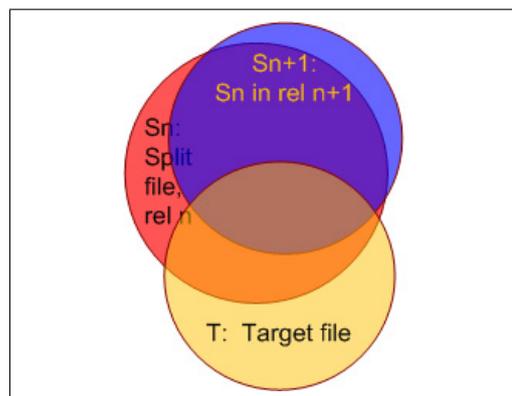


Figure 7.12: A split file where the extracted code is placed in an existing target file. The trigrams from S_n are also mostly in either of the files S_{n+1} or T, but file T has more trigrams unique among this set of files than Fig. 7.11.

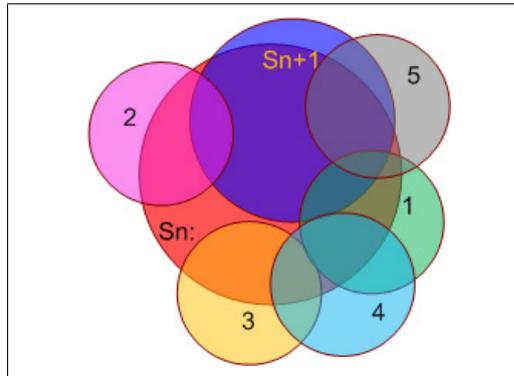


Figure 7.13: A more complex split, where the code moved from S_n has been put in target files 1, 2 and 3. Targets 4 and 5 are selected as likely recipients of the extracted code because of their incidental similarity to S_n .

in Table 6.4 (p.90). Most of the trigrams from file S_n are either in S_{n+1} or T. A few trigrams are unique to the two larger files, the result of minor adjustments to the code, and background similarity means that six trigrams are shared by all three files. Figure 7.12 shows another file, from which code is moved to an existing file. This is similar to the simple split in Figure 7.11, but the target file T has more trigrams unique among this set of files.

A multiway split is depicted in Figure 7.13. The target files, labelled 1–5, are selected because they are similar to S_n . Of these targets, files 1–3 are the real targets, and files 4 and 5 are incidentally similar. File 4 shares code with files 1, 3 and S_n , but the code shared with S_n is covered by files 1 and 3. File 5 shares code with S_n , and is why it has been selected as a possible target, but the code is also shared with file S_{n+1} , so has not been moved.

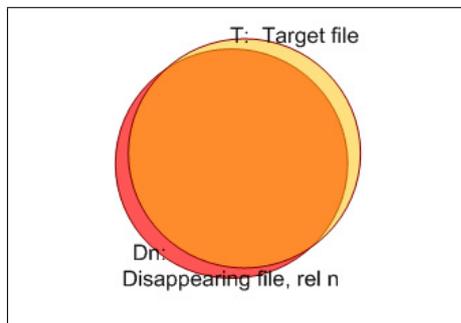


Figure 7.14: The file D_n has been renamed or moved to become file T.

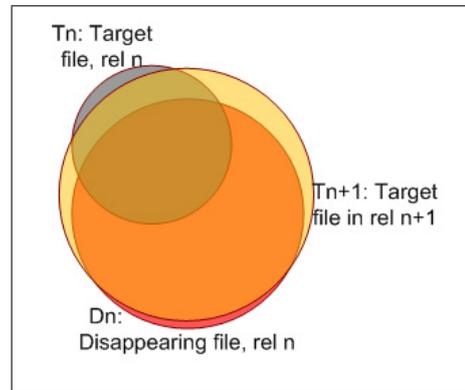


Figure 7.15: The file D_n has merged with the file T_n to create the larger file T_{n+1} .

7.3.2 Disappearing files

Figures 7.14–7.16 depict three possible destinations for a file which has disappeared. Each of the disappearing files is labelled D_n and shown in red. The target files are labelled $T[m]$. In the first diagram, 7.14, the file has been renamed or moved, so that there is another file in the system which is either identical, or, as in this case, very similar.

Figure 7.15 shows that the file D_n has been merged with the file T_n to produce T_{n+1} , which contains the code from each of the files. In this case, some of the trigrams are shared by the two original files, but this is not a precondition of merging. The last diagram, 7.16, shows a disappearing file which has split, with the code going to the target files $T1$ and $T2$. This is similar to the simple split file except that the name of both files has changed.

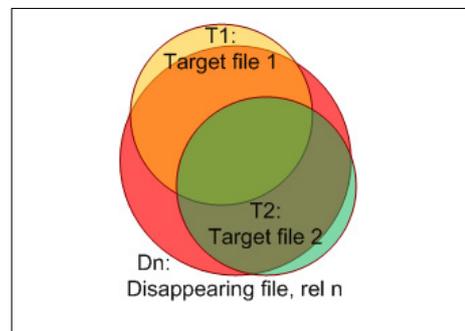


Figure 7.16: This disappearing file, D_n , has split to become files $T1$ and $T2$. This is like the simple split in Figure 7.11, but both files have new names.

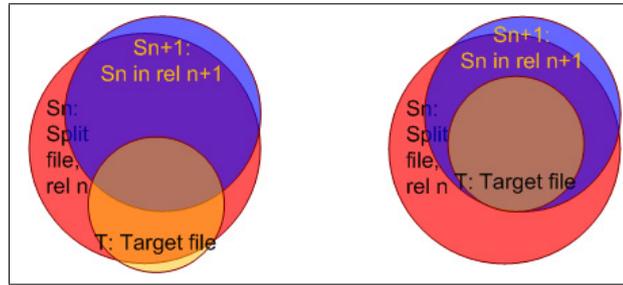


Figure 7.17: The diagram on the left is similar to Figure 7.11. The files on the right are the same size, but the target file trigrams are a subset of those in the amended file, making the similarity $\frac{(S_n \cap T) \setminus S_{n+1}}{(S_n \cup T) \setminus S_{n+1}} = 0$.

Other splitting configurations are not shown here.

7.3.3 Trigram-based measures

The file similarity measures suggested for collusion detection in Sections 7.2.2 and 7.2.3 exclude common code from the calculations. The same idea can be used when measuring similarity between files for origin analysis.

The left-hand diagrams in Figures 7.17 and 7.18 are similar to those in Figures 7.11 (split to a new file) and 7.12 (split to an existing file) respectively. In the right-hand diagram in each figure, the files are similar to those on the left, but do not represent split files, as the trigrams shared by S_n and T overlap those of S_{n+1} . The trigrams in the group can be analysed to produce various measures. For example: $\frac{(S_n \cap T) \setminus S_{n+1}}{(S_n \cup T) \setminus S_{n+1}}$ which excludes the trigrams

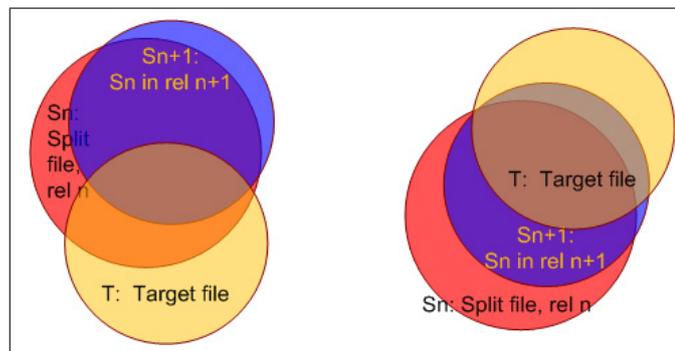


Figure 7.18: The left-hand diagram repeats Fig. 7.12. On the right, the target and amended files overlap. Here $\frac{(S_n \cap T) \setminus S_{n+1}}{(S_n \cup T) \setminus S_{n+1}}$ will also be close to zero.

in the revised version of file S_n from the similarity measure between the original and target files. In the examples, the measure will be zero for non-split files and thus provide useful separation from split files. For real examples, the distinction may be less clear, but should still be valuable.

The measure $\frac{S_n \cap [(S_{n+1} \cup T) \setminus (S_{n+1} \cap T)]}{S_n \setminus (S_{n+1} \cap T)}$ excludes commonly occurring code, i.e. that which occurs in all three files. This measure will be high if most of the code is in only one other file, as the left-hand diagrams of Figures 7.17 and 7.18, and lower if the files are related as the right-hand diagrams.

Another example is the proportion of trigrams unique to a file to the total in the file, such as $\frac{S_n \setminus (S_{n+1} \cup T)}{S_n}$. The meaning of this measure depends on the file in question. For the candidate split file, S_n , a high proportion of unique trigrams indicates deletion from the file, or heavy editing. For S_{n+1} , the amended version of the file, it means there is edited or additional code in the file. If the target file has more than a few unique trigrams, this may mean that the file already existed, or that other code was added to the file.

These, and other proportional measures are used as features for the classification of restructured files by machine learning, and are described in full in Chapter 13. Trigram counts, such as the number of trigrams in the candidate file which are shared with one other file, or the sum of the trigrams in the amended and target files, are also used as features in the machine learning task. The number of trigrams shared by the candidate file and each of the target files is used in ranking target files to select the most likely recipients of the code removed from a candidate file (see Chapter 15).

7.4 Summary

This chapter has covered techniques for identifying interactions between files in a set from the distribution of trigrams. Alternatives to the standard Ferret similarity measure were introduced and their application to collusion detection discussed. Also, features which describe various relationships between files were suggested for use in origin analysis.

Chapter 8

Visualising file relationships

The use of trigram analysis to display information about the interaction between files in the context of a group is explored in this chapter. As explained in Chapter 5, file comparison tools do not generally provide text-based comparisons which show how the code shared by two files relates to the code found in the rest of the files in the group. Also, there do not appear to be any tools which relate the text in one file to that in more than two others. As part of this research, the trigram-file index was analysed to create text-based displays of comparisons between one or more files in the context of a group. The next section looks at a selection of displays based on trigram analysis, used to show the interaction between student assignments. The following section describes a method for showing the relationship between one file and many others, which has proved invaluable in origin analysis in this research.

8.1 Displaying student assignments

Three ways of highlighting aspects of a file in relation to a group are included here. First, to show the text unique to one file. Second, to extend the standard Ferret display which shows text shared by two files. The extensions take two forms: one adds a distinction between shared and uniquely shared code; and the other further extends this idea to show other cate-

gories, such as code shared by the two documents and by few others in the group. The examples given here suggest particular colours for highlighting, but font colour and style can be adjusted to suit the user's preferences.

8.1.1 Unique trigrams

Figure 8.1 shows two extracts from a document with the elements unique within the comparison group highlighted in black. The top part is typical of a "function" introduced in class and amended to suit the student's project. The lower part is typical of independently written (or sourced) "code".

Picking out the unique parts of a student project may be useful if there is doubt about whether the code has been produced by the student or sourced from a third party, or to see what interesting ideas the student has implemented. This display may also be useful in comparing other documents, such as contracts, where it is often important to be able to pinpoint differences in the texts, rather than their similarity.

This is a pretend function which was introduced in class or in the exercises set during the course. Some of the **identifier names** have been changed to suit the student's work and will **therefore be unique**.

However, this pretend function is one written by the student to suit the needs of their project and has little in common with the functions in other student's projects. This function will stand out from the rest of the code when the unique parts are coloured.

Figure 8.1: An example file with trigrams unique within the comparison group highlighted in black. The upper part of the diagram shows "code" mostly used by other students, the lower part shows an independently written function. Commonly occurring "code" fragments are in cyan.

8.1.2 Extending the standard Ferret display

The standard Ferret display, see the top half of Figure 8.2, shows the text covered by trigrams common to two files in blue, with the rest in black. The lower part of this diagram shows a simple extension to the colour scheme where code which is “uniquely shared” by the two files is coloured red to highlight possible collaboration between the two students. Each token appears in three trigrams, if two or three of these are uniquely shared between the two files then the token is coloured red.

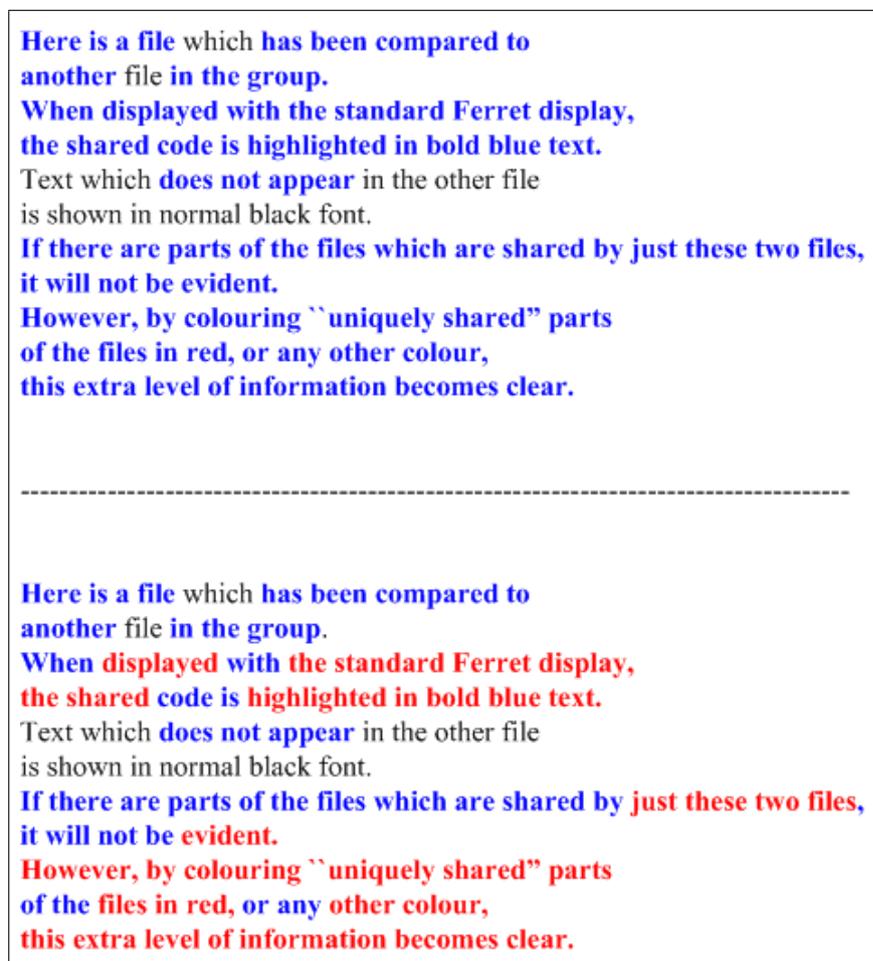


Figure 8.2: An example of the revised colouring. This passage is repeated, the upper version is standard Ferret display. In the lower version, parts of the text which are uniquely shared by the two files are in red.

8.1.3 Graduated similarity information

The black-blue-red colour scheme can be extended to add information about trigrams shared by only a few others in the group. The idea is to display not only the “uniquely shared” code but also evidence of possible group collaboration. Following the idea of the weighted trigram count, parts of the code shared by the two students and by few others are highlighted as shown in Figure 8.3. Assume that a group of files, including file A and file B, are compared. The text of file A is coloured to reflect its similarity to file B within the context of the group. Where text is:

- **black** it is unique to file A;
- **red** it is “uniquely shared” with file B;
- **dark orange** it is shared by file B and up to 2 other files;
- **paler orange** it is present in files A and B and 3–5 other files;
- **blue** it is in files A and B as well as many other files; and
- **pale blue** it is either provided for the course, or is not in file B; in other words, parts of the file of no interest when checking for collusion.

Figure 9.8, on page 144, shows this colour scheme applied to a pair of files from the set of student projects analysed in the next chapter. The size of the subgroups is arbitrary, and it is possible that a larger group warrants larger subgroups. However, Burrows [34, p.14] states that “*Our test data and anecdotal evidence indicates that students generally do not work in very large groups*”, and this is taken into account.

When comparing two files, shared trigrams are differentiated as follows:

1. “Code” which is unique to this file (A) is coloured black.
2. If “uniquely shared” by file A and the file to which it is compared (B), then it is red.
3. When the two files share code with any one or two other files, it is in dark orange.
4. If A, B, and 3, 4 or 5 other files contain the code, it is in orange.
5. Code shared by A, B, and 6 or more other files is in blue.
6. Uninteresting code, either that provided for the course, or code not shared by the two files, is pale blue.

Figure 8.3: One way to colour the code in a file (A) compared with another (B). The colours and gradations can be altered to suit the user’s needs.

8.1.4 Groups

A method for displaying the relationship between one file and a number of other files is explained in Section 8.2. Although developed for visualising the relationships between files in origin analysis, it is also useful for other file comparison tasks. In collusion detection, it can provide information about which file contains what code in a group of similar files. This is particularly useful where one student is thought to be either the main developer, or the main recipient of the code, as in Figure 7.10 (p.107), when their file can be used as the base against which the other files are compared. Figure 9.9 (p.146) gives an example of this visualisation applied to student code.

8.2 Comparing one document with a group of others

A large part of the work involved in creating marked-up datasets for origin analysis is in manually comparing groups of files to try to understand their interaction. Most file comparison tools only show the similarities (or differences) between pairs of files, although some tools, including KDiff3 (see Figure 5.5c, p.71), allow for three files to be viewed together. Although comparison between one file and a group of others can be accomplished by arranging pairwise comparisons alongside each other, as in Figure 8.4, it is inefficient. Not only must each comparison be selected and arranged on the screen separately, but also to view the comparisons side-by-side, each one must be scrolled separately, tasks which are awkward with a few target files, and which become more difficult as the number of files increases.

What is required is a tool which can show the relationships between one file and any number of other files. The development of such a tool is explained in this section. The ideas are illustrated using one file from the split file dataset, which is introduced in Chapter 12. This file (fa.c) is large, having nearly 1800 lines, and has been split to produce three files. Two different ways of displaying this information are described here.

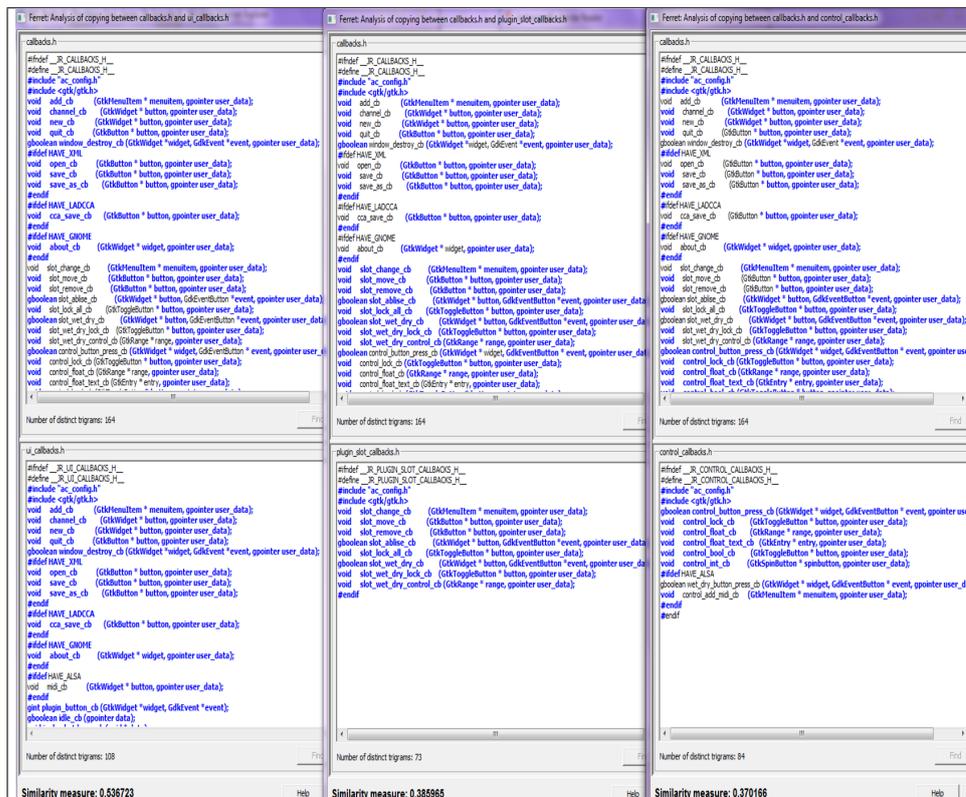


Figure 8.4: In looking at how one file relates to a set of others based on 2-way file comparison, the user must align each window on the screen, and scroll through them individually. Here `callbacks.h` is compared with each of `ui_callbacks.h`, `plugin_slot_callbacks.h` and `control_callbacks.h`.

8.2.1 Multiple blue-red-black displays

The first idea was to apply the colouring technique described in Section 8.1.2 (blue shared, red uniquely shared, otherwise black) several times to comparisons between the candidate file and each of the potential target files, then to view the results side-by-side to help determine the destination of code from a split file, or the sources of a merged file.

On the left of Figure 8.7 (p.122), three versions of the example file are shown side-by-side. The first is compared with the amended file in the next release, the second and third versions are compared to the two target files. The mostly red coloured text shows that the amended version of the file has retained code from the first and last parts of the file. The most similar

target file contains most of the rest of the first half of the code. The other target file contains the remaining code from the second half of the file, as well as a small section from the middle of the part marked by dashed lines.

This marked section is shown in more detail in Figure 8.8 (p.123). This figure shows that the small function 'validate-reply' is in the third file while the remainder of the code in this section is in the second file. Using red to colour those trigrams uniquely shared by the two files gives a clearer picture of code destination than having all common code the same colour.

8.2.2 Displaying comparisons between one file and three others

Although the three comparisons shown on the left of Figure 8.7 are effective in identifying the destinations of a file, it would be more convenient to have this information in one column rather than three. A three-way colour

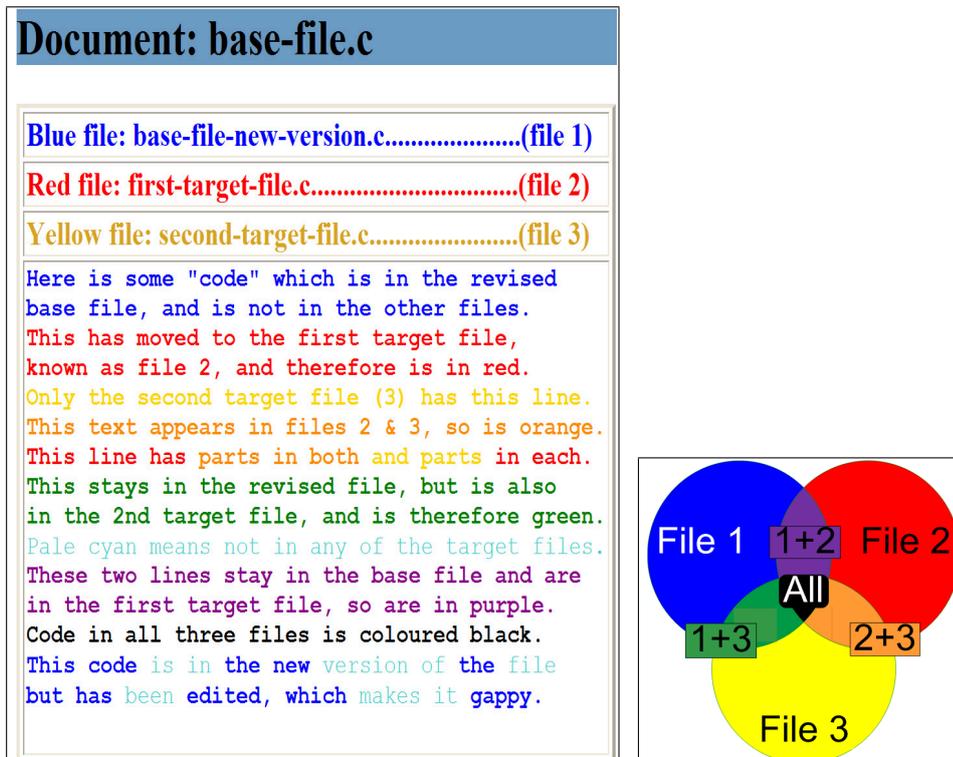


Figure 8.5: Each target file is assigned a primary colour. The base file text is shown and coloured according to which of the target files its trigrams are in.

scheme is developed here to show which of the target files share code with the file of interest, and so condense the information into one column.

The idea is to colour the base file code which is “uniquely shared” with the first file blue, with the second file red, and the third, yellow. If the code appears in two files, the standard mixes of primary colours, purple, green, and orange are used. If in all three files, then it is black, and if the code does not appear in any of the three files, the tokens are coloured pale cyan. The scheme is outlined on the left of Figure 8.5 and the colour mixes shown on the right.

To accustom the reader to the colour scheme, simple example comparisons are provided in Figure 8.6. On the left, a file which has apparently disappeared from the system is the base file. It is compared with the most similar file in the next release. The text is coloured blue where the trigrams appear in the other file, and cyan otherwise. The file on the right is compared with two other files. Where text is in the first file, in this case the new version of the base file, it is coloured blue. Text which appears in the other file, resulting from a split, is coloured red, and text in both files is purple.

The larger example file, *fa.c*, which is split three ways, is shown using this colour scheme to the right of Figures 8.7 and 8.8, where the blue, red, and yellow sections match the red parts of the comparisons on the left of the figure.

The pairwise comparisons between callbacks and three other files, first seen in Figure 8.4, are repeated in Figure 8.9 along with the equivalent group comparison, demonstrating its compactness. The group comparison overcomes the problems with multiple pairwise comparisons: it is fast, does not have to be organised on the screen and the comparisons scroll together. Also, by zooming out, a quick overview of large comparisons is possible.

UH-Ferret: Trigram analysis

Comparison generated by analysis of the trigram report produced by the Ferret Copy-Detection Tool, (c) School of Computer Science, University of Hertfordshire, 2010.

Document: ../pbbUTTONS-n/pbbcmd/src/pbbipc.h

Blue file: ../pbbUTTONS-n+1/pbbUTTONSD/libpbbipc/pbbipc.h

Red file: No second file

Yellow file: No third file

```
#ifndef INCLUDE_PBBIPC_H
#define INCLUDE_PBBIPC_H
#define CHUNK(a,b,c,d) ((a)<<24 | (b)<<16 | (c)<<8 | (d))
#define SERVERPORTKEY CHUNK('P', 'B', 'B', 'S')
#define MESSAGE_TYPE CHUNK('P', 'B', 'B', 'M')
#define MAXCLIENTS 10
#define REGISTERCLIENT 1
#define UNREGISTERCLIENT 2
#define REGFAILED 3
#define CLIENTEXIT 4
#define CHANGEVALUE 10
#define CHANGEERROR 11
#define WARNING 12
#define READVALUE 13
struct pbbmessage {
    long          message_type;
    long          returnport;
    long          action;
    struct tagitem taglist[1];
};
#endif
```

UH-Ferret: Trigram analysis

Comparison generated by analysis of the trigram report produced by the Ferret Copy-Detection Tool, (c) School of Computer Science, University of Hertfordshire, 2010.

Document: ../nap-n/sscr.h

Blue file: ../nap-n+1/sscr.h

Red file: ../nap-n+1/winio.h X

Yellow file: No third file

```
#ifndef _NAP_SSCR_H
#define _NAP_SSCR_H
#include "cmds.h"
struct sscr_s
{
    unsigned char ln[256];
    unsigned char d;
    chans_t *chan;
    struct sscr_s *own;
    struct sscr_s *prev;
    struct sscr_s *next;
};
typedef struct sscr_s sscr_t;
void sscr(sscr_t *);
void plist(sscr_t *, WINDOW *, int);
int sinput(WINDOW *, sock_t *);
void usr(void);
#endif
```

Figure 8.6: In each comparison, the base file text is coloured blue if it appears in the 1st target file, red for the 2nd, and cyan if neither. The left-hand file is compared with a very similar file, where 2 lines have been edited. On the right, the file sscr.h is split, forming the new file winio.h. Trigrams in the amended sscr.h are blue, in winio.h red, and purple if in both.

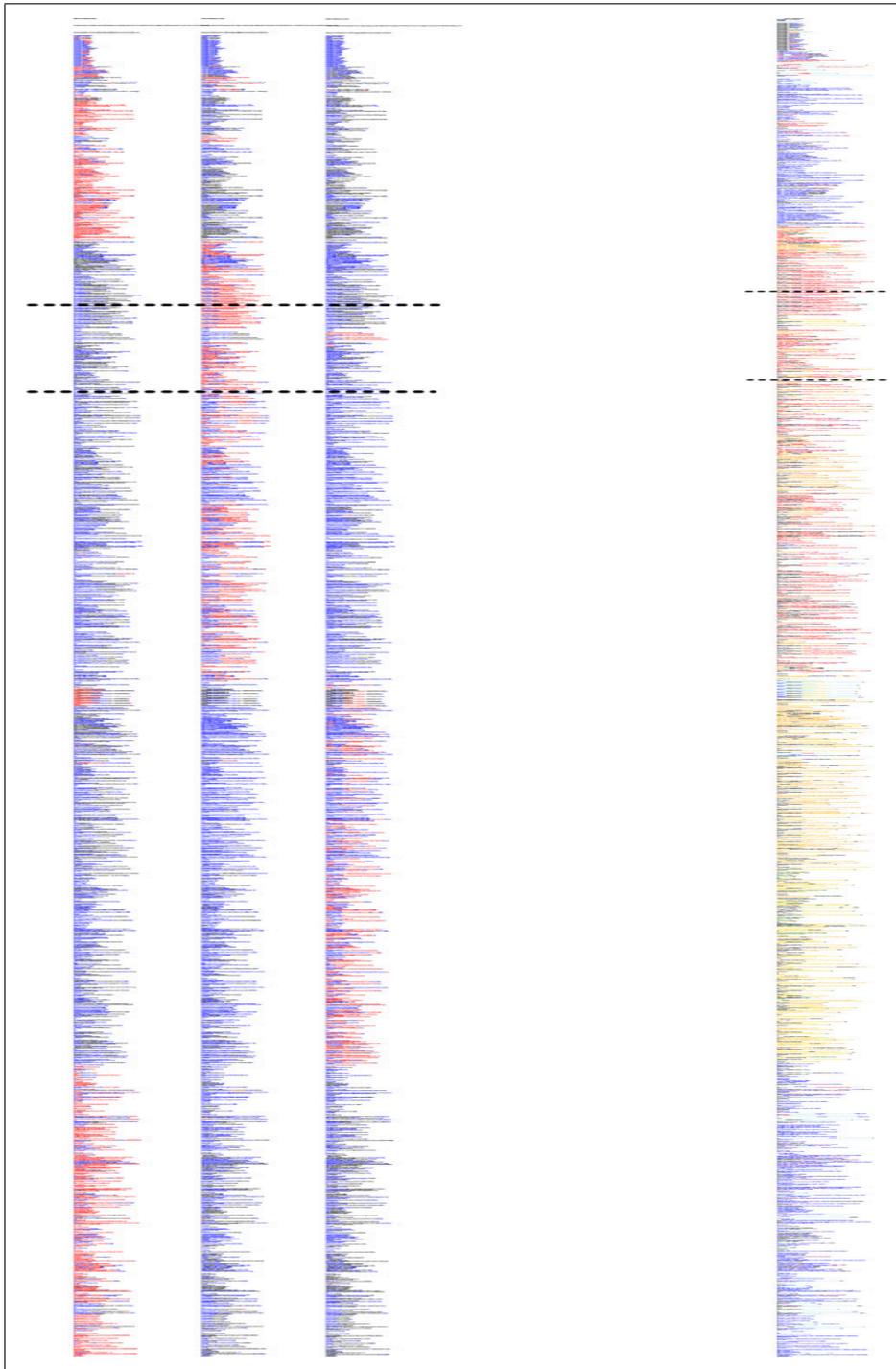


Figure 8.7: On the left, the candidate file is compared with each of three target files. Looking at the mostly red coloured code, it is clear that the majority of the file has been placed in one or other of the target files. On the right the same file is compared with the same 3 target files. Here code in the first file is coloured blue, in the second, red, and in the third, yellow.

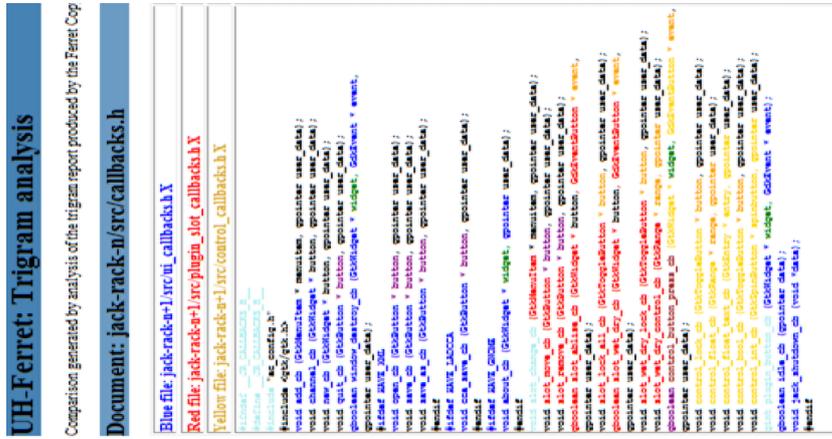
8.2. COMPARING ONE DOCUMENT WITH A GROUP OF OTHERS 123

```
LOG(LOG_WARNING, "validate_request: reverse tunneling not allowed!");
error_code = REGEXP_REVERSE_TUNNEL_UNAVAIL_FA;
}
else if ((test_req_opts &
REGEXP_REVERSE_TUNNEL) == 0
&& config_enable_reverse_tunneling != 0
&& config_enable_triangle_tunneling == 0)
{
LOG(LOG_WARNING,
"Denied request for triangle tunnel - "
"reverse tunnel required!");
error_code = REGEXP_REVERSE_TUNNEL_MANDATORY_FA;
}
else if ((test_req_opts &
REGEXP_REVERSE_TUNNEL) == 0
&& config_enable_reverse_tunneling == 0)
{
LOG(LOG_WARNING, "Denied request for
triangle tunnel!");
error_code = REGEXP_ADMIN_PROHIBITED_FA;
}
else if ((authorized_check (config-
authorized_networks, cil_addr, sin_addr))
{
LOG(LOG_WARNING, "request from an
unauthorized address 'x'");
inet_addr (cil_addr, sin_addr);
error_code = REGEXP_ADMIN_PROHIBITED_FA;
}
else if ((is_sender_public (test) && config-
allow_public_nodes == FALSE)
{
LOG(LOG_WARNING, "RM trying to access -
RM not allowed in this RM");
error_code = REGEXP_ADMIN_PROHIBITED_FA;
}
if (error_code != 0)
{
send_failure_reply (error_code, ext_req,
cil_addr);
ext_req_auth = NULL ? ext_req_auth : opt;
return 1;
}
return 0;
}
static int
validate_reply (struct msg_extensions
*ext)
{
if ((test_req_auth && ext_req_auth ==
REGEXP_ACCEPTED)
{
DEBNO (DEBNO_FLAG, "validate_reply:
missing sk_auth()");
return 1;
}
if ((test_req_lifetime != 0 && test-
fa_publickey && test_fa_publickey)
{
DEBNO (DEBNO_FLAG, "validate_reply:
missing fa_keyref(fa_publickey)");
return 1;
}
return 0;
}
static int
send_failure_reply (int code, struct
req_req *req,
struct sockaddr_in *cil_addr, int spi,
unsigned char *sk,
int sk_len)
{
unsigned char msg[MSGNDL];
struct req_req *reply;
int result;
int result;
if (last_failure_time == 0)
if (last_failure_time == time (NULL))
return 1;
time (last_failure_time);
DEBNO (DEBNO_FLAG, "Sending failure reply
code %i to %i:%i", code,
cil_addr->sin_addr, ntohs
(cil_addr->sin_port));
memset (reply, 0, sizeof (struct
req_req));
reply->lifetime = 0;
reply->type = REG_REQ;
reply->code = code;
reply->sender_addr &= req-
>home_addr;
reply->home_addr &= req-
>home_addr;
memset (reply->id, req->id, sizeof (req-
>id));
memset (reply->cid, req->cid, sizeof (req-
>cid));
if (sk != NULL)
{
memset (sk, vendor (AUTH_AIG_MES, sk, sk_len,
msg),
(struct vendor_msg_auth *) msg + msglen);
VENDOR_EXT_DYNAMICIS_OR_AUTH, spi);
if ((send_address_ok (cil_addr->sin_addr))
return 1;
}
result = req_socklen, req, spi, 0,
(struct sockaddr *) cil_addr,
sizeof (struct sockaddr_in));
check_result (result, msglen);
send_failure_reply();
return 0;
}
static int
add_key_request (struct msg_extensions
*ext, int *len, char *msg)
{
struct in_addr dest;
struct fa_spi_entry *fa_spi;
struct msg_key *key;
if (config_highport_FA)
dest = ext_req->home_addr;
else
dest = upper_fa_addr.sin_addr;
fa_spi = get_fa_spi (0, dest);
if (fa_spi != NULL)
{
if (*len && sizeof (struct msg_key) >
MSGNDL)
return 1;
key = (struct msg_key *) (msg + *len);
init_key_extension (key,
VENDOR_EXT_DYNAMICIS_FA_RETRYSD, htonl
(fa_spi->spi));
*len += GET_KEY_EXT_LEN (key);
DEBNO (DEBNO_FLAG, " + adding fa_keyreq
[lan => %i]%", *len);
}
else
{
if (*len && GET_KEY_EXT_LEN (fa_public_key)
> MSGNDL)
return 1;
memset (msg + *len, fa_public_key,
GET_KEY_EXT_LEN (fa_public_key));
*len += GET_KEY_EXT_LEN (fa_public_key);
DEBNO (DEBNO_FLAG, " + adding
fa_public_key [lan => %i]%", *len);
}
return 0;
}
LOG(LOG_WARNING, "validate_request:
reverse tunneling not allowed!");
error_code =
REGEXP_REVERSE_TUNNEL_UNAVAIL_FA;
}
else if ((test_req_opts &
REGEXP_REVERSE_TUNNEL) == 0
&& config_enable_reverse_tunneling != 0
&& config_enable_triangle_tunneling == 0)
{
LOG(LOG_WARNING,
"Denied request for triangle tunnel - "
"reverse tunnel required!");
error_code =
REGEXP_REVERSE_TUNNEL_MANDATORY_FA;
}
else if ((test_req_opts &
REGEXP_REVERSE_TUNNEL) == 0
&& config_enable_reverse_tunneling == 0)
{
LOG(LOG_WARNING, "Denied request for
triangle tunnel!");
error_code = REGEXP_ADMIN_PROHIBITED_FA;
}
else if ((authorized_check (config-
authorized_networks, cil_addr, sin_addr))
{
LOG(LOG_WARNING, "request from an
unauthorized address 'x'");
inet_addr (cil_addr, sin_addr);
error_code = REGEXP_ADMIN_PROHIBITED_FA;
}
else if ((is_sender_public (test) && config-
allow_public_nodes == FALSE)
{
LOG(LOG_WARNING, "RM trying to access -
RM not allowed in this RM");
error_code = REGEXP_ADMIN_PROHIBITED_FA;
}
if (error_code != 0)
{
send_failure_reply (error_code, ext_req,
cil_addr);
ext_req_auth = NULL ? ext_req_auth : opt;
return 1;
}
return 0;
}
static int
validate_reply (struct msg_extensions
*ext)
{
if ((test_req_auth && ext_req_auth ==
REGEXP_ACCEPTED)
{
DEBNO (DEBNO_FLAG, "validate_reply:
missing sk_auth()");
return 1;
}
if ((test_req_lifetime != 0 && test-
fa_publickey && test_fa_publickey)
{
DEBNO (DEBNO_FLAG, "validate_reply:
missing fa_keyref(fa_publickey)");
return 1;
}
return 0;
}
static int
send_failure_reply (int code, struct
req_req *req,
struct sockaddr_in *cil_addr, int spi,
unsigned char *sk,
int sk_len)
{
unsigned char msg[MSGNDL];
struct req_req *reply;
int result;
int result;
if (last_failure_time == 0)
if (last_failure_time == time (NULL))
return 1;
time (last_failure_time);
DEBNO (DEBNO_FLAG, "Sending failure reply
code %i to %i:%i", code,
cil_addr->sin_addr, ntohs
(cil_addr->sin_port));
memset (reply, 0, sizeof (struct
req_req));
reply->lifetime = 0;
reply->type = REG_REQ;
reply->code = code;
reply->sender_addr &= req-
>home_addr;
reply->home_addr &= req-
>home_addr;
memset (reply->id, req->id, sizeof (req-
>id));
memset (reply->cid, req->cid, sizeof (req-
>cid));
if (sk != NULL)
{
memset (sk, vendor (AUTH_AIG_MES, sk, sk_len,
msg),
(struct vendor_msg_auth *) msg + msglen);
VENDOR_EXT_DYNAMICIS_OR_AUTH, spi);
if ((send_address_ok (cil_addr->sin_addr))
return 1;
}
result = req_socklen, req, spi, 0,
(struct sockaddr *) cil_addr,
sizeof (struct sockaddr_in));
check_result (result, msglen);
send_failure_reply();
return 0;
}
static int
add_key_request (struct msg_extensions
*ext, int *len, char *msg)
{
struct in_addr dest;
struct fa_spi_entry *fa_spi;
struct msg_key *key;
if (config_highport_FA)
dest = ext_req->home_addr;
else
dest = upper_fa_addr.sin_addr;
fa_spi = get_fa_spi (0, dest);
if (fa_spi != NULL)
{
if (*len && sizeof (struct msg_key) >
MSGNDL)
return 1;
key = (struct msg_key *) (msg + *len);
init_key_extension (key,
VENDOR_EXT_DYNAMICIS_FA_RETRYSD, htonl
(fa_spi->spi));
*len += GET_KEY_EXT_LEN (key);
DEBNO (DEBNO_FLAG, " + adding fa_keyreq
[lan => %i]%", *len);
}
else
{
if (*len && GET_KEY_EXT_LEN (fa_public_key)
> MSGNDL)
return 1;
memset (msg + *len, fa_public_key,
GET_KEY_EXT_LEN (fa_public_key));
*len += GET_KEY_EXT_LEN (fa_public_key);
DEBNO (DEBNO_FLAG, " + adding
fa_public_key [lan => %i]%", *len);
}
return 0;
}
LOG(LOG_WARNING, "validate_request: reverse
tunneling not allowed!");
error_code =
REGEXP_REVERSE_TUNNEL_UNAVAIL_FA;
}
else if ((test_req_opts &
REGEXP_REVERSE_TUNNEL) == 0
&& config_enable_reverse_tunneling != 0
&& config_enable_triangle_tunneling == 0)
{
LOG(LOG_WARNING,
"Denied request for triangle tunnel - "
"reverse tunnel required!");
error_code =
REGEXP_REVERSE_TUNNEL_MANDATORY_FA;
}
else if ((test_req_opts &
REGEXP_REVERSE_TUNNEL) == 0
&& config_enable_reverse_tunneling == 0)
{
LOG(LOG_WARNING, "Denied request for
triangle tunnel!");
error_code = REGEXP_ADMIN_PROHIBITED_FA;
}
else if ((authorized_check (config-
authorized_networks, cil_addr, sin_addr))
{
LOG(LOG_WARNING, "request from an
unauthorized address 'x'");
inet_addr (cil_addr, sin_addr);
error_code = REGEXP_ADMIN_PROHIBITED_FA;
}
else if ((is_sender_public (test) && config-
allow_public_nodes == FALSE)
{
LOG(LOG_WARNING, "RM trying to access -
RM not allowed in this RM");
error_code = REGEXP_ADMIN_PROHIBITED_FA;
}
if (error_code != 0)
{
send_failure_reply (error_code, ext_req,
cil_addr);
ext_req_auth = NULL ? ext_req_auth : opt;
return 1;
}
return 0;
}
static int
validate_reply (struct msg_extensions
*ext)
{
if ((test_req_auth && ext_req_auth ==
REGEXP_ACCEPTED)
{
DEBNO (DEBNO_FLAG, "validate_reply:
missing sk_auth()");
return 1;
}
if ((test_req_lifetime != 0 && test-
fa_publickey && test_fa_publickey)
{
DEBNO (DEBNO_FLAG, "validate_reply:
missing fa_keyref(fa_publickey)");
return 1;
}
return 0;
}
static int
send_failure_reply (int code, struct req_req
*req,
struct sockaddr_in *cil_addr, int spi,
unsigned char *sk,
int sk_len)
{
unsigned char msg[MSGNDL];
struct req_req *reply;
int result;
int result;
if (last_failure_time == 0)
if (last_failure_time == time (NULL))
return 1;
time (last_failure_time);
DEBNO (DEBNO_FLAG, "Sending failure reply
code %i to %i:%i", code,
cil_addr->sin_addr, ntohs
(cil_addr->sin_port));
memset (reply, 0, sizeof (struct
req_req));
reply->lifetime = 0;
reply->type = REG_REQ;
reply->code = code;
reply->sender_addr &= req-
>home_addr;
reply->home_addr &= req-
>home_addr;
memset (reply->id, req->id, sizeof (req-
>id));
memset (reply->cid, req->cid, sizeof (req-
>cid));
if (sk != NULL)
{
memset (sk, vendor (AUTH_AIG_MES, sk, sk_len,
msg),
(struct vendor_msg_auth *) msg + msglen);
VENDOR_EXT_DYNAMICIS_OR_AUTH, spi);
if ((send_address_ok (cil_addr->sin_addr))
return 1;
}
result = req_socklen, req, spi, 0,
(struct sockaddr *) cil_addr,
sizeof (struct sockaddr_in));
check_result (result, msglen);
send_failure_reply();
return 0;
}
static int
add_key_request (struct msg_extensions
*ext, int *len, char *msg)
{
struct in_addr dest;
struct fa_spi_entry *fa_spi;
struct msg_key *key;
if (config_highport_FA)
dest = ext_req->home_addr;
else
dest = upper_fa_addr.sin_addr;
fa_spi = get_fa_spi (0, dest);
if (fa_spi != NULL)
{
if (*len && sizeof (struct msg_key) >
MSGNDL)
return 1;
key = (struct msg_key *) (msg + *len);
init_key_extension (key,
VENDOR_EXT_DYNAMICIS_FA_RETRYSD, htonl
(fa_spi->spi));
*len += GET_KEY_EXT_LEN (key);
DEBNO (DEBNO_FLAG, " + adding fa_keyreq
[lan => %i]%", *len);
}
else
{
if (*len && GET_KEY_EXT_LEN (fa_public_key)
> MSGNDL)
return 1;
memset (msg + *len, fa_public_key,
GET_KEY_EXT_LEN (fa_public_key));
*len += GET_KEY_EXT_LEN (fa_public_key);
DEBNO (DEBNO_FLAG, " + adding
fa_public_key [lan => %i]%", *len);
}
return 0;
}
}
```

Figure 8.8: Extracts from the file comparisons in Figure 8.7, expanding the section marked by dashed lines. In this section, most of the code from the base (candidate) file has moved to the second file, with one function, 'validate_reply', in the third file. On the left are a set of three comparisons, one for each target file. The comparison on the right echoes the three on the left in compact form. Yellow code is the same as the red in the third file, and red code, the same as the red code in the second file on the left.



(a) Ferret comparisons between `callbacks.h` and three other files



(b) Compact comparison

Figure 8.9: 3CO and Ferret comparisons between `callbacks.h` and `ui_callbacks.h`, `plugin_slot_callbacks.h`, and `control_callbacks.h`, demonstrating the compactness of the three colour approach against the pairwise comparisons used by many file comparison tools.

8.2.3 Scheme for colouring the text

The scheme for colouring the tokens in the base file is described next, with the aid of an example. As the target files are represented by primary colours, the maximum number of target files which can be displayed in one group is three. The example therefore has three target files.

Each trigram in the base file can appear in any of the eight combinations of the target files. The colours allocated to these combinations are listed in Table 8.1. Each token in the file is a member of three trigrams, except for the first and last two tokens in the file. Given that there are eight possible groups of files that each trigram can appear in, and that each token is in 3 trigrams, a token can be in one of 120 possible combinations of file groups. The formula, where n is the number of choices, i.e. file combinations, and r is the number of places, i.e. trigrams, is:

$$\binom{n+r-1}{r} = \frac{(n+r-1)!}{r!(n-1)!} = \frac{10!}{3!7!} = \frac{10 \times 9 \times 8}{3 \times 2} = 120.$$

There are 8 combinations where all of the trigram file memberships are the same, for example, [(1 3) (1 3) (1 3)]; 56 combinations where two are of one file membership and one of another, for example, [(2) (2) (1 2 3)]; and also 56 combinations where there are three different file membership groups, for example, [() (2) (1 2)]. The tokens are coloured as follows:

- If there is a majority file combination, use its colour.
For example, in the group [(1 2)(3)(1 2)], (1 2) is the majority combination, so colour purple, a mix of red and blue, the colours of files 1 and 2.
- Otherwise find the most common file in the combinations. If one file is in the majority, colour to match.
For example, [(1 2)(2 3)(1 2 3)] - file 2 is in the majority - colour red.

One file	Primary colour	Two files	Secondary mix	Other	Colour
(1) (2) (3)	blue red yellow	(1 2) (1 3) (2 3)	purple green orange	(1 2 3) ()	black pale cyan

Table 8.1: File combinations and the colours associated with them

- If two files share the majority, colour with the secondary colour which combines the two primaries representing the files.

For example, [(1 2)(3)(2 3)], the majority is shared by files 2 and 3, so colour orange, a mix of red and yellow.

- Otherwise, colour brown, the tertiary mix. (e.g. [(1 2 3)(2 3)(1)])

To give an idea of the number of tokens belonging to trigrams of the same or different types, a random sample of 10 files, each with three target files, taken from the split file dataset (see Chapter 12) were analysed. In these files, ranging from 700 to 12,000 tokens, 64% of the tokens belong to trigrams with the same file set, 32% have 2 sets, and only 4% have 3.

As an example, these rules are applied to the data in Table 8.2, which is based on a sequence of 15 tokens, labelled a–o. The trigrams are listed on the left, and the files in which they occur in the next column. The tokens are listed in the third column and the sets of files which the trigrams are part of are in column four. The first column of the ‘Token colouring’ section shows the majority file group, where there is one; the next column, the majority file(s) where there is no majority; and the last column is the colour for the token. The file groups are chosen to exemplify the different

Trigram	Files	Tkn	Sets of containing files	Token Colouring		
				Majority groups	Majority files	Colour
a b c	(0 2 3)	a	[(0 2 3)]	(0 2 3)	n/a	Orange
b c d	(0 1)	b	[(0 2 3) (0 1)]	n/a	1 2 3	Brown
c d e	(0 1 2)	c	[(0 2 3) (0 1) (0 1 2)]	n/a	1 2	Purple
d e f	(0 2 3)	d	[(0 1) (0 1 2) (0 2 3)]	n/a	1 2	Purple
e f g	(0 2)	e	[(0 1 2) (0 2 3) (0 2)]	n/a	2	Red
f g h	(0 2 3)	f	[(0 2 3) (0 2) (0 2 3)]	(0 2 3)	n/a	Orange
g h i	(0 3)	g	[(0 2) (0 2 3) (0 3)]	n/a	2 3	Orange
h i j	(0 1 3)	h	[(0 2 3) (0 3) (0 1 3)]	n/a	3	Yellow
i j k	(0 1 2)	i	[(0 3) (0 1 3) (0 1 2)]	n/a	1 3	Green
j k l	(0 1 2 3)	j	[(0 1 3) (0 1 2) (0 1 2 3)]	n/a	1	Blue
k l m	(0 1 2 3)	k	[(0 1 2) (0 1 2 3) (0 1 2 3)]	(0 1 2 3)	n/a	Black
l m n	(0)	l	[(0 1 2 3) (0 1 2 3) (0)]	(0 1 2 3)	n/a	Black
m n o	(0)	m	[(0 1 2 3) (0) (0)]	(0)	n/a	Pale
		n	[(0) (0)]	(0)	n/a	Pale
		o	[(0)]	(0)	n/a	Pale

Table 8.2: Example of the token colouring scheme described in Section 8.2.3. Trigrams, the files they are in, the tokens in the base file, and the set of files that the trigrams they belong to are in. Note that file 0 is the base file.



Figure 8.10: Colouring scheme applied to the tokens from Table 8.2

colouring patterns, see Figure 8.10; real examples are shown elsewhere, such as Figures 8.11 and 8.12. However, the figure shows that the sequence "a b c d e f" appears in file 2 as the letters are coloured red, purple, orange or brown, that "f g h i" is in file 3, coloured yellow, orange or green, and that "c d" and "i j" are in file 1, being blue, green or purple. The sequence "k l" is in all three files and "m n o" in none of the files.

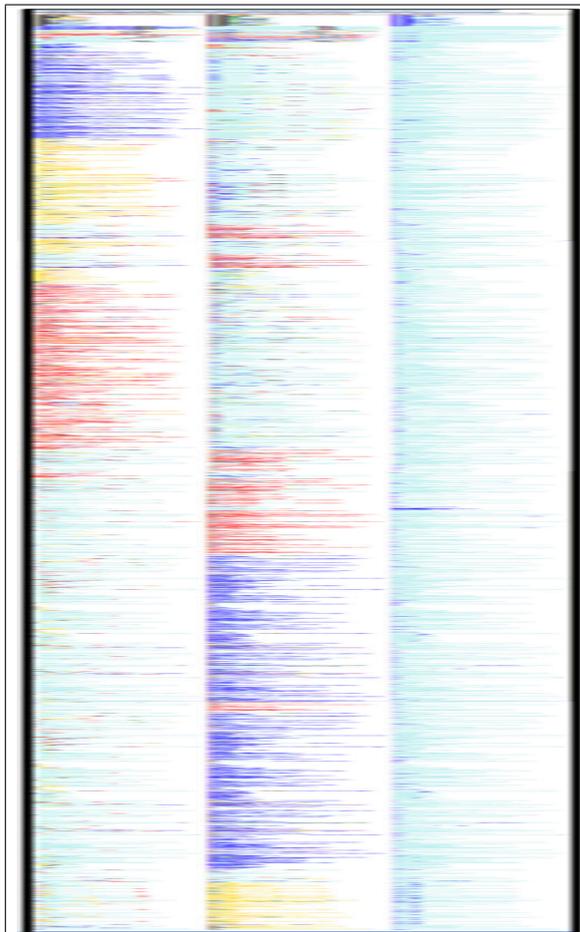


Figure 8.11: A large file split 7 ways, compressed to fit the page. This example is unusual in being a well-defined split with little incidental similarity.

8.2.4 Displaying comparisons between one file and many others

The colouring scheme is well-suited to groups of three target files, if more files and therefore more colours were added, there would be insufficient separation between colour mixes. However, this does not mean that the visualisation is limited to a comparison between the base file and three others. By repeating the base file and comparing it with each group of three target files, information about the similarities between any number of target files can be displayed.

Figures 8.11 and 8.12 show multiple comparison examples. The file in Figure 8.11, from the PostgreSQL project, is split seven ways, and therefore displayed 3 times. This file contains around 2,000 lines and is compressed to fit the page. Most of the code from the first half of the base file is in the files in the first column, with two small sections going to the “red” file in the second column. Most of the rest of the code moves to the second “red” and “blue” files, with some going to the second “yellow” file and one small function to the seventh file, shown in blue in the third column.

In Figure 8.12, the file `callbacks.h`, from the jack-rack project, has nine target files. `callbacks.h` appears to be split into three new files, `ui_callbacks.h`, `plugin_slot_callbacks.h` and `control_callbacks.h`, which are respectively the “blue”, “red” and “yellow” files in the first column. The top two lines are not in any of the files, but otherwise the code is in blocks in one or other of the three files. The other target files are incidentally similar.

This tool is named 3CO,¹ further examples of its use are available at <http://homepages.stca.herts.ac.uk/~gp2ag/trigram-analysis-examples.htm>.

8.3 Summary

In this chapter, methods for displaying different aspects of the interactions between the files in a group were described. In particular, a novel method for displaying a comparison between one file and a number of others in a convenient and reasonably compact manner, based on colouring text.

¹3CO is for COCOCO, which in turn stands for COmmon COde COlouring

Chapter 9

Trigram analysis applied to student assignments

Trigram-based measures of unusual similarity for use in collusion detection were introduced in Chapter 7, and Chapter 8 showed a way to highlight interesting parts of the code, such as unique trigrams, or those shared by just the two files, in a comparison between files in the context of a group.

The application of these techniques to a set of student projects is reported in this chapter. The group in the study were taking the 'Web Scripting and Content Creation' course, during which they were asked to develop a community website using ASP.NET and VB.NET. Unique trigram counts were used in generating reports for feedback. At the end of the course, the highest ranked file pairs identified by each of the set of collusion measures were compared.

The students were expected to commit their code to an online repository regularly, and staff had access to the code for analysis. Use of a repository meant that staff could both review progress and provide feedback to students. The twin aims were to encourage a steady development pattern and to discourage contract cheating. Trigram analysis was used in the feedback, to measure originality, and in the monitoring, to measure similarity.

9.1 Data

Sixty-two students were initially registered on the course. Their projects were developed using Visual Studio^(TM), which produces a large amount of auto-generated code. The files were filtered, so that only those expected to contain the student's own code were selected for analysis. These were .aspx, .css and .master files, with scripts from .aspx files, and classes from .vb files. Although filtering is not necessary for the trigram analysis, except to exclude image files, other analysis of the code made it necessary here.¹

9.2 Method

Each student's code was concatenated into one large file to simplify comparisons between their work, as shown on the left-hand side of Figure 9.1. This concatenation technique is used by others in plagiarism detection and

¹Information in this report is anonymised and has UH Ethics Approval 1011-129

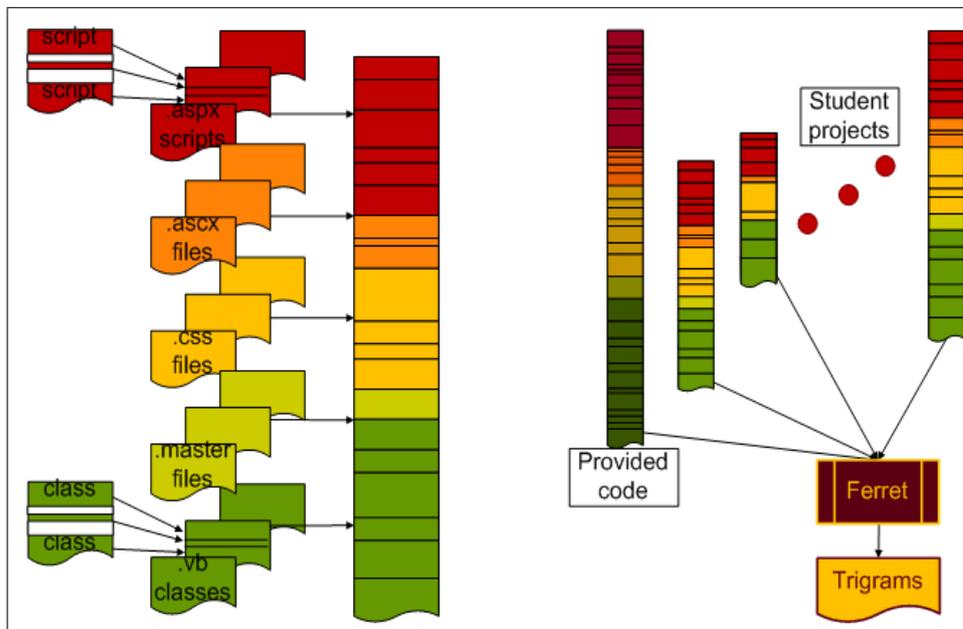


Figure 9.1: The selected files are concatenated to form one file for each student (shown on the left). Provided code is also placed in one file. This file and those of student code are presented to Ferret to obtain information about the distribution of trigrams in the files.

in clone detection. For example, Lancaster and Tetlow [144], and Kamiya et al [125], who also add file delimiters. The concatenation introduces new trigrams at the file margins and these vary depending on file order. However, this is a small price for making the rest of the process more straightforward.

Ferret currently has a tokeniser for C-type languages, but in this work is used for other languages and appears to be effective with the varied code. A C-type tokeniser will give a slightly different token set to a language-specific tokeniser. For example, “not equals” in Visual Basic, “<>”, is treated as two tokens, “<” and “>”, instead of one.

The code provided for the course, in the form of examples or exercises, was also concatenated into one file. This file, and all of the student files were compared by Ferret to provide a trigram-file index and similarity scores.

Feedback The students’ projects were downloaded weekly for analysis throughout the course. The primary purpose was to provide feedback to staff and students, showing progress in terms of the amount of code produced, and its individuality. An anonymised report was posted every week showing the “top 30” students in each of these respects. The amount of code was measured in non-blank lines. Individuality was measured by counting the unique trigrams in a project. An excerpt of a weekly report is shown in Figure 9.2. Each student was allocated a flag, known only to them and to staff. Twenty flags are shown for the student with the largest number of unique trigrams, the number of flags for the remaining students is proportional to this baseline figure.²

Collusion detection At the end of the course, the files of student code and of provided code were compared to check for inappropriate collusion. The four proportional similarity measures were applied to the projects. These measures are the standard Ferret similarity score, those with trigrams in the provided code, in common code, or in both, excluded. Counts of shared

²This work was undertaken with Steve Bennett, the course tutor, who set up the repositories, and whose ideas included monitoring development patterns and providing anonymous feedback. The implementation of these ideas and use of trigram analysis were mine.

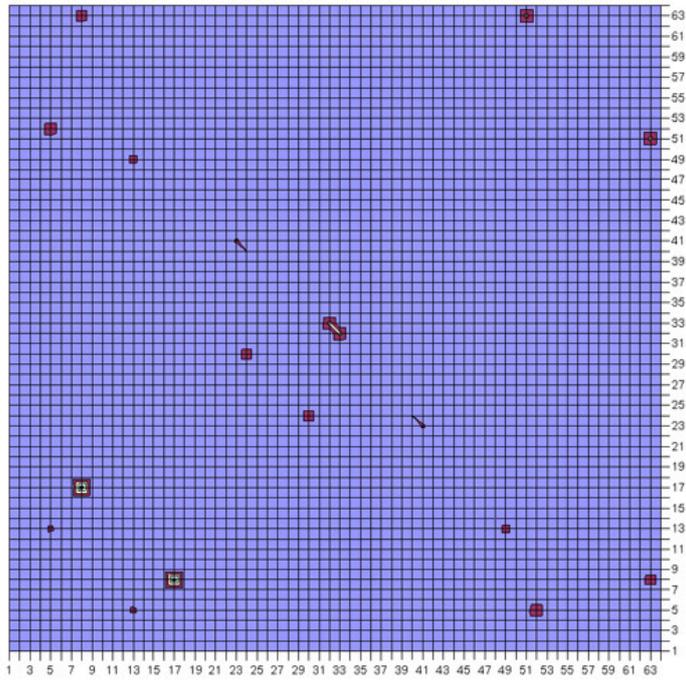
9.3.1 Proportional similarity measures

Table 9.1 shows the pairs of files with the 20 highest scores under each measure. In the first column, file pairs are listed. In the next 4 columns, there is a tick under each measure where the similarity between the pair is ranked 1–20. As expected, the measures which exclude common code, in columns 4 and 5, mostly agree. However, only four of the fifty-five selected pairings score high under all four proportional measures. These pairings are (3, 15), (3, 25), (24, 40) and (57, 60). The last two columns in the table relate to count-based measures, which are discussed in the next section.³

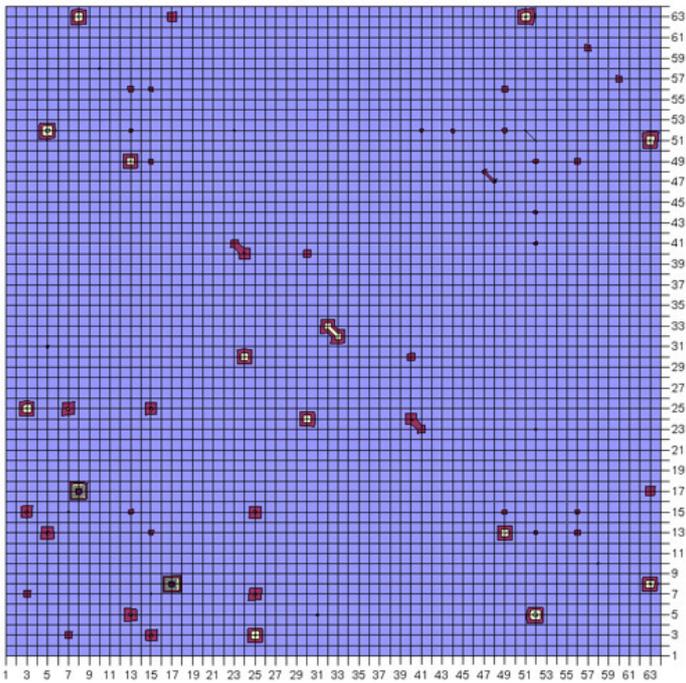
³Table G.1 in the Appendix shows the numbered rankings.

Proj-	Ratios				Counts		Proj- ects	Ratios				Counts	
	Fer ret	Ex. P	Ex. O	Ex. PO	Wt. 2	Wt. 4		Fer ret	Ex. P	Ex. O	Ex. PO	Wt. 2	Wt. 4
3, 7		✓				✓	19, 27			✓	✓		
3, 15	✓	✓	✓	✓		✓	19, 36			✓			
3, 20		✓	✓	✓			19, 43			✓	✓		
3, 25	✓	✓	✓	✓	✓	✓	19, 46	✓					
3, 56		✓					19, 54	✓					
4, 44					✓		21, 28			✓	✓		
5, 13					✓	✓	22, 61	✓					
5, 52					✓	✓	23, 41			✓	✓	✓	✓
7, 15		✓					24, 30		✓	✓	✓	✓	✓
7, 25		✓	✓	✓		✓	24, 40	✓	✓	✓	✓	✓	✓
8, 17		✓	✓	✓	✓	✓	26, 52					✓	✓
8, 23					✓	✓	27, 28	✓	✓	✓			
8, 25					✓	✓	27, 53	✓					
8, 63				✓	✓	✓	27, 61	✓					
10, 29		✓					29, 42	✓	✓				
10, 42	✓	✓					29, 58		✓				
10, 58		✓	✓	✓			30, 40	✓		✓	✓		✓
13, 49					✓	✓	32, 33			✓		✓	✓
13, 52					✓		34, 53						
13, 56		✓					36, 43	✓					
14, 22							41, 58			✓			
14, 53	✓						42, 58		✓		✓		
14, 61	✓						43, 46			✓	✓		
15, 25	✓	✓				✓	47, 48			✓	✓	✓	✓
15, 56	✓	✓					49, 52					✓	✓
17, 34				✓			51, 63				✓	✓	✓
17, 63					✓	✓	53, 61	✓				✓	✓
							57, 60	✓	✓	✓	✓	✓	✓

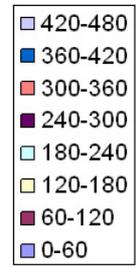
Table 9.1: Top 20 similarities by various measures. Proportional measures are standard Ferret, and those excluding: provided code (P), shared by others (O), and both (PO). Count-based measures are uniquely shared trigrams (Wt.2) and the weighted count for groups of up to 4 members (Wt.4). Pairs of projects with the top 20 scores under one or more measures are in columns 1&8, the remaining columns show for which measure(s).



(a) Max in group = 2



(b) Max in group = 4



(c) Scale

Figure 9.3: Weighted trigram counts: the contour maps depict weighted trigram counts of groups of 2, and up to 4, files sharing the trigrams.

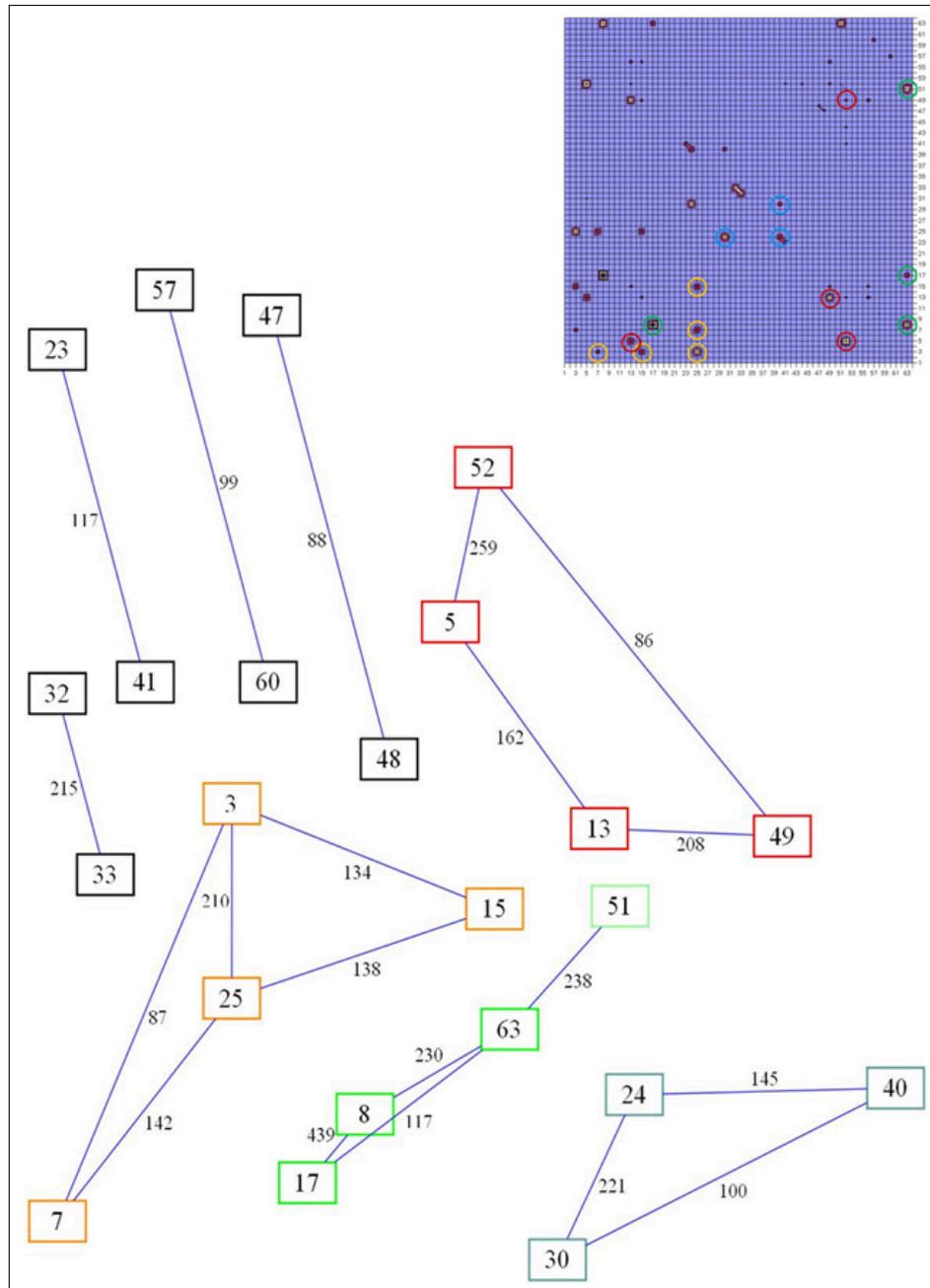


Figure 9.4: Connections between files with a weighted trigram count of at least 85, calculated for a maximum of 4 files. The connections are inversely proportional to the weighted trigram count. The graph in the top right corner of the figure repeats Figure 9.3b; and has the similarities between members of the groups marked by circles which echo the node colours.

9.3.2 Count-based similarity measures

The weighted count of trigrams shared by two files and few others varies depending on the maximum number of other files accounted for. The maximum group sizes for the graphs in Figure 9.3 are two (9.3a) and four (9.3b). These graphs are contour maps of the weighted similarity counts between pairs of project files. The student identifiers are shown on both axes and run from 1-64.⁴ Where the similarity between two files is higher than the “background” level indicated by the scale, the graph shows the level of similarity as a coloured spot. The size and colour of the spot reflects the magnitude of the similarity measure. When the group size is two, the count is of uniquely shared trigrams. When there are four in the group, then trigrams shared by the pair and by up to two others are included in the count. For this set of projects, the similarity rankings were unchanged in groups larger than four, and are therefore not reported.

Graph 9.3a shows that few pairs of students uniquely share more than 60 trigrams. Looking at the second graph, 9.3b, groups emerge, such as the four students 3, 7, 15 and 25, or the group of three, 8, 17 and 63. Most of the larger points in Figure 9.3b are also present in Figure 9.3a. However, the notable exceptions are the group 3, 7, 15 and 25, indicating that these students are likely to have worked together as a group.

To better show the connections, Figure 9.4 depicts the group similarities as an undirected graph, with the projects as the nodes, and the weighted trigram count between them as the edge weights.⁵ Connections of less than an arbitrary weight of 85 are removed, as are the resultant unconnected nodes. This leaves four pairs of files and four groups. The group formed by 24, 30 and 40 is fully connected, the others partially so. In this diagram, the distance between the nodes is inversely proportional to the weight of the connection. The graph in the top right corner of the figure repeats Figure 9.3b, with the similarities between members of the groups circled.

In Figure 9.5, weighted counts of 50 or more are shown, however, for

⁴2 students had extra, unused, accounts (11&39) retained to preserve the numbering.

⁵Graph produced with the aid of Graphviz <http://www.graphviz.org/>

to common code which is not part of the provided code. The similar code in the five pairs ranked in the top 20 by this measure but not by other measures also occurs in other files. This is likely to be because examples given in class were included in many of the students' code, but were not part of the provided code. These student's files were among the smaller ones, so that this code had a greater impact on their similarity scores.

Code shared by others excluded, with and without provided code The column headed "Ex.O" in Table 9.1 shows similarities calculated by excluding code shared by others, and the column headed "Ex.PO" that excluding both that shared by others and by the provided code. As already mentioned, these measures are almost the same because provided code is likely to be used by many students, therefore excluding code shared by others excludes all or most of the provided code. The higher ranked similarities which are not picked by other measures all have project 19 as one half of the pair. These pairs are also more similar than most because they are small files containing code based on the provided code, with parts amended in similar ways, which can happen when two students sit together in class and receive help from their tutor. The other four pairs, 3 and 20, 17 and 34, 21 and 28 and 43 and 46 are also not large files. They either share incidentally similar fragments, such as the same font or colour, or have not renamed automatically named elements, such as "TextBox4.Text". The fragments add up to make the proportional similarity higher than that in large files which share more significant amounts of code.

Weighted trigram count, group size 2 The uniquely shared trigram count, in Figure 9.3a, shows only those pairs sharing at least 60 such trigrams. The main pair here is 8 and 17. An extract of a comparison between these two projects is shown in of Figure 9.8 (p144). The code shown belongs to project 8, in the middle column code not shared by the two projects is black; if shared by the two projects, but also by others, it is coloured blue; and if shared by just the two projects, it is red. There are two sizable functions coloured red, in this case the two students were trying an idea suggested in

class, and had found an unusual solution to the problem.

Five pairs, 5 and 13, 5 and 52, 8 and 63, 13 and 49, and 51 and 63 show up under this measure and not the previous ones. The students involved in these pairings all did very well on the course, and the similarity between them is mostly due to their finding similar solutions to difficult problems. For example, students 8 and 63 have found the same solutions to problems related to handling different image types, and to a problem with coding for a Mac; whereas the trigrams shared by 51 and 63 appear to be because they have both explored the more advanced ideas introduced in class. Third-party code, such as that for dealing with differences in browser display, is properly attributed in these students' files.

Weighted trigram count, group size 4 The similarities found by the weighted trigram count for a group of up to four include all of those agreed on by the four proportional measures, as well as pairs not found by any of the other measures.

Four groups who may possibly have worked together on some of their code are discovered by the weighted trigram count. These are numbers 3, 7, 15 and 25; 24, 30 and 40; 8, 17 and 63, with 51; and 5, 13, 49 and 56.

The first group, who appear to have incorporated each other's code, had low final scores. The second group has used similar code to handle discussion groups, user profiling and the display of ratings. The code they share may be the result of admissible cooperation, or of independently finding the same solution to these problems.

The similarity between students 8 and 17, 8 and 63, and 51 and 63, members of the third group, has already been discussed. Features shared by 17 and 63 have some overlap with project 8, but mostly appear to be the result of cooperation on handling rss feeds.

The code shared by the fourth group is more fragmented and gives no reason for concern, as it is not likely to be the result of copying. The students in the second, third and fourth groups produced good projects.

9.3.4 Showing similarity in a group context

Methods for visualising the interaction between files in the context of a group were described in Section 8.1.3. Examples from the study are shown in this section.

Graded colouring The scheme for graded colouring, introduced in Section 8.1.3, is repeated in Figure 9.7 for reference. Figure 9.8 (p.144) is an extract from project 8 compared with project 17, coloured using three different schemes. The left-hand column shows the code coloured by Ferret in normal use. The code shared by the two files is coloured blue, otherwise it is black. This does not provide information about the nature of the similarity between the projects.

By analysing the trigrams, more information can be shown. In the middle column, the trigrams uniquely shared by the two projects are coloured red. On the right, the idea is developed to provide additional information.

The code is coloured black only if it is unique within the group. Red code is, again, that uniquely shared with the other project. Shades of orange show those trigrams which are shared by the two, but which are also shared by few others. Here, dark orange is used to show one or two others sharing, and pale orange for three, four or five others sharing. Bright blue means that the code is shared by the two projects but by at least six others, and pale blue means that either the code is not shared or is provided for the course, and is therefore uninteresting whether it is shared by 8 and 17 or not.

When comparing two files, shared trigrams are differentiated as follows:

1. "Code" which is unique to this file (A) is coloured black.
2. If "uniquely shared" by file A and the file to which it is compared (B), then it is red.
3. When the two files share code with any one or two other files, it is in dark orange.
4. If A, B, and 3, 4 or 5 other files contain the code, it is in orange.
5. Code shared by A, B, and 6 or more other files is in blue.
6. Uninteresting code, either that provided for the course, or code not shared by the two files, is pale blue.

Figure 9.7: One way to colour the code in a file (A) compared with another (B). The colours and gradations can be altered to suit the user's needs. First shown in Figure 8.3, and repeated here for the reader's convenience.



Figure 9.8: An extract from a comparison between projects 8 and 17. Standard Ferret colouring, where shared trigrams are in blue, and are otherwise black, is on the left. In the middle the trigrams uniquely shared by the two projects are in red. On the right, the excerpt is coloured according to the scheme in Figure 9.7. The red code is two sizable functions uniquely shared by the projects, and the orange shows smaller sections which are shared by the two files but also by a few others in the group.

3CO colouring The 3CO colouring scheme was introduced as a way to depict the source or destination of files which have moved during system restructuring. As already shown in Figure 9.6, the scheme can also be used to display the interaction between a file and others which are found to be similar within the group.

Extracts from the five-way comparison between projects 63, 8, 17, 51 and the provided code are illustrated in Figure 9.9. The base file here is project 63. The colours in the first column are blue for 8, red for 17 and yellow for 51. Text in the second column is coloured blue where trigrams match the provided code. By looking at areas which are coloured in the first column and not in the second column, parts of the code which are shared by one or more of the projects and which are not part of the provided code can be identified. Here these are the blue and yellow sections in the first column. As shown in Chapter 8, this visualisation scales for any number of projects.

9.4 Discussion

Each of the four proportional similarity measures has drawbacks for the style of project investigated here. These drawbacks are likely to apply to other cases where file sizes differ, where a significant amount of code is provided or where code is auto-generated.

Counting trigrams which occur in one pair of assignments, but not in others, appears to offer a reasonable measure of pairwise cooperation, however, counting those which are shared by 2 students and up to 2 others finds several groups not found by the other measures.

It should be noted that these students were aware that their code was monitored and were given feedback on the “most original” projects on a weekly basis. This probably had two effects, first, of encouraging individual development and second, of reducing the amount of collaboration between students. Most of the similarity between projects in this course appears to be due to cooperation in solving difficult tasks, rather than in inappropriate collusion. Incidentally, the feedback generated greater interest in the course

than previously experienced.

The graduated visualisation pinpoints the areas of similar files which warrant investigation, especially helpful for large files with high levels of incidental similarity. Parts of the file which are unique to one student are also highlighted, useful in understanding the novelty of the work, or perhaps where outside sources have been used. No other approach has been found which offers the detail displayed by this source code visualisation. Although the display is based on trigram analysis, other units of comparison, such as clones, could be used as a basis for colouring the code.

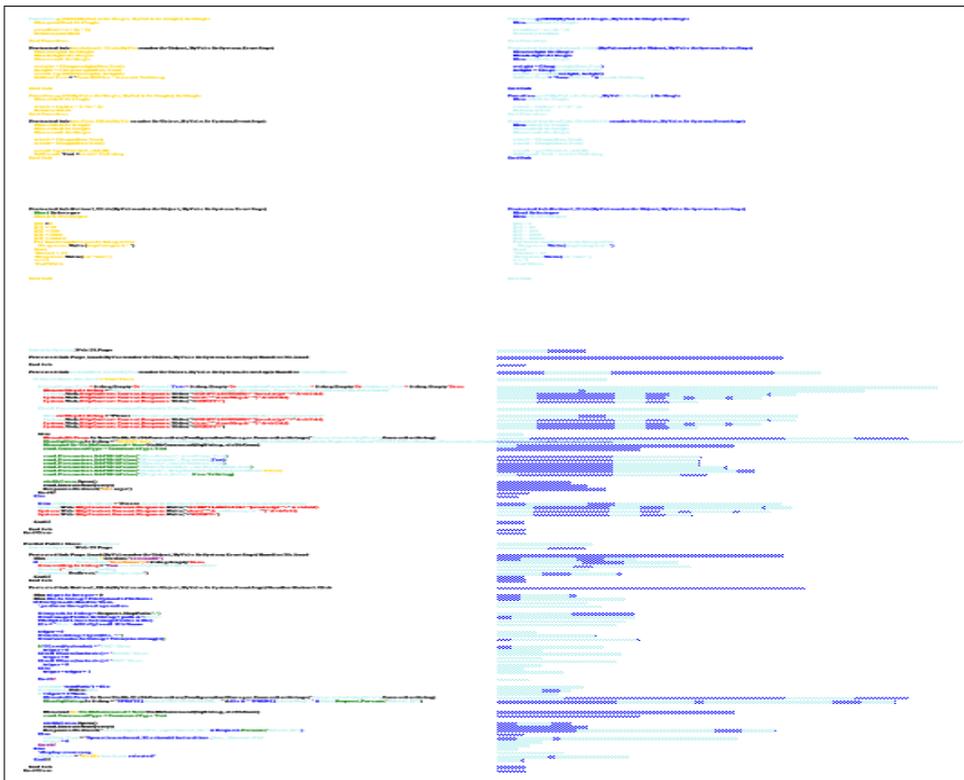


Figure 9.9: An extract from a 5-way comparison between projects 63, 8, 17, 51 and the provided code. The code, from project 63, is coloured blue where shared with project 8, red with 17, yellow with 51, and blue in column 2 for the provided code. Green code is that shared with 8 and 51, and black with 8, 17 and 51. The blue and yellow sections in column 1 do not correspond to the provided code, so are areas for further investigation.

Chapter 10

Overview of the classification system

This part of the dissertation is about the second, and main, practical application of the ideas introduced in Chapter 1. This is an investigation into the use of text analysis and machine learning techniques in analysing software projects. In particular, to find and match files which have been restructured between releases of a project. This chapter outlines the system for finding candidate restructured files, and for classifying these candidates.

Many descriptions of the machine learning, or knowledge discovery, process exist. The steps described by different authors vary, however, the underlying process is similar and usually involves iteration at one or more points (see, for example, Bramer [27], Fayyad et al. [72], Mannila [161], and Pyle [191]). These steps are outlined below, followed by an explanation of the organisation of the chapters relating to the machine learning system.

The steps in the system developed for this research are depicted in Figure 10.1, and are broadly:

- data collection and preprocessing,
- filtering,
- feature construction and data labelling,
- feature selection,
- and model production and selection.

Data collection In spite of the widespread availability of open source software, there is a lack of marked-up datasets in the field of software evolution which are suitable for use directly in data-mining studies [49, 127]. For this research, datasets were created from source code collected from an open source software repository. A range of projects was selected so that a variety of development styles is represented in the datasets.

Preprocessing In preprocessing, the files under investigation, in this case mostly C code files, are selected from each project. Comments and blank lines are removed from the selected files because movement of code is the focus when trying to find restructured files.

Filtering The aim of filtering in this system is twofold: first, to select from the large number of files in the projects the comparatively small number of files which could belong to the category of interest; and second, to find files which are related to them. As many software projects are large, the filtering process needs to be efficient. The similarity measures used for filtering in this system are produced by Ferret, as it runs in approximately linear time. Each file in a release is compared to every other file in that release and to all of the files in the next release. Similarity scores and file sizes are stored so that files of different types can be selected based on this information.

Manual classification To label filtered files, a file and those related to it are inspected, initially with the aid of the 3CO tool (see Chapter 8), and when classification is unclear from this, by inspecting the text of the files.

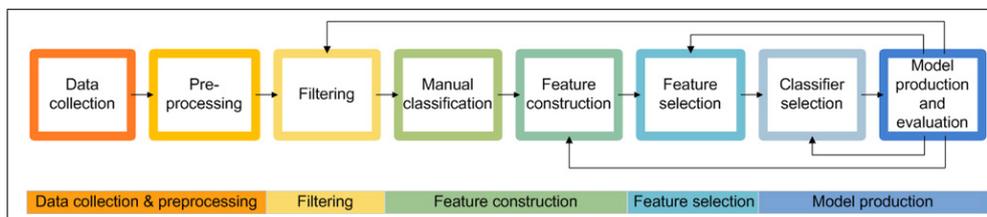


Figure 10.1: Outline of the learning process, indicating steps which can be iterative.

Feature construction The development of machine-learning models from labelled examples assumes the existence of an adequate set of descriptive features or measurements of the data, from which distinctions between classes may be derived. As discussed in Chapter 3, there is no previous work on machine learning in origin analysis, and little work comparing the utility of different similarity measures in matching restructured entities. There is therefore little guidance on suitable features for the task. One observation from the origin analysis survey (see page 46) is that the majority of approaches combine similarity measures from two or more sources when comparing code.

The features in this system are based on the similarity between two or more files in a group, measured using four similarity detection tools with complementary methods. Features are constructed directly from the tools' outputs, and by analysing these outputs to give more detailed information.

One of the recommendations for building features when there are no restrictions, and no obvious choices, is to create a large set which can then be reduced. This is the approach taken here, but rather than selection from the whole set, selection is based on limiting the sources of the features, to simplify the generation of features for new data.

Refining the model In refining the model, the filtering, feature construction, feature selection, model selection and production phases are iterated. The aim is to maximise the correct classification of the training examples,

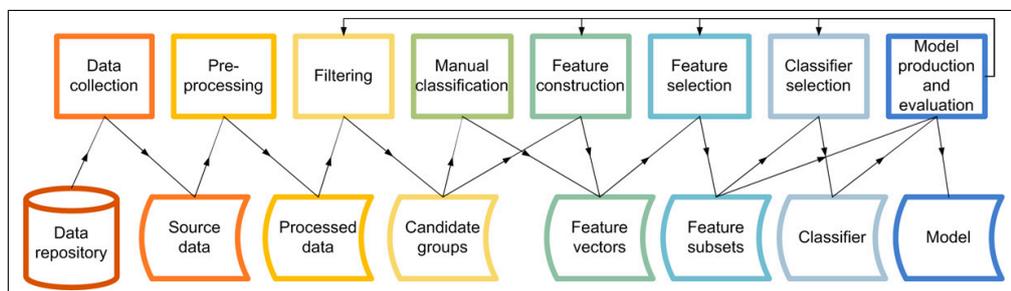


Figure 10.2: An overview of the system used to build a model for classifying restructured files. The processes are on the top row, and the inputs to and outputs from each step are on the bottom row.

while maintaining the ability of the model to generalise. The top row of Figure 10.2 repeats the processes of Figure 10.1, while the bottom row shows the outputs of each step, which act as input to subsequent steps.

Classifying new data When classifying data from new projects, see Figure 10.3, data collection, preparation and filtering are the same. Feature construction differs in that only the subset of features required for input to the particular model needs to be generated. The final step is classification of the filtered files by using these features as input to the learned model.

Organisation of this part of the dissertation There are six further chapters in this part of the dissertation. As already explained, Ferret is the main file comparison tool used in this application, the other comparison tools are described in Chapter 11. The remaining chapters have two purposes, to explain how the datasets are created, and to report experimental results.

Data collection, preprocessing and filtering are described in Chapter 12, and feature construction is explained in Chapter 13. Three chapters describe the experimental part of this research. The classification system is tested and the results compared with the work of two other research groups in Chapter 14. Some weakness in filtering is identified from this study. Experiments aimed at improving the filtering are described in Chapter 15. The system is retested using datasets resulting from the adjusted filtering techniques and the results are reported in Chapter 16.

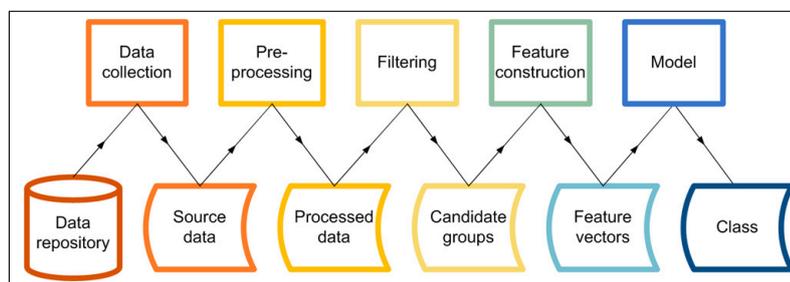


Figure 10.3: Overview of the system for classifying new data. Processes are on the top row, and inputs to and outputs from each step on the bottom row.

Chapter 11

File Comparison Tools

As explained in Chapter 10, features for the machine learning system are created by comparing files with a set of four complementary similarity detection tools. One of these tools is Ferret, already described in Chapter 6. The purpose of this chapter is to introduce the other three tools.

Each tool has strengths and weaknesses in detecting code which has moved between files. By combining tools, gaps in one tool are plugged by another. For example, a tool which matches blocks of identical lines of code will overcome the problem of matching incidentally similar snippets of code, but will not match code with small edits, such as renamed identifiers; and a tool which parameterises identifiers will match such code, but may also match similar structures in the code which are not from the same source.

As noted in Chapter 6, Ferret is very efficient and thus suitable for filtering to find candidate files from large datasets. Its strength is in matching small sections of code, making it moderately robust to many forms of editing. However, incidental matches, due to common constructs in the language or to programming style, are also detected. Repetition of trigrams is not accounted for, therefore quantifying the 'amount' of copying is difficult.

As discussed in Chapter 2, clone detection tools provide complementary information to that given by plagiarism detection tools such as Ferret. The stand-alone clone detection tools used in this research were chosen from the University of Alabama Clone Detection website [182], which lists both

a range of clone-related literature and a variety of clone-detection tools.

Duplo detects blocks of strictly matched lines of code and is useful for finding identical sections of code in the files, information not directly available from Ferret. However, this means that small edits, such as parameter renaming, will prevent the detection of otherwise matched code.

Code Clone Finder (CCFinder), is a mature and widely used token-based clone-detection tool.¹ It parameterises identifiers, so that they can be matched without having the same name, either on an any-to-any or a one-to-one basis. While this overcomes the drawbacks of the matching process used by Ferret and Duplo, CCFinder has its own disadvantages when used for tracking restructured files. These disadvantages are twofold: first, CCFinder disregards code such as preprocessing directives and initialisations, making the tool less informative for the typical header file; and second, because of the parameterisation, similar code structures, such as idiomatic code, may render the matching too general.

Simian falls between Duplo and CCFinder in that it matches lines of code which are first tokenised, and can be parameterised using a range of options. This tool was chosen to reduce the amount of code which is ignored by CCFinder, while having choices in the sensitivity of the matching.

Three tools were developed during this research to alter or analyse the output from the third-party tools. These three tools helped in constructing features for machine learning. The first, P-Duplo, uses a similar method to Duplo but produces output in a form more easily analysed for this application. The second, a clone ‘unscrambler’, analyses the output from clone detection tools to provide information about duplication on a one-to-one basis, more useful for this application than the all-to-all matches output by the tools. The third, explained in Chapter 6, is the density analysis tool, which acts on Ferret’s XML output to discover blocks of mostly copied code, which are possibly the result of copying and editing.

CCFinder, Simian and Duplo are described in Sections 11.1–11.3. P-Duplo and the Unscrambling tool are explained in Sections 11.4, and 11.5.

¹708 citations logged by Google Scholar at 20.1.13, & used in origin analysis [22, 130, 245]

The example file, and the two files resulting from its split, first shown in Chapter 6, are repeated in Figures 11.1 and 11.2, as they are used to illustrate features of the other three file comparison tools. On the left of Figure 11.1, line numbers, or the number of the first token in the line, have been added for reference.

CC	SM	DP	PD	
		1		// cnp-1.c
	1	2	1	#include <stdio.h>
	2			
	3		2	long factorial (int n);
	4	3	3	long combinations (int n, int k;
	5	4	4	long permutations (int n, int k);
	6			
0	7	5	5	int main()
6	8		6	{
7	9	6	7	int setsize, subsetsize;
12	10	7	8	printf ("Set size? ");
18	11	8	9	scanf ("%d", &setsize);
27	12	9	10	printf ("Subset size? ");
33	13	10	11	scanf ("%d", &subsetsize);
42	14	11	12	if (setsize < 0 subsetsize < 0 setsize < subsetsize)
57	15		13	{
58	16	12	14	printf ("Mission impossible\n");
64	17	13	15	return (1);
67	18		16	}
68	19	14	17	printf ("%ld combinations and %ld permutations of %d items taken from %d \n", combinations (setsize, subsetsize), permutations (setsize, subsetsize), subsetsize, setsize);
85	20	15	18	return (0);
98	21		19	}
	23			
99	24	28	20	long factorial (int n)
107	25		21	{
108	26	29	22	long result = 1;
113	27	30	23	int i;
116	28	31	24	for (i = 1; i ≤ n; i++)
130	29	32	25	result *= i;
135	30	33	26	return (result);
139	31		27	}
	32			
141	33	32	28	long combinations (int n, int k)
152	34		29	{
153	35	33	30	return (factorial(n) / (factorial(k) * factorial(n-k)));
166	36		31	}
	37			
168	38	34	32	long permutations (int n, int k)
179	39		33	{
180	40	35	34	return (factorial(n) / factorial(n-k));
196	41		35	}

Figure 11.1: Code for finding combinations and permutations of a subset of items - cnp-1.c. The first 4 columns show the number of the first CCFinder token in each line, and the Simian, Duplo and P-Duplo line numbers. Line 19 (14 or 17) is spread over 3 lines to fit the page.

```
// cnp-2.c
#include <stdio.h>
#include "fact.c"

long combinations (int n, int k);
long permutations (int n, int k);

int main()
{
    int setsize, subsetsize;
    printf ("Set size? ");
    scanf ("%d", &setsize);
    printf ("Subset size? ");
    scanf ("%d", &subsetsize);
    if (setsize < 0 || subsetsize < 0 || setsize < subsetsize)
    {
        printf ("Mission impossible\n");
        return (1);
    }
    printf ("%ld combinations and %ld permutations of %d items
            taken from %d\n", combinations (setsize, subsetsize),
            permutations (setsize, subsetsize), subsetsize, setsize);
    return (0);
}

long combinations (int n, int k)
{
    return (factorial(n) / (factorial(k) * factorial(n-k)));
}

long permutations (int n, int k)
{
    return (factorial(n) / factorial(n-k));
}

// fact.c

long factorial (int n)
{
    long result = 1;
    int i;
    for (i = 1; i ≤ n; i++)
        result *= i;
    return (result);
}
```

Figure 11.2: The amended code for finding combinations and permutations of a subset of items, split into 2 files - cnp-2.c (top) and fact.c (below).

11.1 Code Clone Finder (CCFinder)

CCFinder [124, 125] is a token-based code clone detector. As one of the aims of the tool is to recommend practical code abstractions, a number of reductions and transformations are performed on the code before matching. The reductions include filtering out parts of the file which are considered to be uninteresting, such as comments; or parts which may produce false clones because of their repetitious structure, such as declarations, preprocessing directives and initialisations. The code is transformed using language-specific rules to produce a sequence of tokens consisting of keywords, identifiers, literals and special characters. CCFinder offers flexibility in the size of clone detected and in the type of matching performed.

Clone size is determined by specifying the minimum number of tokens in a sequence and the minimum number of token types. The tokens and token types for the file `fact.c` are shown in Table 11.2. Token types include identifiers (such as `i` - shown as `id-i`), typed literals (`1` - `l-int-1`), one for each operator (`≤` - `op-le`), and for each delimiter (`;` - `suffix:semicolon`).

When a file is moved or renamed, identifiers may be given new names or types. One reason for using CCFinder is that it offers a choice in matching types, constants, and identifiers. To illustrate, consider the code snippets in Table 11.1. Snippet 2 is the same as snippet 1, except that the identifiers are directly replaced: `d` for `a`, and `e`, `f`, `u`, `v`, `w` and `7` for `b`, `c`, `x`, `y`, `z` and `10`, respectively. Snippet 3 has the same operators and identifiers as snippet 1, but the assignments are different; where `a` is used in snippet 1, either a or

1. First example	2. Direct replacement of identifiers	3. Different assignments, same operators	4. Different operators
<pre>int a, b, c; int x = a + b; int y = a - c; int z = 10 + c - b;</pre>	<pre>int d, e, f; int u = d + e; int v = d - f; int w = 7 + f - e;</pre>	<pre>int a, b, c; int x = c + b; int y = a - b; int z = 4 + a - c;</pre>	<pre>int a, b, c; int x = a - b; int y = a * c; int z = 10 - c * b;</pre>

Table 11.1: Code snippets to illustrate one-to-one or p-matching, and any-to-any matching. Snippets 1 and 2 match under p-matching, snippets 1, 2 and 3 with any-to-any matching. Snippet 4 does not match in either case.

b are used in snippet 2; likewise b and c replace b; and b and a replace c. Snippet 4 has the same identifiers as snippet 1, but with different operators.

One-to-one parameter matching, or p-matching [13], only matches snippet 1 and snippet 2. With more relaxed any-to-any identifier matching, snippets 1, 2 and 3 are matched. Snippet 4 does not match because operators are not parameterised. P-matching is used in this application.

Command-line outputs include a list of clone pairs which detail the two files in which the clone occurs, the start and end token numbers in each file, and the number of tokens in the clone. Figure 11.3 shows the output from

1.1.0	0	(def_block	
1.1.0	4	r_int	
1.6.5	9	id_factorial	
1.10.f	0	c_func	
1.10.f	1	(paren	
1.11.10	3	r_int	
1.15.14	1	id_n	
1.16.15	1)paren	long factorial (int n)
2.1.18	1	(brace	{
3.3.1d	4	r_int	
3.a.24	6	id_result	
3.11.2b	1	op_assign	
3.13.2d	1	l_int_1	
3.14.2e	1	suffix:semicolon	long result = 1;
4.3.33	3	r_int	
4.8.38	1	id_i	int i;
4.9.39	1	suffix:semicolon	
5.3.3e	3	r_for	
5.7.42	0	c_loop	
5.7.42	1	(paren	
5.8.43	1	id_i	
5.a.45	1	op_assign	
5.c.47	1	l_int_1	
5.d.48	1	suffix:semicolon	
5.f.4a	1	id_i	
5.11.4c	2	op_le	
5.14.4f	1	id_n	
5.15.50	1	suffix:semicolon	
5.17.52	1	id_i	
5.18.53	2	op_increment	
5.1a.55	1)paren	for (i = 1; i ≤ n; i++)
6.5.5c	0	(brace	
6.5.5c	6	id_result	
6.c.63	2	op_mul_assign	
6.f.66	1	id_i	
6.10.67	1	suffix:semicolon	result *= i;
7.3.6c	0)brace	
7.3.6c	6	r_return	
7.b.74	6	id_result	
7.12.7b	1	suffix:semicolon	return (result);
8.1.7e	1)brace	
8.2.7f	0)def_block	}
a.1.83	0	eof	

Table 11.2: CCFinder token types for fact.c. Columns 1–3 show the token’s source code location, its type identifier, and its CCFinder representation. Line breaks are shown, with the program text in column 4.

a comparison between the three example files, with the parameters:

- minimum tokens in a clone, 10,
- minimum token types in a clone, 2, and
- matching, any-to-any.

Three large clones, marked by asterisks, contain the other smaller clones. Two are between the files `cnp-1.c` (file 1) and `cnp-2.c` (file 2), tokens 0–99 in both files, and tokens 141–207 in file 1, and 99–165 in file 2. The remaining clone is between `cnp-1.c` and `fact.c` (file 3), tokens 99–141 in file 1, and 0–42 in file 3. The first column in Figure 11.1, giving CCFinder token numbers, shows that these are the expected spans for the blocks of matched code, for example, the factorial function is covered by tokens 99–140 in file 1.

```

version: ccfx 10.2.7
format: pair_diploid
option: -b 10
option: -s 2
option: -u -
option: -t 2
option: -w f-g+w-
.....
option: -preprocessed_file_postfix.cpp.2_0_0_2.default.ccfxprep
preprocess_script: cpp
source_files {
1      cnp-1.c   208
2      cnp-2.c   166
3      fact.c    43
}
source_file_remarks { }
clone_pairs {
2      1.0-99      2.0-99      *
10     1.12-27     2.27-42
10     1.27-42     2.12-27
5      1.72-82     2.80-90
5      1.80-90     2.72-82
27     1.141-207   2.99-165   *
26     1.142-152   2.137-147
26     1.179-189   2.100-110
1      1.99-141    3.0-42     *

2      2.0-99      1.0-99
10     2.12-27     1.27-42
10     2.27-42     1.12-27
5      2.72-82     1.80-90
5      2.80-90     1.72-82
27     2.99-165   1.141-207
26     2.100-110  1.179-189
26     2.137-147  1.142-152

1      3.0-42      1.99-141
}
clone_set_remarks { }

```

Figure 11.3: CCFinder clone report for a comparison between the files `cnp-1.c`, `cnp-2.c` and `fact.c`. Parameters and filenames are at the top. Clone pairs are in the lower section, once for each file. Asterisks have been added to show the clones which are not subsumed by another.

11.2 Simian

Simian [105] has a variety of parameterisation options and although, like CCFinder, it disregards some parts of the code, it is less vigorous in this respect. Simian is not an open source tool, nor are its methods documented in detail, so little can be said of its workings.

What is known is that Simian ignores white-space, comments, imports, includes and package declarations. It tokenises the code to allow a number of optional parameters for relaxed matching. Parameters specify which differences between source code elements are ignored; those investigated in this research are shown in hierarchical form in Figure 11.4. For example, if differences between strings are ignored, then string case is irrelevant. The default setting ignores modifier differences. After experimenting with various combinations of options and their effect on classification, literals and identifiers are also parameterised in this research. Matching is based on the hashed values of significant lines of the transformed code. Significant lines exclude blank lines and, for example, those with a single brace.

The user is able to specify the minimum number of lines in a clone. The tool outputs a list of all matches, both within file and between files, which can be parsed to find the start and end line numbers for each inter-file clone. Simian's comparison between the three example files is given in Figure 11.5. Each block is described by the number of significant duplicate lines and the line number at the start and end of the block. Optionally, the duplicate code can be printed, as it has been here. Simian finds the same three blocks as CCFinder in this code.

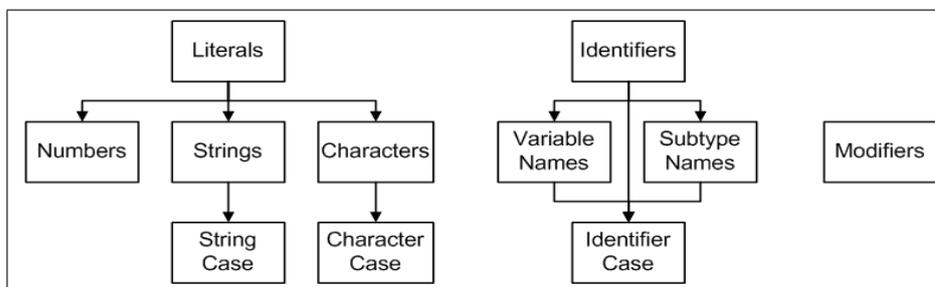


Figure 11.4: Simian parameter hierarchy, the most general parameters at the top

```

Similarity Analyser 2.2.24 -
http://www.redhillconsulting.com.au/products/simian/index.html
Copyright (c) 2003-08 RedHill Consulting Pty. Ltd. All rights reserved.
Simian is not free unless used solely for non-commercial or evaluation purposes.
{failOnDuplication=true, ignoreCharacterCase=true, ignoreCurlyBraces=true,
 ignoreIdentifiers=true, ignoreModifiers=true, ignoreStringCase=true,
 reportDuplicateText=true, threshold=2}

Found 4 duplicate lines in the following files:
Between lines 24 and 31 in /home/.../cnp-2.c
Between lines 33 and 40 in /home/.../cnp-1.c
long combinations (int n, int k){
    return (factorial(n) / (factorial(k) * factorial(n-k)));
}

long permutations (int n, int k)
{
    return (factorial(n) / factorial(n-k));
}
=====
Found 6 duplicate lines in the following files:
Between lines 1 and 7 in /home/.../fact.c
Between lines 24 and 30 in /home/.../cnp-1.c
long factorial (int n){
    long result = 1;
    int i;
    for (i = 1; i <= n; i++)
        result *= i;
    return (result);
}
=====
Found 14 duplicate lines in the following files:
Between lines 4 and 21 in /home/.../cnp-2.c
Between lines 4 and 21 in /home/.../cnp-1.c
long combinations (int n, int k);
long permutations (int n, int k);

int main()
{
    int setsize, subsetsize;
    printf ("Set size? ");
    scanf ("%d", &setsize);
    printf ("Subset size? ");
    scanf ("%d", &subsetsize);
    if (setsize < 0 || subsetsize < 0 || setsize < subsetsize)
    {
        printf ("Mission impossible\n");
        return (1);
    }
    printf ("%ld combinations and %ld permutations of %d items taken from %d\n",
        combinations (setsize, subsetsize), permutations (setsize, subsetsize),
        subsetsize, setsize);
    return (0);
}
=====
Found 48 duplicate lines in 6 blocks in 3 files
Processed a total of 49 significant (86 raw) lines in 3 files

```

Figure 11.5: Simian comparison between cnp-1.c, cnp-2.c and fact.c. Default parameters are used, except that the minimum number of lines required for a clone is 2 instead of 6, and the clone contents are printed.

11.3 Duplo

Duplo [5] is an open source command-line tool similar to ‘Duploc’ [65, 66], which matches hashed lines of code to detect clones. Duplo differs from the other tools in that only exact matches are identified. Blocks of matched code in the files are determined by two parameters: the minimum number of characters in a line, and of consecutive matched lines in the block.

White-space, comments, and, optionally, preprocessing directives are removed, so that a line such as `"a = b + c; /* some comment */"` becomes `"a=b+c;"`. Lines with fewer than the minimum characters specified are ignored. Each line is hashed with the MD5 algorithm [201] before matching. A matrix is coded to show whether each pair of lines in the files match, 1 for a match, or 0 for a mismatch. The matrix is then scanned diagonally, from top left, to find contiguous matches, and those of at least the minimum number of lines are reported. The matrix in Table 11.3 shows the comparison between the files `fact.c` and `power.c` (code in Figure 6.5, p.80), with both parameters set at 2. The sequence of three matched lines is highlighted.

The report in Figure 11.6 is for a comparison between the three example files. Three matched blocks are found, two between `cnp-1.c` and `cnp-2.c`, the first starting at line 3 in each file, and the second on line 32 in `cnp-1.c` and line 23 in `cnp-2.c`, and one block between `cnp-1.c` (starting at line 28) and `fact.c` (at line 0). Following the information about matched blocks are details

	<code>longfactorial(intn)</code>	<code>longresult=1;</code>	<code>inti;</code>	<code>for(i=1;i≤n;i++)</code>	<code>result*=i;</code>	<code>return(result);</code>
<code>longpower(intbase,intn)</code>	0	0	0	0	0	0
<code>longresult=1;</code>	0	1	0	0	0	0
<code>inti;</code>	0	0	1	0	0	0
<code>for(i=1;i≤n;i++)</code>	0	0	0	1	0	0
<code>result*=base;</code>	0	0	0	0	0	0
<code>return(result);</code>	0	0	0	0	0	1

Table 11.3: A Duplo matrix resulting from matching lines in `power.c` and `fact.c`. Matches are indicated by a 1. Sequences of 1s running down diagonally left to right, such as those highlighted, are consecutive copied lines.

of the parameters used for the comparison, and a summary of the results, giving the number of lines in the files and in the duplicated blocks. The line count excludes those with fewer than the minimum number of characters specified, in this example, where the minimum is two, lines with single braces are ignored. Duplo finds the same copied blocks in the example code as Simian and CCFinder, because the example code is not edited.

```

/home/.../cnp-2.c(3)
/home/.../cnp-1.c(3)

long combinations (int n, int k);
long permutations (int n, int k);
int main()
  int  setsize, subsetsize;
  printf ("Set size? ");
  scanf ("%d", &setsize);
  printf ("Subset size? ");
  scanf ("%d", &subsetsize);
  if (setsize < 0 || subsetsize < 0 || setsize < subsetsize)
    printf ("Mission impossible\n");
    return (1);
  printf ("%ld combinations and %ld permutations of %d items taken
          from %d\n",  combinations (setsize, subsetsize),
          permutations (setsize, subsetsize), subsetsize, setsize);
  return (0);

/home/.../cnp-2.c(23)
/home/.../cnp-1.c(32)

long combinations (int n, int k)
  return (factorial(n) / (factorial(k) * factorial(n-k)));
long permutations (int n, int k)
  return (factorial(n) / factorial(n-k));

/home/.../fact.c(0)
/home/.../cnp-1.c(28)

long factorial (int n)
  long  result = 1;
  int  i;
  for (i = 1; i <= n; i++)
    result *= i;
  return (result);

Configuration:
  Number of files: 3
  Minimal block size: 2
  Minimal characters in line: 2
  Ignore preprocessor directives: 0
  Ignore same filenames: 0

Results:
  Lines of code: 52
  Duplicate lines of code: 24
  Total 3 duplicate block(s) found.

Time: 0.062 seconds

```

Figure 11.6: Output from Duplo for a comparison between the files `cnp-1.c`, `cnp-2.c` and `fact.c`. The two `cnp` files share 18 lines of code with at least 2 characters, in 2 blocks of at least 2 lines. The files `cnp-1.c` and `fact.c` share 6 lines of code in 1 block, making a total of 24 lines in 3 blocks.

11.4 P-Duplo

P-Duplo was developed for this research because it is difficult to determine the exact length of the clones detected by Duplo, information needed for removing repeats in the clone pairings, as explained in Section 11.5. P-Duplo uses a modified version of Duplo’s method. P-Duplo is implemented in Racket,² while Duplo is written in C.

Duplo detects matches between lines of code by:

1. Removing comments and white-space from each line in the file.
2. Ignoring lines with fewer than the minimum required characters.
3. Storing the MD5 hash values of the resulting strings.
4. Comparing the hash values between pairs of files.
5. Recording matching hash values, 1 for a match, 0 otherwise.
6. Scanning diagonally for sequences of at least the minimum size.

P-Duplo works in a similar way, with the following changes:

1. Removing white-space (comments are already removed here).
2. Hashing each line with the built-in Racket function “string-hash”.
3. Storing the hash value and the number of characters in the line.
4. Comparing - as Duplo.
5. Recording a match with the number of characters in the line, or 0 for a non-matching pair, see Table 11.4.
6. Scanning the matrix to find sequences of at least the minimum number of lines with at least the minimum number of characters.

The benefit of P-Duplo, apart from finding the exact length of a block, including lines with fewer than the minimum characters, is that once the hash values and number of characters in the line are stored, any combination of line and block sizes can be found from the stored information without re-running steps 1–5. The line numbers are consecutive, unlike those output by Duplo, see, for example, the gap between lines 15 and 28 in Figure 11.1,

²<http://racket-lang.org/>

	longfactorial(intn)	{	longresult=1;	inti;	for(i=1;i<=n;i++)	result*=i;	return(result);	}
longpower(intbase,intn)	0	0	0	0	0	0	0	0
{	0	1	0	0	0	0	0	0
longresult=1;	0	0	13	0	0	0	0	0
inti;	0	0	0	5	0	0	0	0
for(i=1;i<=n;i++)	0	0	0	0	17	0	0	0
result*=base;	0	0	0	0	0	0	0	0
return(result);	0	0	0	0	0	0	15	0
}	0	0	0	0	0	0	0	1

Table 11.4: P-Duplo matrix resulting from matching lines in power.c and fact.c. Matches are indicated by the number of characters in the line. Diagonal sequences of numbers ≥ 1 , such as those highlighted, show consecutive copied lines and can be matched to the given criteria.

making it possible to understand the relationship between the blocks. However, the design is specific to this application where comments and blank lines are removed from the code prior to processing.

The P-Duplo matrix in figure 11.4 shows a line-by-line comparison between the two files power.c and fact.c, the same files compared in Table 11.3. There are two differences: every line, regardless of length, is compared and where a match is found, is recorded by entering the number of characters in the line rather than a "1". In scanning the matrix, instead of looking for "1"s in the grid, the number of characters in the line must be at least the minimum required.

The results may differ slightly from those of Duplo, in that lines with fewer than the minimum characters, ignored by Duplo, must match for a sequence to be recognised by P-Duplo, even though they are not part of the line count for the block. However, in this example, when looking for sequences of at least 2 lines of at least 3 characters, the same block will be found by both tools.

```

long result = 1;
int i;
for (i=1; i<=n; i++)
    
```

11.5 Unscrambling clones

While clone detection tools offer a variety of flexible matching techniques, they have one drawback when used to find the amount of code shared by two files. The drawback is that *every* match between the files which fulfils the criteria of the provided parameters is reported. There can be more code contained in the clones in total than exists in the file. For example, the clones found by CCFinder between `cnp-1.c` and `cnp-2.c` total 244 tokens, while the files contain 208 and 166 tokens respectively (see Figure 11.3).

There are three reasons for this, which are illustrated in Figure 11.7, where the shaded areas represent copied blocks, and identical blocks are given the same shape and shade. The height of the block relates to the size of the clone, but widths vary only to help distinguish between the blocks.

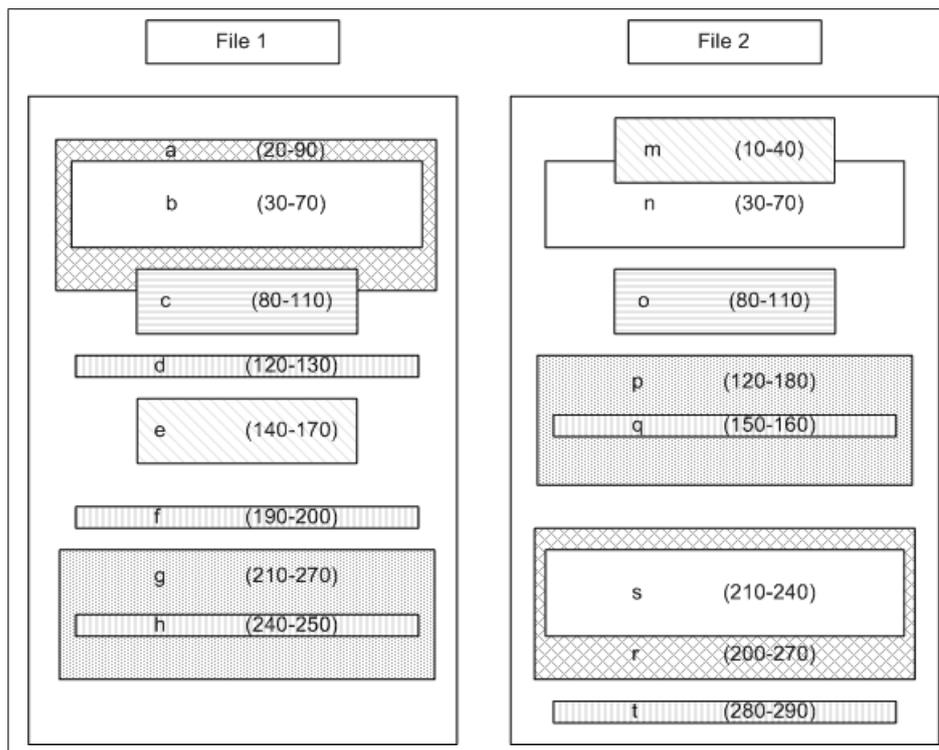


Figure 11.7: Example duplicated blocks in two files. Each block is labelled with a letter, and with start and end numbers. Identical blocks have the same shading and width; for example, block a (file 1) and block r (file 2). Block heights are proportional to their length.

First, when there are multiple copies of the same block of code, each of the blocks in one file is matched to each of the blocks in the other file. For example, if there are 3 copies of a block in one file, such as d, f and h in file 1, and 2 copies in the other, such as q and t in file 2, there will be 6 reported matches of 11 lines, a total of 66 lines; whereas the correspondence between the files is 2 blocks in one file matching 2 in the other, that is 22 lines.

Second, two clones can overlap, so that the overlapping code is counted twice, such as the lines 80–90 in file 1, which are matched as part of block a to r, as well as block c to o.

Third, one clone can be subsumed by another. Consider the block n which is matched to block b. As block a is matched to block r, the clones b and s which are subsumed by blocks a and r, do not need to be accounted for. As block n is matched to block b and has no other match, it should not be considered as part of the correspondence between the files.

Clone detection tools match clones on a many-to-many basis. One-to-one matching is likely to be more useful in matching restructured files. The duplication in clone tool outputs can be made to approximate one-to-one matching by an unscrambling technique described in full in a technical report [94]. The three step unscrambling process is illustrated in Figure 11.8 and described briefly here.

First, subsumed clones are removed, in the example, blocks b and h in file 1, and blocks q and s in file 2, are subsumed by blocks a, g, p and r respectively. Blocks whose only match is a subsumed block, such as block n in file 2, matched to block b, are also removed. Figure 11.8b shows the matching blocks after removal of the subsumed blocks and their dependants.

Second, unmatched multiple copies are removed. The strategy used here is to match the multiple blocks in the order that they appear in the file. This means that block d is matched to block t. Block f is unmatched and therefore removed. The remaining blocks are shown in Figure 11.8c.

Third, where blocks overlap, the strategy is to remove the overlapping code from the smaller of the two blocks. The overlapping code is also

removed from its partner in the other file. In the example, lines 80-90 in block c, which overlap block a are removed. The same eleven lines are also removed from block o which is the block paired with c, see Figure 11.8d.

Once the three steps are complete, the blocks remaining in each file match, the amount of copied code in the two files is the same, and none of the matching code is counted more than once.

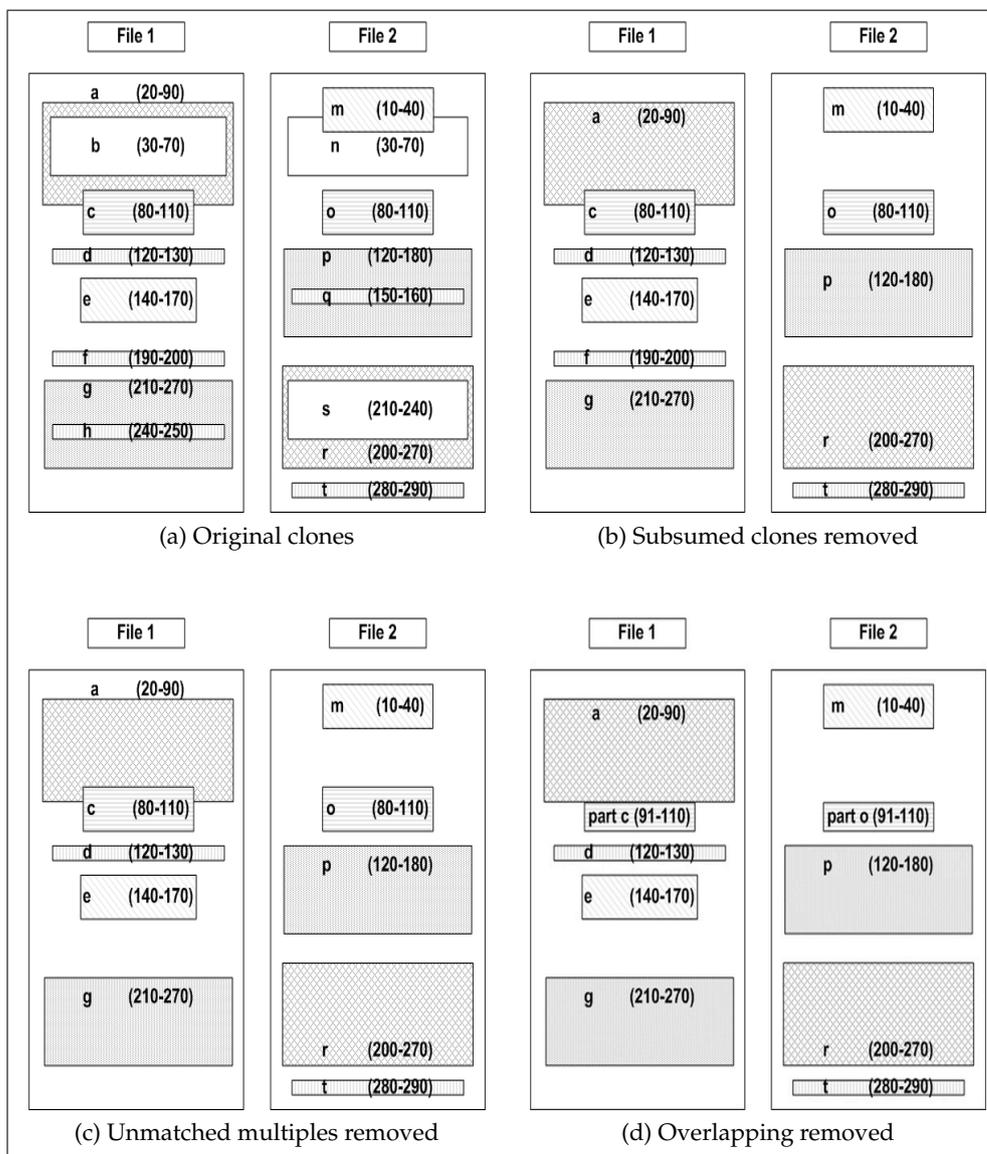


Figure 11.8: Steps in unscrambling clones for one-to-one matching.

11.6 Summary

In this chapter, the third-party tools Code Clone Finder, Duplo and Simian were introduced along with P-Duplo, an adaptation of the method used by Duplo more suited to origin analysis. Each of these tools uses a representation of the code based on tokens or text, rather than more complex transformations, such as abstract syntax trees or program dependency graphs. The tools were described and compared, highlighting their benefits and drawbacks in matching code for origin analysis. The tools provide complementary information about matched code, with the weakness in each one generally covered by the strengths of the others.

The “over-stated” clones reported by the clone detection tools are identified as a problem for matching code in origin analysis. The Unscrambling tool, developed to give a set of blocks which are matched one-to-one, was described in this chapter. As noted in Chapter 6, the Ferret XML report can be analysed to produce information about blocks of code covered by matching trigrams, both directly, and by using the density analysis tool. The raw output from the tools, and the blocks from each of the tools are used as the basis for feature construction, which is described in Chapter 13.

Chapter 12

Data Collection, Preprocessing and Filtering

This chapter describes the data collection, preprocessing and filtering steps outlined in Chapter 10, and is in four main parts. The first describes the source code collection and preparation. Next, terminology used in this and subsequent chapters is introduced. The filtering process is then explained, followed by a description of the resulting datasets and their labelling.

12.1 Source code collection and organisation

The source code for this study was drawn from the open source repository SourceForge [183]. About three hundred projects were initially selected and downloaded. The selection process is explained in Section 12.1.1. These projects were examined and unsuitable ones rejected, resulting in a final set of eighty-nine projects, which are described in Section 12.1.2. Prior to filtering, the data is organised and preprocessed as detailed in Section 12.1.3.

12.1.1 Selection of projects from SourceForge

SourceForge.net lists a large number of open source projects (more than 280,000 in May 2012). It offers a range of selection criteria, such as program-

ming languages used, development status, operating systems supported and topic. The criteria used to select projects for this task were:

- Programming language: C
- Operating system: Linux
- Development status: Production/Stable or Mature

When the projects were selected (2006), the concurrent versioning system, CVS, was more commonly used than Subversion [57, p.4]. To allow projects to be downloaded with a single script, only those using CVS were considered. Prior to downloading, manual filtering, using the Eclipse workbench¹ to inspect the files, excluded projects with few C code files, with fewer than three releases, or those which had been removed from the repository.

Versioning systems record every change committed, but to reduce the amount of material stored and processed, and to build a system which should generalise to software systems where only stable releases are available, only versions tagged by the developers as significant were downloaded for analysis. Although the authors of different projects determine these intervals in different ways, this subset of revisions provides a reasonable granularity for study. Downloading was automated with a bash script based on the tags. Several of the projects were subsequently found to be incorrectly ordered or to contain branches, which was resolved by downloading the affected projects manually, release by release.

Figure 12.1 outlines the steps in data collection and preparation. The top half of the diagram shows the organisation of the code, with each project in a separate directory. Within the projects, each release goes to a separate, consecutively numbered, directory.

12.1.2 The selected data - 89 Projects

Details of the 89 projects selected are provided in two tables in Appendix H. Table H.1, on pages 382–383, shows the project names and purposes, and Table H.2, which follows on pages 384–385, shows the project sizes and the

¹<http://www.eclipse.org/>

amount of code in the source code files. As C code is used in this study, there is a natural bias towards scientific, engineering or technical projects.

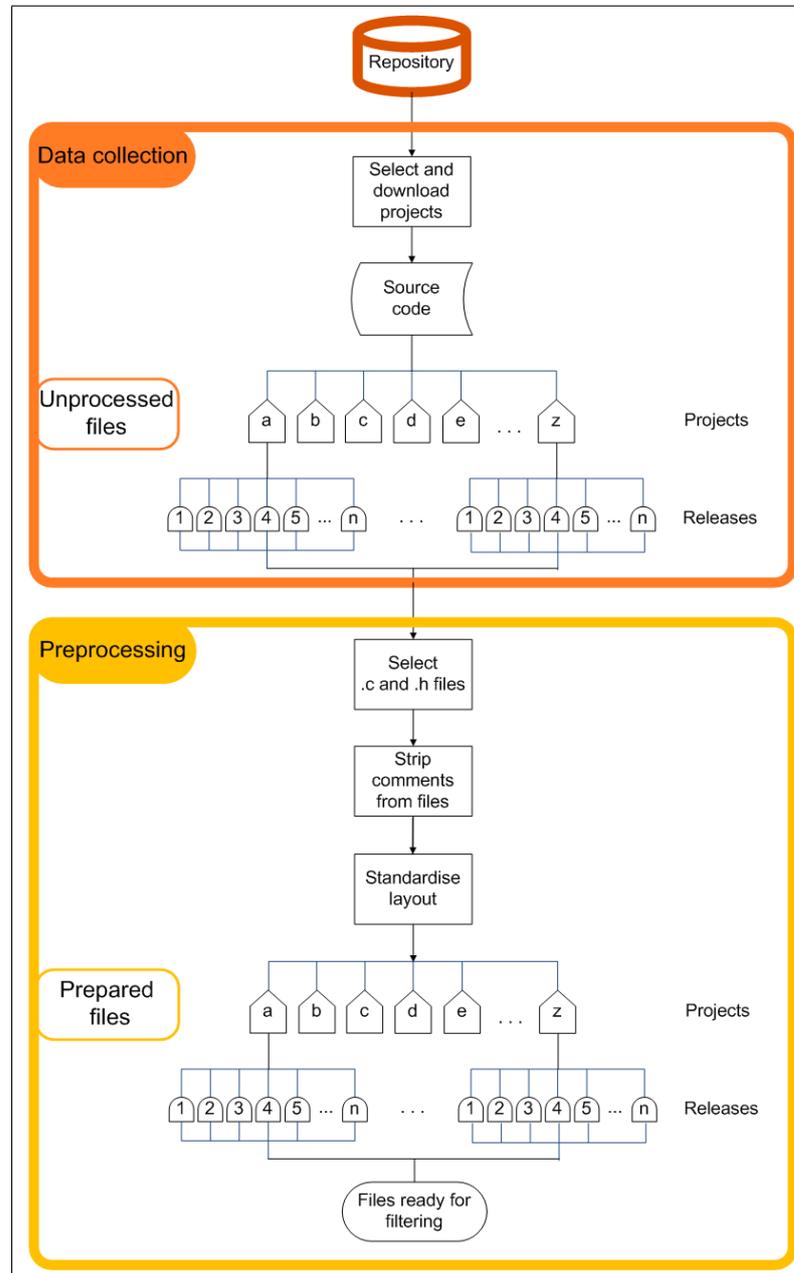


Figure 12.1: Source code collection and preprocessing. Each project is downloaded and stored in its own directory (labelled a...z) with each release in a separate subdirectory (labelled 1...n). C code files are selected and comments stripped from each file, and layout is standardised. The prepared code is stored using the same structure as the raw code.

The projects vary in size and growth rate. Some of the projects, possibly those established before joining Sourceforge, change little between releases, while others show rapid growth. The two extremes are exemplified by the projects 'giw', which remains almost the same size, 10 KLOC, throughout the 3 releases, whereas 'xastir' grows from 47 to 111 KLOC over 140 releases. The granularity of releases varies from those tagged at every change, to those only tagged when the project is deemed stable after major changes.

The C code in individual releases varies from just over 100 lines of code (LOC) to over 200 KLOC, in 2 to 752 files. The number of releases per project ranges from 3 to 140, with a mean of 16. There are 11,210 distinct C code files, with a total of approximately 117,500 files in the 1,405 releases.

12.1.3 Preprocessing

The bottom half of Figure 12.1 shows the steps in preparing the code for comparison. First, all of the C source code files ('.c' and '.h') were selected from the projects. Then comments were removed from the files. This is particularly important in projects where large identical comment blocks, such as those detailing licences, are placed at the head of each file, because the similarity between the files will be distorted by the comments in text-based comparisons. For example, two files from the biew project, illustrated in Figure 12.2, share the same comment block, highlighted in red. The files, ref.c and codeguid.h, have little code in common, but because of the shared comments have a fairly high similarity score of around 0.33. Without the comments their similarity is 0.02.

Removing such comment blocks is useful in this context, however, it is less clear whether the removal of all comments is the best strategy overall. On the one hand, similar comments may point to similarity between files, but on the other hand, it is possible that unrelated comments have elements in common, such as the author's name or the use of standard wording for standard tasks. However, as noted in Chapters 2 and 3, few of the approaches to clone detection, plagiarism detection or origin analysis include comments in their file comparisons. It therefore seems reasonable to do the

```

/**
 * @namespace biew
 * @file refs.c
 * @brief This file contains basic level routines for resolving references.
 * @version -
 * @remark this source file is part of Binary VIEW project (BIEW).
 * The Binary VIEW (BIEW) is copyright (C) 1995 Nick Kurshev.
 * All rights reserved. This software is redistributable under the
 * licence given in the file "Licence.en" ("Licence.ru" in russian
 * translation) distributed in the BIEW archive.
 * @note Requires POSIX compatible development system
 *
 * @author Nick Kurshev
 * @since 1995
 * @note Development, fixes and improvements
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "reg_form.h"
#include "biewutil.h"
#include "bconsole.h"

extern REGISTRY_BIN binTable;
unsigned long __fastcall__ AppendAsmRef(char *str,unsigned long ulShift,int
mode,char codelen,unsigned long r_sh)
{
    static tBool warn_displayed = False;
    unsigned long ret = RAFPREF_NONE;
    if(detectedFormat->bind) ret = detectedFormat-
>bind(str,ulShift,mode,codelen,r_sh);
    else
    {
        if(detectedFormat != &binTable && !warn_displayed)
        {
            WarnMessageBox("Sorry! References resolving for this format still not
supported",NULL);
            warn_displayed = True;
        }
    }
    return ret;
}

/**
 * @namespace biew
 * @file codeguid.h
 * @brief This file contains prototypes code navigator.
 * @version -
 * @remark this source file is part of Binary VIEW project (BIEW).
 * The Binary VIEW (BIEW) is copyright (C) 1995 Nick Kurshev.
 * All rights reserved. This software is redistributable under the
 * licence given in the file "Licence.en" ("Licence.ru" in russian
 * translation) distributed in the BIEW archive.
 * @note Requires POSIX compatible development system
 *
 * @author Nick Kurshev
 * @since 1995
 * @note Development, fixes and improvements
 */
#ifdef __CODEGUID_H
#define __CODEGUID_H

#ifdef __SYS_DEP_H
#include "sys_dep.h"
#endif

#ifdef __cplusplus
extern "C" {
#endif

extern char codeguid_image[];

extern void __fastcall__ GidResetGoAddress( int keycode );
extern void __fastcall__ GidAddGoAddress(char *str,unsigned long
addr);
extern void __fastcall__ GidAddBackAddress( void );
extern unsigned long __fastcall__ GidGetGoAddress(unsigned keycode);
extern char * __fastcall__ GidIncodeAddress(unsigned long
cfpos,tBool aresolv);

extern tBool __fastcall__ initCodeGuider( void );
extern void __fastcall__ termCodeGuider( void );

#ifdef __cplusplus
}
#endif

#endif

```

Figure 12.2: The files refs.c and codeguid.h from the biew project. The shared comments, which are coloured red, are about one-third of the size of the files. Without the comment block, the files have little in common.

same. As some of the tools used to compare the files do so on a line-by-line basis, pretty-printing is used to standardise the layout.²

12.2 Terminology

The terms used when discussing file selection are shown in Table 12.1. A *candidate file* is a file which matches the filtering criteria and is therefore possibly an example of the type of file sought. An *amended file* is a candidate file in the next release. In the same way as many others (e.g. [22, 90, 128, 130, 179]), the approach in this research first matches by name, forming two groups of files: those which are matched by files of the same name in the following release, and those which are not, known as *disappearing files*. The words similarity, similarity score or similarity coefficient mean the Jaccard

²Gnu Indent with the options -gnu -bli0 -di0 -i0 -ip0 -nhnl, removing indentations and ignoring breaks within a line, combined with the removal of leading spaces and blank lines

Term	Notation	Description
Candidate file	f_a^n	Any file a in release n, matching the filter criteria
Amended file	f_a^{n+1}	Revised version of the candidate file in release n+1
New file	$f_b^{n+1} \bullet \nexists f_b^n$	A file in release n+1 not in release n
Disappearing file	$f_a^n \bullet \nexists f_a^{n+1}$	A file in release n not in release n+1 i.e. a renamed, moved or deleted file
Similarity	$\text{Sim}(f_1, f_2)$	Jaccard coefficient of similarity between files f1 and f2, based on token trigrams $\frac{ f_1\text{trigrams} \cap f_2\text{trigrams} }{ f_1\text{trigrams} \cup f_2\text{trigrams} }$
Consecutive-similarity	$\text{Sim}(f_x^n, f_y^{n+1})$	Similarity between 2 files in consecutive releases
Self-similarity	$\text{Sim}(f_a^n, f_a^{n+1})$	Consecutive similarity of files of the same name
Same-similarity	$\text{Sim}(f_x^n, f_y^n)$	Similarity between files in the same release
Similar file	$f_x^{n+1}, \text{Sim}(f_a^n, f_x^{n+1}) > \text{Min-sim}$	File x • $\text{Sim}(f_{\text{candidate}}^n, f_x^{n+1}) \geq$ minimum similarity threshold

Table 12.1: Terminology

coefficient of similarity between two files based on token trigrams in the files. Similarities between different groups of files are given special names which are explained in Section 12.3.1.

Two other terms used are *target file* and *candidate group*. A target file is a file related to the candidate file by similarity and proximity. For example, a candidate split file's targets are files in the next release similar to the candidate file, which may therefore be recipients of the code removed from the candidate. The targets for a disappearing file are the file or files in the next release which are similar enough to the file to be the possible new location(s) of the file. A candidate group consists of the candidate file, the amended version of this file, where it exists, and the target file or files. There can be one or more target files in a group, and they can be any mix of new files similar to the candidate, and existing files which have become more similar to the candidate than they were in the previous release.

12.3 Filtering

There are two steps in filtering the code to find potentially interesting files. The first step is to gather information about the relationships between files throughout a project, and the second step is to combine aspects of this information to find files of different types. The information gathering step is described in Section 12.3.1 and the filtering in Section 12.3.2.

12.3.1 Information Gathering

Once the source code is collected and prepared, the files in each release are compared to all of the other files in the same release, and all of the files in the next release. There are two aims in comparing the files. First, to find files with changes between releases which indicate possible membership of the category of interest; and second, to find other files which may be related to the change in the candidate file.

Ferret is used for comparing files in the filtering stage because of its relatively fast processing time [194]. There are a large number of comparisons made during the filtering stage. For this group of projects, with an average of 16 releases per project and 84 files per release, an estimated 14 million comparisons between pairs of files are made.³

Vectors of file sizes, in bytes, and of similarities between the files in each project are stored for use in filtering. The three types of similarity calculated are: *self-similarity*, of each file to the file of the same name in the next release; *consecutive-similarity*, of each file to every other file in the next release; and *same-similarity*, of each file to every other file in the same release.

An example mini-project is used to illustrate the three types of similarity. This mini-project has 3 releases and 2 original files, a and b. File b is split between releases 1 and 2 to form file c. Table 12.2 lists example similarities and file sizes. The similarities are marked in Figure 12.3, except those between files a and c, as they would make the diagram too cluttered.

Self-similarity shows whether a file has changed between releases. Cons-

³(Estimate: $89 * [((84 * 83)/2) * 16] + (84 * 84 * 15) = 14,383,824$)

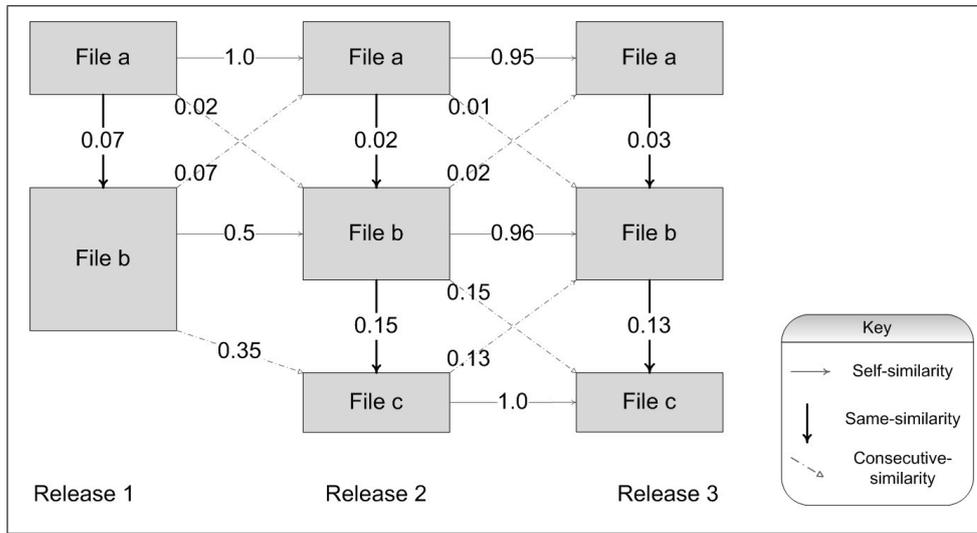


Figure 12.3: The example mini-project with self-, same- and consecutive-similarities shown, except those between files a and c.

secutive-similarity is used to find similarities between the file of interest and files in the next or previous releases. Comparisons between same-similarity and consecutive-similarity are used to detect the changes to similarity between releases. The vectors include information for every release or pair of releases. Where a file does not exist in a release, its size is set to 0 and its similarities to -1.

Size in bytes	a <3000, 3000, 3100>	b <6000, 4000, 4200 >	c <0, 2400, 2400>
Self-similarity	a <1.0, 0.95>	b <0.5, 0.96 >	c <-1, 1.0>
Same-similarity	a-b <0.07, 0.02, 0.03>	b-c <-1, 0.15, 0.13>	a-c <-1, 0.02, 0.01>
Consecutive-similarity	a-b <0.02, 0.01> b-a <0.07, 0.02>	b-c <0.35, 0.15> c-b <0.13>	a-c <n/a here> c-a <n/a here>

Table 12.2: Size and similarity vectors for the mini-project. Same-similarity is between files in the same release, self similarity between a file in one release and the same file in the next release, and consecutive-similarity is that between files in consecutive releases, excluding self-similarities.

In the example, file a is unchanged between releases 1 and 2, and therefore has a self-similarity of 1.0, it has minor edits before release 3, when the similarity between releases is 0.95. File b splits after release 1, resulting in a smaller file b, and the new and similar file c. File b has self-similarity of 0.5, and a consecutive-similarity with file c of 0.35. As file c does not exist in release 1, the value -1 is used in place of similarity scores between this file and other files.

12.3.2 Selecting Candidate Files

The stored vectors are scanned to find possible examples of the required type of file. Different filtering criteria are used for each application.

The main filter for renamed or moved files is size, to find files which exist in release n , but not in the next release, $n+1$. Consecutive-similarity provides a second filter, finding target files in release $n+1$ which are similar to the file which has disappeared.

Split file filters are based on size and similarity. A split file is expected to become smaller, and the self-similarity to be lower than it would be after edits such as within-file abstraction. In addition, one or more target files, those which now contain the extracted code, will increase their similarity to the old file.

Merged files can be viewed as the reverse of split files, and by scanning the releases in reverse order, a merged file will appear to be a split file. They can also be filtered by looking for files which have increased in size and have become more similar to a file from the previous release, which may either have disappeared or have become smaller. However, the view taken in this study is that for code from one file to be merged with another file, it must have originated somewhere in the system. As such, a merge can only happen if a file in the previous release has either split or disappeared, so that looking for merged files repeats the work of finding the other two types of restructuring.

Other aspects of file evolution can also be found. From size alone, growing, shrinking, static and new files can be pinpointed. Other changes

to files can be traced from self-similarity.

The groups formed with each category of candidate file differ. Split file candidate groups consist of the candidate file, from release n , the amended file from release $n+1$, and one or more target files, also in release $n+1$: either new files whose similarity exceeds the minimum threshold, or existing files which have increased their similarity to the candidate file by at least the

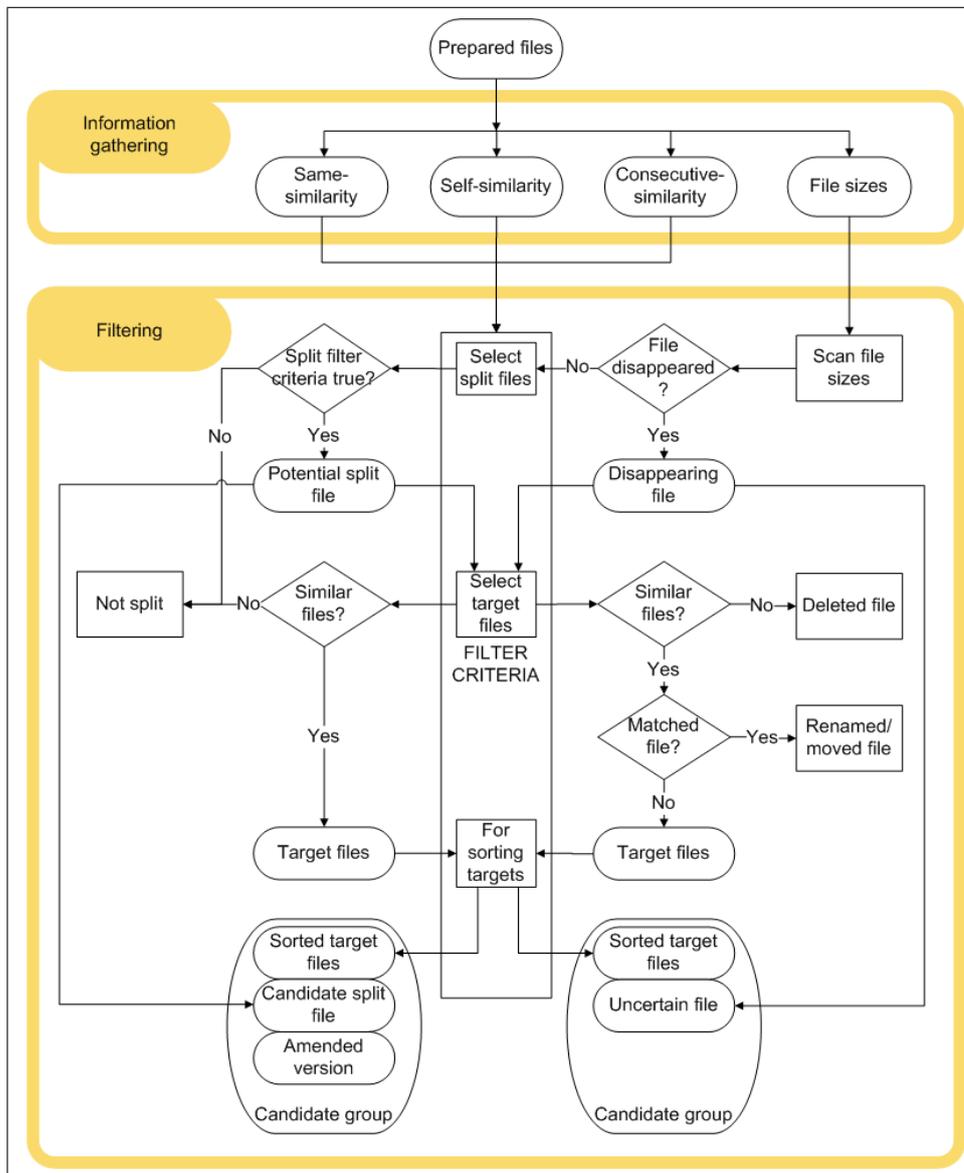


Figure 12.4: Information gathering and filtering

minimum margin between releases n and $n+1$.

Disappearing files encompass moved, renamed or deleted files, files which merge with others, or files which are split, where both newly created files are renamed. Groups related to these files consist of any files in the next release with a similarity to the disappearing file which is above a chosen threshold.

Figure 12.4 gives an overview of the filtering processes for split and disappearing files and their targets. Disappearing files are found by scanning the file size vectors. Target files are found by looking at consecutive- and same-similarities. If there are no similar files, then the file is assumed to have been deleted. If the most similar file is a close match (similarity >0.85), then the file is assumed to have been moved or renamed. Disappearing files which belong to neither of these two categories are called the *uncertain set*. The target files for the uncertain set are sorted according to the chosen criteria and grouped with the disappearing file for further investigation.

Candidate split files are selected based on size and self-similarity. Target files are selected using the filter criteria based on consecutive- and same-similarity. If there are no target files, the file is assumed to have been edited but not split. Otherwise, the target files are sorted and grouped with the candidate split file and its amended version for further investigation.

12.4 Experimental Data

Two different datasets were extracted from the 89 project source code database: candidate split files, described in Section 12.4.1; and disappearing files, described in Section 12.4.2. For each set, both theoretical and real examples are provided to give an idea of the range of relationships between files in each of the classes. Also provided are examples where classification is difficult. In the text, the real examples are mostly based on small files, and headers are sometimes removed, to fit the page. Full versions of the files and larger file examples are available online.⁴ These examples are

⁴<http://homepages.stca.herts.ac.uk/~gp2ag/trigram-analysis-examples.htm>

produced by the 3CO tool, see page 117, Chapter 8.

For each of these datasets, the class of the candidate files was decided by visual inspection of all files in the comparison group. Much of the file classification was undertaken prior to the development of 3CO, which would have streamlined the process.

To put the choice of filtering thresholds in context, the mean consecutive-similarity across all projects in this study is around 0.04. This figure excludes self-similarity, where the mean is 0.95.

12.4.1 Split file dataset

Candidate split files for this experiment were selected by filtering to find files with a reduction in size of at least 10% and no more than 0.9 self-similarity. Target files were selected if their consecutive-similarity was at least 0.1, and, for existing files, an increase in similarity of at least 10%. Two classes were assigned to the set of candidate split files, either split or not. There were a few files for which classification was not clear, and these were excluded from the dataset.

12.4.1.1 Split files

Split files vary in their complexity. Three example split files are shown as block diagrams in Figure 12.5. The first of these, 12.5a, a simple split, will generally only apply to .h files, as it is usually difficult to split a .c file without the need to include at least some of the original code in both the resulting files. The second block diagram, 12.5b, shows a file from which some code has been deleted, some added and some edited. One section of code has also been moved to an existing file. The last diagram, 12.5c, shows a file split three ways, one part forming a new file and one part added to an existing file. There is also another file in the system which has been edited and has incidentally become more similar to the candidate file. This file is therefore selected as a target file, although the code from the split file has not been moved to it.

Three examples taken from the dataset illustrate a simple two-way split,

a three-way split and a multi-way split. The simple split file example is in Figure 12.6.⁵ Code which remains in the amended version of the file diagram.h is coloured blue, and code going to globals.h is red.

A slightly more complex example is shown in Figure 12.7, which is laid out in two columns to fit the page.⁶ This file, gemdos.h, has been split into three parts and some of the code has been edited or deleted. The code remaining in the new version of the file is coloured blue. Code which goes to the new file gemdos.defines.h is red, and that going to the existing file gemdos.c is yellow. Edited code is not matched in the other files and is therefore coloured cyan.

There is a wide range of patterns in the candidate groups, which are mostly more complex than these two examples. Even simple splits vary: the amount of code moved can be very small relative to file size, for example

⁵<http://homepages.stca.herts.ac.uk/~gp2ag/xmls/diagram.xml>

⁶<http://homepages.stca.herts.ac.uk/~gp2ag/xmls/gemdos.xml>

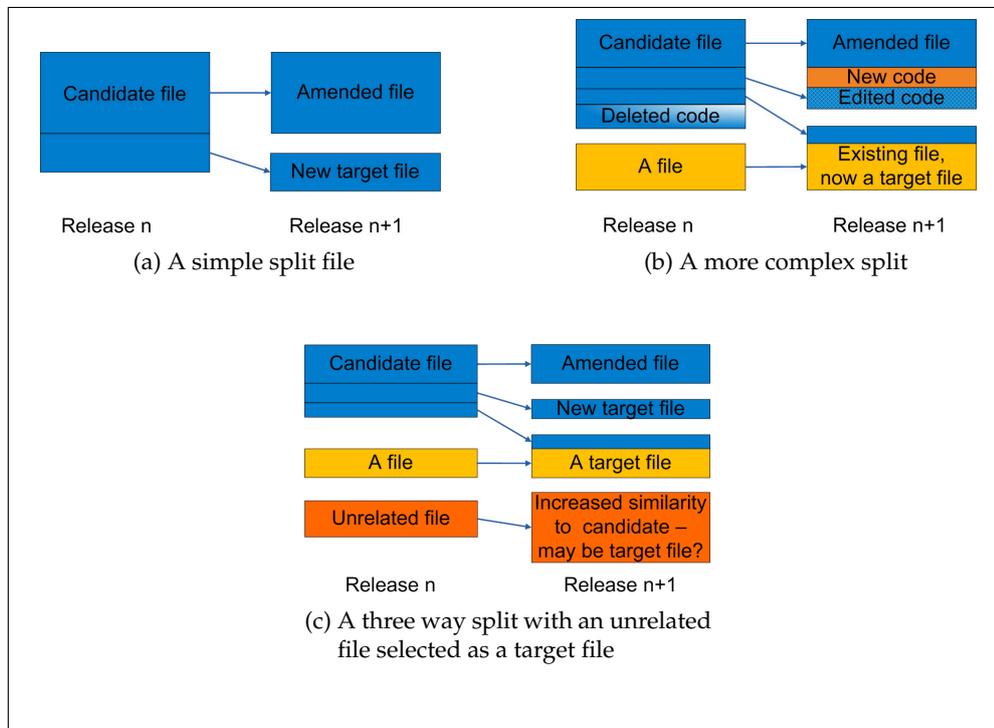


Figure 12.5: Example split files

```

#include "capture.h"
#ifndef 0
typedef enum
{
    LINEAR = 0,
    LOG = 1,
    SQRT = 2
}
size_mode_t;
#endif
typedef struct
{
    guint8 *canvas_node_id;
    node_t *node;
    GnomeCanvasItem *node_item;
    GnomeCanvasItem *text_item;
    GnomeCanvasItem *accu_item;
    gchar *accu_str;
    GnomeCanvasGroup *group_item;
}
canvas_node_t;
typedef struct
{
    guint8 *canvas_link_id;
    link_t *link;
    GnomeCanvasItem *link_item;
    GdkColor color;
}
canvas_link_t;
gdouble get_node_size (gdouble average);
gdouble get_link_size (gdouble average);
gint reposition_canvas_nodes (guint8 * ether_addr,
    canvas_node_t * canvas_node,
    GtkWidget * canvas);
gint update_canvas_links (guint8 * ether_link, canvas_link_t * canvas_link,
    GtkWidget * canvas);
gint update_canvas_nodes (guint8 * ether_addr, canvas_node_t * canvas_node,
    GtkWidget * canvas);
gint check_new_link (guint8 * ether_link, link_t * link, GtkWidget * canvas);
gint check_new_node (guint8 * ether_addr, node_t * node, GtkWidget * canvas);
guint update_diagram (GtkWidget * canvas);
void init_diagram ();

```

Figure 12.6: An example of a simple split file. This shows the file diagram.h from the etherape project. It is split, with some of the code going to globals.h (coloured red), and most of the rest remaining in diagram.h (blue).

15 of 970 lines; or can be a large proportion, such as 1067 out of 1080 trigrams. There may be multiple target files, for example, one of the candidate files in the set is split into 11 of its 13 target files.⁷ This large file (over 1000 lines) is shown in Figure 12.8, the parts of the file moved to some of the target files is more scattered than in the previous examples. The text of the candidate file is repeated five times as there are thirteen target files. The first of these is the amended version of the candidate file in the next release. The rest of the files are new ones created as a result of the split, or existing files which are similar enough to the candidate file to be considered as targets. Vertical bars have been added to the right of the text to highlight the file to which

⁷<http://homepages.stca.herts.ac.uk/~gp2ag/xmls/dialog-8.xml>

```

#ifdef HATARI_GEMDOS_H
#define HATARI_GEMDOS_H
#define GEMDOS_EOK 0
#define GEMDOS_ERROR -1
#define GEMDOS_EDRVNR -2
#define GEMDOS_EUNCMD -3
#define GEMDOS_E_CRC -4
#define GEMDOS_EBADRQ -5
#define GEMDOS_E_SEEK -6
#define GEMDOS_EMEDIA -7
#define GEMDOS_ESECNF -8
#define GEMDOS_EPAPER -9
#define GEMDOS_EWRITF -10
#define GEMDOS_EREADF -11
#define GEMDOS_EWRPR0 -12
#define GEMDOS_E_CHNG -14
#define GEMDOS_EUNDEV -15
#define GEMDOS_EINVFN -32
#define GEMDOS_EFILNF -33
#define GEMDOS_EPTHNF -34
#define GEMDOS_ENHNDL -35
#define GEMDOS_EACCDN -36
#define GEMDOS_EIHNDL -37
#define GEMDOS_ENSMEM -39
#define GEMDOS_EIMBA -40
#define GEMDOS_EDRIVE -46
#define GEMDOS_ENSAME -48
#define GEMDOS_ENMFIL -49
#define GEMDOS_ELOCKED -58
#define GEMDOS_ENSLOCK -59
#define GEMDOS_ERANGE -64
#define GEMDOS_EINTRN -65
#define GEMDOS_EPLFMT -66
#define GEMDOS_EGSBF -67
#define GEMDOS_ELOOP -80
#define GEMDOS_EMOUNT -200
#define GEMDOS_FILE_ATTRIB_READONLY 0x01
#define GEMDOS_FILE_ATTRIB_HIDDEN 0x02
#define GEMDOS_FILE_ATTRIB_SYSTEM_FILE 0x04
#define GEMDOS_FILE_ATTRIB_VOLUME_LABEL 0x08
#define GEMDOS_FILE_ATTRIB_SUBDIRECTORY 0x10
#define GEMDOS_FILE_ATTRIB_WRITECLOSE 0x20
#define TOS_NAMELEN 14
typedef struct
{
    unsigned char index[2];
    unsigned char magic[4];
}
continued ...

... continued
char dta_pat[TOS_NAMELEN];
char dta_sattrib;
char dta_attrib;
unsigned char dta_time[2];
unsigned char dta_date[2];
unsigned char dta_size[4];
char dta_name[TOS_NAMELEN];
} DTA;
#define DTA_MAGIC_NUMBER 0x12983476
#define MAX_DTAS_FILES 256
#define CALL_PEXEC_ROUTINE 3
#define BASE_FILEHANDLE 64
#define MAX_FILE_HANDLES 32
typedef struct
{
    unsigned short word1;
    unsigned short word2;
} DATETIME;
#ifdef MAX_PATH
#define MAX_PATH 256
#endif
typedef struct
{
    char hd_emulation_dir[MAX_PATH];
    char fs_currpath[MAX_PATH];
    int hd_letter;
} EMULATEDDRIVE;
extern EMULATEDDRIVE **emudrives;
#define (SHARDDRIVE(Drive) (Drive!=1)
#define GEMDOS_EMU_ON (emudrives != NULL)
extern BOOL bInitGemDOS;
extern unsigned short int CurrentDrive;
extern void GemDOS_Init (void);
extern void GemDOS_Reset (void);
extern void GemDOS_InitDrives (void);
extern void GemDOS_UnInitDrives (void);
extern void GemDOS_MemorySnapShot_Capture
(BOOL bSave);
extern void GemDOS_CreateHardDriveFileName
(int Drive, char *pszFileName,
char *pszDestName);
extern BOOL GemDOS (void);
extern void GemDOS_OpCode (void);
extern void GemDOS_RunOldOpCode (void);
extern void GemDOS_Boot (void);
#endif

```

Figure 12.7: This file is gemdos.h from the hatari project which is split 3 ways, with some editing. The blue code appears in the file in the next release. That coloured red is in the new file gemdos_defines.h and the yellow has moved to the existing file, gemdos.c. The cyan code has been edited.

each section of the code has moved. For example, the first part of the code is found in the amended file (blue, first column); the next section of code has been moved to the fourth of the thirteen other target files (red, second column); and the next section has been moved to the ninth of the targets (blue, fourth column). The last two target files (yellow, column 4 and the blue, column 5) are not true targets as they are only incidentally similar to the candidate file.

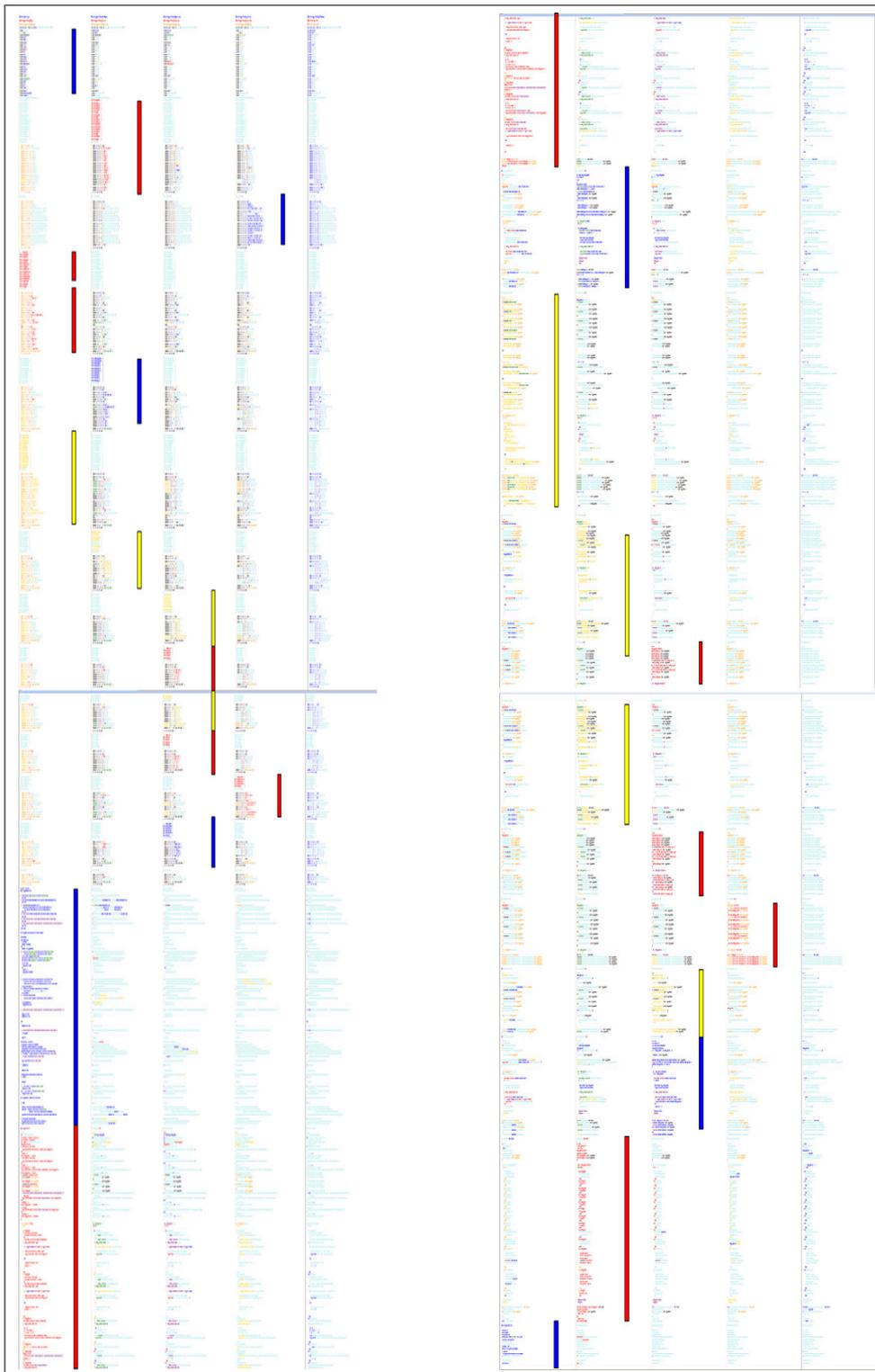


Figure 12.8: The file dialog.c from the hatari project, has over 1000 lines and therefore laid out in two columns to fit the page. The file is split into 11 of the 13 target files. Coloured bars added to the right of the text show which of the target files contains each section of code.

12.4.1.2 Non-split files

Not all of the selected candidate files turn out to be split. Block diagrams of examples of such files are shown in Figure 12.9. There are two conditions for a file to be selected. First, it must reduce in size, and this is represented by the block of deleted code common to each example. Second, there must either be a similar file introduced to the system, or a file which has become more similar to the potential candidate than it was in the previous release. This can occur when an existing file is edited, either by adding code which is incidentally similar to the candidate (Figure 12.9c), or by removing code which is not. A new file may be similar to the candidate either incidentally (Figure 12.9a), or because a section of the candidate file has been used in a copy-paste-edit operation when creating the new file (Figure 12.9b).

An example of a non-split candidate, `inglossarydlg.h` from the interest project, is shown in Figure 12.10. Here the file has been edited, thus becoming smaller. The two files which share code with the file do so because of in-project similarities. Apart from the edited code in cyan, all of the remaining code is coloured blue or a mix of colours which contain blue.

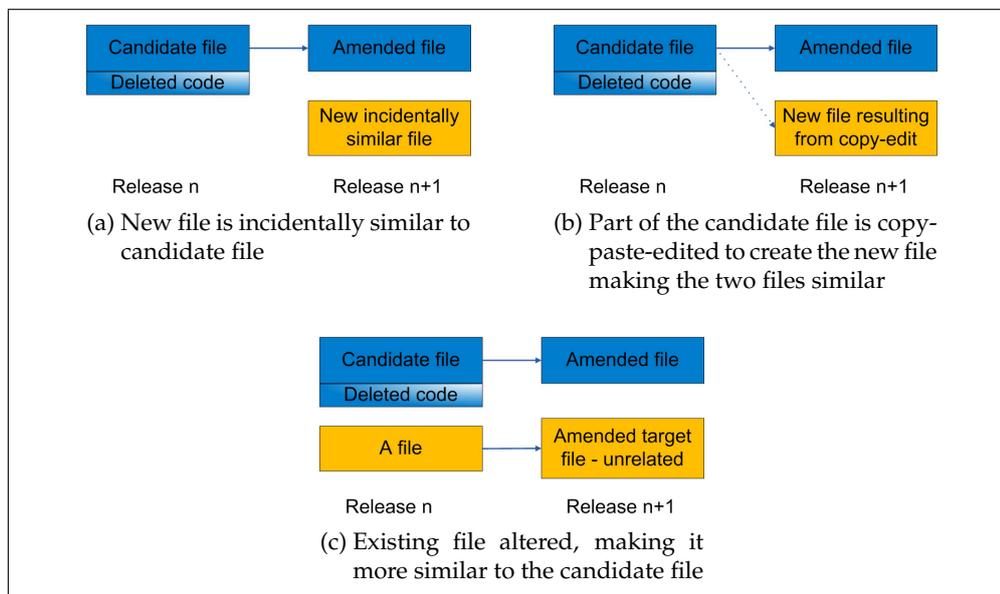


Figure 12.9: Example non-split files selected as candidate split files. In each case, the file becomes smaller, fulfilling part of the selection criteria.

```

#ifdef __IN_GLOSSARYDLG_H__
#define __IN_GLOSSARYDLG_H__
#include <gtk/gtk.h>
#ifdef __cplusplus
extern "C"
{
#endif
#define IN_GLOSSARYDLG(obj)
GTK_CHECK_CAST(obj, in_glossarydlg_get_type(), InGlossaryDlg)
#define IN_GLOSSARYDLG_CLASS(klass)
GTK_CHECK_CLASS_CAST(klass, in_glossarydlg_get_type(), InGlossaryDlg)
#define IS_IN_GLOSSARYDLG(obj)
GTK_CHECK_TYPE(obj, in_glossarydlg_get_type())
typedef struct _InGlossaryDlg InGlossaryDlg;
typedef struct _InGlossaryDlgClass InGlossaryDlgClass;
struct _InGlossaryDlg
{
    GtkWidget win;
    GtkWidget *clist;
    GtkWidget *desc_label;
}
struct _InGlossaryDlgClass
{
    GtkWidgetClass parent_class;
};
GtkType in_glossarydlg_get_type(void);
GtkWidget *in_glossarydlg_new ();
#ifdef __cplusplus
}
#endif
#endif

```

Figure 12.10: An example of a non-split file, interest/src/inglossarydlg.h, selected as a split candidate because it has reduced in size, and two files in the system are similar enough to be possible target files. However, code in the target files is also in the revised version of the candidate.

The green code is also shared by the existing file inchartcfg.h, as is the black code which also appears in the new file widgets/innumentry.h.

In the example in Figure 12.11, ac1d.c from the xmp project, a first look at the red and orange coloured code might appear to be a split to the “red” file. In fact, the code occurs commonly throughout the project, which can be seen from the other columns where the code is coloured black meaning it is in each of the three files in that column. The amount of common code is also magnified here because a few lines are repeated (see e.g. the boxed code in the lower right corner of the figure).

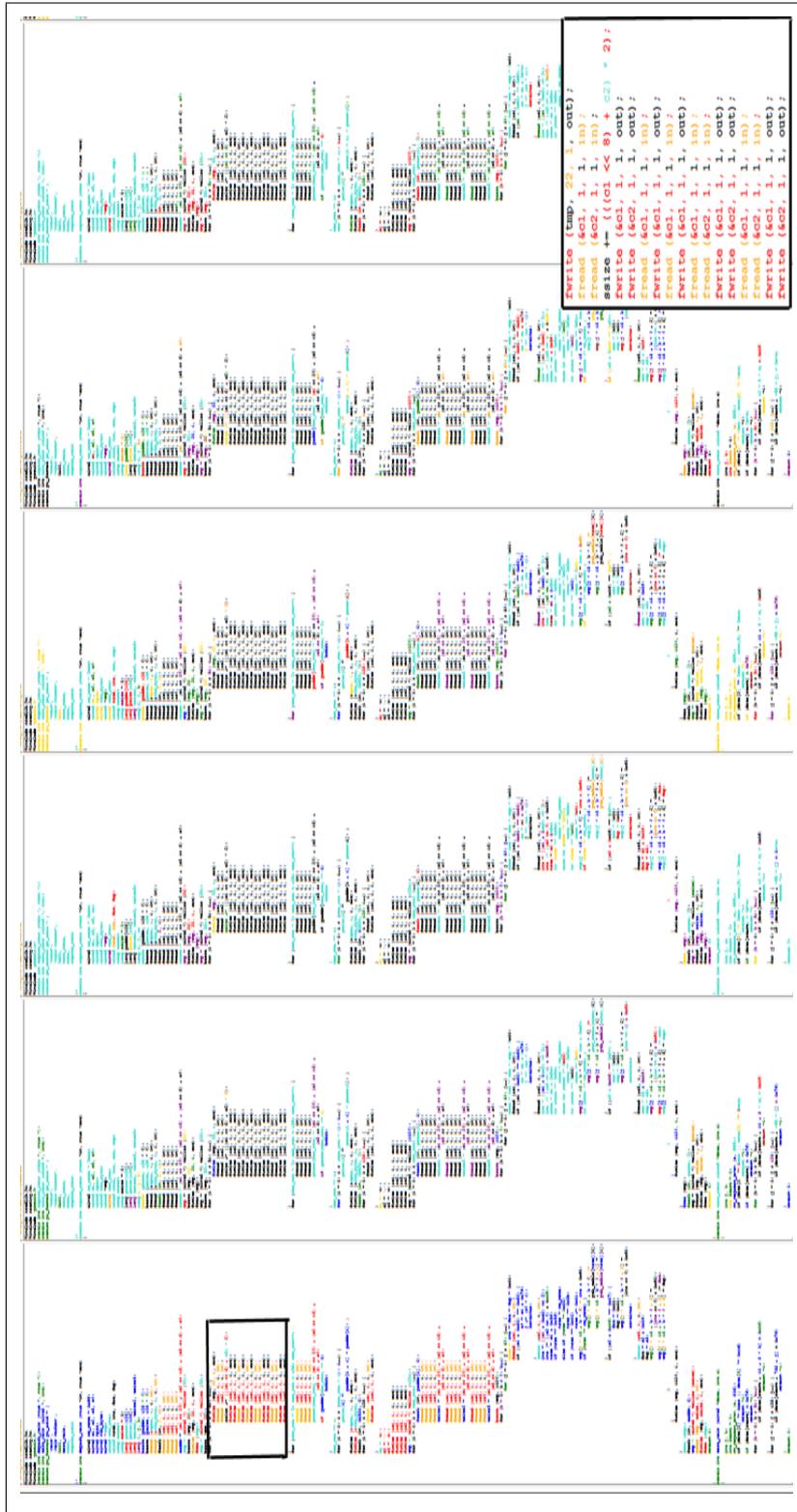


Figure 12.11: This is an edited file which has not been split. Looking at the first column of the .xml file, the file might appear to be split to the "red" file. However, there are two things which prompt further checking. First, the red/orange code is repetitive (see insert), and second, there are 17 other files in which all or most of the code appears, indicating incidental similarity.

File type	.c	.h	Total
Split	130	64	194
Not	146	47	193
Total	276	111	387

Table 12.3: Analysis of file type and classification of candidate split files

12.4.1.3 Not classified

In discussing her work with the project PostgreSQL, Zou states that “*Cases of structural changes can be so complicated that they are hard to detect even by manual examination.*” [261, p.23]. This is also true for some of the examples taken from the 89 projects, where not all of the files selected by filtering could be classified with confidence; these examples were excluded from the dataset. The file `config_MUSENKI.h` is one such example.⁸ This is difficult to classify by manual comparison of the files in the candidate group. The code colouring shows several small groups of lines which appear in only one of the target files. However, the overlaps between blocks of code in different files make the relationship between them confusing.

This example contains many of the features which made it difficult to classify files. The factors include having a large number of target files, having high incidental similarity, and the code which has been edited or removed is scattered throughout the candidate file. Also excluded from the dataset are files of examples or tests, and duplicated files.

12.4.1.4 Dataset composition

Table 12.3 shows the composition of the resulting dataset. Of the 387 instances, 194 are classed as split (130 .c and 64 .h); and 193 are negative examples (146 .c and 47 .h); so that the dataset is nearly balanced. File sizes vary from 5 to 9377 lines and there are between 1 and 19 target files in the comparison groups. An analysis of the number of target files in the groups is shown in Figure 12.12. More than half of the candidates have only one target file, approximately a quarter have two, few have more than eight.

⁸http://homepages.stca.herts.ac.uk/~gp2ag/xmls/config_MUSENKI.xml

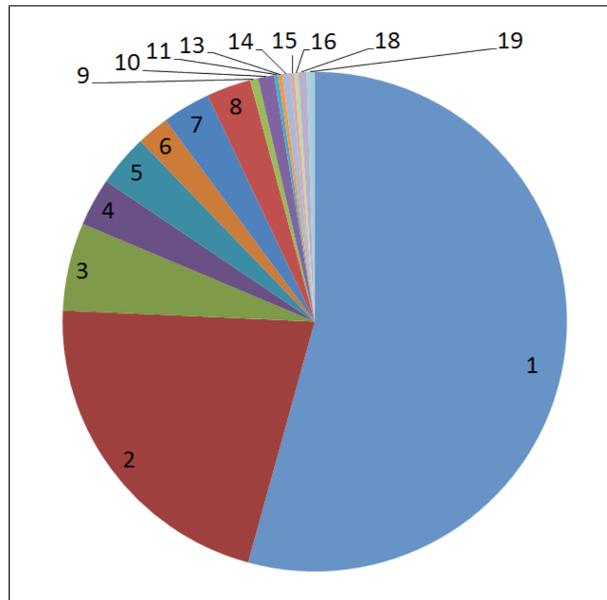


Figure 12.12: Analysis of the number of targets in the split file candidate groups

The x-axis of the graph in Figure 12.13 is labelled with the projects for which filtering selects candidate split files. The bars show the total for each project, with the composition represented by colours. Files classified as split are in red, as non-split in blue, and the unused files are in grey. In each case, the .h files are shown in a paler shade than the .c files. Fifty-five of the 89 projects have candidate split files and of these, the majority, 41, have fewer than 7 candidates.

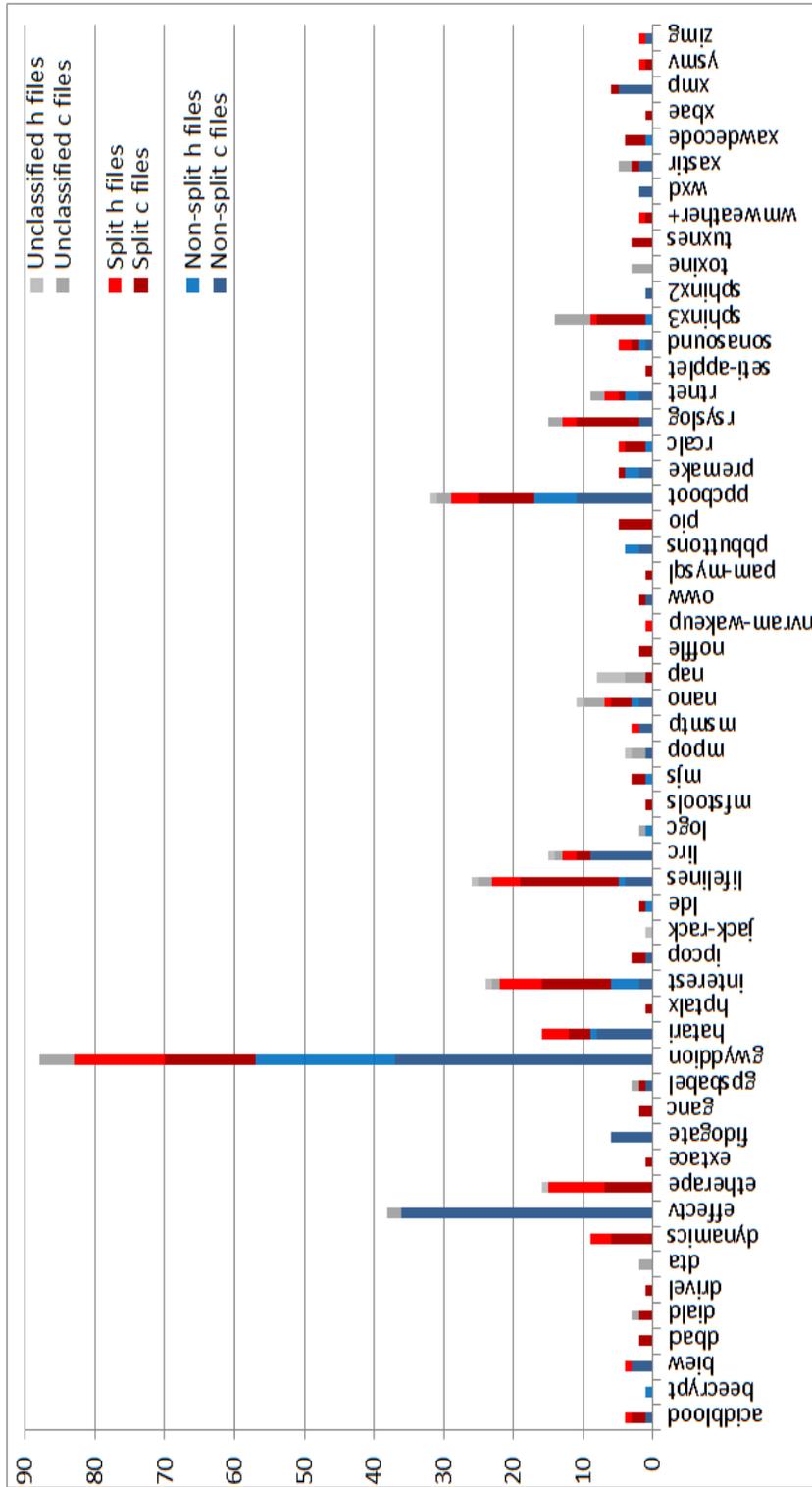


Figure 12.13: The number of candidate files shown by project, with split files in red, non-split in blue and unclassified files in grey. In each case .c files are represented by a darker colour than .h files.

12.4.2 Disappearing file dataset

The disappearing files were found by scanning file size vectors. Possible destinations for the code from disappearing files were selected based on similarity. Files with an identical target file are assumed to be renamed or moved files. Manual classification of a test subset of disappearing files found that files with a high similarity to their main target file, all of those above 0.788 and most above 0.55, had been renamed or moved; and that files with low similarity, all of those below 0.0925 and most below 0.15, were incidentally similar, see Figure 12.14. Adding a margin to these figures to allow for unusual cases, files with a target similarity of at least 0.85 are automatically considered to be moved or renamed files. Those with a similarity of 0.05 or less are considered to be incidentally similar. The remaining set of disappearing files (the uncertain set) were those to which machine learning was applied. The files were partitioned into two groups: those with just one target file and those with more. When there are at least two target files, one of three classes are assigned to the disappearing files.

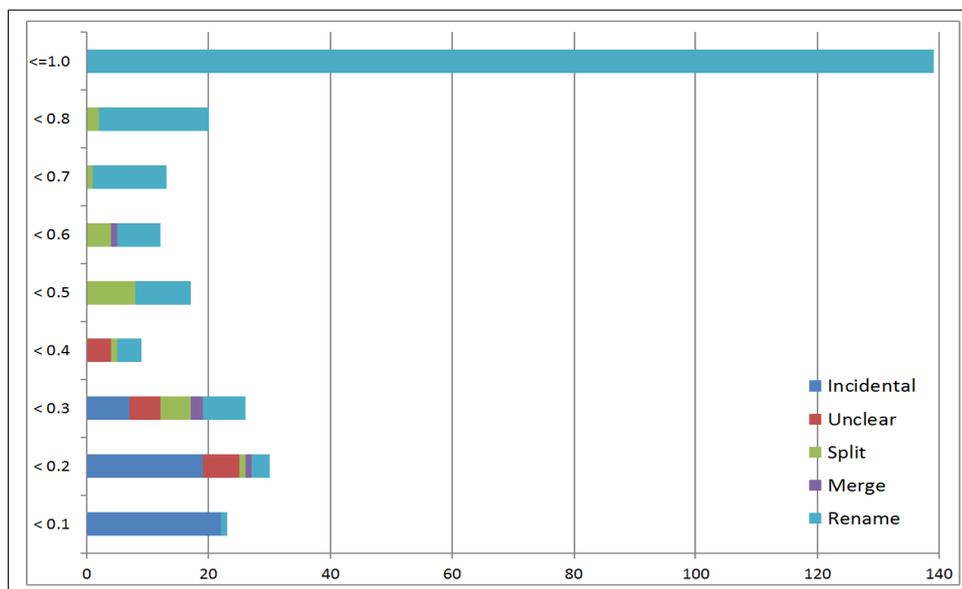


Figure 12.14: Classification of the test set of disappearing files. The y-axis shows the range of similarity between the file and the most similar file to it in the system. The x-axis shows the number of files in the range. Classifications are shown by the colours in the bars.

One class is split, another is rename, which includes moves and merges, and the last is that no related file is found. The group with only one target file cannot be split and therefore has only two possible classes.

12.4.2.1 Renamed, moved or merged files

Two examples are shown in this section. One from `gwyddion`, `libgwyprocess.h`, on the left of Figure 12.15 has been moved and edited. The new file `gwyprocess.h`, on the right, has the same code, with five new lines.

The other file, on the left of Figure 12.16 is `sysexits.h`, a file which disappears from the `fidogate` system. The file is compared to `fidogate.h`, an existing file, and the only target. All of the code from `sysexits.h` is also in `fidogate.h`. There are two possible reasons for this, either the files have merged, or the code was already in `fidogate.h`, and `sysexits.h` is deleted to remove the duplication from the system. The file on the right is `fidogate.h` in the next release, and it is compared to `sysexits.h` (in blue) and to `fidogate.h` in the previous release (in red), showing their merge.

12.4.2.2 Other classes of disappearing files

Disappearing files which have split, or are incidentally similar to their targets, are like those discussed in the section on split files. Examples of disappearing files which have split,⁹ and one which is incidentally similar to the target files¹⁰ are available online.

12.4.2.3 Dataset composition

Apart from 997 files moved to new directories in the project “`mkcdrec`”, there are 1,893 files which disappear from the projects. Of these, 525 are matched by other files in the system, 322 identically. Among the 525 matched files, 21 are ambiguous because two or more of the most closely matched files are identical,¹¹ otherwise the matches are the same as made

⁹<http://homepages.stca.herts.ac.uk/~gp2ag/xmls/player.xml&protos-7.xml>

¹⁰<http://homepages.stca.herts.ac.uk/~gp2ag/xmls/indatedlg.xml>

¹¹16 with 2 identical targets, 4 with 3, and 1 with 7

```

#ifdef __GWY_GWYPROCESS_H__
#define __GWY_GWYPROCESS_H__
#include <libprocess/datafield.h>
#include <libprocess/dataline.h>
#include <libprocess/interpolation.h>
#include <libprocess/simplefft.h>
#include <libprocess/cwt.h>
G_BEGIN_DECLS G_END_DECLS
#endif

#ifdef __GWY_GWYPROCESS_H__
#define __GWY_GWYPROCESS_H__
#include <libprocess/datafield.h>
#include <libprocess/dataline.h>
#include <libprocess/interpolation.h>
#include <libprocess/cwt.h>
#include <libprocess/correl.h>
#include <libprocess/correlation.h>
#include <libprocess/filters.h>
#include <libprocess/ractais.h>
#include <libprocess/grains.h>
#include <libprocess/simplefft.h>
G_BEGIN_DECLS G_END_DECLS
#endif

```

Figure 12.15: The file on the left, gwyddion/libgwyprocess.h, has disappeared. Its code, along with 5 new lines is in the new file gwyprocess.h, shown on the right. Two other files, responsible for the green, purple and black code, are incidentally similar to the disappearing file.

```

#include "config.h"
#include "paths.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <unistd.h>
#ifdef OS2
#include <io.h>
#include <process.h>
#endif
#include <ctype.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <errno.h>
#include <dirent.h>
#ifdef DO_HAVE_SYSEXITS_H
#include <sys/exits.h>
#else
#define EX_OK 0
#define EX_USAGE 64
#define EX_DATAERR 65
#define EX_NOINPUT 66
#define EX_NOHOST 68
#define EX_UNAVAILABLE 69
#define EX_SOFTWARE 70
#define EX_OSERR 71
#define EX_OSFILE 72
#define EX_CANTCREAT 73
#define EX_IOERR 74
#define EXIT_OK 0
#define EXIT_ERROR 1
#define EXIT_BUSY 2
#define EXIT_CONTINUE 3
#define EXIT_KILL 32
#define TRUE 1
#define FALSE 0
#define OK 0
#define ERROR (-1)
#define EMPTY (-1)
#define INVALID (-1)
#define WILDCARD (-2)
#undef _toupper
#undef _tolower
#undef toupper
#undef tolower
#define _toupper(c) ((c)-'a'+'A')
#define _tolower(c) ((c)-'A'+'a')
#define toupper(c) (islower(c) ? _toupper(c) : (c))
#define tolower(c) (isupper(c) ? _tolower(c) : (c))
#define exit(x) fidogate_exit(x)
#include "declare.h"
#include "node.h"
#include "packet.h"
#include "structs.h"
#include "prototypes.h"

```

Figure 12.16: On the left, the file sysexits.h is compared with fidogate.h, which shares all of the code. On the right, fidogate.h is the base file, code in its previous version is red, and sysexits code is blue, showing a merge with no overlapping code.

by visual inspection. Another 443 are unmatched, leaving 925 for which the destination of the code is uncertain.

Note that the “uncertain” dataset reported in Table 12.4 is the result of filtering with the improved filter criteria described in Chapter 15. The number of disappearing files is unchanged whatever filter is used. Matched files also remain the same here. The differences between the original and new filtering criteria mean that more marginal target files are selected, so that more of the files (142 .c files and 38 .h files) fall into the uncertain rather than the unmatched category.

As with the split files, some of the files selected by filtering are excluded. Most of these are examples or tests, with a few files which could not be classified with confidence. Of the remaining 752 files, 177 have 1 target file, 575 have more; 478 are .c files and 274 are .h. These are mostly unrelated and renamed files, 360 (47.9%) and 333 (44.3%) respectively, with only 59 (7.8%) split files. There are up to 99 target files in a candidate group.

12.5 Summary

In this chapter, the collection, selection and preprocessing of files is described along with their filtering to find two different sets of candidate groups. The composition of these datasets by project, file type and class are described. In the next chapter, the features constructed for these datasets as input for machine learning are explained. These features are derived from comparison of the files by the tools described in Chapters 6 and 11.

File type	2 or more target files			1 target file			Totals		
	.c	.h	Total	.c	.h	Total	.c	.h	Total
Unrelated	158	97	255	73	32	105	231	129	360
Renamed	175	86	261	28	44	72	203	130	333
Split	44	15	59	-	-	-	44	15	59
Unclassified	116	21	137	24	12	36	140	33	173
Total	493	219	712	125	88	213	618	307	925

Table 12.4: File type and classification of “uncertain” disappearing files

Chapter 13

Feature Construction

This chapter explains how the features for classifying evolving files are constructed. These features are used to build machine learning models which aim to classify candidate split files as positive or negative examples, and to classify disappearing files as split, renamed, or deleted.

Feature construction is a two-stage process in this research. In the first stage, pairs or combinations of files are selected from those in the candidate group. Second, features are constructed by comparing the pairs or combinations of files generated in the first stage.

The main idea is to build sets of features based on each comparison tool, and investigate the predictive power of the sets, both individually and in combination. The goal is to find a set of features which is simple to compute while providing good discrimination between the classes.

The two tasks, classifying split and deleted files, are related in that both look at the movement of code from one file to another, and consider how the group of files is related. Assessing split files is possibly the more difficult task because it is uncertain whether the code removed from the file is significant, whereas a file which disappears is automatically of interest.

The features were originally developed to classify split files, and explanations in this chapter generally refer to split files. However, because the task of classifying a disappearing file as renamed, split or deleted is similar, the features are also used to classify disappearing files.

When deciding whether a file is split, questions asked about the relationship between files, and the code they contain, include:

- Are blocks of code missing from the candidate file in the next release?
- Are the blocks large enough to be interesting?
- Do these blocks appear in other files?
- How similar are the associated files?
- What proportion of the files are matched?
- How is the matched code distributed within one file?
- How is the matched code distributed among the group of files?

The features constructed for this data aim to answer such questions about the files in a group and their interaction, and so provide suitable information for the file classification tasks.

There are two parts to the chapter. Section 13.1 looks at the way that files in a candidate group are selected and combined ready for comparison. Section 13.2 describes the features constructed from the results of comparing the files using the tools described in Chapter 11.

13.1 File combinations and comparisons

Two candidate groups were introduced in Chapter 12. Split file groups consist of a candidate split file, an amended version of this file in the next release, and a group of target files. Disappearing file groups are just the disappearing file and a group of target files.

The files in a group are compared in two ways. Either pairwise comparisons between selected single files, described in Section 13.1.1, or comparisons between the candidate file and various combinations of the other files in the group, explained in Section 13.1.2.

13.1.1 Comparing single files.

The three single files compared pairwise are the candidate file, the amended version of the file and the ‘main’ target file, or, for disappearing file groups, the two main targets. The ‘main’ target file is the one selected under some similarity measure as most likely to contain the majority of the code removed from the candidate file. A straightforward approach is taken to selecting target files in the experiment reported in Chapter 14. Potential target files are ranked by the change in similarity of the target file, from release n to $n+1$, to the candidate file in release n [$Sim(C_n, T_{n+1}) - Sim(C_n, T_n)$].

This selection has a bias towards choosing new files over existing files. Based on earlier findings [93], if a file has been restructured and there is a similar new file in the system, it is likely to result from the restructuring.

Figure 13.1 illustrates the selection of the main target file by change in similarity score. The example candidate group has four target files, shown on the left of the diagram. Of these, two are new, and two are existing files. These files are shown with two similarity values, chosen to illustrate the selection criteria. The similarity values are labelled “similarity up from x to y ”, where $x = Sim(C_n, T_n)$, (same-similarity); and $y = Sim(C_n, T_{n+1})$,

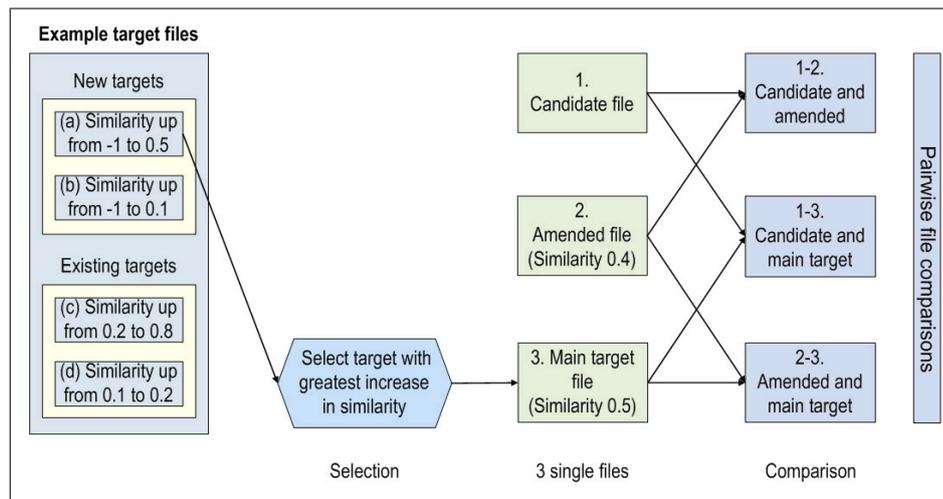


Figure 13.1: The target file with the largest increase in similarity to the candidate file is selected as the main target file. Pairwise comparisons are then made between the candidate, amended and main target file.

(consecutive-similarity). When a file does not exist in release n , it is assigned a similarity value of -1, assuring the selection of new files over existing files. This is just one possible strategy, which could be altered; for example, by reducing the bias, or by removing it altogether.¹

The selected target file, the candidate file and the amended file are then compared to each other. Comparisons between the three files as a group are explained in Sections 13.2.1.2 and 13.2.2.1, and are not depicted here.

13.1.2 Comparing the candidate and concatenated files

The other approach to file comparison looks at the candidate file and various groups of files taken from the amended and target files. Each group of files is concatenated to make one file, which is compared to the candidate. Five combinations are considered below, with the target files which would be selected from the example set, files a–d in Figure 13.1, noted in brackets:

- A The amended file and the main target file (a),
- B the amended file and all target files (a, b, c and d),
- C the two files among the group most similar to the candidate (a and c),
- D the amended file and all new target files (a and b), and
- E the amended file and the target file most similar to the candidate file (c).

One example of a comparison between the candidate file and a concatenation is shown in Figure 13.2. All of the new target files and the amended file are combined (concatenation D); the file resulting from the concatenation is compared to the candidate file. At the end of this Section, on page 202, Figure 13.5 illustrates all of the concatenations and comparisons.

There are two reasons for combining the files. First, comparisons between the candidate file and the concatenations can provide useful information. For example, if a file is simply split to produce an amended file and one or more new files, a high proportion of the original file's code is likely to appear in a concatenation of the new and amended files.

¹Other similarity measures can be used for selection, alternatives are explored in Chap. 15.

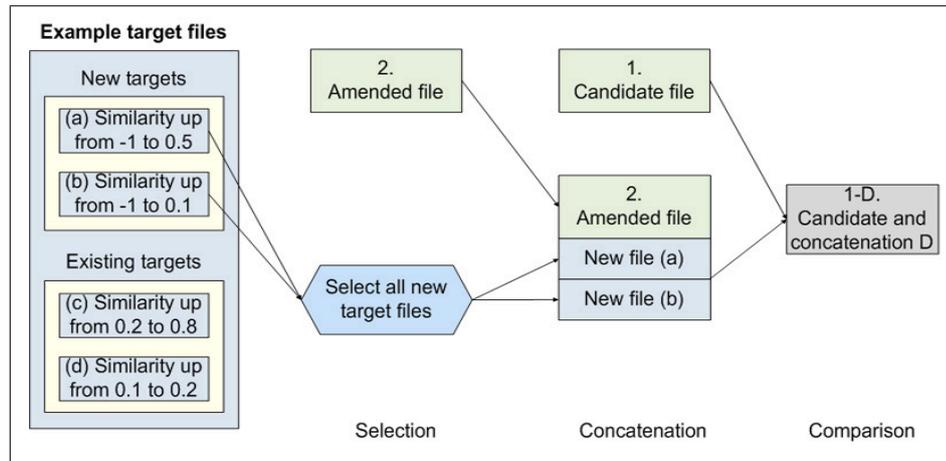


Figure 13.2: An example concatenation, that of all the new target files with the amended file, and its comparison to the candidate file.

Two examples are shown in Venn diagram form in Figure 13.3. On the left is the file fa.c, introduced on page 122, which is split three ways. The similarity between each of the three targets and the candidate file is around 0.3. The similarity between the candidate file and the concatenation of the three files resulting from the split is 0.666. The containment of the candidate file trigrams in the concatenation is 0.89, a high value reflecting the fact that most of the candidate file code appears in one of the three files.

A contrary example is pictured on the right of Figure 13.3. Imagine that instead of being split, a file has been edited, so that, as in the previous example, the amended file has a similarity of 0.3 to the candidate. The two target files for this candidate also have similarities of around 0.3. However,

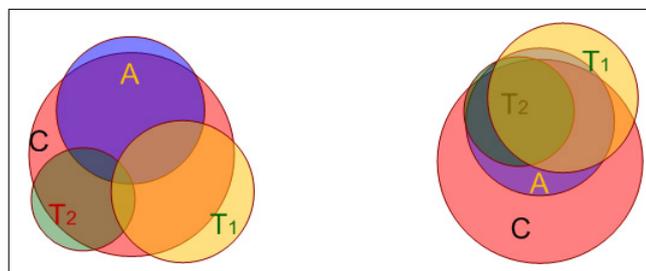


Figure 13.3: In each diagram, the amended file (A, in blue) and two target files (T_1 , T_2 , in yellow and green) have a similarity to the candidate file (C, red) of around 0.3. The file on the left is split, the one on the right is not.

most of the code common to the candidate and target files is a subset of that shared by the amended and candidate files. In this case, the containment of the candidate in the concatenation is about the same as the containment of the candidate in the amended file.

Another example, in Figure 13.4, shows a file split into three. One part forms the amended file, the others are placed in existing files. Although the similarity between the candidate file and a concatenation of the two target files with the amended file will not be as high as the fa.c example, the containment of the candidate file in the concatenated file should be high.

The second reason for combining files is that as the number of target files in the groups varies, the concatenations provide a standard number of comparisons of the set of target files, regardless of their number, while including all members of the group in at least one comparison.

The combinations were chosen with two aims. First to select combinations of files which are more likely to be the targets of the code removed from the candidate file, such as groups A (amended and main target files), C (the two files with the highest similarity) and E (the target file with the highest similarity and the amended file). Two of the groups, A and E, pair the amended file with one target file. These two groups are aimed at simple splits where code is moved from one file to another, such as cnp-1.c where some of the code is moved to the new file fact.c. Group C is similar, but

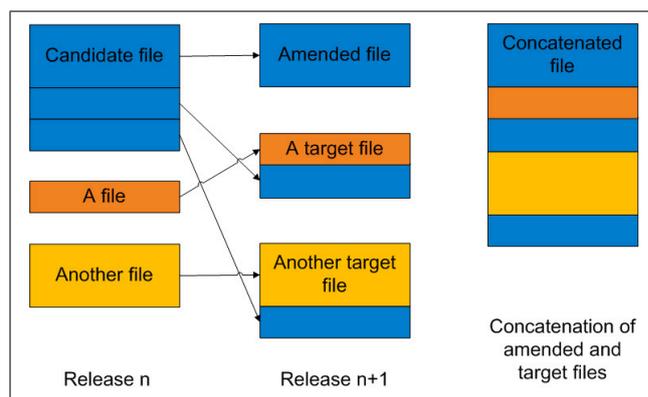


Figure 13.4: A file with two sections split out to two existing files. A concatenation of the resulting files should contain the majority of the code from the original candidate file.

allows for two recipients, like the example in Figure 13.4.

The second aim is to include as many of the true targets as possible for a multiway split. Group B, where the amended file is concatenated with all potential target files, is motivated by the fact that a file may be split into more than two target files, and that it may not always be the most similar files which are the true targets. Group D is similar, but is based on the assumption that new files are likely to be true target files.

There is a disadvantage to this approach. With just one or two target files, the groupings cause some duplication of features when the feature sets are combined. For example, if there is just one new target file, then all concatenations are the same, and consist of the amended file and the target file. With a single existing target file, all except concatenation D are the same, being the amended file and target file. D will simply be the amended file, thus repeating the single file comparison between the candidate and amended files. This duplication is a cost of including all target files in the combinations, and may sometimes affect classification. For example, where random subsets of features are selected in creating classifiers.

Figure 13.5 shows the file groups, with the same four target files as Figures 13.1 and 13.2. Target file selections are shown in the second column of the diagram. As in Figure 13.1, the file with the largest change in similarity is selected as the main target. This means that the new file (a) is chosen, although the similarity of the existing file (c) is higher. To balance this method of selection, two of the groups, C and E, look instead for files with the highest similarity to the candidate file, regardless of previous similarity; in this case file (c) with a similarity of 0.8 is chosen ahead of file (a) at 0.5.

The lower part of the third column in Figure 13.5 shows the file combinations. Each file group is concatenated into a single file. The last column shows the pairs of files passed to the similarity tools for comparison. There are eight file pairs, three pairwise combinations of the candidate, amended and main target files, with the other pairs made up of the candidate file and each of the five concatenations. The information output by the tools is used to construct features which are explained in the next section.

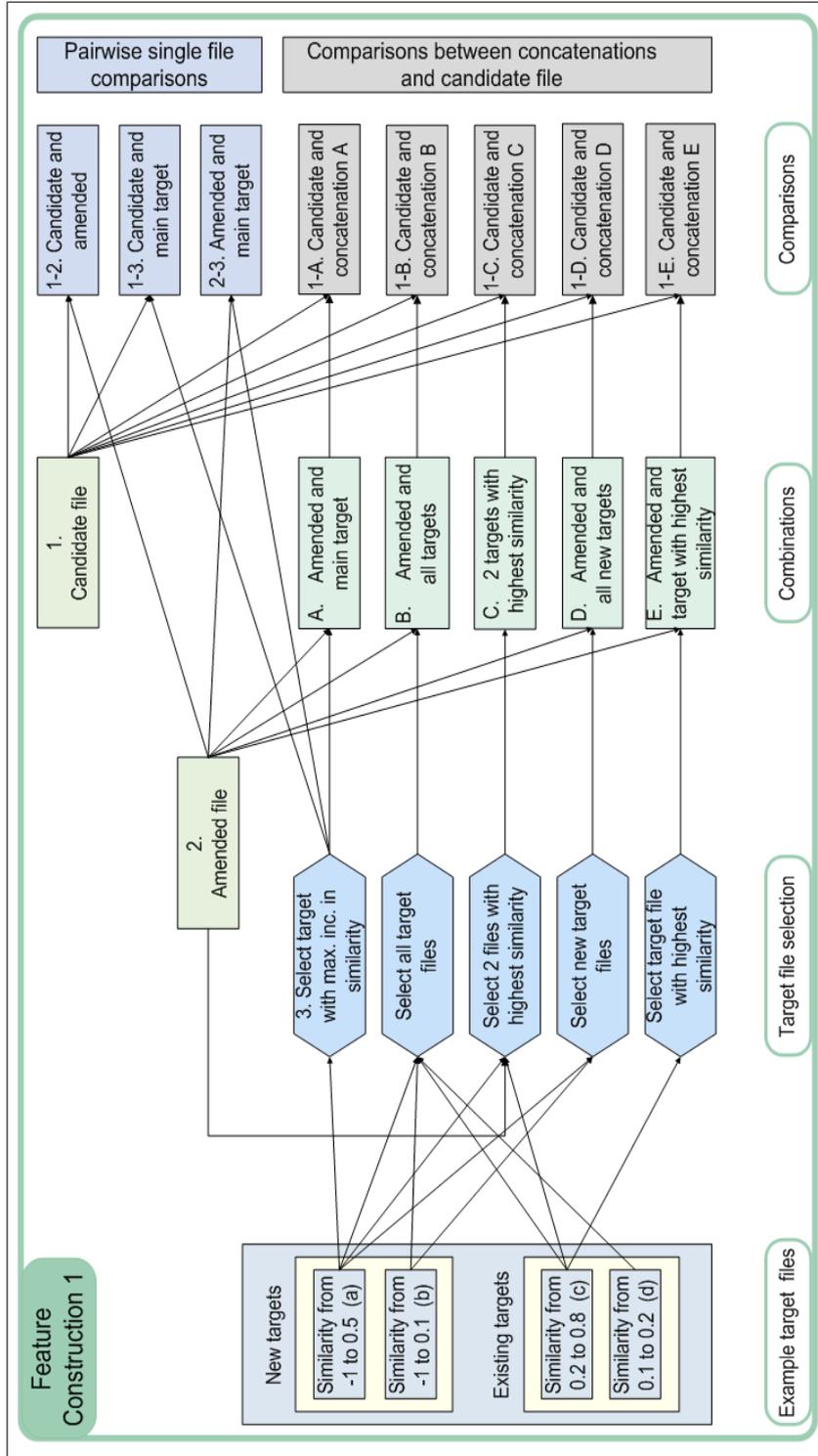


Figure 13.5: File combination and comparison, the example candidate group has 4 target files, 2 new and 2 existing. This diagram shows the selection of the main target file and the single file comparisons at the top. It also shows the criteria for file selection for the concatenated files which are then compared to the candidate.

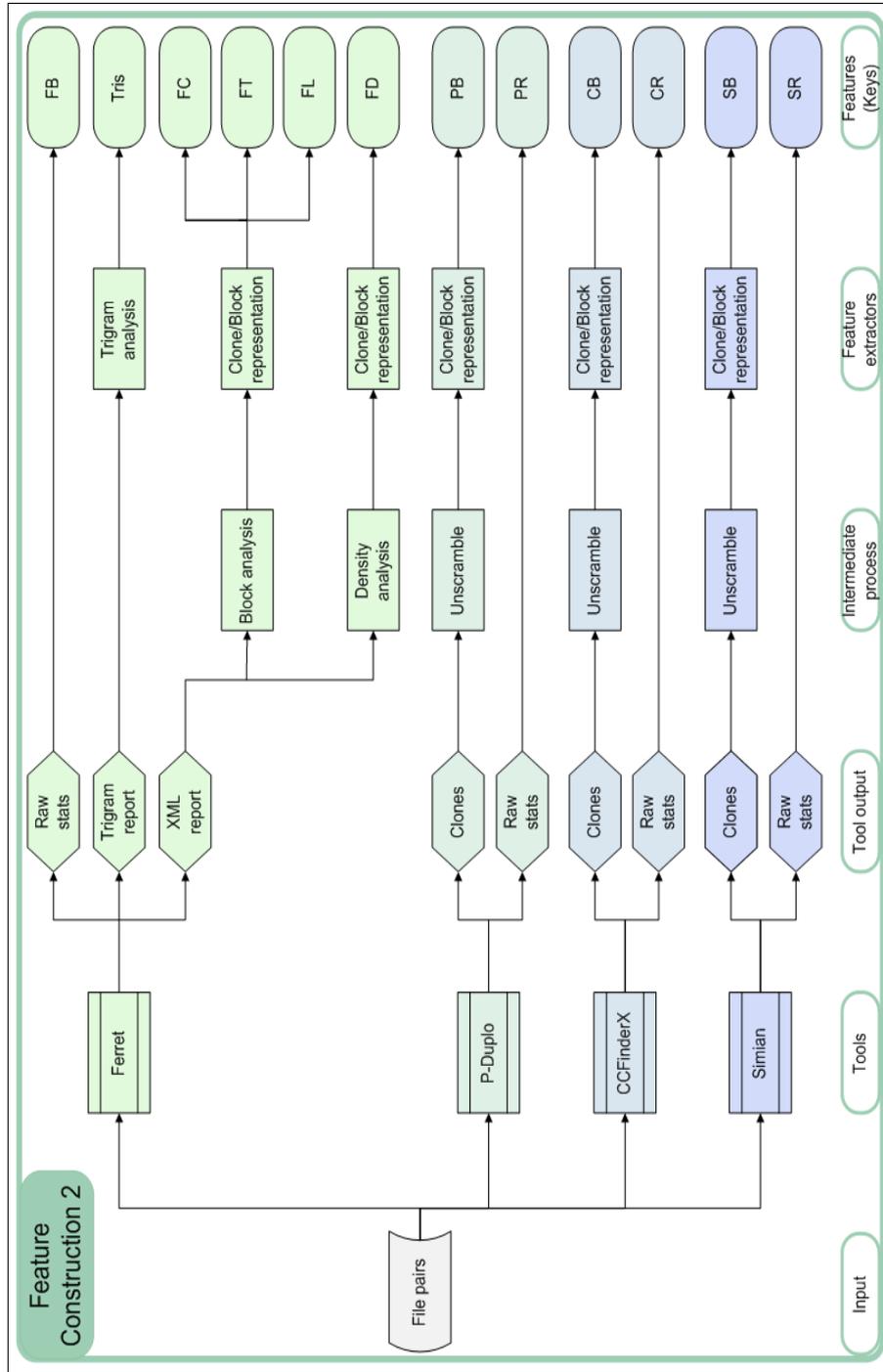


Figure 13.6: Features are constructed by comparing the candidate file groups with each of the tools and analysing the output. The meanings of the abbreviated feature group names, such as FB or SR, are listed in Table 13.1.

13.2 Features

As noted in Chapter 4, the choice of features is important in machine learning. It can be difficult to make a suitable selection when the possibilities are not well constrained. In general, for restructured files, the features should aim to describe the relationships between the contents of the selected files. In practice, as discussed in Section 12.4.1.1, file sizes, the amount of code moved, and the complexity of its distribution vary widely, making the choice of descriptive features difficult.

The second stage in feature construction is explained in this section and illustrated in Figure 13.6. The eight file pairs described in the previous section are compared using the four tools Ferret, P-Duplo, CCFinder and Simian. The outputs from the tools are analysed to provide twelve sets of features, represented in the diagram by keys, which are listed in Table 13.1.

There are two groups of features, raw and block-based. Raw features, described in Section 13.2.1 are taken directly from the output of each tool. Block-based features, explained in Section 13.2.2, are constructed by analysing the clones or matched blocks reported by each of the tools.

13.2.1 Raw feature sets

Among the five raw sets of features, four sets are derived from the statistics reported by each of the four tools. The fifth set comes from the Ferret trigram-file index; a little analysis is needed to construct this set, but as the seven other feature sets have much in common, being based on blocks or clones, the trigram features are grouped with the four raw sets.

Key	Raw features	Key	Block-based features
FB	Ferret Basics	FC, FT, FL	Ferret XML Blocks (measured in characters, tokens, lines)
Tris	Ferret Trigrams	FD	Ferret Dense Blocks
PR	P-Duplo Raw	PB	P-Duplo Blocks
CR	CCFinder Raw	CB	CCFinder Blocks
SR	Simian Raw	SB	Simian Blocks

Table 13.1: Keys and names of the feature sets

Raw features are included because they are simple to collect and because several of the approaches to origin analysis reviewed in Chapter 3 use measures taken directly from similarity detection tools. Also, in earlier experiments [95], 2 of the 32 features chosen by a selection algorithm (correlated feature subset selection [101]) from a set of over 2,000 were based on the direct output from a clone detection tool. These 2 features are ratios of the code in all of the clones to the total in a file. As previously discussed (Section 11.5, p. 164), the amount of code in these clones can be greater than that in the file, because of the many-to-many matching used in clone detection. It is possible that a large number of such clones indicates a file ripe for code abstraction, which increases its chance of being split.

13.2.1.1 Ferret basics

Features constructed from the raw output from Ferret are called Ferret basics. The example of Ferret's default output in Table 13.2 lists file names, the number of trigrams in each file and shared by the files, similarity score, and containment of each file in the other. All of these measurements are used directly as features. The proportion of trigrams in the non-candidate files to the candidate file is also calculated, as is the ratio of the target file trigrams to those in the amended file.

13.2.1.2 Ferret trigrams

The Ferret trigram-file index, an example of which can be seen in Figure 6.3 on page 90, lists all of the trigrams in the files, and shows which files contain each trigram. This index is analysed to create the features listed in

file 1	file 2	common trigrams	file 1 trigrams	file 2 trigrams	similarity	cont't 1 in 2	cont't 2 in 1
cnp-1.c	cnp-2.c	142	179	151	0.755319	0.793296	0.940397
cnp-1.c	fact.c	35	179	35	0.195531	0.195531	1.000000
cnp-2.c	fact.c	2	151	35	0.016393	0.019868	0.085714

Table 13.2: Basic Ferret output for the three example files (cont't - containment)

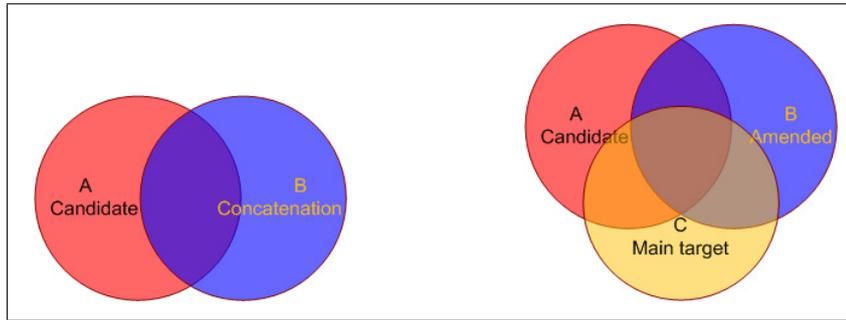


Figure 13.7: The features in Table 13.3 are based on these Venn diagrams, representing comparisons between the candidate and a concatenation (left), and between the candidate, amended and main target files (right).

Table 13.3, where A, B, and C refer to the files shown in the Venn diagrams in Figure 13.7. In concatenation comparisons, file A is the candidate file and file B the concatenated file. File A is also the candidate file in three-way comparisons, while file B is the amended file, and file C the main target file.

Nine features are constructed from the comparisons between the candidate file and each of the concatenated files. These features include the number of trigrams unique to each file; the proportion of these to the total in the file; and the proportion of shared trigrams to the total in each file.

Features	Trigrams in	
	2 files	3 files
In one file	A, B	A, B, C
In amended and target files		$B \cup C$
Shared by all	$A \cap B$	$A \cap B \cap C$
Shared by two of three		$A \cap B \setminus C, A \cap C \setminus B, B \cap C \setminus A$
Shared by candidate and one other file		$(A \cap (B \cup C)) \setminus (B \cap C)$
Cand. trigrams not shared by one file		$(A \setminus (B \cup C)) \cup (A \cap B \cap C)$
Unique to one file	$A \setminus B, B \setminus A$	$A \setminus (B \cup C), B \setminus (A \cup C), C \setminus (A \cup B)$
Unique proportions	$\frac{A \setminus B}{A}, \frac{B \setminus A}{B}$	$\frac{A \setminus (B \cup C)}{A}, \frac{B \setminus (A \cup C)}{B}, \frac{C \setminus (A \cup B)}{C}$
Shared proportions	$\frac{A \cap B}{A}, \frac{A \cap B}{B}$	$\frac{A \cap B}{A}, \frac{A \cap B}{B}, \frac{A \cap C}{A}, \frac{A \cap C}{C}, \frac{B \cap C}{B}, \frac{B \cap C}{C}$
Shared by all to each file		$\frac{A \cap B \cap C}{A}, \frac{A \cap B \cap C}{B}, \frac{A \cap B \cap C}{C}$
Proportion of candidate shared by one file		$\frac{(A \cap (B \cup C)) \setminus (B \cap C)}{A}$
Proportion of candidate shared by any file		$\frac{A \cap (B \cup C)}{A}$

Table 13.3: Trigram-based features for comparisons between the two and three files. A, B and C refer to the files shown in Figure 13.7.

The remaining three features repeat information found in the Ferret basic set: the shared trigrams and the trigrams in each file. However, because this research explores the comparative classification rates of the different feature sets, the features are also included in this set.

The three-way comparison between the candidate, amended and main target files provides twenty-four features which mostly represent relationships between the three files. The features in the set are split into two types: counts of trigrams in various combinations of files and the proportions of these counts to the total in one or more files in the combination.

The counts include the numbers of trigrams in each file, unique to one file in the group, shared by each pair of files, and shared by all three files. Other count-based features are the sum of the trigrams in the amended and target files, the number of trigrams shared by the candidate and just one other file, and this figure subtracted from the total in the candidate file.

Proportional features relate the counts to the total of one of the files compared. For example, the number of trigrams shared by the candidate file and just one of the other two files to the number of trigrams in the candidate file. For a simple split, with little incidentally similar code, this figure should be close to one. In general, the proportion should be high for two-way splits, although it will be reduced by editing or by code common to the three files.

Some of the proportional features echo the basic Ferret features. For example, the proportion of trigrams from the candidate file which appear in either of the other two files is the same as the containment of the candidate file in the concatenation of the amended and target files. Most features, such as the proportion of a file covered by the trigrams which are unique to that file, are not calculated elsewhere.

13.2.1.3 Simian, Code Clone Finder and P-Duplo

The features constructed from the clones reported by each of the clone-detection tools are:

- the number of clones of 5 minimum sizes,
- proportions of lines/tokens in the above to the size of file 1, and
- proportions to the size of file 2.

Both file sizes and the amount of code moved vary, making it difficult to choose one clone size to fit all situations. A range of minimum clone sizes are set, the specification of which depends on the tool. P-Duplo and Simian are based on line counts and the values chosen for the five minimum clone sizes are 2, 4, 8, 16 and 32 lines.

CCFinder clones are based on tokens instead of lines. Analysis of all of the C code files in the most recent release of each of the 89 projects used in this research shows that the average number of tokens per line is around 7 (mean 6.95, median 10, and mode 7). The CCFinder clones are therefore selected using minimum sizes of 14, 28, 56, 112 and 224 tokens.

An edited version of the file `cnp-2.c`, `cnp-2-edit.c`, is introduced here. This file is the same as `cnp-2.c` except that the variable names “combinations” and “permutations” are changed to “combs” and “perms”. The code is shown with `cnp-1.c` in Figure 13.8 (p. 210). The files are compared to provide examples to show how clone and block sizes are derived from each of the tools.

P-Duplo The raw features from P-Duplo are taken from a summary of the clones. In each file there is one clone of 13 lines (5–17).

```
Lines in file 1: 36
Lines in file 2: 28
Clones: ((5 17) (5 17))
```

Simian The raw Simian features are taken from the summary statistics reported by the tool. An extract from the Simian report for a comparison between the files `cnp-1.c` and `cnp-2-edit.c` is shown below.

```
Found 36 duplicate lines in 4 blocks in 2 files
Processed a total of 43 significant (64 raw) lines in 2 files
```

Simian records the number of duplicate blocks and duplicate lines, which are counted twice, once for each file. Simian disregards ‘includes’, and lines with one character, therefore records fewer total lines than P-Duplo. However in its ‘raw’ count, these lines are included.

With parameterised identifiers, Simian finds two clones in each file, 18 significant lines in total. The first spans lines 3 to 20 in both files, the second, lines 28 to 36 in file 1, and lines 20 to 28 in file 2. The first clone is counted as 14 lines long and the second as 4 lines, as lines with a single brace are not counted. This differs from P-Duplo in that lines with altered identifier names are matched, but single character lines are not counted.

CCFinder The extract from the CCFinder report below is also for a comparison between the files `cnp-1.c` and `cnp-2-edit.c`.

```
source_files {
1      cnp-1.c          208
2      cnp-2-edit.c    166
}
clone_pairs {
1      1.0-99          2.0-99          1      2.0-99          1.0-99
8      1.12-27         2.27-42         8      2.12-27         1.27-42
8      1.27-42         2.12-27         8      2.27-42         1.12-27
2      1.74-92         2.74-92         2      2.74-92         1.74-92
15     1.141-207       2.99-165        15     2.99-165        1.141-207
}
```

This report first gives the number of CCFinder tokens for each file. This number differs from the number of actual tokens, because of the exclusions and transformations which occur during preprocessing by the tool. The location and size of each clone is reported and these are totalled. The total number of tokens in the clones reported here is $100 + 16 + 16 + 19 + 67 = 218$; whereas the unscrambled clones, that is the first and last of the clones, contain $100 + 67 = 167$ tokens.

CCFinder allows the minimum token types for a clone to be specified; this is aimed at users wanting to exclude small repeated structures in the code but is not used in this research where all matches may be interesting.

The “raw” features are listed in Table 13.4. In addition to those described in Sections 13.2.1.1–13.2.1.3, the total units in each of the single files and the five concatenated files measured by each of CCFinder, Simian and P-Duplo, as well as the number of characters, tokens and lines in the file are included; as are the type of file (.c or .h), and the number of target files.

```

1. #include <stdio.h>
2. long factorial (int n);
3. long combinations (int n, int k);
4. long permutations (int n, int k);
5. int main()
6. {
7.     int  setsize, subsetsize;
8.     printf ("Set size? ");
9.     scanf ("%d", &setsize);
10.    printf ("Subset size? ");
11.    scanf ("%d", &subsetsize);
12.    if (setsize < 0 || subsetsize
13.        < 0 || setsize < subsetsize)
14.    {
15.        printf ("Mission impossible\n");
16.        return (1);
17.    }
18.    printf ("%ld combinations and %ld
19.        permutations of %d items
20.        taken from %d\n",
21.        combinations (setsize, subsetsize),
22.        permutations (setsize, subsetsize),
23.        subsetsize, setsize);
24.    return (0);
25. }
26. long factorial (int n)
27. {
28.     long  result = 1;
29.     int  i;
30.     for (i = 1; i <= n; i++)
31.         result *= i;
32.     return (result);
33. }
34. long combinations (int n, int k)
35. {
36.     return (factorial(n) /
37.         (factorial(k) * factorial(n-k)));
38. }
39. long permutations (int n, int k)
40. {
41.     return (factorial(n) /
42.         factorial(n-k));
43. }

```

The file cnp-1.c

```

1. #include <stdio.h>
2. #include "fact.c"
3. long combs (int n, int k);
4. long perms (int n, int k);
5. int main()
6. {
7.     int  setsize, subsetsize;
8.     printf ("Set size? ");
9.     scanf ("%d", &setsize);
10.    printf ("Subset size? ");
11.    scanf ("%d", &subsetsize);
12.    if (setsize < 0 || subsetsize
13.        < 0 || setsize < subsetsize)
14.    {
15.        printf ("Mission impossible\n");
16.        return (1);
17.    }
18.    printf ("%ld combinations and %ld
19.        permutations of %d items
20.        taken from %d\n",
21.        combs (setsize, subsetsize),
22.        perms (setsize, subsetsize),
23.        subsetsize, setsize);
24.    return (0);
25. }
26. long combs (int n, int k)
27. {
28.     return (factorial(n) /
29.         (factorial(k) * factorial(n-k)));
30. }
31. long perms (int n, int k)
32. {
33.     return (factorial(n) /
34.         factorial(n-k));
35. }

```

The file cnp-2-edit.c

Figure 13.8: Files used to provide examples of the blocks or clones found by the tools. Cnp-2-edit.c is an edited version of the original cnp-2.c, with the variable names “combinations” and “permutations” changed.

Tool	Measure	No.	Total
Ferret	Similarity score (simscore) Number of trigrams in each file, and those common to both files Proportion of trigrams in other file to those in the original file Containment of each file in the other Total	1 3 1 2 7	56
Ferret Trigrams Two files - used for cand-concat comparison	Trigrams unique to each file Trigrams shared by the 2 files Total trigrams in each file Proportion of trigrams in only one file to the total in that file Proportion of shared trigrams to the number in each file Total	2 1 2 2 2 9	45
Ferret Trigrams Three Files - used for 3 main files	Trigrams unique to each file Trigrams shared by just 2 files Trigrams shared by all 3 files Total trigrams in each file Sum of trigrams in amended and target files Trigrams shared by the candidate and just one of the other files Trigrams in the candidate file not shared by just one of the other files Proportion of trigrams shared by one other file to total in candidate file Proportion of trigrams shared by any other file to total in candidate file Proportion of trigrams unique to each file to the total in the file Proportion of trigrams shared by two files to totals in each file Proportion of the shared trigrams to totals in each file Total	3 3 1 3 1 1 1 1 1 3 6 3 27	27
All sets (7) CCFX, Sim, PDp FC, FT, FL, FD	Total units in single files 3×7 Units in concatenated files 5×7 Total	21 35 56	56
P-Duplo	The number of clones of min m lines of min n characters For (m n) of (2 2)(4 3)(8 4)(16 5)(32 6) Proportion of lines in the above to file 1 size Proportion of lines in the above to file 2 size Total	5 5 5 15	120
CCFX	The number of clones of min m tokens, m is 14, 28, 56, 112, 224 Proportion of tokens in the above to file 1 size Proportion of tokens in the above to file 2 size Total	5 5 5 15	120
Simian	The number of clones of min m lines For m = 2, 4, 8, 16 and 32 Proportion of tokens in the above to file 1 size Proportion of tokens in the above to file 2 size Total	5 5 5 15	120
Sundries	File type (.c or .h) Number of files in target group	1 1	2
Grand Total			546

Table 13.4: Raw features taken directly from the tools' outputs. All of the features, except the sundries and trigram-based ones, are constructed for the 8 file pairs. Column 3 shows the number of features for each pair of files, and column 4, for all relevant pairs.

13.2.2 Feature sets based on blocks

The other seven feature sets, based on either the blocks from the Ferret XML and density analyses,² or the clones found by CCFinder, Simian or P-Duplo, are similar to each other. Features are constructed by analysing the number of blocks (or clones), their sizes, and relationship to the containing file size. The word *block* is used here to describe matched sections of code.

The two Ferret analyses result in a list of block sizes. The clone-detection tools provide information about matching blocks of code as a list of start and end points from which their sizes are calculated. As described in Chapter 11, for the purposes of this research, the clones are post-processed to approximate one-to-one matching (unscrambled).

As an example, Table 13.5 shows the results of comparing the two files, `cnp-1.c` (file 1) and `cnp-2-edit.c` (file 2), using each of the clone-detection tools. The tool and the unit used to measure clones are in the first two columns of the table. The unscrambled clone pairs are in the third column, where each clone is represented by a start and end (token or line) number (start end), and each clone pair is a file 1 clone paired with a file 2 clone ((f1-start f1-end)(f2-start f2-end)). The clones for files 1 and 2 are separated, as in columns 4 and 5. Any contiguous blocks are merged, such as the blocks (0 99) and (100 165) found in file 2 by CCFinder. The length of each block is

²The results of classification tests to find suitable parameters for density analysis on the data used in this research are listed in Appendix I.

Tool	Unit	Clones file1 file2 ((start end)(start end))	File 1 blocks	File 2 blocks	File 1 block lengths	File 2 block lengs
CCFX	Token	(((0 99)(0 99)) ((142 207)(100 165)))	((0 99)(142 207))	((0 99)(100 165))	(100 66)	(166)
Simian	Line	(((3 19)(3 19)) ((29 35)(21 27)))	((3 19)(29 35))	((3 19)(21 27))	(17 7)	(17 7)
PDuplo	Line	(((5 17)(5 17)))	((5 17))	((5 17))	(13)	(13)

Table 13.5: Clones reported by the three tools when `cnp-1.c` (file 1) and `cnp-2-edit.c` (file 2) are compared. The clones are listed in column 3, with those in each file shown in columns 4 and 5. Contiguous clones are merged, and the lengths of the resulting blocks listed in the last column.

then calculated, giving the list of block sizes in the last two columns.

As this is a small example, the results for the tools are not dissimilar. The main difference between CCFinder, which works with tokens, and the other tools, which are line-based, is one of scale. Other differences are caused by features of the matching algorithms. For example, Simian breaks the blocks in file 2 on the “insignificant” line () between the two sections, but would otherwise have one block in file 2 like the other tools.

As described in Chapter 6, analysis of the Ferret XML report produces a list of block sizes, which can be recorded in terms of tokens, lines or characters. A density analysis of the same report produces a list of dense blocks, usually in tokens, but optionally, either of the other two units.

The results of analysing the Ferret XML report are in Table 13.6. The upper part of the table shows the sizes of the matched blocks in each of the units. More tokens are found than by CCFinder, partly because CCFinder does not account for includes and declarations, and partly because the minimum block size specified for CCFinder comparisons was 14 tokens, which approximates to 2 lines, whereas Ferret, because it works with token trigrams, finds matches of three or more consecutive tokens. The number of lines found by the Ferret analysis falls between those found by P-Duplo and Simian, because P-Duplo ignores lines of fewer than the minimum specified characters, and Simian matches code in which the identifiers are changed, and therefore matches more lines. Character counts can differentiate between short statements, such as “ i = 1 ” and longer, and therefore more un-

Ferret Analysis	Unit	File 1 block lengths	Total	File 2 block lengths	Total
XML:	Tokens	(7 6 9 110 6 18 3 3 4 32 24)	222	(7 9 110 6 18 32 24)	206
	Lines	(13 2 3 3)	21	(13 2 3 3)	21
	Chars	(17 11 16 284 21 58 6 8 7 68 49)	545	(17 16 284 21 58 68 49)	513
Density: Parameters: 0.9, 50, 10	Tokens	(166 66)	232	(221)	221

Table 13.6: Blocks resulting from different analyses of the Ferret XML report of the comparison between the files `cnp-1.c` and `cnp-2-edit.c`.

usual sequences, such as “some.long.and.not.so.common.identifier = 2.975”.

The lower part of the table shows the results of density analysis with density 0.9, minimum block size 50, and minimum gap 10. Density analysis provides a means of matching edited code, which is reflected in the results, where more tokens are “matched” than by straightforward matching.

13.2.2.1 The candidate difference set

Trigram analysis provides information about the relationship between the candidate, amended and main target files. To find similar information from the three clone detection tools, a further set of blocks taken from the output of each of the tools is analysed. These blocks are those shared by the main target file and the candidate difference set (defined in Table 13.7), like the examples in Figure 13.9.

The blocks in the candidate file shared by its amended version are:

((20 30)(50 90)(130 150)(180 220))

which means that the blocks in the difference set are:

((0 19)(31 49)(91 129)(151 179))

The blocks shared by the candidate and main target files are:

((10 60)(70 120)(140 220))

Therefore blocks shared by the main target file and the difference set are:

((10 19)(31 49)(91 120)(151 199))

As CCFinder, Simian, P-Duplo, their difference sets, and the XML anal-

Term	Notation	Description
A file	f_a^n	A file a in release n
Amended (revised) file	f_a^{n+1}	Revised version of file a in release n+1
	A^n	The set of trigrams in a file a, release n, f_a^n
	A^{n+1}	The set of trigrams in the revised version of file a, f_a^{n+1}
Difference set	$A^n \setminus A^{n+1}$	Trigrams which disappear from the file in the revised version
Reverse difference set	$A^{n+1} \setminus A^n$	New trigrams in the revised file

Table 13.7: Difference sets and reverse difference sets explained in terms of trigrams, also apply to tokens or lines (as Figure 13.9)

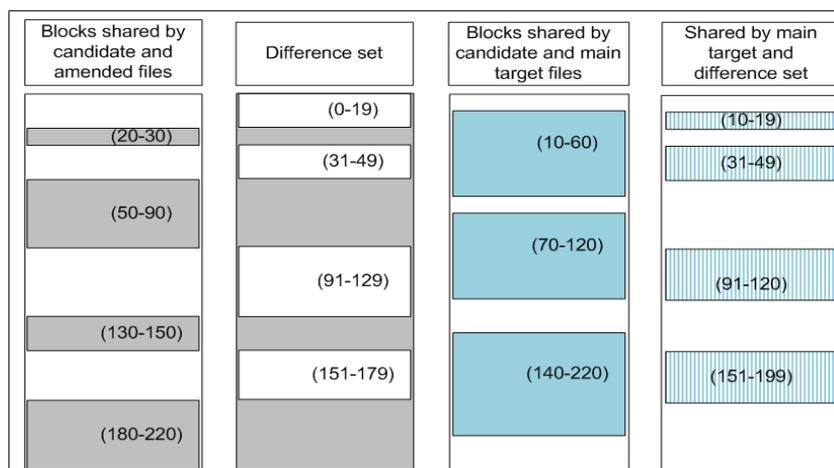


Figure 13.9: In the first representation of the candidate file, blocks shared by the amended file are labelled. In the second, the difference set is shown. The third labels blocks shared by the main target file, and the last, blocks shared by the main target file and the difference set.

yses each produce a list of block sizes, features can be constructed in the same way. Given the block sizes and the file size, there are a number of ways to present the information, which are discussed in the next section.

13.2.2.2 Measurements

Standard machine learning algorithms require feature vectors of uniform length. This requirement can be a challenge when the amount of information to be represented varies between instances. In the split file dataset, the number of blocks in a file varies from one to over a thousand, with maximum block sizes ranging from 2 to around 5,000 lines. This section considers how to present the data in a uniform but meaningful way.

Descriptive statistics summarise such information with measurements based on central tendency (mean, median and mode), dispersion (standard deviation, variance and range) and distribution, which can be given in terms of frequencies.

To construct a frequency distribution it is useful to know two things about the data: the range of values of the variable under consideration and the sample size. For this data, the number of blocks and their sizes are

unknown a priori. There is no standard method of selecting the number of bins in a frequency distribution, although if the sample size is known there are several suggested guidelines, such as Sturges formula, $[1 + \log_2 n]$, where n is the sample size [226].

In the absence of this information, the important thing is to choose sufficient bins to provide some separation between different sized blocks, while not selecting too many, as each extra bin means another set of features ($7 \times 32 = 224$ features here). The spacing between intervals is normally equidistant. For example, evenly distributed intervals might be chosen at 5, 10, 15, ... lines per block. For this data, the importance of blocks of a particular size is unknown, though it can be estimated that the data is likely to be skewed, with more small blocks due to incidental similarity. The intervals chosen are evenly distributed intervals on the \log_2 scale: powers of two, from 2 to 32.

It is not only the absolute size of the blocks which can be important, but also the size of the blocks relative to the size of the file. Therefore another set of intervals was selected based on file size. Powers of two are employed again, with the bin intervals set at $\frac{1}{64}$ to $\frac{1}{4}$ of file size.

Another choice is whether to count the members of each bin or to report the cumulative counts of these blocks. Here the number of blocks is accumulated as the block size decreases. A further choice is how to represent the blocks. This can be a simple count of the blocks or, more expressively, a count of the units within the qualifying blocks.

Measurements can be absolute, or relative to file size. In the context of one file, the only difference between these measures is the scale. However, each type of measure has a role in describing the features of files in general. In characterising a block of copied code, absolute units give the true size of a block, and relative units determine the importance of a block in the file. For example, a block of 25 tokens is a large chunk of a file containing 50 tokens, but is possibly less interesting as part of a file of 5,000 tokens, where this similarity could occur incidentally. However, a block of 20 lines is likely to be significant in any file, even if it is a small part of a very large

file. The ratio of block size to the total copied lines or tokens in a file may indicate whether all or most of the copied code is in one chunk.

13.2.2.3 Block-based features

Table 13.8 shows the measures chosen for the block-based features. The idea was to include both general descriptive and frequency-based features.

Eight features (1-8), describe the blocks in a general way. Four feature sets divide the blocks into five groups, giving cumulative measures; one set (9) gives the total units in blocks of at least 2, 4, 8, 16 and 32 lines (or 14–224 tokens, or 40–640 characters); another set (11) uses fractions of file size for the interval values, $\frac{1}{64}-\frac{1}{4}$. Both sets are also represented as a proportion of file size. Each feature is constructed for both files in a comparison.

To provide an example of the measurements described in Table 13.8, imagine a file of 800 units in which there are 20 blocks. The first block starts at unit 201 and the last ends at unit 700, a spread of 500 units. The sorted block sizes are: (2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 5, 6, 8, 9, 12, 15, 22, 30, 50, 175). The measurements based on these blocks are listed in Table 13.9.

Ref.	Measure	No.
1	Number of blocks	2
2	Number of units in the blocks	2
3	Proportion of (2) to total	2
4	Mean block size	2
5	Proportion of (4) to total	2
6	Largest block size	2
7	Proportion of (6) to total	2
8	Proportion of copied lines to spread	2
9	Cumulative frequency of units in blocks $\geq m$ lines or 7m tokens m=2, 4, 8, 16, 32	10
10	Proportion of (9) to total	10
11	Cumulative frequency of units in blocks \geq (file size / n) n = 4, 8, 16, 32, 64	10
12	Proportion of (11) to total	10
	Total	56
	Grand total = 56 x 7 sets x 8 comparisons = 3136 + 28 x 3 difference sets = 84	3220

Table 13.8: Features constructed from block sizes. Each measurement is for both of the files in the comparison, except for difference set measures.

13.2.3 Final feature sets

The proportions of the different subsets within the full feature set are depicted in Figure 13.10. This diagram shows both the relative sizes of the single and concatenated features, and the Ferret- and non-Ferret-based features in the full set. The proportions of concatenated and single features apply to each subset of features. The non-Ferret features are split evenly between the three other tools. These sets incorporate the raw features which account for around a sixth of the set. The Ferret features are constructed in a different way, so that the raw feature sets are separated from block-based features, and these are about one-eighth of the total number. The block-based feature sets echo those of the three other tools.

The feature sets are listed in Table 13.10. The abbreviated names are hyphenated strings, which for features derived from Code Clone Finder, P-Duplo and Simian generally follow the pattern:

Source	-	Raw/block/both	-	Single/concatenated/both
ccf	-	raw	-	singles
pdp	-	blocks	-	cats
sim	-		-	

The first part of the name shows which tool is used to measure the similarity between files; the second part, whether the measures are direct from the

Block size list:	2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 5, 6, 8, 9, 12, 15, 22, 30, 50, 175	Start 201, end 700.
Ref.	Measure	No.
1	Number of blocks	20
2	Number of units in the blocks	360
3	Proportion of (2) to total units in file	0.45
4	Mean block size	18
5	Proportion of (4) to total	0.0225
6	Largest block size	175
7	Proportion of (6) to total	0.219
8	Proportion of copied units to spread	$\frac{360}{500} = 0.72$
9	Number of units in blocks $\geq m$ lines m=2, 4, 8, 16, 32	400, 340, 321, 277, 225
10	Proportion of (9) to total	0.5, 0.425, 0.4, 0.35, 0.28
11	Number of units in blocks \geq (file size / n) n = 64, 32, 16, 8, 4	292, 255, 225, 175, 0
12	Proportion of (11) to total	0.365, 0.32, 0.28, 0.22, 0

Table 13.9: Features built from the example blocks.

tool (raw), or result from analysis of the blocks (blocks), or both (); and the last part, whether the comparison is between the candidate, amended and main target files (singles), or the candidate and a concatenation of the target files (cats) or both (). For example, the set of features built from the blocks found by Simian when comparing the candidate file with a concatenation of files is called “sim-blocks-cats”, and those built from the raw output of CCFinder with single file comparisons is called “ccf-raw-singles”. When both raw and block-based features are included in a set, neither raw or blocks are stated, for example, pdp-cats means all P-Duplo-based features from comparisons between the candidate file and each concatenated file.

The pattern for Ferret-based features is different, as each set is treated separately, so that its source is either raw or block-based and no distinction is necessary. The trigram and basic similarity information is “raw”, all

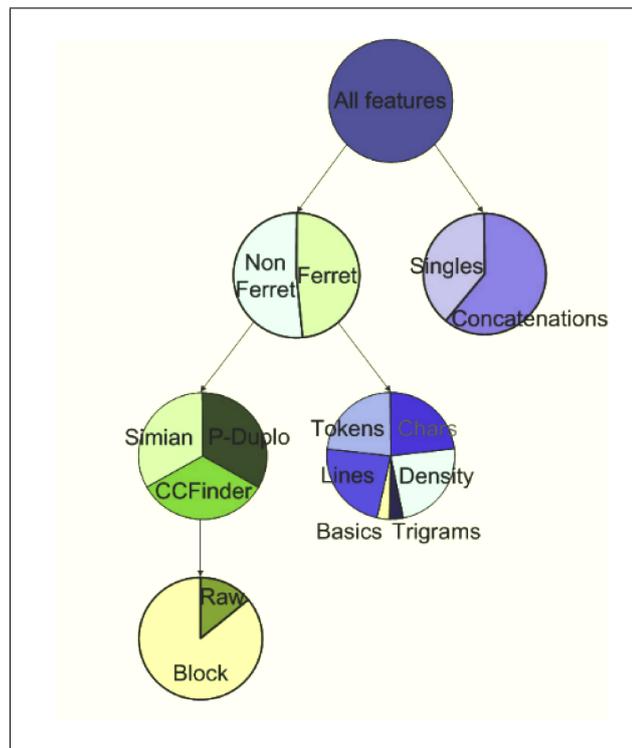


Figure 13.10: Feature set composition, shown proportionally. For example, just under half of the features are from analysis of Ferret outputs, the rest are derived equally from the other three tools.

other sets are based on blocks. However, there is another dimension to these features: different units are used to measure the blocks, thus “fc”, “ft”, “fl” and “fd” are used to indicate characters, tokens, lines and dense blocks, measured in tokens in this dataset. Features taken from Ferret’s basic output are labelled “fb” and from trigram-based features, “tris”. Sets labelled “fall” include all of the Ferret features in that category and “not-fall” are all non-Ferret features.

Combining all the sources, there is one set of features for each group of concatenated files, making five in all. For example, the feature set “cat-am-n-alikest” contains all features which are based on the comparison between the candidate file, and the concatenation of the amended file and the target file having the highest similarity to the candidate file.

There are fifty-four feature sets, seven based on each of CCFinder, Simian and PDuplo, and four which combine these, a total of twenty-five; three based on each of the six Ferret sources, and three which combine these, a total of twenty-one; there are five sets of the different concatenation combinations, one set combines all features based on pairwise file comparisons, one set combines all features based on comparisons with concatenated files, and one set combines all features.

Guidance on a suitable approach to feature construction varies. On the one hand the advice is to include as many features as possible. For example, Guyon and Elisseff say, in stressing the importance of not losing information when deciding on features, *“We argue that it is always better to err on the side of being too inclusive rather than risking to discard useful information.”* [98, p.4]. On the other hand, the advice is to keep in mind the ratio between the numbers of features and the number of instances available. For example, Mark Hall recommends that *“You should normally have at least twice the number of instances as attributes.”* [102].

Both of these requirements are taken into account. The full set of features is large, meaning that there is variety in the information available. However, the aim is to find a subset of these features, preferably from one source to make it simple to compute, so that the final set of features should be smaller.

13.3 Summary

The steps taken to construct features were explained in this chapter. First, the selection of target files, the file combinations and their comparison are described. Then basic features constructed from the results of comparing the files with each of the four tools were detailed. Finally, different approaches to describing a set of matched blocks in a file were considered, and the set of features listed.

The final set of 3,766 features comprises 546 taken fairly directly from the tools' outputs and from the Ferret trigram-file index, and 3,220 block-based features. The features are grouped into 54 subsets, which are listed in Table 13.10. Experiments to find the more useful feature subsets are described in the next chapter.

Tool	Singles			Concatenations			All	No.
	Raw	Blocks	Raw + Blocks	Raw	Blocks	Raw+Blocks		
CCFinder	ccf-raw-singles	ccf-blocks-singles	ccf-singles	ccf-raw-cats	ccf-blocks-cats	ccf-cats	ccf	7
PDuplo	pdp-raw-singles	pdp-blocks-singles	pdp-singles	pdp-raw-cats	pdp-blocks-cats	pdp-cats	pdp	7
Simtan	sim-raw-singles	sim-blocks-singles	sim-singles	sim-raw-cats	sim-blocks-cats	sim-catssim		7
All of the above (not Ferret)		13-sub-12-blocks	not-fall-singles			not-fall-cats	not-fall	4
Ferret	fb-singles			fb-cats			fb	3
Basic	tris-singles			tris-cats			tris	3
Trigrams					ft-cats		ft	3
Tokens		ft-singles			fl-cats		fl	3
Lines		fl-singles			fc-cats		fc	3
Characters		fc-singles			fd-cats		fd	3
Dense blocks		fd-singles						3
All Ferret			fall-singles			fall-cats	fall	3
Concatenations							cat-xxx†	5
Everything			all-singles			all-cats	all-feats	3
Total								54

Table 13.10: Feature sets, and the names used for them in the result tables. The sets are based on the tools used to produce them, the files used in the comparisons, or the groups of concatenated files. †Where “xxx” is the shorthand name for the file combination described in Chapter 13.

Chapter 14

Experimental results - Part 1: Classifying the split file dataset

The origin analysis experiments are presented in this chapter and the two that follow. The first of two sections in this chapter reports on experiments to test classification of the split file dataset introduced in Chapter 12. Each of the feature sets described in Chapter 13 was used to build models with a range of learning algorithms, with the aim of selecting those most suitable for use with this data.

In the second section, the results of applying selected models to candidate files from two unseen projects, PostgreSQL and DNSjava, are reported. The results are compared with those of other origin analysis research groups. Although classification of the dataset is very good, problems in filtering are identified from this comparison.

Chapter 15 describes investigations into alternative filtering criteria which aim to improve the following aspects of this approach: candidate file selection, target file selection, and target file ordering.

The data is refiltered based on the outcome of these investigations, and experiments with the revised data, and with disappearing files are reported in Chapter 16.

14.1 Building models to identify split files

The 89 project dataset was described in Chapter 12. Filtering these projects for split files results in a set of 387 candidates. The experiments described in Section 14.1.2 compare the classification of these candidates by a range of algorithms with each of the 54 feature sets described in Chapter 13. Further experiments explore the effect of combining feature sets and combining classifiers.

Unless stated otherwise, in the experiments reported in this chapter and in Chapter 16, each of the feature sets was used to classify the data with each of the chosen algorithms. The generalisation of each combination of feature set and algorithm was tested by training with 100 different random selections of 66% of the data, leaving 34% of the instances for testing. The performances recorded are the mean percentage of the results over the 100 test sets for each combination.

14.1.1 Machine learning algorithms

The machine learning algorithms used in this research are provided by the Weka [247] machine-learning toolkit (v.3.7.3). At the start of this research, a broad selection of these algorithms were chosen by excluding from those available any unsuitable for these datasets. The resulting set is shown in Table 14.1. There are three main reasons for the exclusions: algorithms which are extremely slow to run with feature sets of this size, such as MultiLayerPerceptron and NBtree; those unsuitable for numeric attributes, such as PRISM and ID3; and those unsuitable for binary classed data, such as MultiClassClassifier. This initial set of algorithms also excludes heterogeneous meta-classifiers (Grading, MultiScheme, Stacking and Voting), because the choice of base classifiers for combination by the meta-classifiers was dictated by the results from the initial tests. Decisions about further reducing the set of algorithms used in the experiments, and how they are configured, are based on the results of earlier investigations using similar data.

First, previous experiments in this research have shown that for most of

Rank	Algorithm	Rank	Algorithm	Rank	Algorithm
1	F SimpleLogistic	16	M Class'nViaRegression	31	M MultiBoostAB
2	T LMT	17	M AdaBoostM1	32	L IB1
3	M RotationForest	18	T ADTree	33	L LWL
4	T RandomForest	19	M RandomSubSpace	34	R NNge
5	T FT	20	T LADTree	35	T RandomTree
6	F SMO	21	R PART	36	R DecisionStump
7	M Decorate	22	T J48	37	M RBFNetwork
8	F SGD	23	R JRip	38	R ConjunctiveRule
9	M Rand'Comm'ee	24	B BayesNet	39	V VFI
10	M Bagging	25	T SimpleCart	40	F Logistic
11	M LogitBoost	26	B BFTree	41	R OneR
12	R FURIA	27	L IB5	42	B NaiveBayes
13	M RealAdaBoost	28	R Ridor	43	V HyperPipes
14	F SPegasos	29	T REPTree		
15	M Dagging	30	R DecisionTable		

Table 14.1: Algorithms ranked by accuracy over all feature sets. Weka provides the algorithms and each algorithm's type (assigned by Weka) is noted by the key: Bayesian, Function, Lazy, Meta, Rule, Tree, and V miscellaneous.

the algorithms tested, the default settings give results which are at least as good as the alternatives tested. Therefore, default settings are used with the following exceptions. IBk was tested with 3, 5, 7, 11, 15 and 21 neighbours, and 5 selected as the best performer; RandomForest has 25, rather than 10 trees, as this number gave a good balance between performance and speed in earlier experiments on similar data.

Second, in earlier experiments, each feature set was run with each of the 43 algorithms listed in Table 14.1. The results from each of 3 sets of experiments were averaged over all runs for each feature set. The algorithms were ranked for each experiment: 1 for the highest accuracy, 2 for the next, and so on. The ranks in Table 14.1 are the mean of these 3 results. The reduced algorithm set, shown in Table 14.2 was selected by taking 19 of the top 20 algorithms,¹ and adding the best of the simple rule-based (PART), tree-type (J48), Bayesian (BayesNet) and lazy (IB5) algorithms.

Third, in addition to the Weka based classifiers, grid-searches were run on support vector machines (SVMs), with both a radial basis function and a linear kernel.² The SVMs did not outperform the Weka algorithms in the early experiments, and so are not included in experiments reported here.³

¹Not LMT, as results on this data are similar to Simple Logistic, which is faster [227].

²Using Chicken Scheme libsvm implementation: <http://wiki.call-cc.org/eggref/4/libsvm>

³Graphs of the results of two of the grid searches can be found in Appendix J

Functions SGD* SimpleLogistic SMO SPegasos* Bayes BayesNet	Trees ADTree* FT J48 LADTree RandomForest	Rules FURIA PART Lazy IB5	Meta AdaBoostM1 Bagging ClassificationViaRegression Dagging Decorate continued continued LogitBoost RandomCommittee RandomSubSpace RealAdaBoost* RotationForest
--	---	---	---	---

Table 14.2: The reduced set of 23 algorithms, used in the reported experiments, are listed by Weka's groupings. Those marked with an asterisk are not suitable for the multiclassified dataset of disappearing files.

14.1.2 Classifying split files

Experiments undertaken to discover which set(s) of features and which learning algorithms combine to give good classification rates for split files are reported here. Given a number of good combinations, selection can be guided by the cost of computing the features. It is more useful for the feature set to be simple to compute than for the model to be built quickly, because model-building is a one-off cost, while features have to be constructed for each new dataset to which a model is applied.

The first experiment was to find results for the individual models. These results were analysed to determine the better performing algorithms and feature sets overall. The next experiment investigated the effect of combining some of the better performing feature sets. A third experiment looked at the effect on classification of combining the better performing algorithms with heterogeneous meta-classifiers.⁴

In a two-class problem, classification of an individual falls into one of four categories. Positive instances, in this case, split file examples, can be classified correctly, true positives (TP), or incorrectly, false negatives (FN). Negative examples can also be correctly (TN) or incorrectly (FP) classified.

⁴The difference in performance between many of the classifiers cannot be considered significant. However, to make discussion simpler, if one model has a higher mean classification accuracy over 100 train/test partitions than another, it is described as 'better'.

There are a number of standard measures used to describe the results of classification. Those reported in this chapter and Chapter 16 are:

- Success rate, or % correct or accuracy = $\frac{TP+TN}{TP+FN+TN+FP}$
The proportion of correct instances in the dataset.
- Precision (of positive instances) = $\frac{TP}{TP+FP}$
The proportion of instances identified as positive which are correct.
- Recall (of positive instances) = $\frac{TP}{TP+FN}$
The proportion of positive instances which are identified as such.
- F-measure = $\frac{2 \times TP}{(2 \times TP) + FN + FP}$
The harmonic mean of precision and recall $2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$.
- Geometric mean = $\sqrt{\left(\frac{TP}{TP+FN} \times \frac{TN}{TN+FP}\right)}$ for two classes, and similarly, $\sqrt[n]{acc_1 \times acc_2 \times \dots \times acc_n}$ for more classes, where acc is the ratio of the correctly classified to the total instances of the class. This is a useful measure for imbalanced datasets.

Experiment 1. Applying the algorithms to each of the feature sets

Each of the algorithms listed in Table 14.2 was applied to each of the feature sets listed in Table 13.10. The top 40 (an arbitrary choice) of the 1242 results, ranked by mean % correct classification, are listed in Table 14.3. The top 8 results are obtained with the SVM-based algorithms, SMO [106, 126, 188], SGD and SPegasos [214]. The top 17 results are based on 4 closely related feature sets: the basic Ferret (fb), and trigram-based (tris) features, both full (including concatenations) and singles sets in each case.

	Feature set	Algorithm	Mean % correct	Std. Dev.	Mean prec'n	Mean recall	Mean F-measure
1	fb	SMO	94.29	1.89	0.940	0.947	0.943
2	fb	SGD	94.17	1.99	0.938	0.947	0.942
3	tris-singles	SMO	94.16	1.56	0.929	0.957	0.942
4	tris	SMO	94.15	1.73	0.939	0.945	0.941
5	fb-singles	SGD	94.15	1.88	0.939	0.946	0.942
6	tris-singles	SGD	94.01	1.82	0.937	0.945	0.940
7	tris	SGD	93.96	1.94	0.937	0.944	0.939
8	fb-singles	SPegasos	93.84	2.21	0.934	0.945	0.939
9	tris-singles	Dagging	93.78	1.90	0.906	0.978	0.940
10	fb	SimpleLogistic	93.68	2.03	0.930	0.946	0.937
11	fb-singles	SMO	93.46	2.03	0.913	0.962	0.936
12	tris-singles	SimpleLogistic	93.42	1.93	0.926	0.945	0.935
13	tris-singles	SPegasos	93.40	2.14	0.931	0.939	0.934
14	tris	SimpleLogistic	93.37	1.74	0.926	0.943	0.934
15	fb-singles	FT	93.37	1.88	0.919	0.953	0.935
16	tris	RotationForest	93.36	1.80	0.939	0.928	0.933
17	fb-singles	SimpleLogistic	93.36	1.91	0.921	0.950	0.935
18	fall-singles	RandomForest	93.33	1.99	0.921	0.949	0.934
19	fb	FT	93.27	2.19	0.928	0.939	0.933
20	tris-singles	RotationForest	93.20	1.96	0.930	0.936	0.932
21	tris	Dagging	93.12	1.78	0.907	0.962	0.933
22	tris-singles	FT	93.11	1.92	0.920	0.946	0.932
23	fall-singles	RotationForest	93.10	1.84	0.927	0.937	0.931
24	tris	RandomForest	93.05	1.89	0.935	0.926	0.930
25	fall	RotationForest	93.04	1.79	0.931	0.931	0.930
26	fb	Dagging	93.00	1.90	0.909	0.957	0.932
27	all-singles	RandomForest	92.99	1.93	0.926	0.936	0.930
28	tris	FT	92.95	2.06	0.922	0.938	0.929
29	all-singles	RotationForest	92.94	2.35	0.932	0.927	0.929
30	all-singles	SimpleLogistic	92.79	2.20	0.918	0.941	0.929
31	fb	RotationForest	92.71	1.86	0.932	0.923	0.927
32	all-feats	RotationForest	92.71	2.19	0.931	0.924	0.927
33	fall	SimpleLogistic	92.66	1.92	0.916	0.941	0.928
34	fall-singles	SimpleLogistic	92.64	2.01	0.914	0.942	0.927
35	tris-singles	BayesNet	92.63	2.13	0.934	0.918	0.925
36	tris-singles	RandomForest	92.61	1.92	0.927	0.925	0.926
37	all-feats	SimpleLogistic	92.54	2.11	0.917	0.936	0.926
38	tris-singles	ClassificationViaRegr'n	92.53	2.02	0.913	0.941	0.926
39	fall-singles	Rand'Comm'ee	92.46	1.90	0.903	0.953	0.927
40	fb	SPegasos	92.44	2.38	0.927	0.925	0.924

Table 14.3: The top 40 results for split file classification sorted by mean % correct.

Which feature set gives the best classification over the set of algorithms?

Table 14.4 shows the mean classification rate for each feature set over all 23 algorithms. Looking at the top performing sets, those in the left-hand column, 11 of the 18 are Ferret-based. Of the other seven, three combine the three other tools (“not-fall” - all non-Ferret features, “not-fall-singles” - all non-Ferret single file comparisons, and “13-sub-12-blocks-singles” - all non-Ferret block features based on the difference set), three are based on P-Duplo, and the other set combines all of the features. However, none of the smaller non-Ferret feature sets achieve an accuracy above 89.85%.

Two groups of feature sets are noticeably worse than others: the CCFinder based features and those based on concatenations. Reasons for these poor performances are explored after the next section.

Feature set	Mean % correct	Feature set	Mean % correct	Feature set	Mean % correct
tris-singles	92.11	fl	87.96	cat-all	84.21
tris	91.91	fl-singles	87.92	fc-cats	84.18
fb-singles	91.27	fd	87.66	sim-blocks-cats	83.14
fb	91.27	fb-cats	87.41	sim-cats	82.89
all-singles	91.21	fd-singles	87.39	cat-2alikest	82.88
fall-singles	90.89	pdp-raw-singles	87.26	cat-am+alikest	82.24
all-feats	90.88	tris-cats	87.09	cat-am+main	82.12
fall	90.69	sim	86.37	cat-am+news	81.60
13-sub-12-block-singles	89.85	sim-blocks-singles	86.33	sim-raw-singles	81.05
fc-singles	89.79	sim-singles	86.27	ccf-blocks-singles	81.05
pdp	89.73	all-cats	85.63	ccf	80.94
pdp-blocks-singles	89.43	pdp-blocks-cats	85.55	ccf-singles	80.88
not-fall-singles	89.31	pdp-cats	85.46	ccf-raw-singles	79.64
pdp-singles	89.30	fall-cats	85.32	pdp-raw-cats	79.09
fc	89.21	not-fall-cats	85.17	ccf-blocks-cats	78.75
not-fall	89.18	fl-cats	84.83	ccf-cats	78.57
ft-singles	88.77	ft-cats	84.59	sim-raw-cats	78.13
ft	88.72	fd-cats	84.31	ccf-raw-cats	75.49

Table 14.4: Mean classification rate for each feature set over all of the 23 algorithms.

Key	Name	Key	Name	Key	Name
fb	Ferret basics	fl	Ferret blocks in lines	pdp	P-Duplo
tris	Ferret trigrams	fd	Ferret dense blocks	ccf	CCFinder
fc	Ferret blocks in characters	fall	All Ferret features	sim	Simian
ft	Ferret blocks in trigrams	not-fall	All non-Ferret features		

Table 14.5: Keys and names of feature sources, repeated for reference

Which algorithm gives the best classification over all of the feature sets?

Table 14.6 shows the mean classification rate for each algorithm over all of the feature sets. Although the SVM-based algorithms perform well with selected feature sets, Simple Logistic performs best overall. This is possibly because the algorithm incorporates feature selection. The reduced set of algorithms used in the experiments reported in Chapter 16 are selected from these results: the top 8 algorithms (in the left-hand column) and 3 algorithms which appear more than once in the top 40 individual results in Table 14.3: SGD, Dagging and SPegasos.

Algorithm	Mean % correct	Algorithm	Mean % correct	Algorithm	Mean % correct
SimpleLogistic	90.14	Bagging	88.41	RandomSubSpace	87.32
RotationForest	89.98	SGD	88.14	SPegasos	87.18
RandomForest	89.75	Dagging	88.11	LADTree	86.89
FT	89.34	AdaBoostM1	88.07	PART	85.80
SMO	89.27	RealAdaBoost	87.95	IB5	85.63
RandomCommittee	88.90	FURIA	87.90	J48	85.62
Decorate	88.72	Class'nViaRegr'n	87.66	BayesNet	84.74
LogitBoost	88.48	ADTree	87.40		

Table 14.6: Mean classification rate for each algorithm over all of the feature sets.

Concatenation based feature sets

Feature sets based on concatenations on their own do not perform as well as the “singles” sets, and although 4 of the 10 top-performing sets include both sources, mixed sets do not outperform singles sets from the same source. As concatenations are partly aimed at dealing with multi-target cases, features based on these comparisons may better classify candidates with more than one target file. In this dataset, 176 of the 387 files have more than one target. The two top singles sets, “tris-singles” and “fb-singles” were compared to the related sets of concatenated features, “tris-cats” and

Feature set	Full set	Multi-target file set	Change
fb-singles	91.27	91.17	-0.10
fb-cats	87.41	89.45	2.04
tris-singles	92.11	91.60	-0.51
tris-cats	86.93	88.87	1.94

Table 14.7: Mean classification rate over the 23 algorithms, multi-target files only.

“fb-cats”, on the multi-target files only. The results in Table 14.7, show that although the “cats” features improve on this subset of files and the singles sets are slightly worse, the singles sets still outperform the concatenated sets. The difference in performance between datasets is not significant (under the corrected paired two-tailed t-test, with 95% confidence) on the cats features tested, but each algorithm’s accuracy improves by 0.05–5.11%.

Code Clone Finder based feature sets

The feature sets based on CCFinder (ccf) do not perform well on their own, all 6 sets are in the bottom 9 results. CCFinder excludes much of the information found in header files, of which there are 111 in the dataset, and this may be the reason for the poor performance overall. To test this, the .c files were separated and classified using the 2 top ccf sets, “ccf-blocks-singles” and “ccf-singles”, and the 2 top singles sets, “tris-singles” and “fb-singles”. The results are compared to those on the full set in Table 14.8.

The change in classification rate between the full and .c file sets is telling, with a marked improvement for the ccf-based feature sets, while the other features do not do so well. However, the ccf sets still do not outperform the other two sets. The minimum improvement in classification for either of the two ccf sets for one algorithm is 3.5%. Based on the t-test (as above), 15 of the 23 algorithms have significantly improved results on the “ccf-singles” set, and 14 of the 23 on the “ccf-block-singles” set.

These results show that .h files are classified less successfully than .c files by CCFinder features, while the reverse is true for the other two algorithms tested. CCFinder’s lower accuracy for .c files may be because parts of .c files are also excluded by CCFinder, or because the matching is too general.

Feature set	Full set	.c file set	Change
tris-singles	92.11	90.82	-1.29
fb-singles	91.27	89.95	-1.32
ccf-blocks-singles	81.05	87.52	6.47
ccf-singles	80.88	87.42	6.54

Table 14.8: Mean classification rate over the 23 algorithms on .c files only.

Experiment 2. Does combining feature sets improve performance?

The nine singles sets, “fb, fc, fd, fl, ft, ccf, pdp, sim and tris”, were combined pairwise and run with the 23 algorithms, to find out whether features built on different comparisons complement each other. The mean classification accuracy of the paired sets over the algorithms are shown graphically in Figure 14.1. The x-axis shows one of the sets in the combination, and the column colour shows the other one of the pair. Where the two labels correspond, the mean classification for the single set is shown. Adding another set to the “tris-singles” set does not improve its overall performance of 92.11%. Combining the “pdp” with either “fc” or “ft” singles sets improves individual set results by at least 1%. Apart from this, the single sets are only noticeably improved by adding a better set, i.e. one to the left in the graph, where the sets are ranked by overall performance.

Further combinations of feature sets Combinations of three, four and five singles feature sets, also of combining the full fb and full tris features sets with singles sets are reported in Section K.1 in the appendix. These experiments show no improvement over the best results reported here.

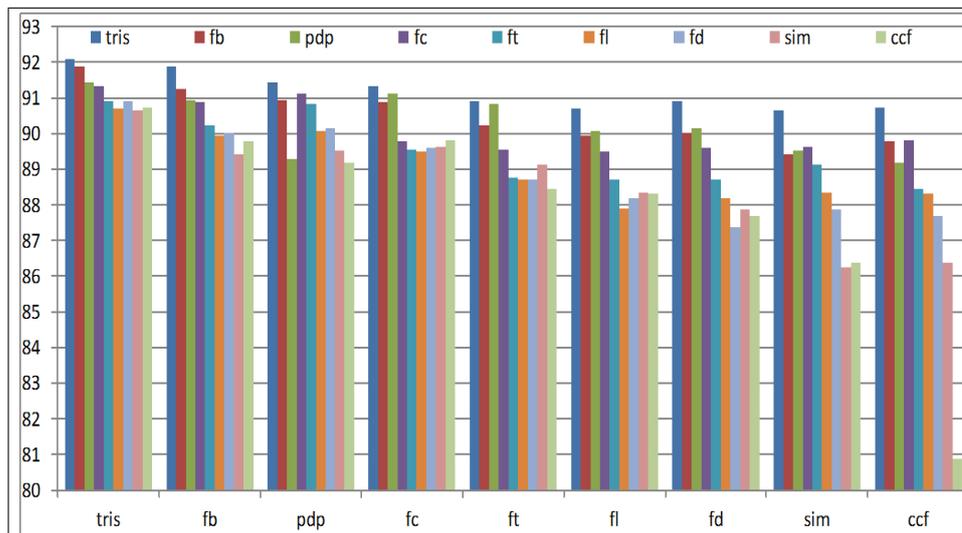


Figure 14.1: Split file classification with single feature sets and their pairwise combinations. X-axis labels show one of the pair, and the column colour the other. Where these are the same, the single set result is plotted.

Combining algorithms Experiments where algorithms were combined with heterogenous meta-classifiers are reported in Section K.2. As with the feature sets, combining algorithms shows no improvement over the best results reported so far. This lack of improvement indicates that the limit of classification accuracy for this data has been approached.

14.2 Tests on other projects

To test the generality of the approach, a selection of the models giving the highest classification rate on the split file dataset from the 89 projects were tested by applying them to unseen data. Two studies, by Zou [261], and by Antoniol et al. [6], provide information about split files found in their origin analysis research.

Zou uses Beagle, a tool with a range of matching techniques (details p.47), to investigate the project PostgreSQL. Although the prime purpose is to trace the movement and restructuring of functions, Zou also uses this information to reason about changes to files.

Antoniol et al. look at restructuring at the class level. They use a text-matching technique, comparing the cosines between weighted frequency vectors of the identifiers in each class of the DNSJava project (details p.51).

The files in each of the PostgreSQL and DNSJava projects were collected and filtered to find sets of candidate split files. The models described in Sections 14.1.2 are used to classify these candidate files, and the results are compared with those of Zou and of Antoniol et al.

In this second part of the chapter, there are three sections. In the first two, the two projects, PostgreSQL and DNSJava are introduced. Lastly, the classifications output by the better performing classifiers reported in Section 14.1.2 are compared. Although the classification of the files in the dataset is very good (accuracy 99%), the comparison with previous results shows that some candidate and target files are missed in the filtering stage of the system.

14.2.1 PostgreSQL

Zou's study covers 12 releases of PostgreSQL⁵, looking at C code in the 'backend' subsystem [90, 261]. Zou reports on two groups of operations which include six file level splits, and also notes two other restructurings where code is moved from one file to one or more others.

- Between versions 6.2 and 6.3, in the subsystem parser [261, p.77]:
 - `analyze.c` is **split** to 7 new files: `analyze.c`, `parse_agg.c`, `parse_clause.c`, `parse_expr.c`, `parse_func.c`, `parse_oper.c`, and `parse_target.c`;
 - `catalog_utils.c` is **deleted and split** into 4 files: `parse_func.c`, `parse_oper.c`, `parse_agg.c` and `parse_type.c`; and
 - `parser.c` is **split**, but details not given, except that the function `parse_agg` from `parser.c` forms part of the new file `parse_agg.c`.
- Between versions 6.5.3 and 7.0, in the subsystem `utils/adt` [261, p.82]:
 - `dt.c` is **deleted** and the functions placed in: `datetime.c` and `time.c`;
 - `datetime.c` functions are all **moved** to: `date.c` and `nabstime.c` ; and
 - `date.c` functions are **moved** to the file: `nabstime.c`
- Between versions 7.1 and 7.1.3, in `access/nbtree` [261, p.84]:
 - `nbtcompare.c`: 2 functions are **moved** to `float.c`, and 1 function to each of `varlena.c` and `nabstime.c`.
- Between versions 6.4.2 and 6.5 [261, p.99]:
 - `geqo_eval.c` and `geqo_path.c` are **merged** with [unspecified] files in the `optimizer/geqo` subsystem.

Candidate split files The backend subsystem explored by Zou is contained in the 'src' directory along with eight other folders. The complete 'src' directory was used in the experiments reported here.⁶

Table 14.9 shows the 28 candidate split files selected by filtering the backend subsystem. Of the nine split files identified by Zou, `analyze.c`, `parser.c` and `date.c` are among the filtered split files. Three others, `catalog_utils.c`,

⁵<http://www.postgresql.org>, releases 6.2 to 7.2

⁶'backend' files are separated from the rest

dt.c and geqo_paths.c, are deleted on splitting and so are part of the disappearing file set (see Chapter 16). The remaining three files, datetime.c, nbtcompare.c and geqo_eval.c are not selected.

Columns 3–5 in the table give the file name and path (postgresql/src/backend/...). The second column has the version number, and the first, the reference number used to give consecutive numbering in this research. The column headed MC gives the manual classification, 1 means split, 0 not. Where Zou identifies a file as split, there is a “Yes” in the next column. The last column gives detail about the splits found by inspecting the code. Table 14.10 (p.236) lists the 45 candidate split files from the include and interfaces directories and the classification assigned to them by inspection.

The 3CO visualisations in Figure 14.2 (p.238) show interesting features

Ref	Vn.	src/backend/...	file	MC	Zou	Comment
12	6.2	parser		1	Yes	5-way, but missing 2 sections
12	6.2	parser		1	Yes	3-way
12	6.2	port	hpux	0		
12	6.2	utils	init	1		2-way
11	6.3.2	utils	error	0		
10	6.4.2	libpq		1		Among many edited functions, one small function, pq_getstr & a heavily edited function, pq_getint, go to libpq/pqformat.c
10	6.4.2	nodes		0		
10	6.4.2	optimizer	plan	1		2-way
8	6.5.1	utils	cache	0		
6	6.5.3	optimizer	path	1		Yes, but heavily edited
6	6.5.3	port		0		
6	6.5.3	port		0		
6	6.5.3	storage	file	1		2-way
6	6.5.3	storage	ipc	1		3 heavily edited functions to sinval.c, TransactionIdsInProgress, GetSnapshotDat, GetXmaxRecent
6	6.5.3	storage	page	0		
6	6.5.3	utils	adt	1	Yes	Almost all code to nabstime.c
6	6.5.3	utils	cache	0		
6	6.5.3	utils	cache	1		TypeDefaultRetrieve moved to lsyscache.c as get_typedefault
4	7.0.3	catalog		0		
4	7.0.3	commands		1		2-way
4	7.0.3	executor		1		2-way
4	7.0.3	executor		0		
4	7.0.3	executor		0		
4	7.0.3	executor		0		
4	7.0.3	regex		0		
4	7.0.3	utils	adt	0		
4	7.0.3	utils	cache	0?		
2	7.1.3	utils	adt	0		

Table 14.9: Columns 2–5 identify 28 possibly split files selected by filtering from the backend subsystem of PostgreSQL. The next two show files identified as split by manual classification and those noted by Zou. Inspection of the source code and 3CO provides the detail in the last column.

among three candidate split files and their target files. They also show up some problems with the selection and ordering of the target files.

`Parser.c`, shown in Figure 14.2a, is split three-ways. The largest block of code, around half of the original file, has moved to `parser_expr.c`, but this is the second target file here, while `parser_agg.c`, to which less code is moved, is selected as the main target file. `Parse_agg.c` is more similar than

Ref	Vrsn.	postgresql/src/...		file	MC	Comment
11	6.3.2	include	port	irix5.h	0	
11	6.3.2	include	port	linux.h	0	
11	6.3.2	include	storage	s_lock.h	1	2-way
10	6.4.2	include	port	aix.h	0	
8	6.5.1	include	utils	datetime.h	0	
6	6.5.3	include	executor	functions.h	0	
6	6.5.3	include	executor	nodeAgg.h	0	
6	6.5.3	include	executor	nodeAppend.h	0	
6	6.5.3	include	executor	nodeHash.h	0	
6	6.5.3	include	executor	nodeMaterial.h	0	
6	6.5.3	include	executor	nodeMergejoin.h	0	
6	6.5.3	include	executor	nodeNestloop.h	0	
6	6.5.3	include	executor	nodeResult.h	0	
6	6.5.3	include	executor	nodeSeqscan.h	0	
6	6.5.3	include	executor	nodeSort.h	0	
6	6.5.3	include	executor	nodeUnique.h	0	
6	6.5.3	include	optimizer	joininfo.h	0	
6	6.5.3	include	optimizer	planmain.h	0	
6	6.5.3	include	optimizer	restrictinfo.h	0	
6	6.5.3	include	optimizer	var.h	0	
6	6.5.3	include	storage	fd.h	1	2-way
6	6.5.3	include	utils	inval.h	0	
4	7.0.3	include	executor	functions.h	0	
4	7.0.3	include	libpq	pqsignal.h	1	2-way
4	7.0.3	include	nodes	nodeFuncs.h	0	
4	7.0.3	include	optimizer	joininfo.h	0	
4	7.0.3	include	optimizer	planner.h	0	
4	7.0.3	include	parser	analyze.h	0	
4	7.0.3	include	port	bsdi.h	0	
4	7.0.3	include	port	hpux.h	0	
4	7.0.3	include	port	sco.h	0	
4	7.0.3	include	port	univel.h	0	
4	7.0.3	include	port	unixware.h	0	
4	7.0.3	include	rewrite	rewriteSupport.h	0	
4	7.0.3	include	tcop	pquery.h	0	
4	7.0.3	include	utils	dynamic_loader.h	0	
4	7.0.3	include	utils	inval.h	0	
4	7.0.3	include		c.h	1	2-way
2	7.1.3	include	commands	view.h	0	
2	7.1.3	include	utils	elog.h	0	
2	7.1.3	include		postgres_fe.h	0	
11	6.3.2	interfaces	libpq	fe-exec.c	1	2-way
11	6.3.2	interfaces	libpq	libpq-fe.h	1	2-way
11	6.3.2	interfaces	odbc	psqlodbc.c	0	
10	6.4.2	interfaces	ecpg/preproc	extern.h	1	2-way

Table 14.10: Columns 2–5 identify 45 candidate split files selected by filtering from the include and interfaces subsystems of PostgreSQL. The next column shows files identified as split (1) or not (0) by manual classification. Inspection of the source code provides the detail in the last column.

parse.c to parse_expr.c, although they share considerably less code, because parse_agg.c contains fewer trigrams than parse_expr.c, thus inflating the similarity score, and the code moved to parse_expr.c contains 5 switch structures making it repetitive, thus reducing the similarity score.

Analyze.c is a seven-way split, shown in Figure 14.2b. In filtering, two target files are missed as their similarity to analyze.c is below 0.1, the threshold, suggesting that either a lower threshold or alternative selection criteria may be better. The similarity of parse_agg.c (second yellow file) to analyze.c is 0.099, and of parse_oper.c (third blue file) just 0.033, as one 5 line function is moved to the file, which has 425 lines. The files share 520 and 190 trigrams with analyze.c, and their containment is 0.612 and 0.193 respectively. Shared trigrams or containment may improve file selection.

Date.c is also recognised as a split file, the 3CO trigram analysis (see Figure 14.2c) shows that almost all of the code has been moved to nabstime.c. This is supported by Zou's explanation that all but one of the functions from date.c moved to the file nabstime.c, while much of the new file date.c consists of functions moved from datetime.c [261, p.83].

Datetime.c, another file from this group, is not selected by filtering because it becomes four times its original size in the new release.

Nbtcompare.c has three target files, float.c, varlena.c and nabstime.c, not selected by filtering because their similarity to nbtcompare.c is below the similarity threshold of 0.1, at 0.027, 0.050, and 0.022 respectively.

Geqo_eval.c has a similar problem, but here the similarity of the target file, optimizer/joinrels.c, reduces from 0.584 to 0.333.

Dt.c, **catalog_utils.c**, and **geqo_paths.c** also identified by Zou as split, ceased to exist at the time of splitting, and are therefore disappearing files.

In summary, three of the six PostgreSQL files noted as split (and not deleted) by Zou are selected in filtering and correctly classified. The three remaining files are not selected, either because the candidate increases in size, or because the target files fall outside the filter criteria. Nine further files in the backend subsystem are found to be split and are correctly classified, as are the seven files in the other subsystems.

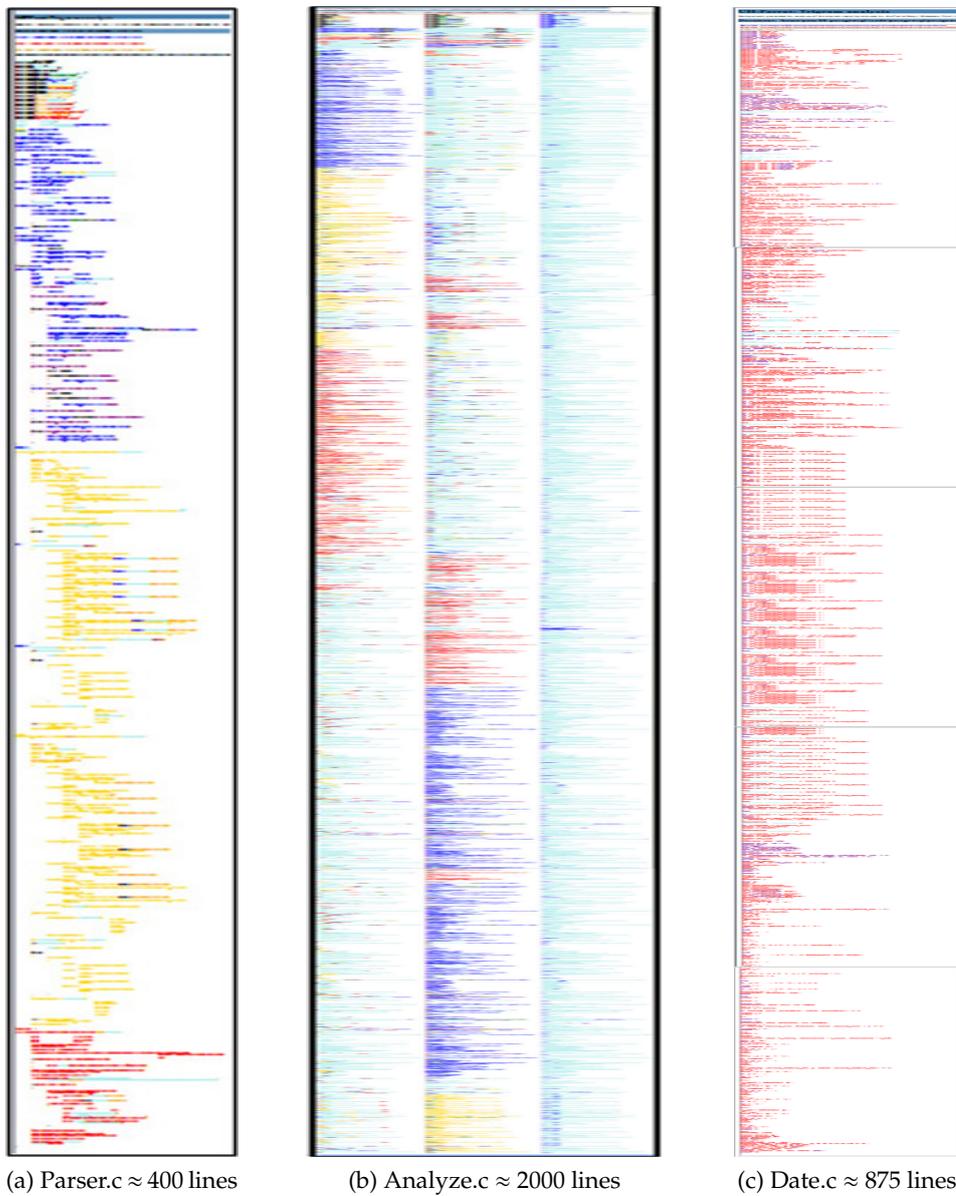


Figure 14.2: Three PostgreSQL split files and their targets

(a) Distribution of code from the file `parser.c` in 3 files in the next version. That in the revised version of `parser.c` is blue, red in `parser_agg.c` and yellow in `parser_expr.c`.

(b) `Analyze.c` is split 7 ways, the majority of the code is fairly evenly split between 5 target files, with less in the 2nd yellow file and just one function in the 3rd blue file. (First seen in Fig.8.11)

(c) Most code from `date.c` is only in the target file `nabstime.c` (red), a little also remains in `date.c` (purple) and there is a little editing.

14.2.2 DNSjava

Antoniol et al. [6] study 40 versions, 0.1 to 1.4.3, of the project DNSjava.² They construct weighted frequency vectors of the identifier names in each class. The cosines between vectors are used to reason about the relationship between files in consecutive releases. Two of their categories of refactoring, class extraction (or factoring out, where the original file retains its name) and class split (where new names are given to both parts of the file) are equivalent to split files when a file is assumed to equate to a class. Generally, a split file will mean a split class, but a renamed file may not mean a renamed class, nor will a renamed class necessarily mean that a file is renamed. The refactorings suggested by their method are listed in Table 14.11, the first five columns of which are taken from [6, Table 2]. In four cases, more than one

²<http://www.dnsjava.org>

Type of Refactoring Performed	Rel.s	Classes involved in release n	Classes involved in release n+1	Cosine	Verified by code inspect'n
Factor out	2-3	dns	dns, Type	0.76	Split
Replacement	3-4	dnsServer	jnamed	0.71	Rename
Replacement Split	4-5 4-5	CountedDataInputStream CountedDataInputStream	DataByteInputStream DataByteInputStream, DataByteOutputStream †	0.32 0.24	Rename X
Replacement	7-8	Resolver	SimpleResolver ‡	0.79	**Split**
Replacement Split	7-8 7-8	FindResolver FindResolver	FindServer ExtendedResolver, FindServer	0.31 0.31	Rename X
Merge	7-8	FindResolver, Resolver	SimpleResolver	0.75	X
Replacement Merge	11-12 11-12	CacheElement CacheElement, IO	Element Element	0.85 0.77	Rename X
Merge	12-13	CacheResponse ZoneResponse	SetResponse,	0.88	Merge
Replacement Merge	32-33 32-33	AXFREnumeration AXFREnumeration	AXFRIterator AXFRIterator, Enumerator	0.9 0.88	Rename X

Table 14.11: Refactorings suggested by Antoniol et al.'s system, see [6, Table 2], here sorted by release and name. Their code inspection showed only 1 of the 3 suggested splits, 1 of 4 merges, and 5 of 6 renames to be so. ‡Resolver was deemed **split** to SimpleResolver and ExtendedResolver. Also †CountedDataOutputStream was renamed to DataByteOutputStream, not found by vector analysis.

refactoring was suggested. Antoniol et al. inspected the code to find the correct class of refactoring, which is shown in the last column of the table. In one case, marked by asterisks, a new classification was allocated: Resolver was split to become Resolver, SimpleResolver and ExtendedResolver. A further refactoring, not found by their system but during inspection, was CountedDataOutputStream, renamed to DataByteOutputStream.

There is a difference between the systems: Antoniol et al. compare classes, while this research compares files. CacheElement and AXFRIterator are renamed within the files Cache.java and Zone.java respectively, and not picked up by the file-based system because they do not involve other files.

Antoniol et al. find four renamed files, and four suggested merges, only one of which, CacheResponse and ZoneResponse merging to become SetResponse, proved to be correct. Therefore a total of six files are found which equate to the disappearing files of the research reported in this dissertation. Table 14.12 lists these disappearing files and the two split files found by Antoniol et al.

Candidate split files In addition to the forty releases (1–40) studied by Antoniol et al. 17 further releases (41–57) are analysed here. Table 14.13 shows the six candidate split files selected by filtering DNSjava releases 1–40, all of which are classified as split on inspection. Among these six are those found by Antoniol, dns (release 2) and Resolver (release 7). Table 14.14 lists the 55 candidate split files selected from releases 40–57. Although Antoniol et al. analyse releases 1–40, changes to files in release 40 can only

Ref.	Rel.	File	Target(s)	Detail
56	2	dns	Type	2-way split
51	7	Resolver	SimpleResolver ExtendedResolver	3-way split

55	3	dnsServer	jnamed	Rename
54	4	CountedDataByteInputStream	DataByteInputStream	Rename
54	4	CountedDataByteOutputStream	DataByteOutputStream	Rename
51	7	FindResolver	FindServer	Rename
46	12	CacheResponse	}	Merge
46	12	ZoneResponse	} SetResponse	

Table 14.12: The 2 split and 6 disappearing files found by Antoniol et al.

be found with information for release 41, these candidates are therefore in Table 14.14. Of the 55 files listed, 7 are identified by inspection as split. The tables are laid out like the those for PostgreSQL. The 2 highlighted files are those which are mostly labelled incorrectly: KeyBase, where the target files are incorrectly ranked, by 90% of the models; and UNKRecord, where the main target file is not selected, by all of the models tested.

The comments in Table 14.13 differentiate between split and extract operations. Split means that part of the code is moved to another file. Extract means that a superclass is created, and code is moved from the original file to the superclass file. For example, in Figure 14.3 the file MXRecord is shown with its text coloured by 3CO. Most of the code, in red, has gone to the first target file, MX_KXRecord, suggesting that method stubs are left in the original file. KXRecord appears to share the method headers, except for the names, as the code is coloured black, showing that it is in all three files. These suggestions are supported by inspection of the amended version of the file, see Figure 14.4, which shows that a new class MX_KXRecord, see Figure K.1 (p.397), has been created to extend Record. MXRecord and the newly created KXRecord extend MX_KXRecord, see Figure 14.4.

⁴SimpleResolver has 734 of Resolver's 750 trigrams, which evolves from class to interface. ExtendedResolver implements Resolver, but seems not to result from a split as suggested.

Ref.	Vn.	*dnsjava/. . . †dnsjava/org /xbill/DNS/. . .	File	MC	Ant.	Comment
56	2	*	dns.java	1	Yes	3-way, to Type & Rcode, edited
52	6	*	Zone.java	1		Split to Master, extract to NameSet which Zone now extends
51	7	*	Resolver.java	1	Yes	To SimpleResolver ⁴
41	17	†	MXRecord.java	1		Extract to MX_KXRecord which MXRecord now extends
27	31	†security/	DNSSECVerifier.java	1		2-way split to DNSSEC
26	32	†	dns.java	1		2-way split to Lookup, bitty

Table 14.13: Columns 2–3 identify 6 possibly split files selected by filtering DNS-java, releases 1–39. Manual classification is in the column labelled MC, those found by Antoniol et al. are in the column labelled Ant., and inspection of the text provides the detail in the last column.

Ref.	Vn.	dnsjava/org/xbill/DNS/...	MC	Comment
18	40	DClass.java	0	
18	40	KEYRecord.java	1	Extract to Keybase which it now extends
18	40	Section.java	0	
18	40	SIGRecord.java	1	Split to FormattedTime & extract to SIGBase, which it now extends
17	41	A6Record.java	0	
17	41	AAAARecord.java	0	
17	41	ARecord.java	0	
17	41	CERTRecord.java	0	
17	41	CNAMERecord.java	0	
17	41	DNAMERecord.java	0	
17	41	DNSKEYRecord.java	0	
17	41	DSRecord.java	0	
17	41	HINFORecord.java	0	
17	41	KEYBase.java	0	
17	41	KEYRecord.java	0	
17	41	KXRecord.java	0	
17	41	LOCRecord.java	0	
17	41	MXRecord.java	0	
17	41	NAPTRRecord.java	0	
17	41	NSECRecord.java	0	
17	41	NSRecord.java	0	
17	41	NXTRecord.java	0	
17	41	OPTRecord.java	0	
17	41	PTRRecord.java	0	
17	41	RPRRecord.java	0	
17	41	RRSIGRecord.java	0	
17	41	SIGBase.java	0	
17	41	SIGRecord.java	0	
17	41	SOARecord.java	0	
17	41	SRVRecord.java	0	
17	41	TKEYRecord.java	0	
17	41	TSIGRecord.java	0	
17	41	TXTRRecord.java	0	
17	41	UNKRecord.java	1	Extract, but main target not selected
14	44	KEYBase.java	1	2-way split, but to 3rd target
12	46	FindServer.java	1	Extract to ResolverConfig
11	47	AFSDBRecord.java	0	
11	47	ARecord.java	0	
11	47	CNAMERecord.java	0	
11	47	DNAMERecord.java	0	
11	47	MBRecord.java	0	
11	47	MDRecord.java	0	
11	47	MFRecord.java	0	
11	47	MGRecord.java	0	
11	47	MRRecord.java	0	
11	47	NSAP_PTRRecord.java	0	
11	47	NSRecord.java	0	
11	47	PTRRecord.java	0	
11	47	RRSIGRecord.java	0	
11	47	SIGRecord.java	0	
11	47	SimpleResolver.java	0	
11	47	SingleCompressedNameBase.java	0	
9	49	AAAARecord.java	0	
8	50	TXTRRecord.java	1	Split and extract to TXTBase
4	54	NSECRecord.java	1	Split to TypeBitmap

Table 14.14: Columns 2–3 identify 55 possibly split files selected by filtering DNS-java, in releases 40–56 Manual classification is in column 4, and inspection of the text provides the detail in the last column.

In summary, Antoniol et al. found two split files in releases 1–40, both are correctly classified here, as are the four other split files found in these releases. Seven others are found in releases 41–57, two of which are incorrectly classified because of problems in filtering.

The file MXRecord shows one reason why Antoniol et al. did not find all of the split files found here. Table 14.15 shows the unweighted frequency of each of the identifiers in the 3 relevant files. For a split to be identified, the cosine between the summed vectors for the amended version of MXRecord and the new MX_KXRecord should be above the threshold set for the project. Identifiers conforming to the expected pattern are ticked. Among the others is “ab”, renamed to become “sb”, and many of the identifiers, such as “.name”, which appear the same number of times in each file.

	Old MXRecord	Amended MXRecord'	New MX_KXRecord	col.3+col.4 ≈col.2 ?
MXRecord	3	3		✓
out	6		6	✓
priority	7		8	✓
Record	1		1	✓
rrToWire	1		1	✓
rrToWireCanonical	1		1	✓
target	11		12	✓
Type.MX	3	3		✓
ab	5			}
sb			5	}
.dclass	6	6	6	x
.name	6	6	6	x
.priority	3	2	2	x
.target	2	2	2	x
.ttl	6	6	6	x
.type			6	x
c	3	2	3	x
in	4	2	4	x
length	1	2	1	x
MX_KXRecord		1	4	x
origin	2	2	2	x
st	3	2	3	x

Table 14.15: Identifiers in MXRecord and the 2 files resulting from extraction, indicating why Antoniol’s analysis would not find this extraction. The sum of identifier frequencies in columns 3 and 4 should be approximately the same as the figure in column 2, but more than half are not.

UH-Ferret: Trigram analysis

Comparison generated by analysis of the trigram report produced by the Ferret Copy-Detection Tool, (c) School of Computer Science, University of Hertfordshire, 2010.

Document: /home/pam/86-dnsjava/scode/dnsjava/dnsjava-41/org/xbill/DNS/MXRecord.java

Blue file: /home/pam/86-dnsjava/scode/dnsjava/dnsjava-40/org/xbill/DNS/MXRecord.java

Red file: /home/pam/86-dnsjava/scode/dnsjava/dnsjava-40/org/xbill/DNS/MX_KXRecord.java

Yellow file: /home/pam/86-dnsjava/scode/dnsjava/dnsjava-40/org/xbill/DNS/KXRecord.java

```

package org.xbill.DNS;
import java.io.*;
import java.util.*;
import org.xbill.DNS.utils.*;
public class MXRecord extends Record {
private short priority;
private Name target;
private
MXRecord() {}
public
MXRecord(Name _name, short _dclass, int _ttl, int _priority, Name _target)
{
    super(_name, Type.MX, _dclass, _ttl);
    priority = (short) _priority;
    target = _target;
}
MXRecord(Name _name, short _dclass, int _ttl,
int length, DataByteInputStream in, Compression c)
throws IOException
{
    super(_name, Type.MX, _dclass, _ttl);
    if (in == null)
        return;
    priority = (short) in.readUnsignedShort();
    target = new Name(in, c);
}
MXRecord(Name _name, short _dclass, int _ttl, MyStringTokenizer st, Name origin)
throws IOException
{
    super(_name, Type.MX, _dclass, _ttl);
    priority = Short.parseShort(st.nextToken());
    target = new Name(st.nextToken(), origin);
}
public String
toString() {
    StringBuffer sb = toStringNoData();
    if (target != null) {
        sb.append(priority);
        sb.append(" ");
        sb.append(target);
    }
    return sb.toString();
}
public Name
getTarget() {
    return target;
}
public short
getPriority() {
    return priority;
}
void
rrToWire(DataByteOutputStream out, Compression c) throws IOException {
    if (target == null)
        return;
    out.writeShort(priority);
    target.toWire(out, null);
}
void
rrToWireCanonical(DataByteOutputStream out) throws IOException {
    if (target == null)
        return;
    out.writeShort(priority);
    target.toWireCanonical(out);
}
}

```

Figure 14.3: The file MXRecord showing the relationship of its code with possible destination files. Code in the amended version of the original file is coloured blue (or green, purple or black, if also in the other potential destination files). Red code is only in the first selected target file.

```

KXRecord.java
package org.xbill.DNS;
import java.io.*;
import java.util.*;
import org.xbill.DNS.utils.*;
public class KXRecord extends MX_KXRecord
{
private KXRecord ()
{
}
public KXRecord (Name _name, short _dclass, int _ttl, int _preference,
Name _target)
{
super (_name, Type.KX, _dclass, _ttl, _preference, _target);
}
KXRecord (Name _name, short _dclass, int _ttl, int length,
DataByteInputStream in, Compression c) throws IOException
{
super (_name, Type.KX, _dclass, _ttl, length, in, c);
}
KXRecord (Name _name, short _dclass, int _ttl, MyStringTokenizer st,
Name origin) throws IOException
{
super (_name, Type.KX, _dclass, _ttl, st, origin);
}
}
<
Number of distinct trigrams: 105

MXRecord.java
package org.xbill.DNS;
import java.io.*;
import java.util.*;
import org.xbill.DNS.utils.*;
public class MXRecord extends MX_KXRecord
{
private MXRecord ()
{
}
public MXRecord (Name _name, short _dclass, int _ttl, int _priority,
Name _target)
{
super (_name, Type.MX, _dclass, _ttl, _priority, _target);
}
MXRecord (Name _name, short _dclass, int _ttl, int length,
DataByteInputStream in, Compression c) throws IOException
{
super (_name, Type.MX, _dclass, _ttl, length, in, c);
}
MXRecord (Name _name, short _dclass, int _ttl, MyStringTokenizer st,
Name origin) throws IOException
{
super (_name, Type.MX, _dclass, _ttl, st, origin);
}
}
<
Number of distinct trigrams: 105

```

Figure 14.4: Ferret comparison between the amended version of MXRecord.java and the new file KXRecord.java showing their use of MX.KXRecord

14.2.3 Unseen data classified by trained models

The model which best classifies the 89 project dataset is the fb/SMO combination. When this model is used to classify the filtered PostgreSQL and DNSjava files, the classification accuracy is reduced: 84.7% over the two projects (87.5% on PostgreSQL and 80% on DNSjava) and recall of 78.1% (73.7% and 84.6%). Simple Logistic is the algorithm which performs best overall on the 89 project dataset, and is best with both the “fb” and “tris-singles” (the best set overall) feature sets on the unseen data. To find out whether other feature sets produce better models, all of the sets which perform at least as well overall as the “fb” (Ferret basic) set were used to build models with Simple Logistic. Classification of the unseen data with these models is shown in Table 14.16 in terms of accuracy and recall, where the top two models have an overall accuracy of 99.2% and a recall of 93.7% (19 of 19 PostgreSQL, and 11 of 13 DNSjava split files) on the unseen data.

Simple Logistic Feature set	Overall		PostgreSQL		DNSjava	
	Accuracy	Recall	Accuracy	Recall	Accuracy	Recall
fc+tris-singles	0.992	0.937	1.000	1.000	0.983	0.846
fc+fb+pdp-singles	0.992	0.937	1.000	1.000	0.983	0.846
tris-singles	0.978	0.906	0.973	0.947	0.983	0.846
all-fc+all-tris	0.978	0.906	0.973	0.947	0.983	0.846
fb-singles	0.970	0.906	0.960	0.947	0.983	0.846
fb+tris-singles	0.970	0.906	0.960	0.947	0.983	0.846
fb	0.955	0.781	0.945	0.789	0.967	0.769
all-fb+tris-singles	0.955	0.781	0.945	0.789	0.967	0.769
tris	0.940	0.750	0.918	0.737	0.967	0.769
all-fb+fc-singles	0.933	0.687	0.918	0.684	0.950	0.692
fc+fb+tris+pdp	0.909	0.875	0.945	0.895	0.867	0.846
fc+fb+tris+pdp	0.909	0.875	0.945	0.895	0.867	0.846
pdp+tris-singles	0.879	0.906	0.945	0.947	0.800	0.846
all-tris+fc-singles	0.864	0.937	0.932	1.000	0.783	0.846
att-sel-all-feats	0.864	0.969	0.945	1.000	0.767	0.923
fc+tris+pdp	0.864	0.906	0.960	0.947	0.750	0.846
fb+tris+pdp	0.864	0.875	0.960	0.895	0.750	0.846
all-tris+pdp-singles	0.857	0.812	0.918	0.789	0.783	0.846
all-fb+pdp-singles	0.848	0.812	0.945	0.842	0.733	0.769

Table 14.16: The accuracy and recall of PostgreSQL and DNSjava candidate split files with models built with the Simple Logistic algorithm and a variety of feature sets. The figures are shown separately for the two test sets and the feature sets are sorted by descending total accuracy, and, where a place is tied, by the size of the feature set, smallest first.

14.3 Summary

In this chapter, the effect of combining feature sets and algorithms to create models for classifying candidate split files was explored. Investigations into improving the classification rate by combining algorithms with heterogeneous meta-classifiers and by combining feature sets were reported. Little is gained by combining algorithms, which implies that the better performing algorithms are able to correctly classify the same instances, and that one classifier is sufficient. Combining the feature sets generally only improves on the performance of the less good performers.

A number of models were tested on the unseen data. The classification rate of the model which performs best on the 89 project dataset, fb/SMO, drops on the unseen data, achieving only 84.7% classification accuracy (against 94.3%), with recall of 78.1%, implying overfitting.

Each of the models built using the best overall classification algorithm, Simple Logistic, with the top performing feature sets give better results than the fb/SMO model on the unseen data. The best of these, fc+tris-singles/Simple Logistic, classifies with an accuracy of 99.2% (against 93.0% on the 89 project set) and recall of 93.7%.

In the backend subsystem of the project PostgreSQL, Zou identifies nine split files. Of these, three are disappearing split files. Of the remaining six, three are selected by filtering and classified as split. The remaining three files are not selected either because the candidate file has become larger as a result of recombination, the target file has been recombined and become less similar than in the previous release, or the target file has too low a similarity score. Nine other files in the backend subsystem are identified as split by code inspection and correctly classified. In the include and interfaces subsystems, a further seven split files are identified among the filtered files, all of which are also correctly classified.

Zou's system, by operating at function level, is able to detect more subtle changes, such as recombination, and moves of single functions from one file to another, which are not always picked up at the file level. Zou reports fewer splits at file level, but the focus of her study is at function level and

therefore has a different perspective.

Antoniol et al. identify two split files in the project DNSjava. Both are detected by this system, along with four others not found by Antoniol, all six of which are correctly classified by the majority of the models. Seven other files are identified in the releases not studied by Antoniol, of which five are confirmed as split. Two files are misclassified: one because its target file is not selected, and the other because its target file is incorrectly ranked second to an incidentally similar file, meaning that the features created for the singles sets are based on the wrong file.

Antoniol et al. recognise that their system is vulnerable to identifier renaming. The file MXRecord exemplifies this shortcoming, as well as showing that unless a split is fairly “clean”, the thresholds on the cosine between vectors will cause problems. The analysis in Table 14.15 does not weight the identifiers as the tf-idf approach used by Antoniol et al. does, but gives an idea of the problem.

The results of classifying the candidate split files from the two unseen projects with the models built on the 89 project dataset are good. However, comparing the results with those of Zou and Antoniol et al. highlights problems in the selection of the candidate files and the selection and ordering of target files, most of which are addressed in the next chapter.

Chapter 15

Exploring filtering criteria

In the previous chapter, four main problems were identified in finding split files in the unseen projects. Each problem has its root in the filtering stage.

First, the amount of code moved to a target file can be too small to qualify under the filter conditions. Second, if the target files are ordered incorrectly, feature construction is affected, making classification difficult. Third, other changes to a target file reduce its similarity to the candidate file, so that it is not selected. Fourth, a split file may not be selected as a potential candidate because more code is added than is removed during restructuring, making the file larger than in the previous release. The third and fourth problems are caused by the granularity of the data, in that several changes can be applied to a file between releases. They can be seen as the same problem affecting the candidate and target files respectively.

The first and second of the four problems are explored in this chapter.

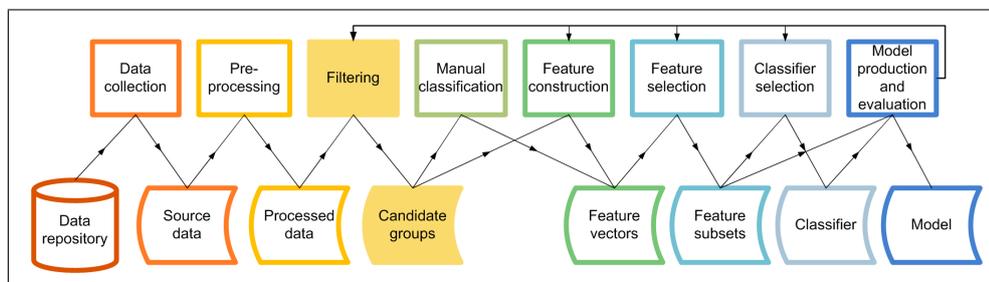


Figure 15.1: System overview, repeated here for reference

Different conditions for filtering and ordering target files are considered, to discover how they affect file selection. Candidate disappearing files are simple to identify, but the same problems exist in selecting their targets as for split files. Finding the correct set of target files and putting them in the right order is important for matching renamed, moved or split files.

In this chapter, a range of similarity measures based on trigram analysis are considered first. Sections 15.2–15.4 cover three sets of investigations: selecting target files, ordering the selected targets, and refining the selection based on knowledge gained from the previous experiments. Next, the split files found by the new filtering criteria are compared to the information given in the change log of a well-documented project. A summary follows.

15.1 Similarity measures

As noted in Chapters 3 and 13, any reasonable measure of similarity can be used to look for target files. There are a number of ways that the similarity between files can be expressed based on trigrams, and include those:

1. taken fairly directly from Ferret, and first described in Chapter 6:
 - (a) similarity score, $Sim(C_n, F_{n+1})$,
where C_n is the candidate file and F_{n+1} is a potential target file,
 - (b) change in similarity score, $[Sim(C_n, F_{n+1})] - [Sim(C_n, F_n)]$,
 - (c) the containment of one file in another $\frac{|C_n \cap F_{n+1}|}{|C_n|}$ or $\frac{|C_n \cap F_{n+1}|}{|F_{n+1}|}$,
where C is the set of trigrams in the file C , and F in F ,
 - (d) the number of shared trigrams, $|C_n \cap F_{n+1}|$, or
 - (e) change in the number of shared trigrams, $|C_n \cap F_{n+1}| - |C_n \cap F_n|$.
2. computed less directly from information about the trigrams in a group of files, as discussed in Chapter 7:
 - (a) the number of trigrams uniquely shared with the candidate file

$$\left| [C_n \cap F_{i_{n+1}}] \setminus [\cup_{j=1}^m F_{j_{n+1}}] \right|$$
 where $j \neq i$, m =number of files in release $n+1$,

- (b) the weighted trigram count, $\sum_i \frac{1}{c_i-1}$
 where i is a trigram $\in C_n \cup [\cup_{j=1}^m \mathcal{F} j_{n+1}]$
 c_i is the no. of files sharing trigram i (and $c \leq$ given maximum)
 (weighted trigram counts are explained on page 104), or
- (c) the number of trigrams shared with the candidate difference set,
 $[C_n \setminus C_{n+1}] \cap \mathcal{F}_{n+1}$, (unsuitable for disappearing files as $\# C_{n+1}$).

There are two tasks in finding target files: selecting the right target files, and ordering the selected files. The first task combines the two opposing aims of selecting all true target files, and not over-selecting incidentally similar files. This opposition leads to the standard trade-off between precision and recall, and makes it difficult to find the right balance when choosing thresholds for target file selection.

The second task is to rank target files according to how likely they are to be the new location of code moved from the candidate file. As explained in Chapter 13, this is an important requirement for feature construction.

The three sets of investigations reported in this chapter look at split files. Similar methods can be used to find and sort targets for disappearing files. Experimenting with all of the data would be the ideal. However, some of the tasks, such as judging the most suitable order for target files, are labour-intensive. Therefore an arbitrary subset of six projects, with around 70 positive examples of split files, was used in these investigations.

The first set of investigations considers each of the proposed filtering conditions, and the effect of different parameters, on file selection. The aim is to reduce thresholds for target file selection to the point where few, and preferably no, split file examples are excluded from the set of candidates, while not including too many negative examples, and making target sets too large. The second set looks at the selected split files, examining differences between the ordering of target files by different similarity measures.

Finally, the criteria explored in the two previous investigations are combined to refine the method for selecting and ordering target files. The chosen method is applied to the best-documented project from the test set, and the results compared to information found in its change log.

15.2 Target file selection

In this section, target file selection is investigated, using various similarity measures and thresholds. Similarity score is a simple measure to explore because the values are stored during the information gathering phase of filtering, and the threshold is easy to vary. The first experiment in this set aims to find a baseline number of candidate files by looking at the effect on target file selection of a range of similarity thresholds. Other measures are more difficult to explore as thoroughly because there are more variables to consider. The results of filtering with these other measures are related to those from the similarity score investigation. Labels relating to the measures listed in Section 15.1 are noted in the headings of the following sections.

15.2.1 Similarity score (1a)

In early experiments [93], similarity score was used to select target files. A threshold value of 0.1 was chosen from 0.05, 0.075, 0.1, 0.15 and 0.2. The two higher values excluded too many split files from the candidate set, because no target files were matched to them. With the similarity scores 0.05 and 0.075, there were too many target files, given the time-consuming method of hand-checking candidate groups prior to the development of 3CO.

In this investigation, lower similarity scores were used to filter the target files: 0.1, 0.083, 0.066, 0.05, and 0.033. The number of candidates are recorded in Table 15.1, along with the number of positive examples of split

Threshold value	Number of candidates	Mean targets	Increase in candidates	New candidates which are split
0.100	209	9.5	–	–
0.083	251	10.5	42	4
0.066	315	13.5	64	3
0.050	351	26	36	3
0.033	370	69	19	0

Table 15.1: The effect of different similarity score thresholds on candidate and target file selection. Column 2 shows the number of candidates selected, with target file set size in column 3. Additional candidate files are noted in column 4, and the number of these which are split in column 5.

files added at each level, and the mean size of the target file sets.

The results show that among these thresholds, 0.05 selects the maximum number of split files as candidates in this dataset. The lower value of 0.033 finds 19 more candidates, none of which are split files.

15.2.2 Containment (1c)

To find target files based on shared trigrams, the threshold can either be a discrete value, or a proportion of file size. Given the range of file sizes, the proportional measure containment was chosen, and various values tested to find a reasonable set to compare with the similarity score set.

At 0.25 of the smallest file size, 351 candidate files are selected, exactly the same number as with a similarity value of 0.05. Each set has 16 candidates which do not appear in the other set. None of the 16 selected by containment are split files, whereas 2 selected by similarity score are split. The mean number of target files per candidate selected by this method is also 26. Based on this test, similarity score seems to be a marginally more useful filter than containment.

15.2.3 Change-based filters (1b, 1e)

Filtering by containment or similarity score alone results in large target file sets. Any file which is similar to a candidate file will be selected as a target under each of the two measures. However, if this similarity is unchanged or is reduced, the file is unlikely to be a true target, unless the code removed from the candidate file was already in the target file, as is sometimes the case when files are merged. The aim of using change-based filters is to exclude incidentally similar, but otherwise unrelated, target files.

Initially the two change-based measures (changes to similarity and to shared trigrams) were explored using just one selection condition. However, to include all of the split files found by the similarity threshold of 0.05, the change-based thresholds have to be lowered to the point where the target file sets become larger (mean files per set >40), making these measures unsuitable for use on their own.

Other strategies for including change-based filters were therefore tested: pairing similarity score with change in similarity, and stratifying the changes in shared trigrams, depending on the ratio of the change to overall file size. Finding parameters for these two groups was an iterative process. Thresholds were estimated for each change measure, candidate files selected, and the resulting sets compared to that reported in Section 15.2.1. The filter thresholds were then adjusted to include the missing split files.

15.2.4 Combining similarity score conditions (1a, 1b)

For a file to qualify under the combined similarity conditions, it must both have a similarity to the candidate file of at least 0.05, the base score established for this dataset, and the similarity of the target file to the candidate must be larger than the similarity of the file in release n . To select files where the code already exists in the target file but is extracted from the split file, targets with a high similarity to the candidate are selected, regardless of change in similarity. The set of parameters chosen after iteration were:

- similarity between files is at least 0.05 and increases by at least 10%,
- or similarity is greater than 0.5, regardless of change.

These parameters result in a set of 263 candidates which include all but one of the positive candidates found with a similarity of 0.05 and above. The mean number of target files is 6.6. The similarity of the target for the missed split file changes by less than 10% and it is therefore not selected.

15.2.5 Stratifying shared trigram conditions (1e)

A range of options was explored to find conditions based on the change in the trigrams shared by the target file in releases n and $n+1$, and the candidate. In the end, the increased numbers of shared trigrams were stratified, depending on the size of the smaller of the two files compared. The idea is that a small increase in shared trigrams may be more significant in a small file than a large one. The conditions are an increase of at least:

- 10 shared trigrams, and at least 30% of the smaller of the file sizes, or

- 20 shared trigrams, and at least 20% of the smaller file size, or
- 50 shared trigrams, and at least 10% of the smaller file size, or
- 100 shared trigrams.

This results in a set of 271 candidates with a mean of 13 target files. One of the split files is also missing from this set, it has a similarity of 0.065 to the target file, and the two files share 201 trigrams. However, the two files, which contain 1445 and 1843 trigrams, share 151 trigrams in release n, so there are 50 new shared trigrams, which is less than 10% of either file size.

15.2.6 Less direct methods (2a, 2b, 2c)

Similarity, shared trigrams, and containment measures are either stored, or can be calculated from measures stored during the information gathering process described in Section 12.3.1. More work is required to compute the three less direct measures (uniquely shared trigram counts, weighted trigram counts, and trigrams shared with the candidate difference set), making them less attractive as a first filter, unless the gain is substantial. Each of the measures in this category was tested on a small set of files taken from the dataset, and the results reported in Appendix L. The tests show that although these measures reduce the number of incidentally similar target files, they do not select true targets as well as the more direct methods.

15.2.7 Discussion on selecting target files

None of the three less direct measures, which require more processing, perform as well as the simpler methods in selecting target files for the more difficult of the files tested. However, they do reduce the number of incidentally similar files selected as targets for other files.

Similarity score or containment result in large target file sets, as do the change in similarity or the change in shared trigrams. However, smaller target file sets result from either combining similarity and change in similarity measures, or stratifying the conditions for changes to shared trigrams.

One positive candidate file is excluded by the proposed filter conditions

for both the similarity combination, and the shared trigrams. Target file sets are reduced to a mean of 13 in the shared trigram change set, and to less than 7 in the similarity score combination set. The set of candidate files is also smaller, as it contains fewer negative examples. The selection of true target files is also a little better with the similarity combination. Investigations into target file ordering are therefore based on this set.

15.3 Ordering target files

The features constructed for this research are based on comparisons between the candidate file and different selections from the group of target files (see Chapter 13). The more successful feature sets contain features based on the three-way comparison between the two versions of the candidate file, and the main target file. When there is more than one target file, it is important that they are ranked correctly, so that the main target file is that most likely to contain the majority of the code removed from the candidate file.

15.3.1 Comparing ranking criteria

The set of target files selected by similarity combination (Section 15.2.4) was used to explore the effect of different measures on target file ranking. Any of the measures discussed for selecting target files can also be used to rank the files. Three of these ranking criteria are compared in this section:

- change in similarity score,
- change in shared trigrams, and
- the number of trigrams uniquely shared with the candidate file.

Method

The two change-based measures are found from stored information. To find the number of unique trigrams, the candidate file, amended file and all of the selected target files are compared by Ferret to produce a trigram-file index. The trigrams uniquely shared by each target file and the candidate file are counted, and the files ranked by this count.

To assess target file order, the 3CO comparisons between each candidate and its targets were examined. Files considered to be true targets were ranked by eye according to the code shared with the parts of the candidate file not covered by the amended file, and their order by each method noted. As an example, Figure 15.2 shows a file whose targets are ordered by uniquely shared trigrams on the left, and by change in shared trigrams on the right. As a reminder, code in the amended file is blue, the file selected as the first (main) target, red, and the second target file, yellow. The true target file, which contains the code not in the amended file, is selected first by uniquely shared trigrams (on the left), and second by the change in shared trigrams. It is easy to see why: the “red” (or first) file on the right has a large number of trigrams shared with the candidate file, however, most of these trigrams (in purple) are also shared by the amended file.



Figure 15.2: Target files ordered by uniquely shared trigrams (left) or by change in shared trigrams (right). The interesting section is coloured red on the left, and yellow on the right, showing this target is ranked first by one method and second by the other. The green or purple section is shared by the second, or first target respectively, and the amended file.

Results

Using the test subset, the three methods of ordering target files are compared in Table 15.2: ordering by uniquely shared trigrams, by change in shared trigrams, and by change in similarity. The first three columns report on the number of rankings which are the same under all measures, showing the number of target files, the number of “true” targets and the number of times the rankings agree. For example, the figures to the left of the asterisk in column 4 mean that there are two instances where there are two target

Total targets	True targets	Count matched		Ordered by uniques	Ordered by shared	Ordered by similarity	Count unmatched
1	1	24					
2	1	4		1	1	2	1
2				1	2	2	4
3	1	3		1	1	2	1
4	1	2		1	1	4	1
5				1	1	3	1
6	1	2					
7	1	2					
8	1	1		1	4	4	1
11				1	2	2	1
17	1	1					
21			\$	1	18	4	1
23				1	2	2	1
44	1	1					
2	1, 2	2	*				
3				1, 2	1, 2	1, 6	1
4	1, 2	1					
5	1, 2	1					
14	1, 2	1		1, 2	1, 2	1, 6	1
14				1, 2	2, 7	2, 6	1
26				1, 2	5, 2	1, 2	1
23				1, 2, 3 ..	4, 2, 10 ..	7, 5, 9 ..	1
24			£	1, 2, 3, 4	1, 2, 12, 7	1, 2, 4, 3	1
6	1, 2, 3 ..	1					
17				1, 2, 3, 4, 5	1, 6, 4, 2, 3	1, 2, 4, 5, 3	1
Totals		46					19

Table 15.2: Comparison of target file order by uniquely shared trigrams, changes to shared trigrams, and to similarity. The first column shows the number of targets selected. The next two give information about the files whose targets are ranked the same by each method. Columns 5–8 show those with different rankings. Symbols in column 4 are referenced in the text.

files, with the files correctly ranked first and second by all methods.

The last four columns show the rankings which disagree. The fifth column has the order of true targets under uniquely shared trigrams, the next column, under change in shared trigrams, and the next, under change in similarity. The last column gives a count of the occurrences.

For example, the figures to the right of the \$ in column 4 mean that there is one instance where the uniquely shared trigrams rank the single true target first, but other methods rank it differently, at positions 18 and 4. Another example, indicated by the £ in column 4, shows a set of 4 true targets from a pool of 24. Under uniquely shared trigrams these are ordered as expected, but under change in shared trigrams or similarity these targets are in positions 1, 2, 12 and 7; and 1, 2, 4 and 3 respectively.

The target files in this set are placed in the “correct” order when ranked by uniquely shared trigrams. The other methods rank target files in the same order in 46 of the 65 cases, of the other 19, 6 are ranked the same by change to shared trigrams and not by similarity, while 2 are ranked the same and 2 nearly the same, by change in similarity, and not by shared trigrams.

15.3.2 Discussion on ranking target files

Using the set of candidate file groups selected by similarity, the target files were ranked by uniquely shared trigrams, change in shared trigrams and change in similarity. Ordering by uniquely shared trigrams outperforms the other methods for this data, giving the “correct” order in every case.

One problem identified while determining target file order is that some candidate files have blocks of code which are unmatched by any of the selected target files, even when the file is apparently split. An example is shown on the left of Figure 15.3, where two blocks of code appear to have moved to another file, but two other blocks are unmatched (mostly cyan). On the right of the figure, the wider selection of target files includes the file which contains the missing code. The next section explores ways to balance the inclusion of such targets while not including too many unwanted files.

15.4 Refining file selection

The idea in refining target file selection is to increase recall. This is achieved by relaxing the selection criteria to generate a larger initial set of files, then refiltering to exclude files which are unlikely to be true targets.

If a file does not share trigrams with the difference set it is not normally a true target, and can be discarded from the set of target files. As previously explained, extra time and space are required to make the comparisons required to select files which share trigrams with the candidate difference set from all of the files in release $n+1$. However, it is simpler to select such files from a smaller set of target files chosen by other criteria.

As a test, the threshold for selection on file similarity, regardless of change, was altered from 0.5 to 0.25. This increases the number of candidates from 263 to 266 and mean targets from 6.6 to 8.4. Files were then

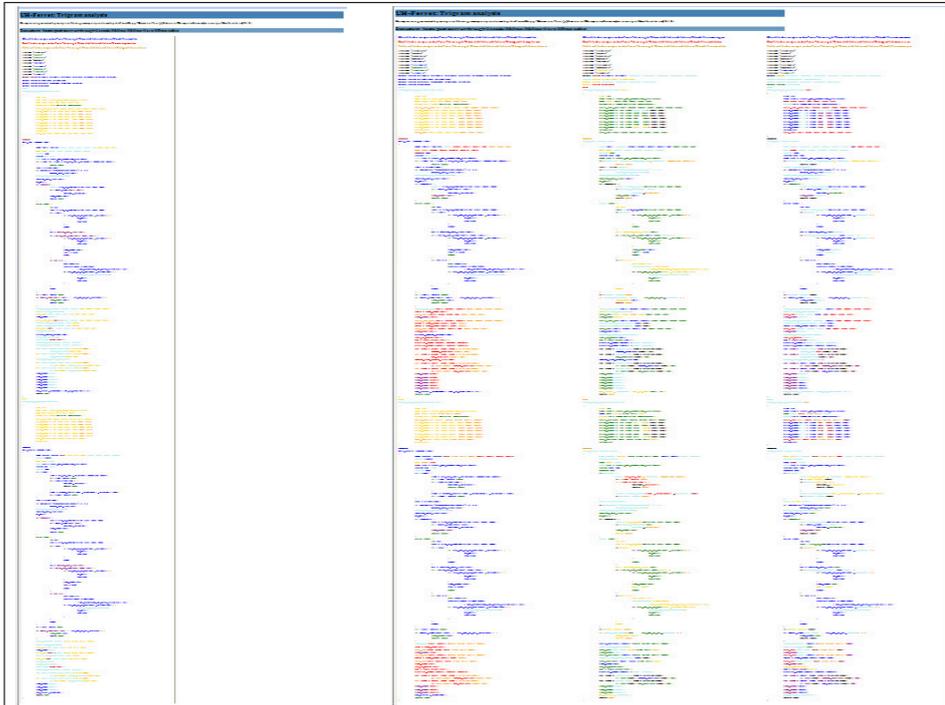


Figure 15.3: On the left, target files are selected by similarity combination (Section 15.2.4). Two sections of the candidate file are not matched by a target file. On the right, additional targets are selected by adding shared trigram conditions. The code is matched by the 1st (red) file.

Trigram shared with difference set	Number of candidates	Mean number target files	Positive splits selected?
no filter	266	8.4	all known
= 0	227	7.2	all known
<5	189	5.7	all known
<10	161	5.2	all known
<20	137	4.8	all known

Table 15.3: Changes to the number of candidate and target files made by removing target files with few trigrams shared with the candidate difference set

removed from the target set if they shared fewer than n trigrams with the difference set. The results for different values of n are in Table 15.3.

In the subset of projects used in these investigations, only irrelevant files are removed, as there is no reduction in the number of positive split file examples and their true targets. There are two cases in the remaining projects where split files will not be matched if target files with fewer than 20 trigrams in common with the difference set are removed.

To explore whether the blocks which are not matched by a target file can be found in target files selected under lower thresholds, several variations to the filter conditions were tested. These are listed in Table 15.4. In each case, target files are removed if they have fewer than twenty trigrams in common with the difference set. The similarity combination has the form

$[(sim \geq a) \wedge (sim \geq (prev \times b))] \vee [sim \geq c]$			Shared added?	Gaps filled	New splits	Candidate files	Positive examples	Mean targets
a	b	c						
0.05	0.05	0.25	n	0	3	137	69	4.8
0.05	0.025	0.25	n	0	3	137	69	4.8
0.05	0.01	0.25	n	0	3	158	69	6.4
0.05	0.05	0.1	n	2	7	154	73	5.9
-	-	-	y(1,2)	4	5	146	71	14.8
0.05	0.05	0.1	y(1)	7	7	166	73	10.8
0.05	0.05	0.1	y(1,2)	9	9	166	75	14.2

(1) means changes to shared trigrams added, (2) means 0.5 containment of smallest file added

Table 15.4: The effect of altering target file selection criteria. Extra shared trigram criteria are noted in Column 4. The next 2 columns show whether newly selected target files cover the gaps found in 22 files, and additional split files detected. The last 3 columns give the total candidate files, those identified as split, and the mean number of target files.

$[(sim \geq a) \wedge (sim \geq (previous \times b))] \vee [sim \geq c]$, a combination like that in Section 15.2.4. The values for a , b and c are shown in the first three columns. In the last three tests, the shared trigram change filters (see Section 15.2.5) are added to the criteria. Two tests also have containment of >0.5 added.

Twenty-two split files have sections not matched by the target files selected, gaps are filled in two of these files by changing the similarity (c) to 0.1, and in seven files by adding the shared trigram change filter conditions. The shared trigram and containment conditions on their own fill gaps in four files, and adding the shared change conditions to the similarity conditions finds nine gap-fillers. Three new files are identified as split by changing similarity (c) from 0.5 to 0.25 and four more by changing this parameter to 0.1. A further two are identified by adding the containment condition.

Although the mean number of target files per group is high (14.2) in the last group in Table 15.4, 99 of the groups contain fewer than 5 files, 125 fewer than 10 and only 25 have more than 20, most of these being in the project ppcboot, which has a large number of parallel subsystems. The number of target files per group is plotted in Figure 15.4.

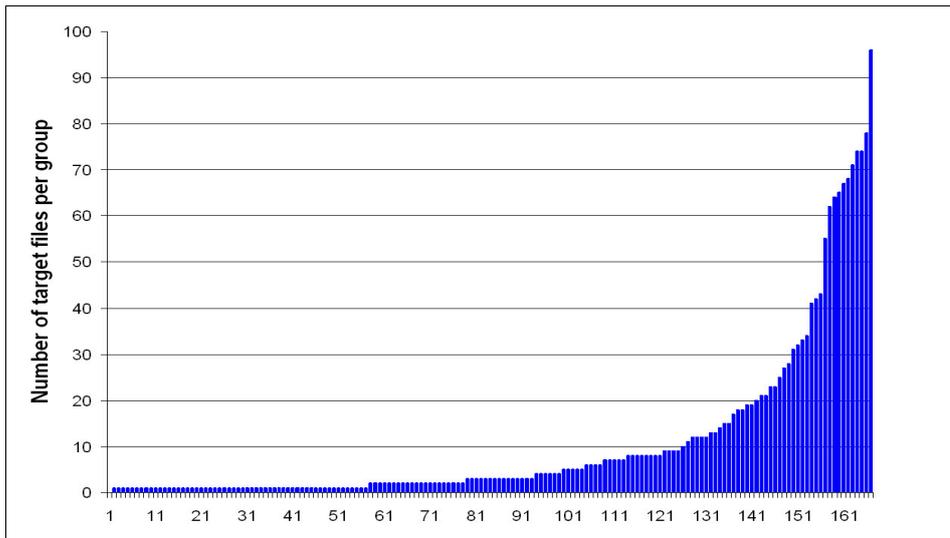


Figure 15.4: Target files selected by the conditions on the bottom row, Table 15.4

15.4.1 Filter conditions summarised

This section provides a summary of the filter conditions.

- Potential split files are selected if the file reduces in size:
 - by at least 5%, or
 - at least 200 bytes.
- Target files are selected for either split or disappearing files under any of the following conditions (stated as formulae in Figure 15.5):
 - similarity is at least 0.05 and is increased by at least 5%
 - similarity is at least 0.1
 - shared trigram change $\geq a$ and $\geq b$ of the smaller file's trigrams where a, b are either 10, 30% or 20, 20% or 50, 30%
 - shared trigram change ≥ 100
 - shared trigrams $\geq 50\%$ of the smaller file size
- For split files, the resulting targets are removed if they do not share at least 20 trigrams with the candidate difference set.
- The remaining targets for split files are sorted by the number of trigrams uniquely shared with the candidate file, or in the case of a tie, the number of trigrams shared with the difference set.
- The disappearing file targets are sorted by unique trigrams, or in the case of a tie, by similarity score.

<ul style="list-style-type: none"> ★ $[Sim(C_n, F_{n+1}) \geq 0.05] \wedge [(Sim(C_n, F_{n+1}) - Sim(C_n, F_n)) \geq (0.05 \times Sim(C_n, F_n))]$ ★ $Sim(C_n, F_{n+1}) \geq 0.1$ ★ $[(C_n \cap \mathcal{F}_{n+1} - C_n \cap \mathcal{F}_n) \geq a] \wedge [(C_n \cap \mathcal{F}_{n+1} - C_n \cap \mathcal{F}_n) \geq (b \times Min(C_n , \mathcal{F}_{n+1}))]$ where $(a, b) \in \{(10, 0.3), (20, 0.2), (50, 0.1), (100, 0)\}$ ★ $[C_n \cap \mathcal{F}_{n+1} \geq (0.5 \times Min(C_n , \mathcal{F}_{n+1}))]$
--

Figure 15.5: Filter criteria, C, \mathcal{F} are the set of trigrams in the files C, F

15.5 Checking a change log to verify selection

To validate the selection of candidate and target files, the change log of one of the projects, “Lifelines”, was searched to find entries relating to split files. Change logs are often incomplete, Chen et al. [40] found that changes omitted from the change logs of three well-established open source projects ranged from 4% to 78%, with a mean of 22%. Parnin discovered that only 7 of 55 refactorings they found in the project Cecil were noted in the change log [187]. However, Lifelines appears to be fairly well-documented. Of the 44 candidates, 11 are not split. Only 5 of the 33 split files are not documented, shown by the explanatory notes in Table 15.5, meaning that a

Rel.	Dir.	File	Change log reference (see Appendix M).
4	hdrs	gedcom.h	13
4	interp	interp.c	2
4	lifelines	screen.c	3, 4, 5
4	lifelines	llexec.c	12
4	stdlib	signals.c	10
4	lifelines	main.c	11
4	hdrs	date.h	15, 16
4	gedlib	init.c	14
4	lifelines	error.c	7, 9
5	gedlib	node.c	23
5	stdlib	mystring.c	24
5	hdrs	cache.h	25
5	gedlib	init.c	17
6	lifelines	llexec.c	— nearly $\frac{1}{2}$ to ui_cli.c, little to screen.c
7	hdrs	interp.h	26, 27
9	gedlib	node.c	29
9	interp	pvalue.c	28
9	lifelines	edit.c	30
9	lifelines	newrecs.c	— ask_for_record to ask.c
10	gedlib	translat.c	— more than $\frac{1}{2}$ to charmaps.c
10	hdrs	gedcom.h	32
10	lifelines	llinesi.h	43
10	hdrs	standard.h	31
10	lifelines	export.c	33
12	stdlib	stdstrng.c	38
12	hdrs	mystring.h	38
12	lifelines	newrecs.c	— nvaldiff to nodeutls.c
12	stdlib	mystring.c	38
13	interp	pvalue.c	42
13	hdrs	interp.h	41
13	lifelines	import.c	40
14	lifelines	delete.c	48
14	gedlib	valid.c	— strings moved to messages.c

Table 15.5: Lifelines split files, identified by release number, directory and name. Column 4 gives the number(s) assigned to the change log extract in Appendix M, or details of the split when no entry is found in the log.

high proportion, 85%, of the split files found by the system are documented.

The log was searched for the words '(re)move', 'split', '(re)factor', and scanned visually for other indications of refactoring. The log was also searched to match the names of files classified as split by inspection of the candidate files, but not found by these terms. The edited log entries are shown in Appendix M, with the split operations numbered from 1–48. Of these 48 log entries, 31 are matched by files found by filtering, and 17 are not matched, for which there are four reasons.

First, some changes involve removal of code introduced in the same release as the change, meaning that the file is unchanged from one release to the next. Second, some candidate files have not reduced in size, as although code has been removed, other code has been added. As previously noted, this is a feature of working at release level. Third, the target file's similarity to the candidate file can fall between releases despite having code moved to it, because of other changes to the file. Lastly, although files are selected as potential candidates, the amount of code moved to the target file is so small that it does not fulfil any of the filter criteria. Table 15.6 lists the Lifelines log entries, noting whether they are matched, or if not, why not.

No.	Matched?	No.	Matched?	No.	Matched?
1	Within release	17	✓	33	✓
2	✓	18	Candidate grows	34	Within release
3	✓	19	Candidate grows	35	Candidate grows
4	✓	20	Candidate grows	36	Within release
5	✓	21	Within release	37	Candidate grows
6	Target similarity falls	22	Candidate grows	38	✓
7	✓	23	✓	39	Candidate grows
8	Change too small	24	✓	40	✓
9	✓	25	✓	41	✓
10	✓	26	✓	42	✓
11	✓	27	✓	43	✓
12	✓	28	✓	44	Within release
13	✓	29	✓	45	Within release
14	✓	30	✓	46	Within release
15	✓	31	✓	47	Within release
16	✓	32	✓	48	✓

Table 15.6: Lifelines: of the 48 change log entries relating to splits, 31 are matched by candidates, and 17 are not; either because a split file has become larger, the target similarity score falls, the change is too small, or the change is made within the release.

15.6 Summary

With a perfect filtering strategy, there would be no need for the classification part of the system, as every candidate selected would be a split file, and every target file a true target. However, it is unlikely that any filtering strategy will achieve this perfect precision and recall. There is a natural conflict in selecting all split files, while not selecting non-split candidates; and in selecting all true target files, while not selecting too many targets for each candidate file. Filtering is nevertheless important, both in reducing the number of candidate files, and in selecting target files. It keeps the negative examples to a reasonable level, helping to balance the dataset for machine learning; as well as reducing the burden of manual classification. The experiments reported in Chapter 14 identified four problems in filtering:

1. too little code moved to qualify under the original conditions,
2. incorrect target file ordering,
3. other changes to the target file, making it less similar, and
4. candidate files are not selected because they increase in size.

The investigations reported in this chapter seek to address problems 1 and 2. By widening the selection criteria, problem 1 should be resolved in the majority of cases, although movement of small amounts of code may only be detected with even lower thresholds. The decision to favour recall against precision, or vice versa, can be made when running the system, depending on requirements. This type of decision occurs in many domains, for example, the detection of craters on Mars, where a minimum size of 5 pixels is used to reduce the number of false positives detected [223]. Problem 2 seems to be resolved by ordering target files by unique trigrams, as each of the test cases appears to be ranked correctly.

In addition, problem 3 will be partially resolved by the change in filtering conditions. If the similarity between a true target file and the candidate file is reduced, because code other than that moved from the candidate file has also been moved to the file, then the conditions based on the change in

shared trigrams may select the target file. A different approach to candidate file selection is required to overcome problem 4, and has still to be explored. The size of a file's difference set may offer a solution to this problem.

The chosen filter conditions combine similarity, change in similarity, change to shared trigrams, and containment, so that target files selected by any of these conditions are added to the set. Re-filtering then removes files which share few trigrams with the candidate difference set, as these files are unlikely to be true targets. The remaining target files are sorted on the number of trigrams uniquely shared with the candidate file.

The difference set for a disappearing file is the file itself, therefore secondary filtering using the difference set is not possible. However, the target files can be selected and ordered in the same way apart from this. The next chapter reports on machine learning experiments with the refiltered data.

Chapter 16

Experimental results - Part 2: Classifying refiltered data

This chapter consists of five main parts. The first compares the refiltered split file dataset with the original set. The second reports on experiments in classifying the refiltered split file dataset, where the accuracy is over 90%. The third part gives the results of applying the top performing models to the two unseen projects, with additional test sets from the Java project Struts, and the Python project PyX. In the fourth part, which is about classifying the disappearing file datasets, the candidates are divided into two groups. Candidates with only one target file are classified with 95% accuracy, and the more challenging group with at least two targets, which is a three class problem, is 88% correct. The last part of the chapter gives the results of using these models to classify the equivalent datasets from the PostgreSQL and DNSjava projects, with PyX providing an additional test set.

The experiments in this chapter are run in a similar way to those reported in Chapter 14, in that each feature set/algorithm combination is run over 100 different random splits. Although all of the feature sets from Chapter 14, including the combined ones, are used in the experiments, a smaller selection of algorithms is applied to these sets.

16.1 Refiltered split file dataset composition

This part of the chapter covers the composition of the refiltered split file dataset and compares it with the original set. The refiltered split file dataset comprises 810 instances, of which 414 are negative, and 396 are positive, more than twice the number of files selected by the previous filtering conditions. The types and classes of the files are shown in Table 16.1 with those of the original set. Only the negative header file category is reduced. All of the positive instances from the original set are included in the refiltered set, except for two, both small header files. In one case, a small 3 line struct is moved, and in the other, 2 lines are moved; neither qualify under the new requirement that at least 20 trigrams belonging to the difference set should be moved between the files.

The pie chart on the left of Figure 16.1 shows the number of target files in the candidate groups. Although there are proportionally fewer files with a small number of target files in the refiltered set than in the original set (on the right), there are around one and a half times as many files with one or two targets because the set is larger. Less than 10% of the files have 19 or more targets, although 16 files have more than 50 targets.

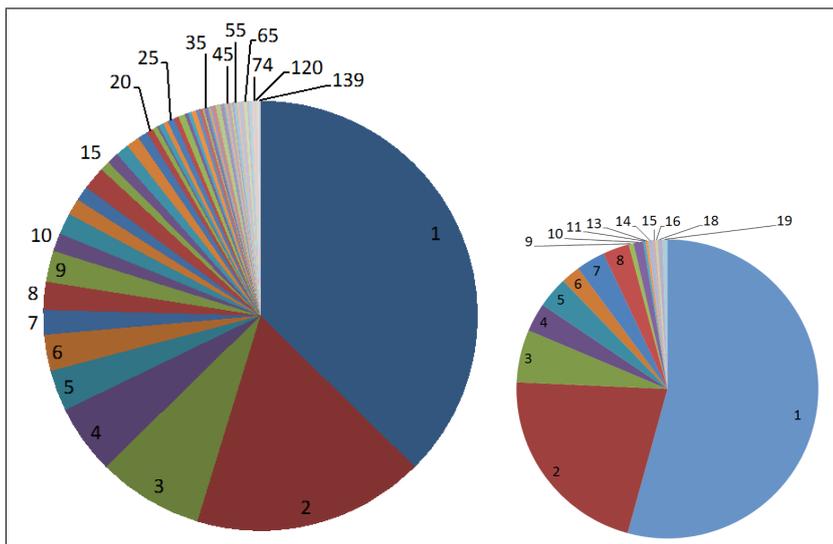


Figure 16.1: The number of target files in refiltered split file candidate groups (left), with the same information for the original set, repeating Fig. 12.12

Refiltered set				Original set			
File type	.c	.h	Total	File type	.c	.h	Total
Split	285	111	396	Split	130	64	194
Not	386	28	414	Not	146	47	193
Total	671	139	810	Total	276	111	387

Table 16.1: File type and classification of candidate split files

Table 16.2 shows the number of candidate files per project: the refiltered and original sets, and the change. Candidate files are selected for 67 projects, rather than 55. The majority of projects have 10 or fewer candidates, 17 projects have between 11 and 42. Exceptionally, Gwyddion, the second largest project, with 6.5 times the mean number of files, which includes a high level of incidental similarity, has 191 candidates. Figure 16.2 shows the distribution of candidate files, their type and class in the same way as the original set (Figure 12.13, p.190). Overall, the new filter conditions select more positive candidate split files, and fewer of the more certain negative candidates. The cost is the increased number of target files, but this should mean that the correct targets are more likely to be among those selected.

Project	Or.	Re.	Δ	Project	Or.	Re.	Δ	Project	Or.	Re.	Δ
acidblood	4	6	2	jack-rack	0	1	1	premake	6	23	17
aqtwo-tng	0	3	3	lde	2	7	5	pxlib	0	1	1
artoolkit	0	5	5	lejos	0	1	1	rcalc	5	6	1
beecrypt	4	3	-1	lgeneral	0	3	3	rsyslog	13	35	22
biew	4	3	-1	libbt	0	1	1	rtnet	7	15	8
cipe-linux	0	0	0	lifelines	22	42	20	seti-applet	1	3	2
dbacl	2	4	2	lirc	13	13	0	sf-xpaint	0	3	3
diald	2	3	1	logc	1	3	2	sonasound	5	8	3
drivel	1	3	2	mfstools	1	1	0	sphinxthree	11	15	4
dta	0	0	0	mjs	3	7	4	sphinxtwo	1	4	3
dynamics	9	22	13	mkcdrec	0	2	2	toxine	0	3	3
effectv	36	39	3	mpop	1	4	3	tulip	0	1	1
etherape	15	28	13	msmtip	3	0	-3	tuxnes	3	5	2
extace	1	2	1	nano	7	9	2	wmweather+	2	3	1
felt	0	3	3	nap	8	10	2	wxd	2	3	1
fidogate	6	7	1	noffle	2	5	3	xastir	3	32	29
ganc	2	3	1	nvramp-wakeup	1	2	1	xawdecode	4	15	11
gpsbabel	2	25	23	oww	2	5	3	xbae	1	17	16
gwyddion	83	191	108	pam-mysql	1	1	0	xmp	6	16	10
hatari	16	27	11	pbbuttons	4	4	0	ysmv	2	5	3
hptalx	1	1	0	pidgin-hotkeys	0	1	1	zimg	2	3	1
interest	22	42	20	pjo	5	7	2				
ipcop	3	7	4	ppcboot	24	38	14				
								Total	387	810	423

Table 16.2: Candidate split files by project: original set, refiltered set, and difference

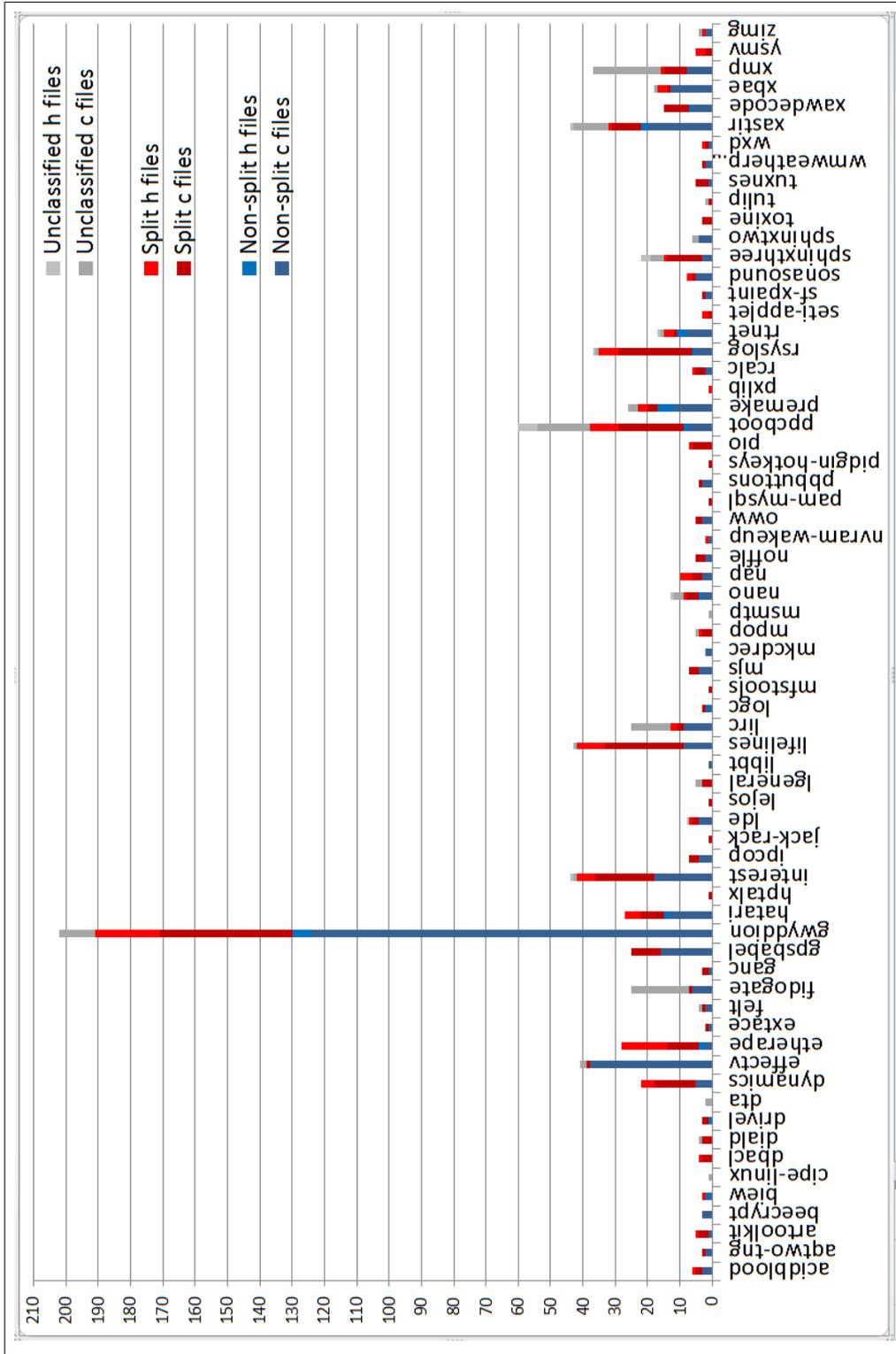


Figure 16.2: Refiltered split file dataset by project, type, and class

16.2 Classifying the refiltered split files

Each of the 11 algorithms listed in Table 16.6 (p.275) was applied to each of the feature sets explored in Chapter 14. The top 40 results, ranked by mean % correct classification, are listed in Table 16.3. The “best” mean classification accuracy on the test partitions for this set is around 4% lower than on the original set, 90% rather than 94%. However, the refiltered set includes more marginal examples, both positive and negative, than the original set, and also excludes some of the more certain negative examples. In this context, marginal means that little code is moved between the files.

	Feature set	Algorithm	Mean % correct	Std. Dev.	Mean prec'n	Mean recall	Mean F-measure
1	fc+ft+tris+pdp-singles	RotationForest	90.32	1.79	0.90	0.92	0.91
2	fc+tris+pdp-singles	LogitBoost	90.26	1.78	0.90	0.91	0.91
3	all-tris+all-fc	SimpleLogistic	90.23	1.56	0.88	0.93	0.91
4	fc+ft+fb+tris+pdp-s's	RotationForest	90.22	1.57	0.90	0.91	0.91
5	all-singles	LogitBoost	90.17	1.57	0.90	0.91	0.90
6	fc+fb+tris+pdp-singles	LogitBoost	90.15	1.81	0.90	0.91	0.90
7	fc+tris-singles	RotationForest	90.07	1.40	0.89	0.92	0.90
8	fc+ft+tris+pdp-singles	LogitBoost	90.04	1.72	0.90	0.91	0.90
9	fc+tris+pdp-singles	RotationForest	90.02	1.63	0.89	0.91	0.90
10	ft+fb+tris+pdp-singles	RotationForest	89.99	1.85	0.90	0.91	0.90
11	fc+ft+fb+tris+pdp-s's	LogitBoost	89.99	1.89	0.90	0.91	0.90
12	fc+fb+tris+pdp-singles	RotationForest	89.93	1.65	0.89	0.91	0.90
13	all-feats	LogitBoost	89.91	1.45	0.90	0.91	0.90
14	fall	SimpleLogistic	89.90	1.49	0.88	0.93	0.90
15	fc+fb+tris-singles	RotationForest	89.89	1.66	0.89	0.92	0.90
16	tris-singles	RotationForest	89.83	1.63	0.89	0.92	0.90
17	fb+tris+pdp-singles	RotationForest	89.80	1.60	0.89	0.91	0.90
18	fc+ft+tris-singles	RotationForest	89.79	1.63	0.89	0.92	0.90
19	fc+tris+pdp-singles	RandomForest	89.73	1.52	0.88	0.93	0.90
20	all-singles	RotationForest	89.73	1.71	0.89	0.91	0.90
21	all-tris+all-fd	SimpleLogistic	89.73	1.42	0.88	0.92	0.90
22	fc+tris-singles	SimpleLogistic	89.71	1.57	0.88	0.93	0.90
23	tris	RotationForest	89.71	1.51	0.89	0.91	0.90
24	pdp+tris-singles	RotationForest	89.71	1.54	0.89	0.91	0.90
25	fc+ft+tris-singles	SimpleLogistic	89.70	1.58	0.88	0.93	0.90
26	ft+tris+pdp-singles	RotationForest	89.69	1.65	0.89	0.91	0.90
27	fc+fb+tris+pdp-singles	RandomForest	89.69	1.46	0.88	0.92	0.90
28	all-feats	SimpleLogistic	89.68	1.43	0.88	0.92	0.90
29	all-tris+fc-singles	RotationForest	89.67	1.61	0.89	0.91	0.90
30	fc+tris+pdp-singles	SimpleLogistic	89.67	1.52	0.88	0.93	0.90
31	all-tris+fb-singles	RotationForest	89.67	1.71	0.89	0.91	0.90
32	all-tris+ft-singles	RotationForest	89.67	1.83	0.89	0.91	0.90
33	fc+ft+tris+pdp-singles	RandomForest	89.67	1.70	0.88	0.93	0.90
34	all-tris+all-pdp	LogitBoost	89.65	1.57	0.89	0.91	0.90
35	all-tris+pdp-singles	RotationForest	89.65	1.65	0.89	0.91	0.90
36	fc+ft+fb+tris-singles	SimpleLogistic	89.64	1.57	0.88	0.93	0.90
37	all-feats	RotationForest	89.63	1.62	0.89	0.91	0.90
38	all-tris+pdp-singles	LogitBoost	89.61	1.55	0.89	0.91	0.90
39	fc+ft+fb+tris-singles	RotationForest	89.60	1.55	0.89	0.92	0.90
40	fl+tris-singles	RotationForest	89.60	1.71	0.89	0.91	0.90

Table 16.3: Classifying refiltered split files: top 40 results, sorted by mean % correct.

Feature set	Mean % correct	Feature set	Mean % correct	Feature set	Mean % correct
fc+trispdp-singles	88.70	ft+fb+pdp-singles	86.66	ft	83.82
fc+fb+trispdp-singles	88.68	all-fb+all-pdp	86.38	fd+ccf-singles	83.69
fc+trisp-singles	88.54	all-fb+fc-singles	86.27	fb+ccf-singles	83.66
all-tris+fc-singles	88.49	ft+pdp-singles	86.25	fl-singles	83.30
fc+ft+trispdp-singles	88.48	fd+pdp-singles	86.08	fl	82.85
fc+fb+trisp-singles	88.45	fc+sim-singles	85.91	sim+ccf-singles	82.80
fc+ft+fb+trispdp-s's	88.44	fb+fc-singles	85.72	fd-singles	82.73
pdp+trisp-singles	88.27	fc+ft+fb-singles	85.64	fb-singles	82.72
fb+trispdp-singles	88.24	all-fb+all-fc	85.63	fd	82.44
all-trispdp-singles	88.23	fl+pdp-singles	85.56	sim-blocks-singles	81.77
fc+ft+fb+trisp-singles	88.20	ccf+pdp-singles	85.56	sim-singles	81.72
fc+ft+trisp-singles	88.17	all-fb+ft-singles	85.56	sim	81.44
ft+fb+trispdp-singles	88.09	fc+ccf-singles	85.53	ccf-blocks-singles	81.13
fl+trisp-singles	88.08	pdp-blocks-singles	85.49	ccf-singles	81.11
all-singles	88.06	pdp-singles	85.46	all-cats	80.93
ft+trispdp-singles	88.05	sim+pdp-singles	85.45	ccf	80.73
ft+fb+trisp-singles	87.91	not-fall-singles	85.44	fb-cats	80.59
all-tris+ft-singles	87.85	fc+fl-singles	85.29	fall-cats	80.56
ft+trisp-singles	87.84	not-fall	85.27	pdp-raw-singles	80.10
all-tris+all-fc	87.77	fc+ft-singles	85.15	fc-cats	79.70
all-feats	87.72	all-fb+ccf-singles	85.07	not-fall-cats	79.50
fd+trisp-singles	87.68	13-sub-12-blocks-s's	85.04	tris-cats	79.49
all-tris+fb-singles	87.59	pdp	85.01	ft-cats	78.99
fb+trisp-singles	87.59	ft+sim-singles	84.97	cat-2alikest	78.96
all-fb+trisp-singles	87.56	fc-singles	84.95	fl-cats	78.66
fall-singles	87.54	fb+fd-singles	84.84	pdp-cats	78.64
tris	87.51	all-fb+all-ft	84.82	pdp-blocks-cats	78.58
all-fb+all-tris	87.46	fc+fd-singles	84.79	fd-cats	78.43
tris-singles	87.45	fc	84.77	ccf-raw-singles	77.10
all-tris+all-pdp	87.43	fb+ft-singles	84.77	cat-am+news	77.08
sim+trisp-singles	87.33	fb+fl-singles	84.75	cat-all	76.97
fc+fb+pdp-singles	87.22	ft+ccf-singles	84.56	ccf-cats	76.82
all-tris+all-fl	87.21	fb	84.45	ccf-blocks-cats	76.69
all-fb+pdp-singles	87.15	fd+sim-singles	84.36	sim-blocks-cats	76.57
fc+ft+fb+pdp-singles	87.08	fb+sim-singles	84.32	sim-cats	76.56
all-tris+all-ft	87.00	ft+fl-singles	84.29	cat-am+main	76.00
ccf+trisp-singles	86.93	fd+fl-singles	84.23	cat-am+alikest	75.25
fc+pdp-singles	86.87	fl+ccf-singles	84.16	pdp-raw-cats	73.53
fall	86.86	all-fb+all-ccf	84.06	sim-raw-cats	72.74
fc+ft+pdp-singles	86.78	ft-singles	84.03	sim-raw-singles	72.66
fb+pdp-singles	86.72	fl+sim-singles	84.02	ccf-raw-cats	72.63
all-tris+all-fd	86.67	ft+fd-singles	83.95		

Table 16.4: Mean classification rates for the feature sets over the 11 algorithms

Feature set	Mean accuracy		Difference
	23 algorithms	11 algorithms	
tris-singles	92.11	93.02	0.99
fb+tris-singles	91.89	92.86	1.06
tris	91.91	92.83	1.00
fb-singles	91.27	92.34	1.17
pdp+tris-singles	91.44	92.05	0.67
all-singles	91.21	92.02	0.88
Mean difference			0.96

Table 16.5: Selected feature sets and their mean accuracy on the original dataset. Means are given over the 23 algorithms used with the original set and the 11 algorithms used with the refiltered set. The mean increase using the reduced set of algorithms over these feature sets is around 1%.

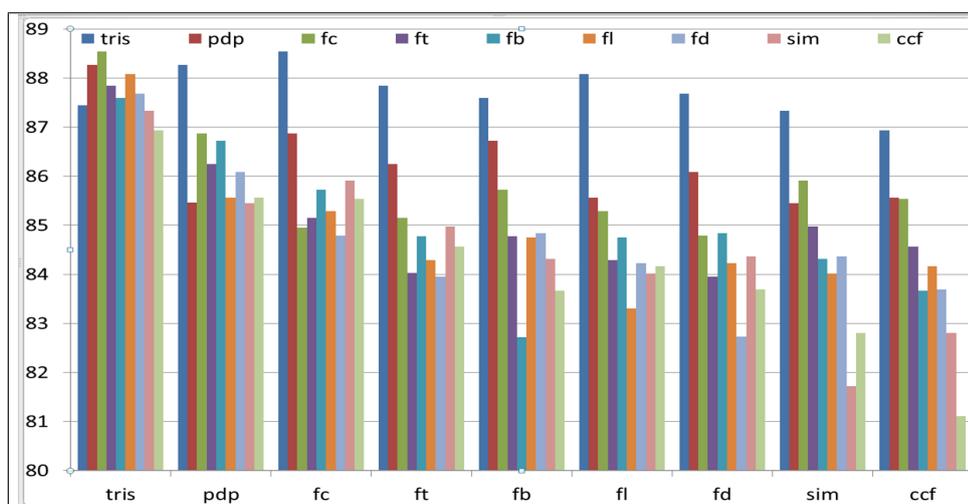


Figure 16.3: Split file classification with single feature sets and their pairwise combinations. X-axis labels show one of the pair, and the column colour the other. Where these are the same, the single set result is plotted.

Feature sets Table 16.4 shows mean classification rates for the feature sets over the 11 algorithms. These rates are not directly comparable with those for the original dataset, where 23 algorithms are averaged. To give an indication of the difference, Table 16.5 lists a selection of the better algorithms on the original dataset, giving the mean accuracy over the original 23 algorithms, and the 11 used here. Over this sample, the mean difference $\approx 1\%$. Figure 16.3 shows results for paired singles sets as before (see p.232). In contrast, here all of the single sets are improved by adding another set. The best combination being Ferret trigram- and character-based features.

Algorithms Table 16.6 gives the mean results for each algorithm over all feature sets. As with the original set, combining models based on different algorithms does not improve the accuracy of the better models.

Algorithm	Mean % correct	Algorithm	Mean % correct	Algorithm	Mean % correct
RotationForest	85.84	Decorate	83.90	LogitBoost	83.25
SimpleLogistic	85.29	RandomCommittee	83.83	SPegasos	82.30
RandomForest	84.80	FT	83.69	Dagging	81.35
SMO	84.08	SGD	83.52		

Table 16.6: Mean classification rate for each algorithm over all of the feature sets.

16.3 PostgreSQL & DNSjava split files

The results of classifying the refiltered split file data from the two unseen projects, PostgreSQL and DNSjava, are reported in this section. First the refiltered datasets are described, then classification results are presented.

16.3.1 Refiltered data

Figure 16.4 shows the relationship between the original and refiltered datasets. The DNSjava set is smaller, with half of the original negative examples removed from the set, and only 8 new ones. PostgreSQL has 42 of the original negative examples replaced by 135 new ones. The composition of the PostgreSQL and DNSjava datasets is summarised in Table 16.7.

There are 49 instances in the refiltered DNSjava set: 17 positive examples, 31 negatives and one indeterminate example, which is therefore not

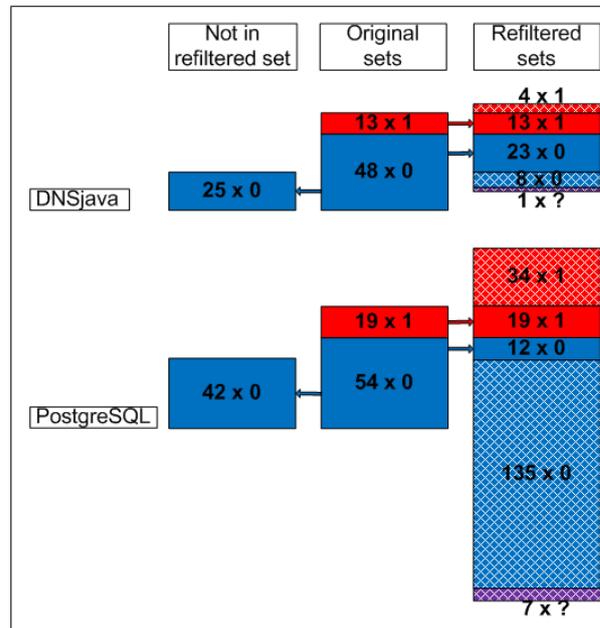


Figure 16.4: The relationship between original and refiltered DNSjava and PostgreSQL datasets. Original sets are in the middle, with split examples in red, and non-split examples in blue. The composition of the refiltered sets are shown on the right, with new instances cross-hatched. Examples from the original sets not in the refiltered sets are to the left.

	Refiltered set		Original set	
	PostgreSQL	DNSjava	PostgreSQL	DNSjava
Total split	53	17	19	13
Not split	147	31	54	48
Indeterminate	7	1	-	-
Total	207	49	73	61

Table 16.7: The composition of the refiltered PostgreSQL and DNSjava datasets

labelled. The positive examples include the 13 previously found; of the 4 additional examples, 3 have little code moved between files.

The refiltered PostgreSQL dataset comprises 207 instances, of which 147 are not split, 7 are indeterminate, and 53 are split files. These files include the 19 found previously; of the other 34 examples, 12 are marginal. An example of a file of indeterminate classification is shown in Figure 16.5¹ where 7 target files have the same similarity score, 8 of which share the same code with the disappearing file, and there are no unique trigrams in the group. It is difficult to judge whether the file is renamed or just incidentally similar to the target files, half of which are new.

¹also at <http://homepages.stca.herts.ac.uk/~gp2ag/xmls/port-protos-11.xml>

UH-Ferret: Trigram analysis		
Comparison generated by analysis of the trigram report produced by the Ferret Copy-Detection Tool, (c) School of Computer Science, University of Hertfordshire, 2010.		
Document: /home/.../postgresql-n/src/backend/port/linuxalpha/port-protos.h		
Blue file: /home/.../postgresql-n+1/src/backend/port/dynloader/unixware.h X	Blue file: /home/.../postgresql-n+1/src/backend/port/dynloader/univel.h O	Blue file: /home/.../postgresql-n+1/src/backend/port/dynloader/dgux.h O
Red file: /home/.../postgresql-n+1/src/backend/port/dynloader/sunos4.h O	Red file: /home/.../postgresql-n+1/src/backend/port/dynloader/irix5.h O	Red file: /home/.../postgresql-n+1/src/backend/port/dynloader/sco.h O
Yellow file: /home/.../postgresql-n+1/src/backend/port/dynloader/solaris_i386.h X	Yellow file: /home/.../postgresql-n+1/src/backend/port/dynloader/solaris_sparc.h X	Yellow file: /home/.../postgresql-n+1/src/backend/port/dynloader/freebsd.h X
<pre>#ifndef PORT_PROTOS_H #define PORT_PROTOS_H #include "fmgr.h" #include "utils/dynamic_loader.h" #include "dlopen.h" #define pg_dlopen(f) dlopen(f, 2) #define pg_dlsym dlsym #define pg_dlclose dlclose #define pg_dlerror dlerror #endif</pre>	<pre>#ifndef PORT_PROTOS_H #define PORT_PROTOS_H #include "fmgr.h" #include "utils/dynamic_loader.h" #include "dlopen.h" #define pg_dlopen(f) dlopen(f, 2) #define pg_dlsym dlsym #define pg_dlclose dlclose #define pg_dlerror dlerror #endif</pre>	<pre>#ifndef PORT_PROTOS_H #define PORT_PROTOS_H #include "fmgr.h" #include "utils/dynamic_loader.h" #include "dlopen.h" #define pg_dlopen(f) dlopen(f, 2) #define pg_dlsym dlsym #define pg_dlclose dlclose #define pg_dlerror dlerror #endif</pre>

Figure 16.5: The disappearing PostgreSQL file port-protos.h and 9 of its 12 target files. The file cannot be classified, as 8 targets share the same code with port-protos.h, and 7 have the same similarity to the file, 0.745.

16.3.2 Classifying unseen refiltered split file candidates

The models which best classify the 89 project refiltered set were used to classify the unseen data. Their accuracy and recall is listed in Table 16.8, sorted by overall accuracy. Also included in the table are other models which perform well on the unseen data. Combining groups of the better models (top 3, 5, 7, or 10 based on accuracy on the training data) by voting, either by majority or by mean probability, does not improve on the classification of the “fc+tris”/Rotation Forest model.

16.3.2.1 PostgreSQL

The three PostgreSQL files listed by Zou [261], `analyze.c`, `parser.c` and `date.c`, correctly classified by the original models, are correctly classified again. The other split file listed, `datetime.c`, is also now found and correctly classified. The new filter criteria also select all seven of the target files for `analyze.c`, unlike the original criteria. The two other files Zou reports are: `geqo_eval.c`, where eight functions are moved to `joinrels.c`, which is also correctly matched and classified; and `nbtcompare.c`, where four functions are moved to three other files. However, only one of the three qualifies as a target, the others have too few additional trigrams (12 and 22, $\approx 3\%$),

Rk.	Feature set	Alg'm.	Overall		PostgreSQL		DNSjava	
			Acc'y	Recall	Acc'y	Recall	Acc'y	Recall
7	fc+tris	ROT	0.944	0.873	0.950	0.906	0.917	0.824
46	fl+tris	SL	0.927	0.803	0.920	0.774	0.958	0.941
22	fc+tris	SL	0.923	0.803	0.935	0.811	0.896	0.824
4	fc+ft+fb+tris+pdp	ROT	0.919	0.831	0.920	0.849	0.917	0.824
3	all-tris+all-fc	SL	0.915	0.746	0.920	0.774	0.896	0.706
9	fc+tris+pdp	ROT	0.915	0.845	0.915	0.868	0.917	0.824
40	fl+tris	ROT	0.915	0.845	0.920	0.849	0.896	0.882
1	fc+ft+tris+pdp	ROT	0.915	0.803	0.910	0.811	0.938	0.824
6	fc+fb+tris+pdp	LB	0.915	0.803	0.905	0.792	0.958	0.882
8	fc+ft+tris+pdp	LB	0.903	0.761	0.905	0.774	0.896	0.765
5	all-singles	LB	0.891	0.775	0.890	0.792	0.896	0.765
10	fb+ft+tris+pdp	ROT	0.891	0.817	0.885	0.811	0.917	0.882
2	fc+tris+pdp	LB	0.891	0.775	0.885	0.774	0.917	0.824

Table 16.8: Classification accuracy and recall of selected models on PostgreSQL and DNSjava candidate split files, sorted by total accuracy. The column headed ‘Rk.’ shows the model’s rank in Table 16.4. ROT - Rotation Forest, SL - Simple Logistic, LB - Logit Boost, RAND - Random Forest.

consequently only one-third of the models tested classify the file as split.

In summary, there are 32 positive split file instances in the backend subsystem and 21 in other subsystems. The “fc+tris”/Rotation Forest model misclassifies 5 of these 53, and 5 of the 147 negative instances.

16.3.2.2 DNSjava

All of the DNSjava files found by the original filter conditions and correctly classified by the models are correctly classified after refiltering. UNKRecord.java and KEYBase.java were incorrectly classified by the original models. The KEYBase main target was ranked third, making classification difficult; the new ranking orders the target files correctly. The UNKRecord main target was not selected previously, but the new filtering both selects and orders the targets correctly. Both files are now correctly classified.

Four additional split candidates are found by refiltering, three of which are marginal examples: 2 lines, 4 lines (2 edited), or a few lines (all edited) are moved. The new models classify the thirteen original split file candidates and the one new non-marginal example accurately. However, the marginal examples are incorrectly classified by the majority of models tested.

In summary, both of the split files found by Antoniol et al. [6] are found and correctly classified by the “fc+tris”/Rotation Forest model. Nine other split files are found in the releases studied by Antoniol, of which seven are correctly classified. In the remaining releases, eight split files are found, all but one of which are correctly classified, as are all of the negative examples.

16.3.2.3 Overall

The best of the models on the unseen data, “fc+tris”/Rotation Forest, has an overall accuracy of 94.4% and recall of 87.3% (48 of 53 PostgreSQL, and 14 of 17 DNSjava) on the unseen data. The same feature set is in third place, with the Simple Logistic algorithm, the same combination which best classified the original unseen datasets. The reasons for misclassification varies: either little code (<3 lines) is moved, the code is heavily edited, or, as in the nbtcompare.c example, target files are not selected.

16.3.3 Additional unseen data: Struts and PyX

The models were also tested on two further projects: the Java project Struts, which has previously been analysed by several other origin analysis research groups [64, 208, 250], and the Python project PyX, which was analysed manually with reference to the change log.

16.3.3.1 Struts

Dig et al.'s [64] approach (see p.49), which first matches method bodies textually, then refines by call analysis, was tested on a pair of releases taken from each of three Java projects : Eclipse.UI (2.1.3–3.0), JHotDraw (5.2–5.3) and Struts (1.1–1.2.4).² Among the changes considered in their study, those found and reported are: renamed classes and methods, pulled up and moved methods, and changes to method signatures. Of these, moved and pulled up methods relate to split files. JHotDraw does not include any such changes; of the other two, Struts was chosen to provide a further test for Java files here. Dig et al.'s results for Struts are given in Appendix P, and show that the three files listed in Table 16.9 were split between releases 1.1 and 1.2.4.

Other than test and example files, which are not reported by other researchers, 14 candidate split files are found by filtering, and these are listed in Table 16.10. The models built with the “fc+tris-singles” or “fl+tris-singles” feature sets with Simple Logistic or Rotation Forest algorithms assign the same class to all of the files as the manual classification. Among these files are the three in Table 16.9, RequestUtils, ResponseUtils and ActionMapping, all of which are correctly classified. Four other files, Dy-

²Results are published at <http://netfiles.uiuc.edu/dig/RefactoringCrawler>

File	Split to
RequestUtils ResponseUtils ActionMapping	ModuleUtils and TagUtils TagUtils ActionConfig

Table 16.9: Split files found by Dig et al., Struts release 1.1 to 1.2.4

Candidate split files	MC	Cl'n
/src/share/org/apache/struts/action/ActionMapping.java *	1	1
/src/share/org/apache/struts/util/RequestUtils.java *	1	1
/src/share/org/apache/struts/util/ResponseUtils.java *	1	1
/contrib/struts-faces/src/java/org/apache/struts/faces/taglib/FormTag.java	1	1
/src/share/org/apache/struts/tiles/DefinitionsUtil.java	1	1
/src/share/org/apache/struts/validator/DynaValidatorActionForm.java	1	1
/src/share/org/apache/struts/validator/ValidatorActionForm.java	1	1
/src/share/org/apache/struts/action/Action.java	0	0
/src/share/org/apache/struts/action/ActionServlet.java	0	0
/src/share/org/apache/struts/config/ConfigHelper.java	0	0
/src/share/org/apache/struts/taglib/html/MessageTag.java	0	0
/src/share/org/apache/struts/taglib/logic/MessagePresentTag.java	0	0
/src/tiles-documentation/org/apache/struts/webapp/tiles/rssChannel/Channels.java	0	0
/src/tiles-documentation/org/apache/struts/webapp/tiles/rssChannel/ RssChannelsAction.java	0	0

Table 16.10: Classification of Struts candidate split files. MC is manual classification, Cl'n is class assigned by the fc+tris-singles/Rotation Forest or fc+tris-singles/Simple Logistic models.

naValidatorActionForm, ValidatorActionForm, DefinitionsUtil and FormTag are also split. The first two are confirmed by Wu et al. [250],³ (for details see p.51). Four methods are removed from DefinitionsUtil, these methods are already present in other files, one in ReloadableDefinitionsFactory, and three in TilesUtilImpl. Five methods are removed from contrib/.../struts/faces/taglib/FormTag, all of which are already present in the file src/.../struts/taglib/FormTag. Neither of these two files are reported by Wu et al. or Dig et al. The changes between these two releases were also tested by Schafer et al. [208], but these results are no longer available online.

16.3.3.2 PyX

The approaches to origin analysis surveyed in Chapter 3 are tested on C, Java or Smalltalk code. The Smalltalk code used by DeMeyer et al. [60] is no longer available online. Without relevant data from previous research, to test projects in other languages, they must be well-documented, so that the results can be assessed. Many projects in other languages were inspected with the aim of finding one with reasonable documentation, including around 40 projects in Python. PyX⁴ is a Python graphics package

³Whose results are available at www.ptidej.net/downloads/experiments/icse10b

⁴<http://pyx.sourceforge.net/index.html>

Ref.	Vn.	Candidate split files	MC	Doc.	Class'n
3	0.10	/examples/axis/log.py	0	-	1 *
3	0.10	/setup.py	0	-	0
5	0.8	/examples/axis/rating.py	0	-	0
5	0.8	/pyx/box.py	0	-	0
5	0.8	/pyx/pattern.py	0	-	0
5	0.8	/pyx/pdfwriter.py	0	-	0
5	0.8	/pyx/text.py	0	-	0
6	0.7	/pyx/graph/axis/axis.py	0	-	0
6	0.7	/pyx/graph/graph.py	0	-	0
6	0.7	/pyx/text.py	0	-	0
8	0.5	/pyx/mathtree.py	0	-	0
9	0.5	/pyx/graph.py	0	-	0
9	0.5	/pyx/t1strip/_init_.py	0	-	0
10	0.4	/pyx/data.py	0	-	0
3	0.10	/pyx/pdfwriter.py	1	Y	1
3	0.10	/pyx/pswriter.py	1	Y	1
5	0.8	/examples/bargraphs/compare.py	1	Y	1
5	0.8	/pyx/path.py	1	Y	1
5	0.8	/pyx/type1font.py	1	Y	1
6	0.7	/pyx/canvas.py	1	Y	1
6	0.7	/pyx/dvifile.py	1	N	1
7	0.6	/pyx/deco.py	1	Y	1
9	0.4	/pyx/canvas.py	1	Y	1
9	0.4	/pyx/helper.py	1	Y	1
9	0.4	/pyx/text.py	1	Y	1
10	0.3	/pyx/canvas.py	1	N	1
10	0.3	/pyx/graph.py	1	N	0 *

Table 16.11: Classification of PyX candidate split files. MC is manual classification; Doc: Y if in log, N if not; Class'n is class assigned by the models built with fc+tris-singles or fl+tris-singles feature sets and Rotation Forest or Simple Logistic algorithms.

for creating PostScript and PDF files. This project is the best documented among those considered, and was therefore selected as an additional test of the classification system.

There are 27 candidate split files, which are listed in Table 16.11, with their manual classification, whether the code movement is documented in the Change Log, and the system's classification.⁵ Moves which are not logged are assigned their manual classification only after confirmation by detailed code inspection. All but two (which are asterisked) of the candidates are correctly classified, a precision of 93%.

⁵The three "canvas.py" split files can be viewed at

[http://homepages.stca.herts.ac.uk/~gp2ag/xmls/canvas-0.3.xml\(/0.4.xml/0.7.xml\)](http://homepages.stca.herts.ac.uk/~gp2ag/xmls/canvas-0.3.xml(/0.4.xml/0.7.xml))

16.4 Disappearing files

This section reports on classifying the 89 project disappearing file dataset, introduced in Chapter 12. As already noted, a disappearing file for which there is no similar file in the system is assumed to have been deleted. When there is a close match between a disappearing file and a new file in the next release, it is assumed to have been moved (different directory, same name), renamed (same directory, different name) or both (different name and directory). It is less clear what has happened to the uncertain set, the files whose similarity falls between the two conservative thresholds, 0.05 and 0.85, and these are the files to which machine learning is applied. The datasets are described in Section 16.4.1 and their classification in Section 16.4.2. The experiments are similar to those for classifying split files reported in Section 16.2.

16.4.1 Data

Three classes are assigned to the uncertain set of candidate files:

1. There is no meaningful relationship between the disappearing file and any of the target files (class 0),
2. the file is renamed, moved, or merged with another (class 1), or
3. the file is split, forming new files or merging with existing files (class 2).

The candidates are divided into two groups: those with one target file and those with more. A disappearing file with only one target file cannot be split, it is also impossible to create a full set of features, as comparisons can only be made between the disappearing file and the target file. There are 177 files with one target and 575 with two or more. Of the 177 files, 105 are unrelated to their target, and 72 are renamed or merged, a ratio of 59:41. The set with more than one target is less balanced, with 255 unrelated, 261 renamed or merged, and 59 split files, a ratio of approximately 45:45:10.

Figure 16.6 (p.285) shows the target files per candidate. Approximately 85% of the files have 12 or fewer targets, and 4% have 50 or more files.

16.4.2 Classifying disappearing files with one target file

The eleven algorithms applied to the split file dataset were also used to classify the disappearing files with one target file (see Table 16.13). As shown in Table 16.12, the model with the highest mean classification rate, 95.06%, is the “fb-singles” set with the SGD algorithm. The mean accuracies of the feature sets over all of the algorithms are listed in Table 16.14.

	Feature set (all sets are singles)	Algorithm	Mean % correct	Std. Dev.	Mean prec'n	Mean recall	Mean F- measure
1	fb	SGD	95.06	2.36	0.950	0.969	0.959
2	fc+sim	FT	94.94	2.41	0.949	0.968	0.958
3	fb	RotationForest	94.89	2.68	0.954	0.962	0.957
4	fb+tris	SMD	94.84	2.52	0.946	0.969	0.957
5	fb+tris	SGD	94.79	2.62	0.950	0.964	0.957
6	fc+fb+tris+pdp	RandomForest	94.79	2.30	0.947	0.968	0.957
7	fc+fb+tris	RandomForest	94.64	2.29	0.946	0.966	0.955
8	tris	SGD	94.61	2.57	0.952	0.958	0.955
9	tris	SMD	94.51	2.42	0.944	0.966	0.954
10	fc+sim	LogitBoost	94.50	2.44	0.948	0.962	0.954
11	fc+fb+pdp	RotationForest	94.49	2.38	0.948	0.962	0.954
12	fc+sim	RotationForest	94.47	2.74	0.951	0.958	0.954
13	fb+pdp	SMD	94.44	2.40	0.941	0.969	0.954
14	fc+fb+tris	RotationForest	94.42	2.66	0.952	0.956	0.953
15	fb+fc	RotationForest	94.41	2.40	0.951	0.956	0.953
16	fb+fc	RandomForest	94.40	2.62	0.948	0.960	0.953
17	fc+tris+pdp	RandomForest	94.39	2.51	0.943	0.965	0.953
18	fc+sim	Decorate	94.38	2.49	0.939	0.970	0.954
19	fb+pdp	RotationForest	94.38	2.67	0.952	0.955	0.953
20	fc+sim	SimpleLogistic	94.38	2.62	0.939	0.969	0.954
21	fc+ft+fb+tris+pdp	RandomForest	94.37	2.40	0.942	0.966	0.953
22	fb+tris	RotationForest	94.37	3.20	0.948	0.960	0.953
23	fc+fb+tris+pdp	RotationForest	94.37	2.45	0.945	0.963	0.953
24	fb	SMD	94.36	2.52	0.936	0.972	0.954
25	fb+fd	RotationForest	94.36	2.94	0.949	0.959	0.953
26	fc+fb+pdp	LogitBoost	94.35	2.68	0.947	0.960	0.953
27	fc+fb+tris	SMD	94.34	2.58	0.941	0.968	0.953
28	fc+pdp	RotationForest	94.32	2.30	0.949	0.957	0.952
29	fc+fb+pdp	Decorate	94.31	2.50	0.940	0.968	0.953
30	fb+fc	RandomComm'ee	94.31	2.40	0.940	0.967	0.953
31	fc+tris	RandomForest	94.30	2.31	0.945	0.962	0.953
32	fc+fb+pdp	RandomForest	94.29	2.61	0.944	0.962	0.953
33	fc+pdp	SimpleLogistic	94.28	2.44	0.942	0.965	0.953
34	fc+sim	RandomForest	94.28	2.42	0.938	0.969	0.953
35	fc	LogitBoost	94.27	2.63	0.945	0.962	0.952
36	fc+ft+fb+pdp	LogitBoost	94.27	2.68	0.945	0.961	0.952
37	fc	RandomForest	94.26	2.47	0.944	0.962	0.952
38	fc+ft+fb	LogitBoost	94.24	2.69	0.944	0.961	0.952
39	fb	SimpleLogistic	94.23	2.83	0.941	0.965	0.952
40	fc+ft+fb+tris	RotationForest	94.22	2.57	0.948	0.957	0.952

Table 16.12: Classifying disappearing files with one target file: the top 40 results, sorted by mean % correct.

Algorithm	Mean % correct	Algorithm	Mean % correct	Algorithm	Mean % correct
RotationForest	91.63	SimpleLogistic	91.07	Dagging	89.75
RandomForest	91.41	SMO	90.78	SGD	89.68
LogitBoost	91.30	RandomCommittee	90.76	SPegasos	89.05
Decorate	91.08	FT	90.74		

Table 16.13: Mean classification rate for each algorithm over all feature sets on the disappearing files with one target file

Feature set (all singles sets)	Mean % correct	Feature set	Mean % correct	Feature set	Mean % correct
fc+fb+tris	93.89	fc+fd	93.04	fb+sim	91.32
fc+sim	93.89	ft+fb+tris	93.03	fl+pdp	90.89
fb+fc	93.82	fc+ccf	92.95	fl	90.66
fb+tris	93.74	ft+fb+tris+pdp	92.95	fd+fl	90.59
tris	93.74	fb+fd	92.86	fd+pdp	90.57
fc+tris	93.72	ft+tris	92.85	fl+sim	90.54
fc+ft+fb+tris	93.65	fb+ft	92.81	fl+ccf	90.28
fc+ft+fb+tris+pdp	93.64	pdp+tris	92.76	fd+sim	89.98
fc+fb+tris+pdp	93.63	ft+tris+pdp	92.68	fd	89.66
fb	93.62	fb+pdp	92.68	fd+ccf	88.98
fc	93.62	ft+fb+pdp	92.68	pdp-blocks	88.72
fc+fb+pdp	93.59	fd+tris	92.66	pdp	88.60
fc+ft+fb+pdp	93.55	ccf+tris	92.64	ccf+pdp	87.87
fc+ft+fb	93.54	sim+tris	92.40	pdp-raw	87.86
fc+tris+pdp	93.53	fl+tris	92.36	sim+pdp	87.23
fc+ft+tris+pdp	93.50	fb+ccf	92.16	not-fall	86.71
fc+pdp	93.49	fb+fl	92.15	sim-blocks	82.52
fc+ft+tris	93.49	ft+sim	92.08	sim+ccf	82.26
fc+ft	93.43	ft+pdp	92.07	sim	82.07
fc+ft+pdp	93.35	ft	92.07	ccf-blocks	75.21
fb+tris+pdp	93.16	ft+fd	91.85	ccf	75.19
fc+fl	93.08	ft+fl	91.78	sim-raw	74.51
all	93.06	ft+ccf	91.34	ccf-raw	72.33
fall	93.05				

Table 16.14: Mean classification rate for each feature set over the 11 algorithms on the disappearing files with one target file

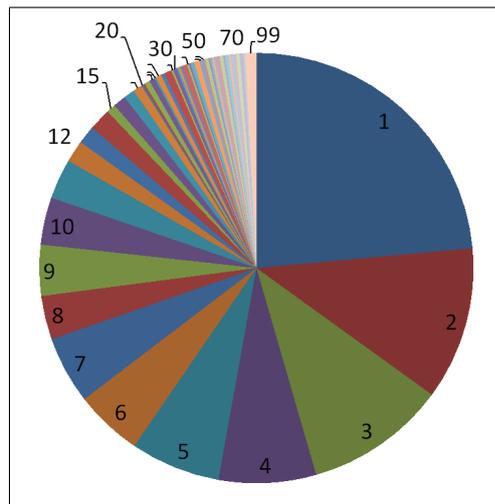


Figure 16.6: Analysis of the number of target files for the set of disappearing files

16.4.3 Disappearing files with more than one target file

As disappearing files with more than one target have three classes, two of the algorithms used previously, SGD and SPegasos, cannot be used with this dataset. Otherwise the experimental process is the same as before.

The top 40 results of classifying these disappearing files are listed in Table 16.15, sorted by mean % accuracy. It is noticeable that only one of the top 20 combinations (no.8) is based solely on comparisons between single files. This is in contrast to the results for classifying split files, where only 4 of the top 20 combinations include concatenated file comparisons.

	Feature set	Algorithm	Mean % correct	Std. Dev.	Mean prec'n	Mean recall	Mean F-measure
1	all-tris+all-fl	RotationForest	88.13	1.93	0.90	0.92	0.91
2	fall	RotationForest	87.75	1.72	0.90	0.92	0.91
3	all-feats	RotationForest	87.71	2.05	0.90	0.92	0.91
4	all-tris+fc-singles	RotationForest	87.57	1.90	0.90	0.90	0.90
5	all-fb+all-pdp	SimpleLogistic	87.50	2.02	0.90	0.92	0.91
6	all-tris+all-fc	RotationForest	87.31	1.89	0.90	0.90	0.90
7	tris	RotationForest	87.22	2.13	0.90	0.89	0.89
8	fl+tris-singles	RotationForest	87.20	2.04	0.88	0.92	0.90
9	all-cats	RotationForest	87.18	2.16	0.90	0.92	0.91
10	fall-cats	RotationForest	87.17	1.91	0.90	0.91	0.91
11	all-fb+tris-singles	RotationForest	87.14	2.02	0.90	0.90	0.90
12	all-fb+all-tris	RotationForest	87.13	1.92	0.90	0.90	0.90
13	all-tris+pdp-singles	RotationForest	87.08	1.95	0.89	0.90	0.90
14	tris	RandomForest	87.08	1.81	0.89	0.91	0.90
15	all-tris+all-pdp	RotationForest	87.06	2.04	0.89	0.91	0.90
16	all-tris+fc-singles	RandomForest	87.00	1.84	0.89	0.92	0.90
17	all-tris+pdp-singles	RandomForest	86.99	1.80	0.88	0.91	0.90
18	all-tris+all-pdp	SimpleLogistic	86.93	2.05	0.89	0.92	0.90
19	all-tris+all-fl	LogitBoost	86.87	2.09	0.89	0.91	0.90
20	all-tris+all-pdp	LogitBoost	86.86	2.05	0.89	0.91	0.90
21	fc+fb+tris+pdp-singles	RotationForest	86.84	2.19	0.88	0.91	0.90
22	all-fb+all-tris	RandomForest	86.81	1.91	0.89	0.91	0.90
23	all-feats	LogitBoost	86.80	2.18	0.89	0.91	0.90
24	pdp+tris-singles	RotationForest	86.78	1.94	0.88	0.91	0.89
25	fc+tris+pdp-singles	RotationForest	86.76	1.96	0.88	0.91	0.89
26	fall	LogitBoost	86.73	1.99	0.89	0.91	0.90
27	all-feats	RandomForest	86.72	1.90	0.90	0.92	0.91
28	all-fb+tris-singles	RandomForest	86.70	1.75	0.89	0.91	0.90
29	all-fb+pdp-singles	SimpleLogistic	86.67	2.07	0.90	0.91	0.90
30	all-fb+pdp-singles	RotationForest	86.66	1.99	0.89	0.90	0.90
31	fc+tris-singles	RotationForest	86.60	1.92	0.88	0.91	0.89
32	all-tris+ft-singles	RotationForest	86.59	1.90	0.89	0.90	0.89
33	all-tris+all-fc	RandomForest	86.57	1.91	0.89	0.92	0.90
34	all-fb+all-pdp	FT	86.56	2.09	0.90	0.90	0.90
35	all-fb+pdp-singles	RandomForest	86.54	1.86	0.88	0.92	0.90
36	fc+ft+tris-singles	RotationForest	86.53	2.14	0.88	0.91	0.89
37	all-tris+all-pdp	FT	86.52	2.13	0.89	0.91	0.90
38	fc+ft+fb+tris+pdp-singles	RotationForest	86.52	2.20	0.88	0.91	0.89
39	all-tris+all-ft	RotationForest	86.50	2.03	0.88	0.90	0.89
40	all-fb+all-fc	RotationForest	86.47	1.95	0.90	0.90	0.90

Table 16.15: Classifying disappearing files with at least two target files: the top 40 results, sorted by mean % correct.

A similar pattern can be seen in Table 16.16, where the feature sets are ranked by classification accuracy over the 9 algorithms. None of the sets in the top 20 are based solely on single file comparisons. Also noticeable is that two of the sets (“all-cats” and “fall-cats”) are based only on concatenated file comparisons, whereas for split files, the 22 concatenation sets are in the bottom 27 of the 116 sets. For both split and disappearing files, the CCFinder and Simian sets perform poorly, as do the raw P-Duplo sets.

As this dataset is imbalanced, it is useful to assess the performance of the models using the geometric mean, as discussed in Chapter 14. The geomet-

Algorithm	Mean % correct	Algorithm	Mean % correct	Algorithm	Mean % correct
all-tris+all-fl	86.02	fc+fb+tris-singles	83.84	pdp-singles	82.08
all-tris+all-pdp	85.78	fc+tris-singles	83.84	pdp-cats	82.05
all-tris+fc-singles	85.74	fb+tris+pdp-singles	83.81	fc+fd-singles	82.04
all-feats	85.63	ft+tris+pdp-singles	83.77	ft-singles	82.02
all-tris+all-fc	85.61	pdp+tris-singles	83.75	fl+ccf-singles	81.98
tris	85.53	cat-am+alikest	83.68	pdp-blocks-cats	81.97
all-tris+pdp-singles	85.48	fc+fb+pdp-singles	83.63	fc+sim-singles	81.91
all-fb+all-tris	85.43	fc+ft+fb+tris-s's	83.56	fb+fd-singles	81.78
all-fb+all-pdp	85.38	fc+ft+tris-singles	83.50	cat-all	81.75
fall	85.33	fd	83.48	ft+sim-singles	81.74
all-fb+tris-singles	85.30	fd-cats	83.40	cat-am+main	81.65
all-tris+ft-singles	85.26	ft+fb+tris-singles	83.39	ft+fd-singles	81.63
all-fb+pdp-singles	85.23	fc+pdp-singles	83.37	fl+sim-singles	81.62
all-fb+all-fc	85.19	fc+ft+fb+pdp-s's	83.36	ft+ccf-singles	81.61
fall-cats	85.16	fc+ft+pdp-singles	83.35	sim+pdp-singles	81.36
all-fb+fc-singles	85.00	ft+tris-singles	83.27	fb+ccf-singles	81.29
all-cats	84.96	not-fall-cats	83.27	ccf+pdp-singles	81.21
all-tris+all-ft	84.75	not-fall	83.25	fd-singles	81.21
fb	84.71	fall-singles	83.21	fd+ccf-singles	81.18
all-fb+ft-singles	84.70	all-singles	83.17	fd+sim-singles	81.01
fc	84.63	fl+pdp-singles	83.14	fb+sim-singles	80.94
fl	84.62	ft+fb+pdp-singles	83.13	cat-am+news	80.75
fb-cats	84.59	fb+fl-singles	83.11	not-fall-singles	80.74
all-tris+all-fd	84.44	ft+pdp-singles	83.07	pdp-raw-singles	79.61
fc+fb+tris+pdp-singles	84.36	fc+fl-singles	83.04	pdp-raw-cats	79.61
all-fb+all-ft	84.31	sim+tris-singles	82.83	sim	77.67
fl+tris-singles	84.29	fb+fc-singles	82.82	sim+ccf-singles	77.43
tris-cats	84.29	fd+tris-singles	82.78	sim-blocks-singles	76.79
fc-cats	84.28	ccf+tris-singles	82.77	sim-singles	76.67
fl-cats	84.24	ft+fl-singles	82.73	sim-blocks-cats	76.42
fc+tris+pdp-singles	84.24	fb+pdp-singles	82.67	sim-cats	76.41
pdp	84.20	fc+ft+fb-singles	82.59	ccf	73.14
fc+ft+fb+tris+pdp-s's	84.03	fc+ft-singles	82.57	sim-raw-cats	72.69
fc+ft+tris+pdp-singles	84.02	fl-singles	82.51	ccf-blocks-singles	71.67
all-fb+all-ccf	83.99	fc-singles	82.50	ccf-singles	71.54
ft	83.97	fb-singles	82.42	ccf-cats	71.25
all-tris+fb-singles	83.95	fd+fl-singles	82.41	sim-raw-singles	71.05
fb+tris-singles	83.91	cat-2alikest	82.35	ccf-blocks-cats	71.02
all-fb+ccf-singles	83.91	pdp-blocks-singles	82.25	ccf-raw-singles	68.78
ft-cats	83.86	fb+ft-singles	82.23	ccf-raw-cats	68.43
ft+fb+tris+pdp-singles	83.84	fc+ccf-singles	82.15		
tris-singles	83.84	fd+pdp-singles	82.13		

Table 16.16: Mean classification rate of disappearing files with at least two target files for each feature set over the 9 algorithms.

Algorithm	Mean % correct	Algorithm	Mean % correct	Algorithm	Mean % correct
RotationForest	83.71	SimpleLogistic	81.98	SMO	80.50
RandomForest	83.42	Decorate	81.96	FT	80.29
RandomCommittee	82.58	LogitBoost	81.96	Dagging	78.97

Table 16.17: Mean classification rate for each algorithm over all feature sets on the disappearing files with at least two target files

ric mean of accuracies is not available directly from the Weka experimenter which was used to run the experiments; to calculate it, accuracy values were taken from individual 10-fold cross-validated runs. In Table 16.18, the accuracy on each class of each of the top ten models from Table 16.15 is noted along with their geometric mean. Under this measure, the singles set “fl+tris-singles” is most accurate.

Strategies for dealing with imbalance in the classes are reported in Appendix O. These strategies are: over-sampling and under-sampling the data, and use of cost-based algorithms. In most of the tests reported there are no significant changes. Models built using data over-sampled by the SMOTE algorithm tend to increase the correct classification of the minority class by one instance on the Random Forest models tested. However, this effect is not repeated in the better models, such as the “all-tris+all-fl”/SimpleLogistic model, which already classifies 11 of the 14 instances correctly. Consequently, tests on the unseen data are reported for models built without adjustment for the imbalance.

Feature set	Algorithm	Unrelated	Renamed	Split	Geo.mean
fl+tris-singles	ROT	0.929	0.885	0.729	0.843
all-fb+all-pdp	SL	0.906	0.897	0.712	0.833
all-tris+all-fl	ROT	0.922	0.904	0.661	0.820
all-tris+fc-singles	ROT	0.890	0.908	0.678	0.818
all-tris+all-fc	ROT	0.898	0.893	0.678	0.816
tris	ROT	0.906	0.877	0.661	0.807
fall-cats	ROT	0.910	0.893	0.627	0.799
fall	ROT	0.902	0.897	0.627	0.797
all-feats	ROT	0.918	0.889	0.576	0.778
all-cats	ROT	0.929	0.893	0.508	0.750

Abbreviations: ROT - Rotation Forest, SL - Simple Logistic

Table 16.18: Geometric means of accuracy for the top 10 results from Table 16.15

16.5 PostgreSQL and DNSjava disappearing files

Disappearing files from the two unseen projects were classified using the models built from the refiltered 89 project datasets. Lists of the matched and unmatched files for these projects can be found in Appendix N. The results for candidates with one target from the uncertain set are reported in the next section, and those with two or more in Section 16.5.2.

16.5.1 Candidates with one target file

Nineteen of the PostgreSQL disappearing files have only one target, and are listed in Table 16.19, with those in the backend subsystem at the top. All of these files are correctly classified by the ten top-performing models on the 89 project dataset, except for the “fc+sim”/FT model which misclassifies one positive example.

The five DNSjava disappearing files listed in Table 16.20 have one target; three are renamed, one unrelated, and one, KeyConverter, is difficult to classify, for several reasons. Although some of the functionality from KeyConverter is merged with the file DNSSEC, the code is edited and scattered.

Ref.	Vn.	Path			File	MC	Model
10	6.4.2	backend	access	common	heapvalid.c	0	0
6	6.5.3	backend	optimizer	path	mergeutils.c	0	0
6	6.5.3	backend	utils	sort	psort.c	0	0
4	7.0.3	backend	libpq		be-pqexec.c	0	0
4	7.0.3	backend	port	hpux	fixade.h	0	0
4	7.0.3	backend	storage	lmgr	multi.c	0	0
2	7.1.3	backend	access	transam	transsup.c	0	0
2	7.1.3	backend	storage	buffer	s_lock.c	1	1
2	7.1.3	backend	utils	mb	utfest.c	0	0
12	6.2	include	catalog		pg_defaults.h	0	0
12	6.2	include	catalog		pg_magic.h	0	0
12	6.2	include	catalog		pg_user.h	1	1
10	6.4.2	pl	plpgsql	src	scan.c	1	1
6	6.5.3	include	lib		qsort.h	0	0
4	7.0.3	include	port		solaris_i386.h	1	1
4	7.0.3	include	regex		cdefs.h	0	0
4	7.0.3	include	utils		module.h	0	0
2	7.1.3	include	catalog		pg_inheritproc.h	0	0
2	7.1.3	include	storage		multilev.h	0	0

Table 16.19: Disappearing PostGreSQL files with one target and their classifications determined both by inspection and by the fb-singles/SGD model. Every file is correctly classified by the fb-singles/SGD model.

Ref.	Vn.	Path				File	MC	Model	Notes
46	12			DNS		CacheResponse.java*	1	1	
46	12			DNS		ZoneResponse.java*	1	1	
12	48	org	xbill	DNS	utils	hmacSigner.java	1	1	Edited Merged?
2	56	org	xbill	DNS	security	KEYConverter.java	1?	0	
2	56	org	xbill	DNS	security	SIG0Signer.java	0	0	

Table 16.20: Disappearing DNSJava files with one target and their classifications determined both by inspection and by the fb-singles/SGD model. Files from the versions 1–39 are at the top of the table, the rest at the bottom.

It is also a small part of the code in the new version of DNSSEC which is more than four times larger than in the previous release.

16.5.2 Candidates with two or more target files

PostgreSQL has 106 uncertain disappearing files with two or more targets, and DNSJava has 23. Six of the PostgreSQL files are indeterminate, these are small header files with a large number of similar files in the system. This leaves 100 files, of which 57 are unrelated to their targets, 30 are renamed or merged, and 13 are split. The DNSJava files consist of 8 unrelated files, 13 renamed or merged files, 1 split file, and 1 indeterminate file. The two split and renamed PostgreSQL files identified by Zou, `catalog_utils.c` and `dt.c`, are asterisked in Table 16.23. However, the target file for `geqo_paths.c` (the other asterisked file) is not selected because their shared trigrams are reduced from 231 to 119, (see Figure N.1, p.416).

Over one hundred models were tested with this data, including those selected using the results from the 89 project dataset, those based on datasets balanced by under- and over-sampling, and cost-based classifiers. The model built using the combination which best classifies the 89 project dataset, (“all-tris+all-fc”/Rotation Forest) correctly classifies 108 of 122 instances (88.5%) and 8 of the 14 instances in the minority class. Of those tested, three models based on the 89 project dataset, with no adjustment for the imbalance, give the best classification of the unseen data. Although the Simple Logistic algorithm is ranked fourth overall in classifying the 89

Model	Feature set	Algorithm	Classn.	Splits
'Best'	all-tris-all-fl	Rotation Forest	108/122	8/14
A	all-tris+all-fl	Simple Logistic	110/122	11/14
B	all-tris+all-fc	Simple Logistic	110/122	8/14
C	fc+tris-singles	Simple Logistic	111/122	9/14
D 'Best' singles set	fl+tris-singles	Rotation Forest	109/122	10/14

Table 16.21: The 'best' model over the 89 project dataset and the four models, A, B, C, and D, and their classification of the PostgreSQL and DNSjava disappearing files with at least two targets.

project dataset, each of these models use this algorithm, with the feature sets "all-tris+all-fl" (model A), "all-tris+all-fc" (B) and "fc+tris-singles" (C). A fourth model, D, "fl+tris-singles"/Rotation Forest, gives the best classification on the 89 project data among the singles sets. The classifications of the four models are shown in Tables 16.21–16.23. These models correctly classify 110, 110, 111 and 109 instances out of 122, respectively, with 11, 8, 9 and 10 of the 14 split files correctly classified.

Ref.	Vn.	Path				File	MC	Model			
								A	B	C	D
55	3					dnsServer.java*	1				
54	4			DNS	utils	CountedDataInputStream.java*	1				
54	4			DNS	utils	CountedDataOutputStream.java*	1	0	0	0	0
51	7			DNS		FindResolver.java*	1				
47	11			DNS		IO.java	1				2
45	13			DNS		DNSSEC.java	1				
45	13			DNS		EDNS.java	1				
45	13			DNS		ExtendedResolver.java	1				
45	13			DNS		Resolver.java	1				
45	13			DNS		ResolverListener.java	1				
45	13			DNS		RRset.java	1				
45	13			DNS		WorkerThread.java	2				
38	20	org	xbill	DNS		EDNS.java	0				
32	26	org	xbill	DNS		TypeClassMap.java	0	1			
24	34	org	xbill	DNS	utils	MyStringTokenizer.java	0				
17	41	org	xbill	DNS	utils	DataByteInputStream.java	0				
17	41	org	xbill	DNS	utils	DataByteOutputStream.java	0				
17	41	org	xbill	DNS		MX_KXRecord.java	1?	0	0	0	0
17	41	org	xbill	DNS		NS_CNAME_PTRRecord.java	0				
11	47	org	xbill	DNS		dns.java	0				
11	47	org	xbill	DNS		FindServer.java	0				
11	47	org	xbill	DNS		Inet6Address.java	1				

Table 16.22: Disappearing DNSjava files with two or more targets and their classifications, determined both by inspection and by the four models. In the last 3 columns, the incorrect class is noted against instances which are misclassified by the models.

Three instances are misclassified by every one of the models tested, those shaded grey in Tables 16.22 (DNSjava), and 16.23 (PostgreSQL). A question mark indicates files which are difficult to classify manually.

Ref.	Vn.	Path			File	MC	Model			
							A	B	C	D
12	6.2	backend	commands		purge.c	0				
12	6.2	backend	lib		qsort.c	0				
12	6.2	backend	optimizer	prep	archive.c	0				
12	6.2	backend	parser		catalog_utils.c*	2				
12	6.2	backend	parser		dbcommands.c	1				
12	6.2	backend	parser		parse_query.c	2				
12	6.2	backend	parser		sysfunc.c	0				
12	6.2	backend	port	aix	dlfcn.h	1				
12	6.2	backend	port	aix	port_protos.h	1				
12	6.2	backend	port	alpha	port.c	1				
12	6.2	backend	port	hpux	port.c	2		0	0	0
12	6.2	backend	port	i386_solaris	port.c	2		1	1	
12	6.2	backend	port	i386_solaris	port_protos.h	1	0		0	
12	6.2	backend	port	irix5	port.c	0				1
12	6.2	backend	port	sco	port.c	1				
12	6.2	backend	port	sparc_solaris	port.c	2		0		
12	6.2	backend	port	svr4	port.c	2				
12	6.2	backend	port	ultrix4	port.c	0				
12	6.2	backend	port	ultrix4	port_protos.h	0				
12	6.2	backend	port	ultrix4	strdup.c	0		1		
12	6.2	backend	port	univel	port.c	2				
12	6.2	backend	port	univel	port_protos.h	0				
12	6.2	backend	tcpop		variable.c	1				
11	6.3.1	backend	optimizer	util	internal.c	1?	0	0	0	0
11	6.3.1	backend	regex		utils.c (4 line edit)	2	1	1	1	1
11	6.3.1	backend	regex		wstrcmp.c	1				
11	6.3.1	backend	utils	adt	oidint2.c	0				
11	6.3.1	backend	utils	adt	oidint4.c	0				
11	6.3.1	backend	utils	adt	oidname.c	0				
10	6.4.2	backend	libpq		pqcomprim.c	1?				2
10	6.4.2	backend	optimizer	geqo	geqo_paths.c*	0				
10	6.4.2	backend	optimizer	path	joinutils.c	1				
10	6.4.2	backend	optimizer	util	clauseinfo.c	1				2
6	6.5.3	backend	optimizer	geqo	minspantree.c	0				
6	6.5.3	backend	optimizer	path	hashutils.c	0				
6	6.5.3	backend	optimizer	path	prune.c	0				
6	6.5.3	backend	optimizer	util	keys.c	0				
6	6.5.3	backend	optimizer	util	ordering.c	0				
6	6.5.3	backend	utils	adt	dt.c*	2				
4	7.0.3	backend	lib		fstack.c	0				
4	7.0.3	backend	optimizer	geqo	geqo_params.c	0				
4	7.0.3	backend	optimizer	util	indexnode.c	0				
4	7.0.3	backend	port	dynloader	solaris_i386.h	1				
4	7.0.3	backend	port	dynloader	solaris_sparc.h	1				
4	7.0.3	backend	port	hpux	port_protos.h	0				
4	7.0.3	backend	rewrite		locks.c	1?	0	0	0	0
4	7.0.3	backend	storage	lmgr	single.c	0				
4	7.0.3	backend	utils	adt	filename.c	0				
4	7.0.3	backend	utils	adt	lztext.c	0				
4	7.0.3	backend	utils	init	enbl.c	0				
4	7.0.3	backend	utils	mb	variable.c	1				
4	7.0.3	backend	utils	misc	trace.c	0				
4	7.0.3	backend	utils	mmgr	oset.c	0				
4	7.0.3	backend	utils	mmgr	palloc.c	1?	0	0	0	0
2	7.1.3	backend	access	nbtree	nbtscan.c	0				
2	7.1.3	backend	lib		hasht.c	0				
2	7.1.3	backend	libpq		pqpacket.c	0				
2	7.1.3	backend	storage	ipc	spin.c	1			2	2

Continued on next page

cont'd.							Model			
Ref.	Vn.	Path			File	MC	A	B	C	D
2	7.1.3	backend	utils	cache	rel.c	0				
2	7.1.3	backend	utils	mb	common.c	1				
2	7.1.3	backend	utils	mb	liketest.c	1	0	0		
2	7.1.3	backend	utils	mb	palloc.c	0				
2	7.1.3	backend	utils	mb	sjistest.c	0				
6	6.5.3	bin	psql		psql.c	2	0	0	0	0
4	7.0.3	bin	pg.version		pg.version.c	0				
12	6.2	include	commands		purge.h	0?				
12	6.2	include	parser		catalog_utils.h	2				
12	6.2	include	parser		dbcommands.h	1				
12	6.2	include	parser		parse_query.h	2				
12	6.2	include	parser		parse_state.h	1				
12	6.2	include	port		BSD44_derived.h	1				
12	6.2	include	tcop		variable.h	1				
11	6.3.1	include	regex		pg_wchar.h	1				
10	6.4.2	include	executor		nodeTee.h	0				
10	6.4.2	include	optimizer		clauseinfo.h	1				
10	6.4.2	include	optimizer		geqo_paths.h	0				
6	6.5.3	include	optimizer		keys.h	0				
6	6.5.3	include	optimizer		ordering.h	0				
6	6.5.3	include	utils		dt.h	2				
6	6.5.3	include	utils		rel2.h	0				
4	7.0.3	include	optimizer		internal.h	0				
4	7.0.3	include	port		alpha.h	0				
4	7.0.3	include	port		bsd.h	1				
4	7.0.3	include	port		solaris_sparc.h	1	0			
4	7.0.3	include	rewrite		locks.h	0				
4	7.0.3	include	utils		fcache2.h	0				
4	7.0.3	include	utils		lztext.h	0				
4	7.0.3	include	utils		mcxt.h	0				
4	7.0.3	include	utils		trace.h	0				
2	7.1.3	include	access		giststrat.h	0				
2	7.1.3	include	access		rtstrat.h	0				
2	7.1.3	include	lib		hasht.h	0				
11	6.3.1	interfaces	libpq		fe-connect.h	0				
6	6.5.3	interfaces	ecpg	lib	ecpglib.c	2				
6	6.5.3	lextest			lextest.c	0				
10	6.4.2	pl	plpgsql	src	gram.c	1	2			
4	7.0.3	test	examples		testlo2.c	1				
4	7.0.3	utils			version.c	0				
4	7.0.3	win32			endian.h	0				

Table 16.23: Disappearing PostgreSQL files with more than one target. The manual classification is in the column labelled 'MC'. In the last 4 columns, the incorrect class is noted against instances which are misclassified.

16.5.2.1 PostgreSQL

Zou notes the following file moves:

- 6.2 to 6.3.2 [261, p.77]:
 - tcop/acclchk.c to catalog/acclchk.c
 - tcop/variable.c to commands/variable.c
 - parser/dbcommands.c to commands/dbcommands.c
- 6.4.2 to 6.5 [261, p.80]:

- `commands/defind.c` to `commands/indexcmds`
- `optimizer/joinutils.c` to `optimizer/pathkeys.c`
- `optimizer/clauseinfo.c` to `optimizer/restrictinfo.c`

The files `aclchk.c` and `defind.c` are automatically matched to their new destinations because the similarities are 0.94 and 0.96 respectively. The other four files are matched to the correct targets and correctly classified by all of the models tested. Their similarities are 0.70, 0.82, 0.40 and 0.80.

16.5.2.2 DNSjava

In release 13 of the DNSjava project, forty-nine files are moved from `/DNS` to `/org/xbill/DNS`, and in release 11, one file is moved from `/DNS` to `/DNS/Utils`. These files have a similarity of at least 0.85 and bear the same file name as each other. Files with no qualifying targets in the next release are assumed to be deleted. These sets of files are listed in appendix N, in Tables N.3 and N.4.

Files falling between the two similarity values 0.05 and 0.85 are listed in Tables 16.20 and 16.22. Of these 27 disappearing files, 5 are part of the large-scale move of files in the directory `/DNS` to the new subdirectory `/org/xbill/DNS`, and 1 is split, all are correctly classified.

Six files are identified by Antoniol et al. as renamed or merged, and are asterisked in the two tables. The merge of `CacheResponse` and `ZoneResponse` is captured, see Figure 16.7, as both of these files are renamed to the same target, `SetResponse`. Both `dnsServer` and `FindResolver` are successfully identified as renamed files. The files `CountedDataInputStream` and `CountedDataOutputStream` are successfully matched with their destination files as the first target file, but because the 'Output' file is heavily edited, it is not classified as renamed. This file was not found by Antoniol et al.'s system either, and the 'Input' file caused difficulties because it appeared to have split to become `DataByteInputStream` and `DataByteOutputStream`. In this system classification of the 'Input' file is indefinite, in that around a half of the models tested correctly classify the file, and the majority of models have low probabilities one way or the other.

UH-Ferret: Trigram analysis

Comparison generated by analysis of the trigram report produced by the Ferret Copy-Detection Tool, (c) School of Computer Science, University of Hertfordshire, 2010.

Document: /home/. . ./dnsjava-DNS/SetResponse.java

Blue file: /home/. . ./dnsjava-(n+1)/DNS/CacheResponse.java

Red file: /home/. . ./dnsjava-(n+1)/DNS/ZoneResponse.java

Yellow file: No third file

```

package DNS;
import java.util.*;
import DNS.utils.*;
public class SetResponse
{
    static final byte UNKNOWN = 0;
    static final byte NEGATIVE = 1;
    static final byte NOGAIN = 2;
    static final byte NODATA = 3;
    static final byte PARTIAL = 4;
    static final byte SUCCESSFUL = 5;
    private byte type;
    private Object data;
    private Vector backtrace;
    private SetResponse ()
    {
    }
    SetResponse (byte _type, Object _data)
    {
        type = _type;
        data = _data;
    }
    SetResponse (byte _type)
    {
        this (_type, null);
    }
    void set (byte _type, Object _data)
    {
        type = _type;
        data = _data;
    }
    void addRRset (RRset rrset)
    {
        if (data == null)
        data = new Vector ();
        Vector v = (Vector) data;
        v.addElement (rrset);
    }
    void addCNAME (CNAMERecord cname)
    {
        if (backtrace == null)
        backtrace = new Vector ();
        backtrace.insertElementAt (cname, 0);
    }
    public boolean isUnknown ()
    {
        return (type == UNKNOWN);
    }
    public boolean isNegative ()
    {
        return (type == NEGATIVE);
    }
    public boolean isNOGAIN ()
    {
        return (type == NOGAIN);
    }
    public boolean isNODATA ()
    {
        return (type == NODATA);
    }
    public boolean isPartial ()
    {
        return (type == PARTIAL);
    }
    public boolean isSuccessful ()
    {
        return (type == SUCCESSFUL);
    }
    public ERset[] answers ()
    {
        if (type != SUCCESSFUL)
        return null;
        Vector v = (Vector) data;
        ERset[] rrssets = new ERset[v.size ()];
        for (int i = 0; i < rrssets.length; i++)
        rrssets[i] = (ERset) v.elementAt (i);
        return rrssets;
    }
    public CNAMERecord partial ()
    {
        if (type != SUCCESSFUL)
        return null;
        return (CNAMERecord) data;
    }
    public Vector backtrace ()
    {
        return backtrace;
    }
    public String toString ()
    {
        switch (type)
        {
            case UNKNOWN:
                return "unknown";
            case NEGATIVE:
                return "negative";
            case NOGAIN:
                return "NOGAIN";
            case NODATA:
                return "NODATA";
            case PARTIAL:
                return "partial: reached " + data;
            case SUCCESSFUL:
                return "successful";
            default:
                return null;
        }
    }
}

```

Figure 16.7: The merge of the two DNSjava files CacheResponse (in blue) and ZoneResponse (in red) into SetResponse. The file SetResponse is the base file and is shown in two columns to fit the page.

16.5.3 Additional unseen data: PyX

There are twenty-five uncertain disappearing files in the PyX releases analysed. The three files with only one target are listed in Table 16.24, all are correctly classified. Those with at least two target files are listed in Table 16.25. In both tables, ‘MC’ means the manual classification, ‘Doc.’ shows whether the change is documented in the log: Y/N, or Y? where the wording appears to support the classification, but not clearly. All of the

Ref.	Vn.	Disappearing files (one target)	MC	Doc.	Class'n
8	0.3	/manual/tex2.py	0	-	0
3	0.10	/examples/bargraphs/months.py	1	-	1
12	0.1	/pyx/datafile.py	1	Y	1

Table 16.24: Classification of PyX disappearing files with one target file. Class'n is the class assigned by the fb/SGD model.

Ref.	Vn.	Disappearing files (two plus targets)	MC	Doc.	Model			
					A	B	C	D
2	0.11	/faq/tipa.py	0	-				
5	0.8	/examples/bargraphs/addontop.py	0	-				
5	0.8	/examples/bargraphs/multisubaxis.py	0	-				
5	0.8	/pyx/mathtree.py	0	-				
6	0.7	/contrib/dvips.py	0	-				
6	0.7	/pyx/prolog.py	0	Y				
7	0.6	/newbox.py	0	-				
8	0.5	/manual/tex1.py	0	-				
2	0.11	/gallery/graphs/mandel.py	1	N				
3	0.10	/pyx/font/afm.py	1	Y?				
5	0.8	/examples/path/springs.py	1	N	0			0
6	0.7	/examples/misc/valign.py	1	N				
7	0.6	/examples/circles.py	1	N				
7	0.6	/examples/graphs/bar.py	1	N	0			
8	0.5	/pyx/data.py	1	Y	0	0	0	0
9	0.4	/pyx/attrlist.py	1	Y				
3	0.10	/pyx/dvifile.py	2	Y				
3	0.10	/pyx/font/encoding.py	2	Y?	1	1	1	1
3	0.10	/pyx/font/t1font.py	2	Y?		1	1	
3	0.10	/pyx/type1font.py	2	Y?			1	
8	0.5	/pyx/graph.py	2	Y				
8	0.5	/pyx/timeaxis.py	2	N	1	1	1	1

Table 16.25: Classification of PyX disappearing files with at least two target files.

unrelated files are correctly classified, but the renamed and split files are less successfully classified, an overall accuracy of 77-82%. Matched and unmatched files are listed in Appendix Q.

16.6 Selected models

In Table 16.26, the classifications of eight models are listed. The models are based on the two simplest feature sets with a mean classification accuracy of over 90% on the 89 project split file dataset: fc+tris-singles and all-tris+all-fc; and the top performing feature set, and the top performing singles feature set on the 89 project disappearing file (2 or more targets) dataset: fl+tris-singles and all-tris+all-fl. Each feature set is tested with the Simple Logistic and Rotation Forest algorithms. The table shows the

Feature set	Algo.	Disappearing files: (2 or more targets)			Split files			Grand Total
		Mean on tests	Unseen data	Total	Mean on tests	Unseen data	Total	
fc+tris-singles	ROT	165	123	288	243	272	515	803
all-tris+all-fc	SL	164	128	292	244	265	509	801
fl+tris-singles	ROT	167	127	294	242	264	506	800
all-tris+all-fc	ROT	167	124	291	241	266	507	798
all-tris+all-fl	ROT	169	125	294	242	262	504	798
all-tris+all-fl	SL	166	127	293	242	263	505	798
fc+tris-singles	SL	159	128	287	242	267	509	796
fl+tris-singles	SL	161	123	284	242	268	510	794
Out of		192	144	336	270	289	559	895
Max. %		88.0	88.9	87.5	90.4	94.1	92.1	89.7

Table 16.26: Summary of the classification by models based on fl+tris and fc+tris, singles and complete feature sets. Results are for the mean of the 89 project test sets and the unseen data: PostgreSQL, DNSjava, Struts and Pyx. Results are ranked by the total for split and disappearing files.

mean classification accuracy over the 100 test sets on the 89 project datasets and the classification of data from all four unseen projects, for each of the split and disappearing files (with 2 or more target files), and the models are ranked by the total. The simpler singles sets give the best performance when combined with the Rotation Forest algorithm. The fc+tris set, which combines trigrams and character-level analysis of their distribution, is best overall and on the split files, while the fl+tris set (trigrams and line-level analysis) is best for the disappearing files. Both of these models give the same results on the disappearing files with one target as the best (fb/SGD) model.

16.7 Selected features

The two models discussed above use the Rotation Forest algorithm. This is a ‘black-box’ algorithm, in that it is not easy to understand which features are important in classification. To give an idea of important features, those selected by the Simple Logistic algorithm are presented in Figure 16.8.

This figure shows the selected features in terms of the relationships between files and how their similarity measure is represented. There are five diagrams for each feature set: those for classifying split files, for disappear-

ing files with one target file, and one for each class for disappearing files with two or more targets (labelled Dis.0–2). Two features are not shown in the diagram: the number of target files in a group, and the type of file. Detailed lists of the selected features are provided in Appendix S. In Figure 16.8 the large numbers 1–3 represent the 3 files in a single file comparison:

1. the candidate file,
2. the amended file (or most similar target for disappearing files), and
3. the main target file (second similar target for disappearing files).

Examples from the top-left group (split files, *fc+tris-singles*) help to explain the layout. Many of the features relate the similarity between two files to one of the two. For example, 4 features are selected which relate the code shared by files 1 and 3 to file 1, denoted by the blue 4, and 3 to file 3, by the yellow 3. Features accounting for three files are shown on the edge of the circle opposite the file to which they are proportional. For example, the red 1 on the outer circle means that there is 1 feature which relates some aspect of the 3 files to the size of file 2. Quantitative features (in boxes) include code shared by files 1 and 3, shown by the green 1; a measure of the code in file 3, by the yellow 1. Further explanations are provided in the key.

In general, more features are selected for the *fc+tris* set than for the *fl+tris* set, but the patterns are broadly similar. This does not mean that the features selected are the same. For example, for both sets the disappearing and renamed files (Dis.1) have two features relating the similarity between files 1 and 3 to file 1. One of these features is the same in both sets (the lines/characters in blocks of at least 32 lines/640 characters), and one in each set is different: **lines** in blocks of at least one-sixteenth of file size, and **characters** in the largest block to file size.

As might be expected, the relationship between the two target files is less important for disappearing files than the relationship between the amended and target files for classifying candidate split files. At least one feature which takes account of the interaction between all three files is selected for split files and disappearing files in classes 0 and 1. It is less clear why features which involve all three files are not selected for the disappearing split files.

Features which are the same for both sets are highlighted in Table S.2 in Appendix S. For each class of file, approximately even numbers of trigram-based and block-based features are selected. The containment of the target file (or second target file) in the candidate file is the feature selected more than any other.

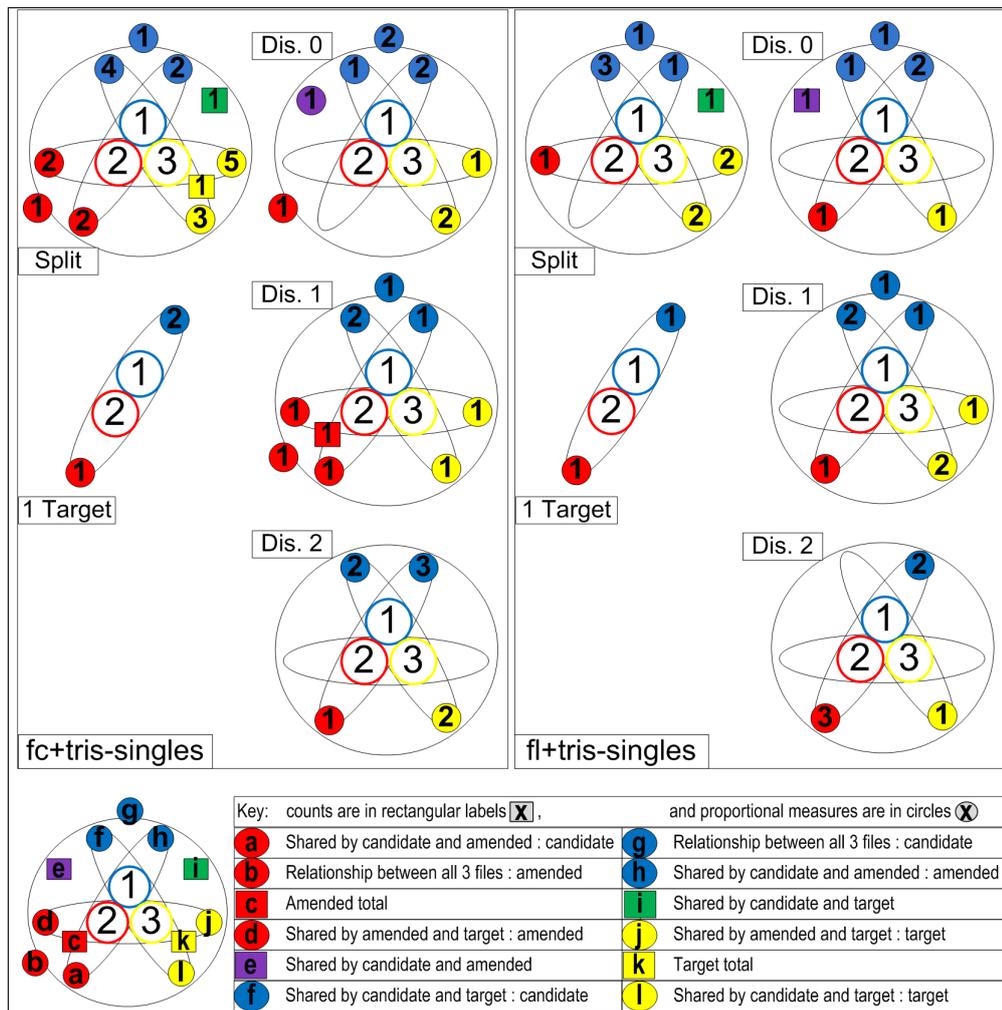


Figure 16.8: File relationships represented by features selected from the sets fc+tris-singles and fl+tris-singles, by the Simple Logistic algorithm. The 5 diagrams for each set are: split files, disappearing files with 1 target, and the 3 classes for those with 2 or more targets (Dis.0–2). Relationships between files are shown with the number of features of this type noted in the circles. The key to the relationships describes files as “amended and target”, for disappearing files, these are the 1st and 2nd targets.

16.8 Summary

16.8.1 Split files

The effect of the revised filter criteria on the split file dataset has been explored in this chapter; with the revised criteria, twice the number of candidate files are selected, and include all except for two of the previous positive examples. Although there are more ‘marginal’ candidates and fewer definite negative examples, the refiltered data is classified with over 90% accuracy.

The model which best classifies the 89 project dataset, “fc+ft+tris+pdp-singles”/Rotation Forest, classifies candidates from the unseen projects PostgreSQL and DNSjava with 91.5% accuracy, and these candidates are classified with 94.4% accuracy by the “fc+tris”/SimpleLogistic model. This same model correctly classifies all of the candidates found in the Struts project (release 1.1) and 93% of those in the PyX project.

The effect of the revised filtering criteria on the candidate split files selected from the two main unseen projects differs between the projects. In both, filtering is able to screen out at least half of the previously selected negative examples. The PostgreSQL system includes sets of files which have the same function for different operating systems, meaning that target files are found in many cases where the files are unrelated except by function, and this accounts for the large increase in negative candidates. In contrast, DNSjava, with no parallel subsystems, has fewer new negative examples.

16.8.2 Disappearing files

The uncertain set of disappearing files is partitioned into two: files with one target, which are classified with 95% accuracy, and those with two or more target files, with 88% accuracy.

The ‘best’ model for classifying the disappearing files with one target from the 89 project dataset is that based on Ferret basic features using the SGD algorithm. This model, along with many others, is able to correctly classify all of the examples from the unseen data except for one, which is

also difficult to classify manually.

For the files with two or more targets, the ‘best’ model based on accuracy is “all-tris+all-fl”/Rotation Forest (88.13% accuracy, 81.97% geometric mean), with “fl+tris-singles”/Rotation Forest having the best geometric mean (84.3%) of those tested. Strategies for improving classification in imbalanced data did not improve classification in the tests.

Results are reported for four models: “all-tris+all-fl”/Simple Logistic (A), “all-tris+all-fc”/Simple Logistic (B), “fc+tris-singles”/Simple Logistic (C) and “fl+tris-singles”/Rotation Forest (D), are based on the original, rather than manipulated, data. The models correctly classify 90.2%, 90.2%, 91.0% and 89.3% of the examples respectively, with 78.6%, 57.1%, 64.3% and 71.4% of the minority class, split files, correctly classified.

16.8.3 Comparison to other approaches

Antoniol et al. find two split classes in releases 1–40 of DNSjava. My system finds eleven split files in these releases, correctly classified except for two marginal examples. Antoniol et al. did not find the additional split files for reasons already discussed. They find two classes which are renamed, but which do not move files, and so not identified by my approach. Antoniol et al. find one merged file, also found by my approach, as are the four renamed files, and although two of these are not reliably classified, their target files are correctly identified. Antoniol et al. also had problems with one of these files, only finding its correct classification by inspection of the code. My approach finds one file not found by Antoniol which is both split and renamed. It also finds ten renamed files, all of which are correctly classified by the majority of models, and two which are possibly renamed, for which classification is mixed.

Zou’s work is at function level, and although file level changes are reported, it is unlikely that every change to functions which results in file restructuring is mentioned, given the additional restructured files found in the PostgreSQL project by my system. As already discussed in Section 16.3.2.1, comparison with the available results shows vulnerabilities in

my approach: first, that a target file will not be selected if other changes to the file mask the relevant changes; and second, that text analysis is not always sufficient to decide between a number of textually similar possibilities.

All of the changes to the Struts project found by Dig et al. are found and correctly classified by the system, as are two others supported by the findings of Wu et al. The PyX project is manually classified with help from the change log, and therefore cannot be compared with other results.

16.8.4 Overview

The revised filter criteria exclude at least half of the negative examples selected by the original filters, and select approximately twice as many positive examples overall, although the changes vary between projects.

The system is trained with C code, the test data comprises four projects: one C, two Java, and one Python. As shown in Table 16.27, every category in each of the projects is classified with at least 77% accuracy, with the mean across all restructured files selected from the unseen projects of over 91%.

The two models, *fc+tris-singles/Rotation Forest* or *fl+tris-singles/Rotation Forest*, are among the top three models overall, and have the benefit of simply constructed feature sets, as all the features are derived from analysis of Ferret's outputs and do not need the concatenated files.

Project	Language	Split files		Disappearing files			
		n.	acc.	1 target		2+ targets	
		n.	acc.	n.	acc.	n.	acc.
89 projects	C	810	90%	177	95%	575	88%
PostgreSQL	C	200	95%	19	100%	100	90-91%
DNSjava	Java	48	91.7%	5	80%	23	87-91%
Struts	Java	14	100%	-	-	-	-
PyX	Python	27	93%	3	100%	22	77-82%

Table 16.27: Summary of the classification results for PostgreSQL, DNSjava, Struts and Pyx by the best model tested in each case. The number of instances and accuracy are shown for each group.

Chapter 17

Discussion and evaluation

The main part of this project explored the use of machine learning techniques in the field of software evolution, using features generated from comparisons between source code texts. There are several challenges in creating a new dataset for machine learning, among which are developing a suitable filtering technique, selecting a means of generating information about the data, creating features from this information, and manual classification of the instances. Features were created by comparing both pairs and groups of files, and looking at the size, distribution and interaction of blocks of matched code in the files. Techniques used in creating these features were also applied to collusion detection, and to visualising textual relationships between files. This chapter is in four sections: in the first section origin analysis is discussed and evaluated, the three following sections cover collusion detection, the visualisations, and other tools.

17.1 Origin analysis

17.1.1 Related work

The approach to origin analysis in this research has its roots in that taken by Rainer et al. [194] in the use of trigram analysis to filter the files in a system to find unusually similar pairs. However, the ideas are developed in many ways: in the different filtering techniques; in the novel set of features;

in exploring the use of complementary approaches to file comparison; in classifying potential matches using machine learning; and in providing the 3CO file comparison tool for the visual inspection of groups of files.

There is also similarity to Antoniol et al.'s approach [6] where text-based measures are used to analyse a system at class level. Antoniol et al.'s approach is one of two (with Zou [261]) of those surveyed which consider the relationship between more than two entities at a time. Antoniol et al. take account of the interactions between vectors from three classes when trying to find split or merged files, and from four classes for recombined files. Their view is that splits, merges or recombinations between more than the minimum number of classes are unlikely. However, multi-way splits occur in many of the 93 projects studied in this research, showing the importance of considering this possibility when working at file level. For example, one fifth of the splits found in the projects 'Lifelines' and 'Hatari' are multi-way, as are nearly one half of those in the project 'Etherape'.

Antoniol et al. report that their approach is vulnerable to renaming. The example in Chapter 14 (p. 243) shows this vulnerability, and another problem, that files must be split fairly "cleanly" to qualify as candidates under the cosine similarity measure. In my work, the first of these problems is reduced by the use of trigram analysis, which is moderately tolerant to renaming; and the second by training the models using a range of features, and examples from projects of varying development styles.

DeMeyer et al. [60] also work partly at class level, using heuristics based on the changes to the metrics of each class to suggest possible refactorings. However, the user must browse the code to find out whether refactoring has taken place, and to discover the other class(es) involved. DeMeyer et al. found a high proportion ($\approx 96\%$) of false positives in their work. This is partly because they consider the changes to only one file, making it impractical to filter out files with no obvious targets.

In common with others [237, 250, 252, 261], S.Kim et al. [130] combine a small set of features from different sources. Unusually, they include measures of text similarity from three third-party tools. These measures are

based on strict matching (*diff*), parameterised matching (*CCFinder*) and *n*-gram matching (*Moss*), a similar combination of tools to that used in this research, respectively the *P-Duplo*, *CCFinder* and *Simian*, and *Ferret* tools. S.Kim et al.'s features are weighted according to each one's accuracy in predicting matches in a set of method pairs determined by their expert panel. Their approach is the closest to machine learning among the surveyed work, and they suggest using machine learning methods in future searches for suitable feature combinations and threshold values.

In the same way as many others (e.g. [22, 90, 128, 130, 179, 250]), the approach in this research first matches by name, forming two groups of files: those matched or unmatched by files of the same name in the following release. Matching entities are not normally considered for change, whereas in this research, changes to matched files are taken into account in looking for split files. Once candidates are established, potential target files are selected based on their similarity to the candidate file, whether the files are new or already exist.

Many of the approaches surveyed match names or method headers textually (e.g. [128, 179, 194]). Only Dig et al. [64], compare the text of the method bodies as an initial filter, whilst others (e.g. [22, 130, 245]) use the text similarity of entity bodies as a second filter. In my work, both the first coarse-grained, and the second fine-grained filters use textual similarity based on trigram analysis of the full text of the files.

Several of the approaches use *n*-grams to match elements in the code [22, 64, 77, 130, 252]. Biegel et al.'s [22] finding, that ranking targets by similarity measured using token bigrams is not only faster, but marginally outperforms both *CCFinder* and an AST-based similarity tool, supports the use of *n*-gram based similarity measures in this task.

Tools which parameterise, such as *CCFinder*, *Simian* and *Moss*, do not seem to be the best way to match code for origin analysis. The reasons relating to file comparison are pointed out in Section 14.1.2. However, both S.Kim et al. and Biegel et al. also found *CCFinder* to be less useful than other features in comparing methods in Java code. While the header

information which is ignored by CCFinder will not be an issue for functions or methods, the relaxed matching allowed by parameterising may be the cause. Although parameterising helps to match edited code, it can also allow matches between standard forms, such as accessors, which are present in many files.

In general, method or function level approaches, especially those which consider split or merged entities, should find most of the same restructurings as found by my approach. They will also find within-file changes, such as method renaming, but do not find changes to the code which falls outside the methods/functions, while my approach will find movement of code from any part of many types of file.

There are four main differences between my approach and those detailed in the review of origin analysis in Chapter 3. First, machine learning is used to classify the candidate files, based on a training set derived from a large number of open source projects. Second, the novel features derived from the similarity detection tools used to compare the source code. Third, comparisons between entities are normally made pairwise; whereas here, group comparisons are used to create some of the more predictive features, and are also used in ranking target files. Lastly, the models built using source code in one language are successfully tested on systems written in other languages.

The task of determining whether code taken from one file should be matched to that in one or more other files is very difficult, given the wide variation in file sizes, the amount of code moved, the extent to which it is edited, the way the code is distributed in both the source and target files, and the inherent similarity within a project. Rather than using a pre-defined set of rules to work out whether files should be matched, this system uses machine learning to discover, among a large number of features, patterns which apply to a collection of projects, with the aim of creating a model which will be generally applicable.

Working at file level with text-based analysis allows the approach to be used on most other programming languages and on text. The approach is

not restricted to files containing methods, functions or classes. Although the models built for this study are based on C code, they work well on the Java and Python projects tested.

The experiments show that Ferret-based features provide sufficient information for good classification. Ferret is used in primary and secondary filtering and in target file ranking. The supporting tool 3CO is also based on Ferret. This means that apart from a machine learning tool such as Weka, Ferret is the only third-party tool needed to classify the restructured files.

In discussing matching techniques for multi-version analysis, M.Kim et al. suggest hybrid matching as a good way forward. However, they also caution that *“combining results from multiple matchers will require tremendous efforts because (1) not every matching tool is available for public use or applicable to popular programming languages and (2) different matchers use different program representations”* [127, p.5]. By applying different techniques to the various outputs of the trigram analysis tool Ferret, some degree of pseudo-hybrid matching, applicable to a wide range of languages, is gained without the need for other matching tools, thus partly addressing Kim’s concerns.

17.1.2 Machine Learning System

Two of the main challenges in creating the machine learning datasets for this application are how best to filter the very large amount of data to identify the items of interest, and what features might best characterise the data to allow discrimination between the classes.

There is a natural trade-off between precision and recall in filtering. The aim here was for good recall, while keeping a reasonable balance between the classes for machine learning.

Feature construction presents many challenges: which tools to use to compare files, and what features to create from among the almost limitless number of potential features available from analysing the output of these tools. Without prior knowledge of suitable features these decisions are difficult. To explore the possibilities, a large pool of features was created using a set of complementary comparison algorithms, and a wide range of

measures. Features were grouped into smaller sets based on their source, and compared so that a suitable subset could be chosen.

In the rest of this section, each step in the origin analysis system is discussed. These steps are: data collection, preprocessing, information gathering, filtering, manual classification, feature construction, feature selection, and model selection.

Data collection The projects chosen for training the models were all written in C. As the features used in this system characterise the distribution of matched text in the evolving files, theoretically the projects could have been written in any one, or a mixture, of most programming languages. Given that the models built with C code perform well on projects in two other languages, it is possible that either strategy would have worked as well.

Preprocessing Relevant source code files (e.g. '.c' and '.h') were selected from the stored versions and prepared by removing comments. Comment removal is the only part of the system where language specific processing is necessary. To assist with Duplo and Simian's line-by-line matching, pretty-printing was originally used, to standardise the code layout. As Duplo and Simian features are not used in the final models, pretty-printing is not necessary.

Information gathering The next step is to compare the files in each release, both within the release, and with files in adjacent releases, and to store the results. The information was stored in hash tables, written to file for reuse. Racket's hash tables are limited in size, therefore using a database for storage may be a better strategy. Once stored, the results of comparing the files are used to filter for candidate restructured files.

Filtering An initial filter for finding potential split files in this system is a reduction in file size. This reduction was found not to be a feature of all split files, as other within-release changes may result in a larger file. One

solution to this problem is to consider the number and distribution of the trigrams in each file's difference set, however, this remains as future work.

Target file selection is also an important part in the process. The investigation reported in Chapter 15 explored criteria and thresholds for target file selection in the test subset of the data. One of the aims of this research was to remove predetermined thresholds. However, as the simple filter criteria first applied were inadequate, new criteria were found by experimenting with a subset of the data, and in balancing precision with recall, the criteria turned out to be fairly complex. Simplifying these conditions is possibly an area for further work. Although the thresholds which suit the test set fit most of the remaining data, there are cases which fall outside the parameters. The main problem is that target files are not selected when the number of shared trigrams does not increase sufficiently between releases, because of other changes. As with the candidate files, the changes to the target file may offer a solution, the interesting trigrams in this case are the new trigrams in the file, in other words, the reverse difference set (see p.214).

Target file order is important for matching files in general [22], and for feature construction in this system. Initially, the target files were sorted on their change in similarity to the candidate file. The experiments reported in Chapter 15 showed that this strategy was not as successful as sorting by unique trigram count for the target files in the test set of split file candidate groups. The order is mainly judged using 3CO, which is also based on trigram distribution, and so may be subject to bias. However, files previously misclassified because of incorrect target file ordering were correctly classified after reordering their targets by unique trigram count.

Manual classification Manual classification of the candidate files was very time-consuming prior to the development of 3CO. The tool usually provides sufficient information for classification. Otherwise, the number of files which need to be inspected is normally reduced by its use.

Given that manual labelling is subjective, independently labelled training data would have been of benefit, but availability was limited. The

unseen data is partly labelled by other research groups, but the same subjectivity is present in labelling the examples not reported by these groups.

Several files were excluded because they could not be classified, a particular problem for projects with parallel subsystems, like the example in Figure 16.5. This file highlights a weakness in using text analysis alone, and is a case where call analysis [64, 250, 261] should provide a solution.

Feature construction Advice on feature construction varies. For example, Guyon and Elisseeff [98] suggest creating a large number of features to include as much information as possible, while Hall [102] recommends that there should be at least twice the number of instances as features. The approach taken in this research considers both of these requirements: by constructing a large set of features to give variety, but comparing classification by reasonably sized subsets of these features. The final models were built with sets of 200 features, well under half the number of instances in the datasets.

At the start of this project, Ferret, Duplo, Simian and CCFinder were selected as complementary tools for comparing files. As the work progressed, P-Duplo was developed using the ideas behind Duplo, to better fit the requirements of the task. The idea that the Ferret XML report could be analysed to provide additional information was also a later development. To some extent, the XML analyses have similarities to the other tools.¹ The line-based analysis has much in common with P-Duplo, and density analysis finds the gapped copies which are the main reason for using both Simian and CCFinder. With hindsight, it would have been sufficient to explore the range of features available from analysing the outputs from Ferret.

Features are based on comparison between the three main files or the candidate file and various combinations of the other files in the group. Comparisons of concatenated files were used as a solution to the problem of representing multiple targets, and as a way to measure how much of a candidate file is covered by combinations of its targets. There is a compu-

¹Comparison between the predictive accuracy of these sets can be found in Appendix T

tational cost in generating these features, as the files must be concatenated, and additional groups of features generated.

Although the difference between the top performing feature sets for each task is not significant, the “top 20” feature sets for split files are mostly based only on features generated from comparisons between single files (“singles” features), while only one of the 20 for disappearing files with two or more targets has only singles features. It is possible that ordering targets by unique trigrams is less suitable for disappearing files than for split files, so that files other than the first two of the ordered targets must be considered. However, as there are singles sets which perform well on the data, the ordering problem may only affect a few cases. Investigating the order of disappearing file targets remains as further work.

Feature selection Standard solutions to finding a good subset of features from a large set include the use of a feature selection algorithm or a machine learning algorithm which incorporates feature selection. However, in this research, one important factor in choosing a feature set was the ease of extracting the features from new data. The features are derived from comparisons made by four tools, and there is further complexity in constructing the concatenated files. The ideal is a small feature set, taken from simple comparisons between files, based on the output of one tool. The combination of trigram and block based features (“fc+tris-singles” or “fl+tris-singles”) derived from Ferret fits these requirements. As noted in Chapter 16, both of these feature sets give good results on split and on disappearing files, although the former is better with split files and the latter with disappearing files. The trigram based features take account of interactions between the three ‘main’ files with no sense of the arrangement or frequency of the trigrams, while the measures of the sizes of the matched blocks, either in characters (fc) or in lines (fl), give a fairly precise analysis of the arrangement of matched parts of a file and their significance.

In the simpler tasks of classifying the original split file dataset and the disappearing files with one target, one feature set is sufficient for classifica-

tion. However, for the more challenging tasks of classifying the refiltered split file set and the disappearing files with more than one target, classification is improved by using complementary feature sets.

Model selection It is not possible to find out which classifier will be the best for *all* new data, but by training classifiers with one portion of the available data and testing with another portion, an estimate of the predictive accuracy of each classifier can be found [27, p.79–80].

Here the training data was split 100 times, with two-thirds of the data used for training and one-third for testing. The “best” models were selected on their mean classification accuracy over the 100 test sets. Results demonstrated that these models were not necessarily the best on the unseen data. Tests on unseen data were used to decide on the “best” models overall.

Various strategies for improving classification were tested during this research. These strategies included exploring algorithm parameters, combining algorithms with heterogeneous meta-classifiers, and for imbalanced data, under- and over-sampling and use of cost-sensitive classifiers. In general, these strategies did not improve classification accuracy on this data.

Although a subset of the possible combinations of feature sets were investigated, given the large number of sets, a full exploration was not undertaken, and may be an area for further work.

Results Classification of the candidate split files reported in Chapter 14 was around 94% for the 89 project dataset and, using models built on this data, 99% on the unseen data. However, comparison with the results from other research groups on this data highlighted weaknesses in the filtering process: not all candidate files were selected, not all target files were selected, and target files were ranked incorrectly.

Extensive experiments in filtering, reported in Chapter 15, found strategies for correcting many of the identified problems. The new set of candidate split file groups selected by refiltering from the 89 projects are classified with a mean accuracy of 90%. Candidates from the four unseen projects are classified with a mean accuracy of over 94%.

Files which disappear from the system are initially matched to their targets by similarity score. Those with identical or nearly identical files in the next release are classified as renamed, and those with no similar file as deleted. The remaining files, the “uncertain set” fall into two groups, those with one target file and those with more. The former group is classified with a mean accuracy of 95%, and 96% on the unseen data. The latter group with mean accuracy of 88% and around 89% on the unseen data.

These percentages are not directly comparable with those of other research groups. The two approaches which compare files [6] or classes [60] directly do not provide estimates of their accuracy. Unless the system analysed is fully documented, or there are other means of tracing all of the changes, it is not possible to know what percentage of changes have been found. Approaches which provide percentages use varied bases. For example, S.Kim et al. [130] give the percentage of matches between disappearing and new methods made by their system to those in their oracle sets as 87.8–91.1%. However, the oracle sets contain 85-91% of the examples judged, and it can be assumed that the pairs left out are the more challenging ones to classify, as the panel does not agree on these pairs. As another example, Dig et al. [64] report 100% precision and 86% recall on the Struts project, using entries in the change log to give the expected results. The results of Wu et al. [250] show that not all of the changes are logged.

The figures presented in my work are also based on a subset of the candidates. On the one hand, examples whose classification is unclear are omitted. This means that 14% of the split file candidates from the 89 projects

	Split			Disappearing			
	Cand- idates	Not included	% not included	Directly classified	Uncertain	Not included	% not included
89 projects	944	134	14.2	1965	925	173	6.0
PostgreSQL	207	7	3.4	81	125	6	2.9
DNSjava	49	1	2.0	65	28	1	1.1
Struts	14	0	0.0	–	–	–	–
PyX	27	0	0.0	44	25	0	0.0

Table 17.1: Proportion of total candidate split and disappearing files not classified.

are excluded, although more than 97% of the unseen data is retained. Other figures are shown in Table 17.1. On the other hand, the results given for the disappearing files include only the uncertain set. The remaining examples are directly classified as matched or unmatched. Although nearly 19% of the uncertain disappearing files from the 89 project dataset cannot be classified with confidence, this represents only 6% of all disappearing files. The directly matched files also affect the overall classification rate of the disappearing files. For example, in the Pyx project there are 34 renamed files and 8 unmatched files. While it is almost certain that all of the renamed files are correctly matched, the unmatched files may not be matched because their targets have been missed. Assuming all of these files to be correctly classified, the percentage of correctly matched files would rise to 93%.

Analysing new projects with the models Table 17.2 gives the steps required to classify a new project with the models. Most of the steps are a subset of those used in building the models. The difference is that should filtering a new project result in few candidate files, then the simplest way to find out what has happened to the code is to view the 3CO files for the candidate groups. With more files, automatic analysis is required and features must be constructed so that the files can be classified using the models. Each step runs automatically, except when manual downloading is necessary at step 1, and at step 7a, where inspection is required.

1	Store the data in consecutively numbered directories.
2	Select file types (e.g. not makefiles, images, binaries)
3	Prepare as necessary, i.e. remove comments
4	Compare the files using Ferret and store the information
5	Filter to find the type of files required (e.g. split, disappearing) and their targets.
6	Process file groups to produce 3CO XML files.
7a	*If not many candidates, then 3CO files should provide sufficient information quickly.
7b	If more candidates, use automatic analysis, extract the features to create a machine learning file.
8	Run through the appropriate model to classify.

Table 17.2: Steps in classifying a new project with the models.

17.2 Collusion detection

Twenty-nine approaches to source-code plagiarism were analysed in Chapter 2. Although many people involved in marking assignments recognise that unusual similarity between students' work prompts suspicion of collusion between students [52], the survey found that few approaches take account of unusual similarity and none measure it directly.

Another observation from the survey is that twenty-one of the approaches initially tokenise the code. Tokenising requires a suitable lexer, but means that the method is otherwise language-independent. A lexer for one language can be used to tokenise other languages in a reasonable manner, as is shown by use of a C-type lexer for Visual Basic and ASP in the application reported in Chapter 9. However, eighteen of these approaches parameterise the tokenised code, meaning that a language-specific lexer is required to identify keywords.

Tools which change the code to graph, tree or metric representations are tied to one language. Those which convert the code into an intermediate language aim for language independence, but are restricted to the languages supported. Among the approaches listed in Table 2.3, few will be able to analyse almost any mix of languages. Those which can are Ferret [146] and PlaGate [52].

Proportional measures will sometimes rank small amounts of similarity between two small files above significant similarity between two larger files. This was found to be a feature of measures based on Ferret where similarity is computed using the presence of trigrams in a file, regardless of their frequency. Where there is a large amount of repetitive code, measures which match sections of code in a file each time they occur, such as [99, 115], may exaggerate this problem. It is important to be aware of what collusion detection tools are measuring and therefore what is being reported.

Almost all of the similarity measures used by the approaches surveyed are proportional to the size of one or both of the files compared. Although proportional measures are likely to perform well with files of similar sizes, there can be a problem when file sizes differ. Count-based measures find

similarity which may be missed by proportional measures, especially where files are large. However, care must be taken in what is counted, otherwise pairs of large files will have more in common than pairs of small files, because of the inherent similarity in program code. By computing similarity based on less common trigrams, the focus is only on unusual similarity.

A number of methods used to disguise plagiarism are reported by Jones [117, p.2]. These are changes to comments, white-space, identifiers and data-types; reordering code within statements, moving blocks of code, adding redundant statements, and exchanging one type of control structure with another. Many of the approaches surveyed are naturally concerned with detecting similarity in the face of these disguises. Strategies such as parameterising and attribute counting lead to a loss of textual information. However, there is a place for approaches which preserve more information, such as those of Liu [151], Cosma [52] and Ferret [146].

The approach to collusion detection presented in Chapters 7 and 9 is language independent, is based on recognising unusual similarity between pairs of assignments, preserves much of the textual information, and does not rely on proportional measures. Despite an increase in web technology courses,² where several programming languages may be used in building a project, there appears to be no other reported work in source code collusion detection where files in more than one language are analysed together.

Although the method is not robust to systematic identifier renaming, it will find unusual shared code which has not been well disguised. This type of similarity is likely to be found naturally by someone marking a small group of moderately sized assignments. However, as the size and number of assignments increases, and particularly when marking is distributed among several tutors, unusual similarity is unlikely to be found without automatic analysis. A full evaluation of this approach would need further tests on groups of different sizes and with different styles of assignment.

²158 courses with a web technology content are listed on the UCAS site for 2013, (36 specifically for web technology, and 9 for multimedia web production)

17.3 Visualisations

17.3.1 Collusion detection

In Chapter 5, the outputs of a selection of collusion detection tool file comparisons were shown. A large number of such tools were examined during the course of this research and only that of Ribler and Abrams [199] was found to show details of the similarity between two files in the context of the group. However, a display of this type is useful in three ways, first in highlighting the more unusual elements shared by two students, second in highlighting the elements unique to one student, and last in showing which parts of the code are of less interest. The adaptation of the Ferret display, based on the analysis of two files in the context of the group, provides all of this information, and also allows the user to see the relevant text.

The visualisation adds group context to the comparison between two files. The more interesting sections of the file are highlighted, which helps in assessing the nature of the similarity between the files. The tool has not yet been applied to other groups of submissions, or by other users, therefore it is difficult to evaluate its usefulness.

17.3.2 File comparison with 3CO

There does not appear to be a tool available which can display comparisons between more than three files at once. This makes comparing files in origin analysis very difficult. As shown throughout this dissertation, the relatively compact display of 3CO has proved invaluable in understanding interactions between a group of files, and in speeding up manual classification of the large number of file groups in this research. It also has a place in other file comparison tasks, such as collusion detection, as shown in Chapter 9.

Without this tool, much of the work in this research would not have been possible. For example, it would have been unthinkable to undertake the very large number of file comparisons in the filtering experiments.

The XMLs are produced in reasonable time, the average over a set of 400 files with a mean of 8 target files each, is 1 second per file, under Ubuntu

on a machine with 2.9GB memory and a 2.4 GHz processor. The analysis is not yet integrated into Ferret but works externally, analysing the trigram-file index, the trigrams, and related white-space, then mapping back to the original code. It should be considerably faster if integrated into Ferret.

There are a couple of negative points about the display. First, because the colour of each token is based on the files each of its three trigrams appear in, heavily edited sections of code may not be coloured as expected, and if there are high levels of incidental similarity the code can appear very fragmented. These conditions occur infrequently, and alternative techniques could be adopted when they do. For example, for gapped copying, a “darker” colour scheme where tokens for which only one of the three trigrams belongs to a file are coloured to match that file, giving a similar result to a standard Ferret comparison between two files for the gapped copy region.

Second, from my limited observation of people’s understanding of the tool, the majority easily grasp how it works. However, for some it takes a little more work to get a “feel” for it. A more subtle colour scheme using intermediate hues [224] would provide more information, but would need tuning to an individual monitor, and the finer differences in colour may be difficult to perceive, and less easily understood. As already noted, the colour scheme can be adjusted to cater for colour blindness. For example, using underlining or shaded backgrounds in place of one or more colours.

Set against the benefits of the tool is the fact that by focusing on the base file, information about the target files, which is available in pairwise comparisons, is lost. One way to overcome this drawback, and to make 3CO suitable for more general use, would be to provide the option to select alternate views. For example, to show a pairwise comparison between the base file and any of the target files, or to select a new base file from among the group displayed. Also, in origin analysis, the ability to select a file to view its evolution would be helpful, in other words, a new display with the file compared with itself in the previous, next or both versions.

17.4 Other tools

17.4.1 Density analysis

The density analysis technique used to measure “gapped” blocks and to create features performs as expected. However, as a top down approach it can be slow when the files being analysed are large and have high levels of incidental similarity (i.e. there are many segments), and therefore may benefit from a different approach. Bottom-up approaches were considered, but not fully explored [94, p.8]. The density tool is an early prototype based on the density analysis. The following additions would make the tool more generally useful:

- profiles of each file with block positions highlighted and selectable,
- links from blocks in one file to the closest match(es) in the other file, and
- a more functional front end, so that the user has only to choose the files and the parameters, rather than, as in the present version, providing the result of a previous Ferret comparison in the form of an XML file.

17.4.2 One-to-one matching of clone output

It is difficult to assess how the Unscrambling tool performs because the only comparisons between tools are the relative merits of the feature sets, for which there are too many other variables. On test data it performs as expected. Although useful for files where there may be a large number of many-to-many clones, this analysis is perhaps less useful in smaller entities, such as methods, because there are unlikely to be enough clones in smaller entities for it to be worthwhile. There are limitations in the approach taken to matching multiple copies, where a greedy first-come first-matched approach is adopted. This could be altered, perhaps by considering the surrounding code.

17.5 Summary

This chapter has discussed the similarities and differences between this research and other research in software origin analysis and collusion detection. Also discussed are the difficulties encountered, their solutions or suggested future solutions, and the limitations of the approaches.

There were a number of difficulties to overcome in this research. One of the major problems is matching code in the face of high levels of inherent similarity. For collusion detection this was tackled by looking instead at the unusual similarities between files, an approach also used in origin analysis, along with a range of other features looking at various aspects of the relationships between pairs and groups of files.

Another difficulty was the need to label the training data and some of the test data for machine learning. Manual classification was extremely time-consuming prior to the development of the 3CO tool. However, this tool enabled a more in-depth study than would otherwise have been possible.

Filtering in origin analysis is also difficult, because of the range of interactions between a candidate file and its potential targets. Experiments to find suitable criteria for filtering were mostly successful, but two areas remain unresolved. Split files which have become larger are not selected, and neither are target files which have become less similar to the candidate file than in the previous release. As discussed, both of these problems may be overcome by taking account of the difference sets.

As a purely textual approach, the collusion measure is limited in that it will not detect well-disguised copying, but does mean that multi-language projects can be processed. The measure is also useful when file sizes vary, there is auto-generated code, or there is group collusion. In very large files, the density tool could also be used to identify the blocks of duplicate code.

Potential improvements to the visualisations are discussed in this chapter, the main improvement to both of these, and to the density tool, is to incorporate them with Ferret.

The next chapter considers further application of the techniques and tools developed, and the contributions made by this research.

Chapter 18

Conclusions and further work

In this chapter, the contributions to knowledge made in this dissertation are reviewed and their application to other problems discussed.

18.1 Contributions to knowledge

The contributions are:

- Application of machine learning and text analysis techniques to the field of software origin analysis, developing models from one set of projects which have been successfully applied to four other projects, three of which help to demonstrate the language independence of the approach.
- Techniques for measuring relationships between files in a group, based on trigram analysis of the source code text, and the development of a novel set of features based on this analysis.
- A method for visualising the similarity of one file to a set of others by colour coding the file text based on trigram analysis.
- A survey of approaches to source code plagiarism detection.
- A novel similarity measure for use in collusion detection.

- A method for colouring text to provide information about pairwise comparisons between files in a group, giving contextual detail not found elsewhere in the field of source code collusion detection.
- Techniques for providing information about the distribution in one file of the trigrams shared with another file, both as sequences and, by developing a flexible density analysis algorithm, as “gapped” blocks.
- A new method for ranking potential matches between entities in origin analysis, by finding “uniquely shared” trigrams.
- A technique for converting the many-to-many results of clone detection tools to match the clones on a one-to-one basis.

18.2 Further work

This section discusses ways in which the techniques developed in this dissertation might be of benefit in other applications. Three areas are considered: origin analysis, n-gram analysis, and visualising file comparisons.

18.2.1 Origin analysis

Methods used in creating the machine learning models for origin analysis could be applied to other projects, either as a whole or individually. The models built to classify both split and disappearing files should apply to other projects, certainly those written in C, and based on the test data, also to those in Java and Python. In principle there is no reason why the models should not work with many other programming languages, or with text, however this would need to be tested.

The method of ranking target files by the number of unique trigrams they share with the candidate file could be applied to other entities, and therefore used in other origin analysis research, and potentially in matching other documents. A version of this method is applied in the weighted trigram count used in the collusion detection application described in Chapter 9.

Individually, other techniques which have been used in this approach and which could be applied to work both inside and outside the field of software origin analysis are: building features based on group comparisons, using complementary measurements to improve generalisation, and analysing similarity detection tool output to create a broad set of features.

18.2.2 N-gram analysis

The techniques described in this dissertation for analysing the trigram-file index and XML output from Ferret can be applied to any n-gram analysis tool which creates an inverted index, and maps the analysis back to the source. However, the future application of these techniques are discussed in terms of additions to the Ferret tool.

Ferret is a well-established copy detection tool. In practical terms, it is simple to use, is very fast, can be used for both text and code, and is a standalone tool, which is important where confidentiality is required. When presented with a set of documents, it provides a similarity measure for each pair. The user can select any of these file pairs to view their text, in which the shared trigrams are highlighted. The techniques developed for analysing other outputs available from Ferret are broadly:

- Analysis of the trigram-file index to
 - find groups of related files,
 - find the relative “importance” of the shared trigrams, based on the number of files which contain them, and
 - produce measures based on less frequently occurring trigrams.
- Analysis of the XML output to
 - find out about the distribution of the shared trigrams in the texts.

The results of these analyses could be used to add features to Ferret to provide a wider range of information for the user. For example:

1. Integrating other code colourings into the present text comparisons,

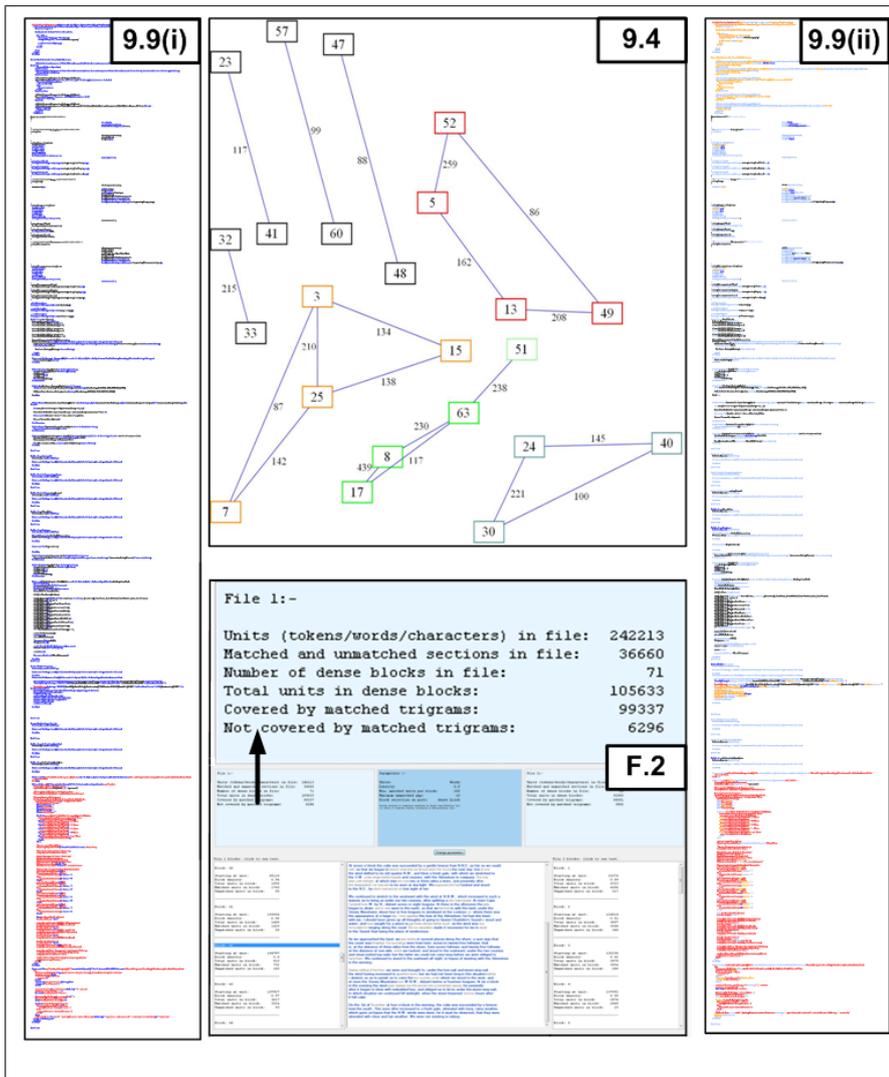


Figure 18.1: Figures 9.4, 9.9: (i) uniquely shared code in red and (ii) code shared by groups in shades of orange to red, and F.2 repeated for reference

such as those shown in the middle and on the right of Figure 9.8, and repeated to either side of Figure 18.1.

2. Adding alternative similarity measures to the table of pairwise similarity scores, with the option to sort on these measures.
3. Providing statistical information about blocks found by density analysis in the pairwise reports, such as that at the top left of Figure F.2

(also at the bottom of Figure 18.1); and giving the full results of the analysis as a click-through screen, based on that shown in Figure F.2, with the amendments recommended in Section 17.4.1.

4. Producing graphical displays of group interactions based on the chosen measure, such as that shown in Figure 9.4 and at the top of Figure 18.1.
5. Identifying unique trigrams among a group to enable use of these trigrams as search terms in cases where their source is uncertain.

These features would also be suitable for application to other collusion detection tools. However, approaches which make comparisons on a purely pairwise basis would not be able to adopt the first two, or the last, of these suggestions.

Also, as discussed in Chapters 6 and 17, the density analysis tool is an early prototype. There are a number of alterations and improvements suggested in Section 17.4.1 which would help in developing this tool for more general use.

18.2.3 File comparison with 3CO

The ability to view relationships between files based on the trigrams they contain has saved time in the task of manually classifying files. It is also shown to be useful in quickly understanding the interaction between one file and a group of others where inappropriate code duplication is suspected.

This way of displaying files should be useful in other origin analysis tasks, as the same technique will work with smaller entities, such as methods. The visualisation is also helpful in understanding the interaction between entities where there are numerous potential matches.

The idea of combining primary colours to show the elements from one file present in up to three others can also be applied to comparisons based on other units, such as lines or sequences. To use the technique with n -grams where $n \neq 3$, the voting mechanism for choosing the colour of a token would need to be adapted.

18.3 Conclusion

In this chapter, the contributions made by this work have been reviewed, and applications for the techniques developed have been suggested as potential avenues for further work.

The aim of this work was to investigate the identification and classification of changes to evolving software code. In particular, to extract group relationships between evolving files using text analysis, and to use machine learning to classify files based on these relationships.

The application of these techniques to origin analysis has been successful. Previous approaches to origin analysis rely on predetermined rules in selecting matches between software entities. The approach taken in this work tackles this difficult problem by allowing machine learning to find patterns which best describe the problem in data from a range of sources. Also, because the approach is text-based, it should be applicable to most programming languages and to text.

A subset of the techniques were applied to collusion detection and have provided a useful similarity measure which overcomes many of the problems identified in detecting collusion in groups of unevenly sized files with high incidental similarity, and consisting of a mix of languages.

The visualisations created to support each of these applications are also based on text analysis and the relationships among files in a group. Further work is needed to assess the worth of the collusion detection visualisation. The 3CO tool has proved particularly useful in this work and could prove to be more generally applicable in file comparison tasks.

Bibliography

- [1] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC-FSE '07: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 25–34, New York, NY, USA, 2007. ACM.
- [2] Aleksi Ahtiainen, Sami Surakka, and Mikko Rahikainen. Plaggie: GPL-licensed source code plagiarism detection engine for Java exercises. In M. Wiggberg A. Berglund, editor, *6th Baltic Sea Conference on Computing Education Research*, pages 141–142. Uppsala University, Sweden, 2007.
- [3] Alex Aiken. Moss system for detecting software plagiarism. <http://theory.stanford.edu/aiken/moss/>, 1997.
- [4] Stephen F. Altschul, Warren Gish, Webb C. Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of molecular biology*, 215(3):403–410, October 1990.
- [5] Christian M. Ammann. Duplo - code clone detection tool. Sourceforge project, 2005. <http://sourceforge.net/projects/duplo/>.
- [6] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo. An automatic approach to identify class evolution discontinuities. In *IWPSE '04: Proceedings of the 7th International Workshop on Principles of Software Evolution*, pages 31–40, Washington, DC, USA, 2004. IEEE Computer Society.
- [7] Giuliano Antoniol, Umberto Villano, Massimiliano Di Penta, Gerardo Casazza, and Ettore Merlo. Identifying clones in the linux kernel. In *SCAM '01: Proceedings of the 1st IEEE International Workshop on Source Code Analysis and Manipulation*, pages 92–99. IEEE Computer Society, 2001.

- [8] Taweewup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. A differencing algorithm for object-oriented programs. *International Conference on Automated Software Engineering*, 0:2–13, 2004.
- [9] Christian Arwin and Seyed M. M. Tahaghoghi. Plagiarism detection across programming languages. In Vladimir Estivill-Castro and Gillian Dobbie, editors, *ACSC*, volume 48 of *CRPIT*, pages 277–286. Australian Computer Society, 2006.
- [10] Lerina Aversano, Luigi Cerulo, and Concettina Del Grosso. Learning from bug-introducing changes to prevent fault prone code. In *IWPSE '07: 9th International Workshop on Principles of Software Evolution*, pages 19–26, New York, NY, USA, 2007. ACM.
- [11] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE*, pages 86–95, 1995.
- [12] Brenda S. Baker. Parameterized pattern matching: Algorithms and applications. *Journal of Computer and System Sciences*, 52(1):28–42, 1996.
- [13] Brenda S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM Journal on Computing*, 26(5):1343–1362, 1997.
- [14] Thomas Ball and Stephen G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.
- [15] Victor R. Basili and Lionel C. Briand, editors. *13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy*. IEEE Computer Society, 2006.
- [16] Ira D. Baxter, Andrew Yahin, Leonardo Mendonça de Moura, Marcelo Sant’Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the 14th IEEE International Conference on Software Maintenance*, pages 368–377, 1998.
- [17] Laszlo A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.
- [18] Boumediene Belkhouche, Anastasia Nix, and Johnette Hassell. Plagiarism detection in software designs. In Seong-Moo Yoo and Letha H. Etzkorn, editors, *ACM Southeast Regional Conference*, pages 207–211. ACM, 2004.

- [19] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007.
- [20] Houda Benbrahim and Max Bramer. Text and hypertext categorization. In Max Bramer, editor, *Artificial Intelligence: An International Perspective*, volume 5640 of *Lecture Notes in Computer Science*, pages 11–38. Springer, 2009.
- [21] Benjamin Biegel and Stephan Diehl. JCCD: a flexible and extensible API for implementing custom code clone detectors. In Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto, editors, *ASE*, pages 167–168. ACM, 2010.
- [22] Benjamin Biegel, Quinten David Soetens, Willi Hornig, Stephan Diehl, and Serge Demeyer. Comparison of similarity metrics for refactoring detection. In Arie van Deursen, Tao Xie, and Thomas Zimmermann, editors, *MSR*, pages 53–62. IEEE, 2011.
- [23] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.
- [24] Avrim Blum. Machine learning theory (essay). <http://www.cs.cmu.edu/~avrim/>, 2007.
- [25] Avrim Blum and Tom Mitchell. Combining labeled and unlabeled data with co-training. In *COLT: Proceedings of the Workshop on Computational Learning Theory*, Morgan Kaufmann Publishers, pages 92–100, 1998.
- [26] Salah Bouktif, Balazs Keg], and Houari Sahraoui. Combining software quality predictive models: An evolutionary approach. In *ICSM '02: Proceedings of the International Conference on Software Maintenance*, pages 385–392, Washington, DC, USA, 2002. IEEE Computer Society.
- [27] Max Bramer. *Principles of Data Mining*. Springer, 2007.
- [28] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.
- [29] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.
- [30] Björn Bringmann, Siegfried Nijssen, and Albrecht Zimmermann. Pattern-based classification: A unifying perspective. *CoRR*, abs/1111.6191, 2011.
- [31] Romain Brixtel, Mathieu Fontaine, Boris Lesner, Cyril Bazin, and Romain Robbes. Language-independent clone detection applied to plagiarism detection. In *SCAM'10: Proceedings of 10th IEEE International Workshop on Source Code Analysis and Manipulation*, pages 77–86. IEEE Computer Society, 2010.

- [32] Andrei Z. Broder. On the resemblance and containment of documents. In *SEQUENCES '97: Proceedings of the Compression and Complexity of Sequences 1997*, page 21, Washington, DC, USA, 1997. IEEE Computer Society.
- [33] Steven Burrows. *Source Code Authorship Attribution*. PhD thesis, School of Computer Science and Information Technology, College of Science, Engineering and Health, RMIT University, Melbourne, Victoria, Australia., November 2010.
- [34] Steven Burrows, Seyed M. M. Tahaghoghi, and Justin Zobel. Efficient plagiarism detection for large code repositories. *Software Practice and Experience*, 37(2):151–175, 2007.
- [35] Raymond P. L. Buse and Westley Weimer. A metric for software readability. In Barbara G. Ryder and Andreas Zeller, editors, *ISSTA*, pages 121–130. ACM, 2008.
- [36] Rich Caruana, Alexandru Niculescu-Mizil, Geoff Crew, and Alex Ksikes. Ensemble selection from libraries of models, 2004.
- [37] Pierre Caserta and Olivier Zendra. Visualization of the static aspects of software: A survey. *IEEE Trans. Vis. Comput. Graph.*, 17(7):913–933, 2011.
- [38] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In H. V. Jagadish and Inderpal Singh Mumick, editors, *SIGMOD Conference*, pages 493–504. ACM Press, 1996.
- [39] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. Smote: Synthetic minority over-sampling technique. *J. Artif. Intell. Res. (JAIR)*, 16:321–357, 2002.
- [40] Kai Chen, Stephen R. Schach, Liguu Yu, Jeff Offutt, and Gillian Z. Heller. Open-source change logs. *Empirical Software Engineering.*, 9(3):197–210, 2004.
- [41] Xin Chen, Brent Francia, Ming Li, Brian McKinnon, and Amit Seker. Shared information and program plagiarism detection. *IEEE Transactions on Information Theory*, 50(7):1545–1551, 2004.
- [42] Michel Chilowicz, Étienne Duris, and Gilles Roussel. Finding similarities in source code through factorization. *Electronic Notes in Theoretical Computer Science*, 238(5):47–62, 2009.

- [43] Kenneth Ward Church and Jonathan Isaac Helfman. Dotplot: A program for exploring self-similarity in millions of lines of text and code. *Computing Science and Statistics*, 24:58, 1993.
- [44] Victor Ciesielski, Brian T. Lam, and Minh Luan Nguyen. Comparison of evolutionary and conventional feature extraction methods for malt classification. In *IEEE Congress on Evolutionary Computation*, pages 1–7. IEEE, 2012.
- [45] David A. Cieslak and Nitesh V. Chawla. Learning decision trees for unbalanced data. In Walter Daelemans, Bart Goethals, and Katharina Morik, editors, *ECML/PKDD (1)*, volume 5211 of *Lecture Notes in Computer Science*, pages 241–256. Springer, 2008.
- [46] Daoud Clarke, Peter Lane, and Paul Hender. Developing robust models for favourability analysis. In *Proceedings of the 2nd Workshop on Computational Approaches to Subjectivity and Sentiment Analysis (WASSA 2.011)*, pages 44–52, Portland, Oregon, June 2011. Association for Computational Linguistics.
- [47] Paul Clough. Old and new challenges in automatic plagiarism detection. In *National Plagiarism Advisory Service, 2003*; <http://ir.shef.ac.uk/cloughie/index.html>, pages 391–407, 2003.
- [48] William W. Cohen. Fast effective rule induction. In *ICML*, pages 115–123, 1995.
- [49] Stephen Cook, Keiichi Nakata, and Paul Wernick. European laboratory for software evolution (else): Vision statement. In *Automated Software Engineering Workshops, 2008. ASE Workshops 2008. 23rd IEEE/ACM International Conference on*, pages 92–95. IEEE, 2008.
- [50] James R. Cordy and Chanchal K. Roy. The NiCad clone detector. In Susan E. Sim and Filippo Ricca, editors, *ICPC*, pages 219–220. IEEE Computer Society, 2011.
- [51] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Trans. on Software Engineering*, 35(5):684–702, 2009.
- [52] Georgina Cosma. *An approach to source-code plagiarism detection and investigation using Latent Semantic Analysis*. PhD thesis, University of Warwick, 2008.
- [53] UCL Crest SBSE Group. Search Based Software Engineering Repository. http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/repository.html.

- [54] Davor Cubranic and Gail C. Murphy. Automatic bug triage using text categorization. In *SEKE '04: Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering*, pages 92–97, 2004.
- [55] Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. In Schäfer et al. [209], pages 481–490.
- [56] Barthélémy Dagenais and Martin P. Robillard. Semdiff: Analysis and recommendation support for API evolution. In Fickas et al. [75], pages 599–602.
- [57] Marco D’Ambros, Harald Gall, Michele Lanza, and Martin Pinzger. Analysing software repositories to understand software evolution. In Mens and Demeyer [167], pages 37–67.
- [58] Neil Davey, Paul Barson, Simon Field, and Ray Frank. The development of a software clone detector. *International Journal of Applied Software Technology*, 1(3/4):219–236, 1995.
- [59] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.
- [60] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 166–177, New York, NY, USA, 2000. ACM.
- [61] Lee R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297–302, 1945.
- [62] Thomas G. Dietterich. Ensemble learning. In M.A. Arbib, editor, *The Handbook of Brain Theory and Neural Networks, Second edition*. The MIT Press, Cambridge, MA, 2002.
- [63] Diff - The linux diff utility.: <http://www.gnu.org/software/diffutils/>.
- [64] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In Dave Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 404–428. Springer, 2006.
- [65] Stéphane Ducasse, Oscar Nierstrasz, and Matthias Rieger. On the effectiveness of clone detection by string matching: Research articles. *Journal of Software Maintenance and Evolution*, 18(1):37–58, 2006.

- [66] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, *ICSM '99: Proceedings of the 15th IEEE International Conference on Software Maintenance*, pages 109–118. IEEE, 1999.
- [67] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Aiding program comprehension by static and dynamic feature analysis. In *ICSM '01: Proceedings of the 17th IEEE International Conference on Software Maintenance*, pages 602–611, 2001.
- [68] Emelie Engström, Mats Skoglund, and Per Runeson. Empirical evaluations of regression test selection techniques: a systematic review. In H. Dieter Rombach, Sebastian G. Elbaum, and Jürgen Münch, editors, *ESEM*, pages 22–31. ACM, 2008.
- [69] Seyda Ertekin, Jian Huang, Léon Bottou, and C. Lee Giles. Learning on the border: active learning in imbalanced data classification. In Mário J. Silva, Alberto H. F. Laender, Ricardo A. Baeza-Yates, Deborah L. McGuinness, Bjørn Olstad, Øystein Haug Olsen, and André O. Falcão, editors, *CIKM*, pages 127–136. ACM, 2007.
- [70] Pedro G. Espejo, Sebastián Ventura, and Francisco Herrera. A survey on the application of genetic programming to classification. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, 40(2):121–144, 2010.
- [71] Jinan A. W. Faidhi and Stuart K. Robinson. An empirical approach for detecting program similarity within a university programming environment. *Computers and Education*, 11(1):11–19, 1987.
- [72] Usama M. Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. From data mining to knowledge discovery in databases. *AI Magazine*, 17(3):37–54, 1996.
- [73] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. In Manfred Paul and Bernard Robinet, editors, *Symposium on Programming*, volume 167 of *Lecture Notes in Computer Science*, pages 125–132. Springer, 1984.
- [74] Stephen Few. *Data Visualization for Human Perception*. The Interaction Design Foundation, Aarhus, Denmark, 2010.

- [75] Stephen Fickas, Joanne M. Atlee, and Paola Inverardi, editors. *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 2009.
- [76] Beat Fluri, Emanuel Giger, and Harald Gall. Discovering patterns of change types. In *Automated Software Engineering*, pages 463–466. IEEE, 2008.
- [77] Beat Fluri, Michael Wuersch, Martin Pinzger, and Harald Gall. Change distilling:tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.
- [78] Martin Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley, Reading, MA, USA., 1999.
- [79] A. Frank and A. Asuncion. UCI machine learning repository, 2010.
- [80] Eibe Frank, Geoffrey Holmes, Richard Kirkby, and Mark Hall. Racing committees for large datasets. In *DS'02: Proceedings of the 5th Int'l. Conference on Discovery Science*, pages 153–164, London, UK, 2002. Springer-Verlag.
- [81] Manuel Freire. Visualizing program similarity in the AC plagiarism detection system. In *Proceedings of Advanced Visual Interfaces (AVI)*, pages 404–407, New York, USA, May 2008. ACM Press.
- [82] Manuel Freire, Manuel Cebrián, and Emilio del Rosal. AC: An integrated source code plagiarism detection environment. *CoRR*, abs/cs/0703136, 2007.
- [83] Manuel Freire, Manuel Cebrian, and Emilio del Rosal. Uncovering plagiarism networks. *arXiv:cs/0703136v7 [cs.IT]*, 2007, revised 2011.
- [84] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *EuroCOLT '95: Proceedings of the 2nd European Conference on Computational Learning Theory*, pages 23–37, London, UK, 1995. Springer-Verlag.
- [85] Jerome Friedman, T. Hastie, and R. Tibshirani. *Additive logistic regression: a statistical view of boosting*, 1998.
- [86] Jon Froehlich and Paul Dourish. Unifying artifacts and activities in a visual tool for distributed software development teams. In *ICSE*, pages 387–396. IEEE Computer Society, 2004.
- [87] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In Schäfer et al. [209], pages 321–330.

- [88] Tudor Gîrba, Stéphane Ducasse, Adrian Kuhn, Radu Marinescu, and Rațiu Daniel. Using concept analysis to detect co-change patterns. In *IWPSE '07: 9th International Workshop on Principles of Software Evolution*, pages 83–89, New York, NY, USA, 2007. ACM.
- [89] David Gitchell and Nicholas Tran. Sim: a utility for detecting similarity in computer programs. In Jane Prey and Robert E. Noonan, editors, *SIGCSE*, pages 266–270. ACM, 1999.
- [90] Michael W. Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.
- [91] Carsten Görg and Peter Weißgerber. Detecting and visualizing refactorings from software archives. In *IWPC '05: Proceedings of the 13th IEEE Int'l. Workshop on Program Comprehension*, pages 205–214. IEEE Computer Society, 2005.
- [92] Pam Green. Ferret density analysis tool. <http://homepages.feis.herts.ac.uk/~gp2ag/density.html>, 2010.
- [93] Pam Green, Peter C. R. Lane, Austen Rainer, and Sven-Bodo Scholz. Building classifiers to identify split files. In Petra Perner, editor, *MLDM Posters*, pages 1–8. IBAI Publishing, 2009.
- [94] Pam Green, Peter C. R. Lane, Austen Rainer, and Sven-Bodo Scholz. Analysing Ferret XML reports to estimate the density of copied code. Technical Report 501, University of Hertfordshire, College Lane, Hatfield, Herts AL10 9AB, UK, April 2010.
- [95] Pam Green, Peter C. R. Lane, Austen Rainer, and Sven-Bodo Scholz. Selecting features in origin analysis. In Max Bramer, Miltos Petridis, and Adrian Hopgood, editors, *SGAI Conference*, pages 379–392. Springer, 2010.
- [96] Pam Green, Peter C. R. Lane, Austen Rainer, and Sven-Bodo Scholz. Unscrambling code clones for one-to-one matching of duplicated code. Technical Report 502, University of Hertfordshire, College Lane, Hatfield, Herts AL10 9AB, UK, April 2010.
- [97] Pam Green, Peter C. R. Lane, Austen Rainer, Sven-Bodo Scholz, and Steve Bennett. Same difference: Detecting collusion by finding unusual

- shared elements. In *Proceedings of the 5th International Plagiarism Conference*, Newcastle-upon-Tyne, UK, July 2012. iParadigms, iParadigms. <http://archive.plagiarismadvice.org//conference-programme>.
- [98] Isabelle Guyon and Andre Elisseeff. *Feature extraction: Foundations and applications*, volume 978-3540354871 of *Studies in fuzziness and soft computing*, chapter An Introduction to Feature Extraction. Springer-Verlag, New York, Secaucus, NJ, USA, August 2006.
- [99] Jurriaan Hage, Peter Rademaker, and Nike van Vugt. A comparison of plagiarism detection tools. Technical Report UU-CS-2010-015, Utrecht University, June 2010. Further information from <http://www.cs.uu.nl/docs/vakken/apa/10plagiarismdetection.pdf>.
- [100] Maria Halkidi, Diomidis Spinellis, George Tsatsaronis, and Michalis Vazirgiannis. Data mining in software engineering. *Intelligent Data Analysis*, 15(3):413–441, 2011.
- [101] Mark A. Hall. *Correlation-based Feature Subset Selection for Machine Learning*. PhD thesis, University of Waikato, Hamilton, New Zealand, 1998.
- [102] Mark A. Hall. Weka forum advice. <http://comments.gmane.org/gmane.comp.ai.weka/10582>, August 2007.
- [103] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.
- [104] David J. Hand, Heikki Mannila, and Padhraic Smyth. *Principles of Data Mining*. MIT Press, Cambridge, MA, 2001.
- [105] Simon Harris. Simian. <http://www.redhillconsulting.com.au/products/simian>. Copyright (c) 2003-08 RedHill Consulting Pty. Ltd.
- [106] Trevor Hastie and Robert Tibshirani. Classification by pairwise coupling. In *NIPS '97: Proceedings of the 1997 conference on Advances in neural information processing systems 10*, pages 507–513, Cambridge, MA, USA, 1998. MIT Press.
- [107] Haibo He and Edwardo A. Garcia. Learning from imbalanced data. *Knowledge and Data Engineering, IEEE Transactions on*, 21(9):1263–1284, Sept. 2009.
- [108] Abram Hindle, Daniel M. Germán, Michael W. Godfrey, and Richard C. Holt. Automatic classification of large changes into maintenance categories. In *ICPC '09: IEEE 17th International Conference on Program Comprehension*, pages 30–39. IEEE Computer Society, 2009.

- [109] Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 20(8):832–844, 1998.
- [110] Timothy C. Hoad and Justin Zobel. Methods for identifying versioned and plagiarized documents. *Journal of the American Society for Information Science and Technology*, 54(3):203–215, 2003.
- [111] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin. A practical guide to support vector classification, 2000.
- [112] Liuliu Huang, Shumin Shi, and Heyan Huang. A new method for code similarity detection. In *Progress in Informatics and Computing (PIC), 2010 IEEE International Conference on*, volume 2, pages 1015–1018, dec. 2010.
- [113] Paul Jaccard. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bulletin de la Socit Vaudoise des Sciences Naturelles*, 37:547–579, 1901.
- [114] Ameera Jadalla and Ashraf Elnagar. PDE4Java: Plagiarism detection engine for Java source code: a clustering approach. *IJBIDM*, 3(2):121–135, 2008.
- [115] Jeong-Hoon Ji, Soo-Hyun Park, Gyun Woo, and Hwan-Gue Cho. Source code similarity detection using adaptive local alignment of keywords. In David S. Munro, Hong Shen, Quan Z. Sheng, Henry Detmold, Katrina E. Falkner, Cruz Izu, Paul D. Coddington, Bradley Alexander, and Si-Qing Zheng, editors, *PDCAT*, pages 179–180. IEEE Computer Society, 2007.
- [116] Lingxiao Jiang, Ghassan Mishergghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.
- [117] Edward L. Jones. Metrics based plagiarism monitoring. *Journal of Computing Sciences in Colleges*, 16(4):253–261, 2001.
- [118] Karen Sparck Jones, Steve Walker, and Stephen E. Robertson. A probabilistic model of information retrieval: development and comparative experiments - parts 1 and 2. *Information Processing and Management*, 36(6):779–808, 809–840, 2000.
- [119] Mike Joy, Georgina Cosma, Jane Yau, and Jane Sinclair. Source code plagiarism - a student perspective. *IEEE Trans. Education*, 54(1):125–132, 2011.

- [120] Mike Joy and Michael Luck. Plagiarism in programming assignments. *IEEE Transactions on Education*, 42(1):129–133, 1999.
- [121] Elmar Jüergens, Florian Deissenboeck, and Benjamin Hummel. CloneDetective - a workbench for clone detection research. In Fickas et al. [75], pages 603–606.
- [122] Vedran Juricic. Detecting source code similarity using low-level languages. In *Information Technology Interfaces (ITI), Proceedings of the ITI 2011 33rd International Conference on*, pages 597–602, June 2011.
- [123] Huzefa H. Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance*, 19(2):77–131, 2007.
- [124] Toshihiro Kamiya. AIST CCFinder official site. <http://www.ccfinder.net/index.html>.
- [125] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.
- [126] S. S. Keerthi, S. K. Shevade, C. Bhattacharyya, and K. R. K. Murthy. Improvements to Platt’s SMO algorithm for SVM classifier design. *Neural Computation*, 13(3):637–649, 2001.
- [127] Miryung Kim and David Notkin. Program element matching for multi-version program analyses. In *MSR ’06: Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 58–64, New York, NY, USA, 2006. ACM.
- [128] Miryung Kim, David Notkin, and Dan Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE ’07: Proceedings of the 29th International Conference on Software Engineering*, pages 333–343, Washington, DC, USA, 2007. IEEE Computer Society.
- [129] Sunghun Kim, E. James Whitehead Jr., and Yi Zhang 0001. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.
- [130] Sunghun Kim, Kai Pan, and E. James Whitehead Jr. When functions change their names: Automatic detection of origin relationships. In *WCRE ’05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 143–152, Pittsburgh, PA, USA, 2005. IEEE Computer Society.

- [131] Rob Kitchin and Martin Dodge. *Code/Space: Software and Everyday Life*. MIT Press, Cambridge MA, 2011.
- [132] Josef Kittler, Mohamad Hatef, Robert P. W. Duin, and Jiri Matas. On combining classifiers. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(3):226–239, 1998.
- [133] Patrick Knab, Martin Pinzger, and Abraham Bernstein. Predicting defect densities in source code files with decision tree learners. In *MSR '06: Proceedings of the 2006 International Workshop on Mining Software Repositories*, New York, NY, USA, May 2006. ACM Press, ACM Press.
- [134] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. *Lecture Notes in Computer Science*, 2126:40–56, 2001.
- [135] Kostas Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *WCRE '97: Proceedings of the 4th Working Conference on Reverse Engineering*, pages 44–, 1997.
- [136] Kostas A. Kontogiannis, Renato De Mori, Ettore Merlo, M. Galler, and Morris Bernstein. Pattern matching for clone and concept detection. *Reverse engineering*, pages 77–108, 1996.
- [137] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In Basili and Briand [15], pages 253–262.
- [138] Yasemin Kösker, Burak Turhan, and Ayse Basar Bener. An expert system for determining candidate software classes for refactoring. *Expert Systems with Applications*, 36(6):10000–10003, 2009.
- [139] Jens Krinke. Identifying similar code with program dependence graphs. In *Proc. Eighth Working Conference on Reverse Engineering*, pages 301–309, 2001.
- [140] Miroslav Kubat and Stan Matwin. Addressing the curse of imbalanced training sets: One-sided selection. In Douglas H. Fisher, editor, *ICML*, pages 179–186. Morgan Kaufmann, 1997.
- [141] Ludmila I. Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms*. Wiley-Interscience, 2004.
- [142] Ahmed Lamkanfi, Serge Demeyer, Emanuel Giger, and Bart Goethals. Predicting the severity of a reported bug. In Jim Whitehead and Thomas Zimmermann, editors, *MSR*, pages 1–10. IEEE, 2010.

- [143] Thomas Lancaster. *Effective and Efficient Plagiarism Detection*. PhD thesis, Birmingham City University, Birmingham, UK, 2003.
- [144] Thomas Lancaster and Mark Tetlow. Does automated anti-plagiarism have to be complex? Evaluating more appropriate software metrics for finding collusion. In *Ascilite 2005*, pages 361–370, Brisbane, Australia, 2005. Ascilite 2005.
- [145] Niels Landwehr, Mark Hall, and Eibe Frank. Logistic model trees. *Machine Learning*, 59(1-2):161–205, 2005.
- [146] Peter C. R. Lane, Caroline M. Lyon, and James A. Malcolm. Demonstration of the Ferret plagiarism detector. In *2nd Int'l. Plagiarism Conference*, 2006.
- [147] Hugh Leather, Edwin V. Bonilla, and Michael F. P. O'Boyle. Automatic feature generation for machine learning based optimizing compilation. In *CGO*, pages 81–91. IEEE Computer Society, 2009.
- [148] M. M. Lehman. Laws of software evolution revisited. In Carlo Montangero, editor, *EWSPT*, volume 1149 of *Lecture Notes in Computer Science*, pages 108–124. Springer, 1996.
- [149] Zhenmin Li and Yuanyuan Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 306–315, New York, NY, USA, 2005. ACM.
- [150] Christian Lindig. Mining patterns and violations using concept analysis. <http://www.st.cs.uni-sb.de/lindig/papers>, 2007.
- [151] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. GPLAG: detection of software plagiarism by program dependence graph analysis. In Tina Eliassi-Rad, Lyle H. Ungar, Mark Craven, and Dimitrios Gunopulos, editors, *KDD*, pages 872–881. ACM, 2006.
- [152] Wei Zhong Liu, Allan P. White, Simon G. Thompson, and Max A. Bramer. Techniques for dealing with missing values in classification. In Xiaohui Liu, Paul R. Cohen, and Michael R. Berthold, editors, *IDA*, volume 1280 of *Lecture Notes in Computer Science*, pages 527–536. Springer, 1997.
- [153] V. Benjamin Livshits and Thomas Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *13th ACM*

SIGSOFT International Symposium on the Foundations of Software Engineering, pages 296–305. ACM Press, 2005.

- [154] Caroline M. Lyon, Ruth Barrett, and James A. Malcolm. Experiments in electronic plagiarism detection. Technical Report 388, University of Hertfordshire, 2003. <http://hdl.handle.net/2299/1774>.
- [155] Caroline M. Lyon, Ruth Barrett, and James A. Malcolm. A theoretical basis to the automated detection of copying between texts, and its practical implementation in the Ferret plagiarism and collusion detector. In *JISC(UK) Conference on Plagiarism: Prevention, Practice and Policies Conference*, 2004.
- [156] Caroline M. Lyon, Ruth Barrett, and James A. Malcolm. Plagiarism is easy, but also easy to detect. *Plagiary: Cross-disciplinary studies in plagiarism, fabrication and falsification*, 1:1–10, 2006.
- [157] Caroline M. Lyon, James A. Malcolm, and Bob Dickerson. Detecting short passages of similar text in large document collections. In *Proceedings of Conference on Empirical Methods in Natural Language Processing*. SIGDAT, Special Interest Group of the ACL, 2001.
- [158] James A. Malcolm and Peter C. R. Lane. An approach to detecting article spinning. In *3rd International Conference on Plagiarism*, 2008.
- [159] Jonathan I. Maletic and Andrian Marcus. Supporting program comprehension using semantic and structural information. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 103–112, Washington, DC, USA, 2001. IEEE Computer Society.
- [160] Udi Manber. Finding similar files in a large file system. In *USENIX Winter*, pages 1–10, 1994.
- [161] Heikki Mannila. Data mining: Machine learning, statistics, and databases. In Per Svensson and James C. French, editors, *SSDBM*, pages 2–9. IEEE Computer Society, 1996.
- [162] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, Cambridge, MA, USA, 2001.
- [163] Andrian Marcus and Jonathan I. Maletic. Identification of high-level concept clones in source code. In *ASE*, pages 107–114. IEEE Computer Society, 2001.
- [164] Stephen Marsland. *Machine Learning: An Algorithmic Perspective*. Chapman & Hall/CRC, 1st edition, 2009.

- [165] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *ICSM*, pages 244–253, 1996.
- [166] Prem Melville and Raymond J. Mooney. Constructing diverse classifier ensembles using artificial training examples. In Georg Gottlob and Toby Walsh, editors, *IJCAI*, pages 505–512. Morgan Kaufmann, 2003.
- [167] Tom Mens and Serge Demeyer, editors. *Software Evolution*. Springer, 2008.
- [168] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.
- [169] Tim Menzies, Bora Caglayan, Ekrem Kocaguneli, Joe Krall, Fayola Peters, and Burak Turhan. The PROMISE repository of empirical software engineering data. <http://promisedata.googlecode.com>, June 2012.
- [170] Ettore Merlo. Detection of plagiarism in university projects using metrics-based spectral similarity. In Rainer Koschke, Ettore Merlo, and Andrew Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, volume 06301 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.
- [171] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill Higher Education, 1997.
- [172] Tom M. Mitchell. The discipline of machine learning. Machine Learning CMU-ML-06-108, Carnegie Mellon University, July 2006.
- [173] Lefteris Moussiades and Athena Vakali. PDetect: A clustering approach for detecting plagiarism in source code datasets. *The Computer Journal*, 48(6):651–661, 2005.
- [174] Maxim Mozgovoy. *Enhancing Computer-Aided Plagiarism Detection*. PhD thesis, Department of Computer Science, University of Joensuu, University of Joensuu, P.O.Box 111, FIN-80101 Joensuu, Finland, November 2007.
- [175] Maxim Mozgovoy, Kimmo Fredriksson, Daniel R. White, Mike Joy, and Erkki Sutinen. Fast plagiarism detection system. In Mariano P. Consens and Gonzalo Navarro, editors, *SPIRE*, volume 3772 of *Lecture Notes in Computer Science*, pages 267–270. Springer, 2005.

- [176] Emerson Murphy-Hill, Andrew P. Black, Danny Dig, and Chris Parnin. Gathering refactoring data: a comparison of four methods. In *Proceedings of the 2nd Workshop on Refactoring Tools, WRT '08*, pages 7:1–7:5, New York, NY, USA, 2008. ACM.
- [177] Sandra Nadelson. Academic misconduct by university students: Faculty perceptions and responses. *Plagiary: Cross Disciplinary Studies in Plagiarism, Fabrication, and Falsification*, 2(2), 2007.
- [178] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *ICSE '06: Proceeding of the 28th Int'l. Conference on Software Engineering*, pages 452–461, New York, NY, USA, 2006. ACM.
- [179] Iulian Neamtiu, Jeffrey S. Foster, and Michael W. Hicks. Understanding source code evolution using abstract syntax tree matching. In *MSR*. ACM, 2005.
- [180] Nils J. Nilsson. Introduction to machine learning: An early draft proposed textbook. <http://robotics.stanford.edu/people/nilsson/mlbook.html>, 1998.
- [181] Seo-Young Noh, Sangwoo Kim, and Cheonyoung Jung. A lightweight program similarity detection model using XML and Levenshtein distance. In Hamid R. Arabnia, editor, *FECS*, pages 3–9. CSREA Press, 2006.
- [182] University of Alabama. Code clones literature. <http://students.cis.uab.edu/tairasr/clones/literature/#standalone>.
- [183] Sourceforge open source software repository. <http://sourceforge.net/>, 1998.
- [184] François Pachet and Pierre Roy. Analytical features: A knowledge-based approach to audio feature generation. *EURASIP J. Audio, Speech and Music Processing*, 2009, 2009.
- [185] Frank Padberg, Thomas Ragg, and Ralf Schoknecht. Using machine learning for estimating the defect content after an inspection. *IEEE Transactions on Software Engineering*, 30(1):17–28, 2004.
- [186] Kai Pan, Sunghun Kim, and Jr. E. James Whitehead. Bug classification using program slicing metrics. In *SCAM '06: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 31–42, Washington, DC, USA, 2006. IEEE Computer Society.
- [187] Chris Parnin and Carsten Görg. Improving change descriptions with change contexts. In Ahmed E. Hassan, Michele Lanza, and Michael W. Godfrey, editors, *MSR*, pages 51–60. ACM, 2008.

- [188] J. Platt. *Fast Training of Support Vector Machines using Sequential Minimal Optimization*, chapter Advances in Kernel Methods - Support Vector Learning. MIT Press, Cambridge, MA, USA, 1999. pp. 185–208.
- [189] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. Finding plagiarisms among a set of programs with JPlag. *The Journal of Universal Computer Science*, 8(11):1016–1038, 2002.
- [190] NASA Metrics Data Program. NASA software metrics dataset. <http://mdp.ivv.nasa.gov>.
- [191] Dorian Pyle. *Data Preparation for Data Mining*. Morgan Kaufmann, 1999.
- [192] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.
- [193] Michael O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.
- [194] Austen W. Rainer, Peter C. R. Lane, James A. Malcolm, and Sven-Bodo Scholz. Using n-grams to rapidly characterise the evolution of software code. In *The 4th International ERCIM Workshop on Software Evolution and Evolvability*, 2008.
- [195] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Path-sensitive inference of function precedence protocols. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 240–250, Washington, DC, USA, 2007. IEEE Computer Society.
- [196] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Static specification inference using predicate mining. *SIGPLAN Notices*, 42(6):123–134, 2007.
- [197] Jacek Ratzinger, Thomas Sigmund, Peter Vorburger, and Harald Gall. Mining software evolution to predict refactoring. *Empirical Software Engineering and Measurement, International Symposium on*, 0:354–363, 2007.
- [198] Evandro N. Regolin, Gustavo A. de Souza, Aurora R. T. Pozo, and Silvia R. Vergilio. Exploring machine learning techniques for software size estimation. In *SCCC '03: Proceedings of the 23rd International Conference of the Chilean Computer Science Society*, page 130, Washington, DC, USA, 2003. IEEE Computer Society.

- [199] Randy L. Ribler and Marc Abrams. Using visualization to detect plagiarism in computer science classes. In *INFOVIS*, pages 173–178, 2000.
- [200] Claudio Riva. Visualizing software release histories with 3DSoftVis. In *ICSE*, page 789, 2000.
- [201] Roland Rivest. The MD5 Message-Digest algorithm. <http://tools.ietf.org/pdf/rfc1321.pdf>, April 1992.
- [202] Stephen E. Robertson, Steve Walker, Susan Jones, Micheline Hancock-Beaulieu, and Mike Gatford. Okapi at TREC-3. In *TREC*, pages 109–126, 1994.
- [203] Juan J. Rodriguez, Ludmila I. Kuncheva, and Carlos J. Alonso. Rotation Forest: A new classifier ensemble method. *IEEE Transactions Pattern Analysis and Machine Intelligence*, 28(10):1619–1630, 2006.
- [204] Chanchal Kumar Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.
- [205] David E. Rumelhart and James L. McClelland. *Parallel distributed processing*, 2 vols. MIT Press, 1986. Explorations in the microstructure of cognition; Vol.1: Foundations; Vol.2: Psychological and biological models.
- [206] Gerard Salton and Chris Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, 24(5):513–523, 1988.
- [207] Arthur L. Samuel. Informal quote. Reported by Andrew Y. Ng, In The Motivation and Applications of Machine Learning, Video Lecture from the Stanford Engineering Everywhere Series., May 2009. http://videlectures.net/stanfordcs229f08_ng_lect01/.
- [208] Thorsten Schäfer, Jan Jonas, and Mira Mezini. Mining framework usage changes from instantiation code. In Schäfer et al. [209], pages 471–480.
- [209] Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn, editors. *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. ACM, 2008.
- [210] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 76–85, New York, NY, USA, 2003. ACM.

- [211] Alexander K. Seewald. How to make stacking better and faster while also taking care of an unknown weakness. In *ICML '02: Proceedings of the 19th International Conference on Machine Learning*, pages 554–561, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.
- [212] Alexander K. Seewald and Johannes Fürnkranz. An evaluation of grading classifiers. In *IDA '01: Proceedings of the 4th International Conference on Advances in Intelligent Data Analysis*, pages 115–124, London, UK, 2001. Springer-Verlag.
- [213] Burr Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.
- [214] Shai Shalev-Shwartz, Yoram Singer, and Nathan Srebro. Pegasos: Primal estimated sub-gradient solver for SVM. In Zoubin Ghahramani, editor, *ICML*, volume 227 of *ACM International Conference Proceeding Series*, pages 807–814. ACM, 2007.
- [215] Victor S. Sheng and Charles X. Ling. Cost-sensitive learning. In John Wang, editor, *Encyclopedia of Data Warehousing and Mining*, pages 339–345. IGI Global, 2009.
- [216] Lin Shi, Hao Zhong, Tao Xie, and Mingshu Li. An empirical study on evolution of API documentation. In Dimitra Giannakopoulou and Fernando Orejas, editors, *FASE*, volume 6603 of *Lecture Notes in Computer Science*, pages 416–431. Springer, 2011.
- [217] Jelber Sayyad Shirabad, Timothy C. Lethbridge, and Stan Matwin. Mining the maintenance history of a legacy software system. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 95, Washington, DC, USA, 2003. IEEE Computer Society.
- [218] Shivkumar Shivaji, E. James Whitehead Jr., Ram Akella, and Sunghun Kim. Reducing features to improve bug prediction. In *ASE*, pages 600–604. IEEE Computer Society, 2009.
- [219] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *MSR '05: Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, New York, NY, USA, 2005. ACM.
- [220] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

- [221] Jeong Woo Son, Seong-Bae Park, and Se-Young Park. Program plagiarism detection using parse tree kernels. In Qiang Yang and Geoffrey I. Webb, editors, *PRICAI*, volume 4099 of *Lecture Notes in Computer Science*, pages 1000–1004. Springer, 2006.
- [222] Dejan Sraka and Branko Kaucic. Source code plagiarism. In Vesna Luzar-Stiffler, Iva Jarec, and Zoran Bekic, editors, *ITI*, pages 461–466. IEEE, 2009.
- [223] Tomasz F. Stepinski, Michael P. Mendenhall, and Brian D. Bue. Machine cataloging of impact craters on mars. *Icarus*, 203(1):77–87, 2009.
- [224] Maureen Stone. Choosing colors for data visualization. Business Intelligence Network http://72.251.211.178/articles/b-eye/choosing_colors.pdf, January 2006. Accessed January 2012.
- [225] Margaret-Anne D. Storey, Davor Cubranic, and Daniel M. Germán. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In Thomas L. Naps and Wim De Pauw, editors, *SOFTVIS*, pages 193–202. ACM, 2005.
- [226] Herbert A. Sturges. The Choice of a Class Interval. *Journal of the American Statistical Association*, 21(153):65–66, Mar 1926.
- [227] Marc Sumner, Eibe Frank, and Mark A. Hall. Speeding up logistic model tree induction. In *9th European Conference on Principles and Practice of Knowledge Discovery in Databases*, Porto, Portugal, pages 675–683. Springer, 2005.
- [228] Richard S. Sutton and Andrew G. Barto. Reinforcement learning: An introduction. *IEEE Transactions on Neural Networks*, 9(5):1054–1054, 1998.
- [229] Robert Tairas and Jeff Gray. Phoenix-based clone detection using suffix trees. In *ACM Southeast Regional Conference*, pages 679–684, 2006.
- [230] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*comment: bugs or bad comments?*/. In *SOSP '07: Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 145–158, New York, NY, USA, 2007. ACM.
- [231] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes?: an exploratory study in industry. In Will Tracz, Martin P. Robillard, and Tevfik Bultan, editors, *SIGSOFT FSE*, page 51. ACM, 2012.

- [232] Suresh Thummalapenta and Tao Xie. NEGWeb: Static defect detection via searching billions of lines of open source code. Technical Report TR-2007-24, North Carolina State University Department of Computer Science, Raleigh, NC, August 2007.
- [233] Suresh Thummalapenta and Tao Xie. Alattin: mining alternative patterns for defect detection. *ASE '11: Proceedings of the 26th IEEE International Conference on Automated Software Engineering*, 18(3-4):293–323, 2011.
- [234] Kai Ming Ting and Ian H. Witten. Stacking bagged and dagged models. In *Proc. 14th International Conference on Machine Learning*, pages 367–375. Morgan Kaufmann, 1997.
- [235] Kai Ming Ting and Ian H. Witten. Issues in stacked generalization. *Journal of Artificial Intelligence Research*, 10:271–289, 1999.
- [236] Qiang Tu. On navigation and analysis of software architecture evolution. Master's thesis, Mathematics and Computer Science, University of Waterloo, Ontario, Canada, 2002.
- [237] Qiang Tu and Michael W. Godfrey. An integrated approach for studying architectural evolution. In *IWPC '02: Proceedings of the 10th Int'l. Workshop on Program Comprehension*, pages 127–136. IEEE Computer Society, 2002.
- [238] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. On detection of gapped code clones using gap locations. In *APSEC '02: Proceedings of the Ninth Asia-Pacific Software Engineering Conference*, page 327, Washington, DC, USA, 2002. IEEE Computer Society.
- [239] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
- [240] Stefan-Lucian Voinea. *Software Evolution Visualization*. PhD thesis, Technische Universitat Eindhoven, 2007.
- [241] Vera Wahler, Dietmar Seipel, Jürgen Wolff von Gudenberg, and Gregor Fischer. Clone detection in source code by frequent itemset techniques. In *SCAM*, pages 128–135, 2004.
- [242] Andrew Walenstein, Nitin Jyoti, Junwei Li, Yun Yang, and Arun Lakhotia. Problems creating task-relevant clone detection reference data. *Reverse Engineering, Working Conference on*, 0:285, 2003.

- [243] Geoffrey I. Webb. Multiboosting: A technique for combining boosting and wagging. *Machine Learning*, 40(2):159–196, 2000.
- [244] Peter Weißgerber. *Automatic Refactoring Detection in Version Archives*. PhD thesis, Universität Trier, 2009.
- [245] Peter Weißgerber and Stephan Diehl. Identifying refactorings from source-code changes. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society.
- [246] Michael J. Wise. Yap3: improved detection of similarities in computer program and other texts. In John Impagliazzo, Elizabeth S. Adams, and Karl J. Klee, editors, *SIGCSE*, pages 130–134. ACM, 1996.
- [247] Ian H. Witten and Eibe Frank. *Data mining: Practical machine learning tools and techniques with Java implementations*. Morgan Kaufman, San Francisco, CA, USA, 2000. <http://www.cs.waikato.ac.nz/ml/weka>.
- [248] David H. Wolpert. Stacked generalization. *Neural Netw.*, 5(2):241–259, 1992.
- [249] Gang Wu and Edward Y. Chang. Aligning boundary in kernel space for learning imbalanced dataset. In *ICDM*, pages 265–272. IEEE Computer Society, 2004.
- [250] Wei Wu, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Miryung Kim. Aura: a hybrid approach to identify framework evolution. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *ICSE '10: Proceedings of the 32nd International Conference on Software Engineering*, pages 325–334. ACM, 2010.
- [251] Zhenchang Xing and Eleni Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In David F. Redmiles, Thomas Ellman, and Andrea Zisman, editors, *ASE '05: Proceedings of the 20th IEEE/ACM Int'l. Conference on Automated Software Engineering*, pages 54–65. ACM, 2005.
- [252] Zhenchang Xing and Eleni Stroulia. Refactoring detection based on UMLDiff change-facts queries. In Basili and Briand [15], pages 263–274.
- [253] Hao Xiong, Haihua Yan, Zhoujun Li, and Hu Li. BUAA AntiPlagiarism: A system to detect plagiarism for C source code. In *International Conference on Computational Intelligence and Software Engineering, CISE 2009.*, pages 1–5, 2009.

- [254] Annie T. T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.
- [255] Du Zhang and Jeffrey J. P. Tsai. *Machine Learning Applications In Software Engineering (Series on Software Engineering and Knowledge Engineering)*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2005.
- [256] Xiaojin Zhu. Semi-supervised learning literature survey. Technical Report 1530, University of Wisconsin-Madison, December 2007.
- [257] Thomas Zimmermann, Sunghun Kim, Andreas Zeller, and E. James Whitehead Jr. Mining version archives for co-changed lines. In Stephan Diehl, Harald Gall, and Ahmed E. Hassan, editors, *MSR*, pages 72–75. ACM, 2006.
- [258] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In Hans van Vliet and Valérie Issarny, editors, *ESEC/SIGSOFT FSE*, pages 91–100. ACM, 2009.
- [259] Thomas Zimmermann, Nachiappan Nagappan, and Andreas Zeller. Predicting bugs from history. In Mens and Demeyer [167], pages 69–88.
- [260] Thomas Zimmermann, Peter Weissgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.
- [261] Lijie Zou. Toward an improved understanding of software change. Master's thesis, Computer Science, Waterloo University, Ontario, 2003.
- [262] Lijie Zou and Michael W. Godfrey. Detecting merging and splitting using origin analysis. In Arie van Deursen, Eleni Stroulia, and Margaret-Anne D. Storey, editors, *WCRE*, pages 146–154. IEEE Computer Society, 2003.

Appendix A

Similarity measures

In Chapter 2, methods for matching program code were discussed. This appendix describes measures for computing the similarity of program code elements. The measures are not exhaustive, but are selected because they are mentioned in this dissertation, especially in the surveys in Chapters 2 and 3.

Measures on sequences can be applied to any type of element, such as letters in a string, tokens in code, or words in a sentence. The example measures in Table A.1 are based on the strings 'nearest' and 'measure'.

The longest common substring (LCS) is the longest unbroken sequence of elements shared by two sequences. The longest common subsequence (LCSq. here, to differentiate) is the longest shared sequence which can be obtained by deleting intervening elements.

The Hamming distance is the number of differences between sequences of the same length. For two sequences s and t , the elements s_1 and t_1 , s_2 and t_2 , ... , s_n and t_n , are compared. The Hamming distance is the total number of these pairs in which the elements differ.

The Levenshtein edit distance is computed by adding 1 for each change, delete or insert operation performed to obtain a match between the two sequences. Weighted versions of this distance, termed local alignment, are also common, especially in string alignment, such as DNA sequencing, and is also used to match source code. For example, Burrows et al. [34]

Measure	Description	Example	S/D	
Longest common substring	The longest consecutive sequence of elements shared by the sequences.	nearest measure ea or re	S	2
Longest common subsequence	The longest sequence shared by the sequences which can be obtained by deleting intervening elements.	nearest measure e a r e	S	4
Hamming distance	The number of differences between two sequences of the same length.	nearest measure 1001111	D	5
Levenshtein distance	The edit distance between two sequences. (number of deletes, inserts, and changes)	nea--rest measure-- 1 change (n/m) + 4 inserts (-)	D	5
Sequence alignment	A weighted edit distance For example, change = -1, insert/delete = 0, match = 1.	nea--rest measure-- 1 change = -1 4 inserts = 0 4 matches = +4	-	3

Table A.1: Similarity(S) or distance(D) measures, based on ‘nearest’ and ‘measure’

score matches as 1, inserts or deletes -2, and mismatches (changes) -3; or Gitchell and Tran [89], allocate lower penalties to mismatched identifiers than to other tokens, reasoning that identifier renaming is a likely disguise in plagiarism. The deliberately simple example in Table A.1 scores mismatches as -1, matches +1, and inserts or deletes 0. Alignment is usually performed by considering a matrix of possible alignments produced by combining the available operations [220], or an optimised method, e.g. BLAST [4].

S.Kim et al. [130] also use the measure LCSC, longest common subsequence count. Their LCSC similarity between two strings is defined as the mean of the proportion of the longest common subsequence to each of the sequence lengths, $\frac{1}{2} \left[\frac{LCSC_{AB}}{|A|} + \frac{LCSC_{AB}}{|B|} \right]$.

Compression distance measures, of which several variants exist, are also used on sequences. The idea is that the more similar two files are, the smaller will be the distance between the size of one compressed file and the compression of a concatenation of the two files. Three variants are reproduced here, where C_m means the size of the compressed file.

1. Freire et al.'s normalised compression distance [82]

$$\frac{Cm(ab) - \min\{Cm(a), Cm(b)\}}{\max\{Cm(a), Cm(b)\}}$$
2. Chen et al.'s conditional compression distance using their compression algorithm, TokenCompress [41]

$$1 - \frac{Cm(a) - Cm(a-b)}{Cm(ab)}$$
3. Lancaster and Tetlow's compression based similarity measure [144]

$$100 \times \frac{Cm(a) + Cm(b)}{Cm(ab) + Cm(ba)} - 1$$

Measures on sets or bags which are commonly used by the approaches discussed in this dissertation are the Jaccard and Dice coefficients. The Dice coefficient is twice the size of the set intersection divided by the sum of the set sizes. The Jaccard coefficient is the size of the set intersection divided by the size of the set union. Containment, a related measure, is the proportion of a set which is shared by another. S.Kim et al. also use the measure ISC, the intersecting 'set' count on the bag of letters which form a string. They define ISC similarity as the mean of the proportion of the bag intersection to each of the bag lengths.

	Measure	Description	Example	S/D	
Sets	Dice coefficient	Set intersection size divided by the sum of the two set sizes $\frac{2 A \cap B }{ A + B }$	{a, e, r, s, n, t} {a, e, r, s, m, u} $\frac{2 \times 4}{6+6}$	S	0.67
	Jaccard coefficient	Set intersection size divided by the size of the set union $\frac{ A \cap B }{ A \cup B }$	{a, e, r, s, n, t} {a, e, r, s, m, u} $\frac{4}{8}$	S	0.5
	Jaccard distance	1 minus the Jaccard coefficient $1 - \frac{ A \cap B }{ A \cup B }$ or $\frac{ A \cup B - A \cap B }{ A \cup B }$	$1 - \frac{1}{2}$	D	0.5
	Containment (A in B)	The amount of A in B $\frac{ A \cap B }{ A }$	{a, e, r, s, n, t} {a, e, r, s, m, u} $\frac{4}{6}$	S	0.67
Bags	Intersecting 'set' count (ISC [130])	The mean of the ratios of intersection to file size $\frac{1}{2} \left[\frac{ A \cap B }{ A } + \frac{ A \cap B }{ B } \right]$	{a, e, e, r, s, n, t} {a, e, e, r, s, m, u} $\frac{5}{7}$	S	0.71

Table A.2: Similarity (S) and distance (D) measures for sets, with examples based on the sets, e.g. {a, e, n, r, s, t} or bags, e.g. {a, e, e, n, r, s, t}

Measure	Description	Example	
Manhattan distance	Sum of the differences between pairs $\sum_i a_i - b_i $	(a, e, i, l, m, r, s, t, u) (1, 0, 2, 1, 1, 1, 1, 0, 0) (1, 2, 0, 0, 1, 1, 1, 0, 1) 0+2+2+1+0+0+0+0+1	6
Euclidean distance	Root of the sum of squared differences between pairs $\sqrt{\sum_i (a_i - b_i)^2}$	(a, e, i, l, m, r, s, t, u) (1, 0, 2, 1, 1, 1, 1, 0, 0) (1, 2, 0, 0, 1, 1, 1, 0, 1) $\sqrt{(0+4+4+1+0+0+0+0+1)}$	$\sqrt{10}$
Cosine similarity	The cosine between two vectors $\frac{\sum_i (a_i \times b_i)}{\sqrt{\sum_i a_i^2} \times \sqrt{\sum_i b_i^2}}$ a.b for normalised vectors	(a, e, i, l, m, r, s, t, u) (1, 0, 2, 1, 1, 1, 1, 0, 0) (1, 2, 0, 0, 1, 1, 1, 0, 1) $\frac{1+0+0+0+1+1+1+0+0}{\sqrt{(1+4+1+1+1+1)} \times \sqrt{(1+4+1+1+1+1)}}$	$\frac{4}{9}$

Table A.3: Similarity measures for vectors. The examples are based on the (unweighted) frequency of the letters in the words ‘similar’ and ‘measure’.

Measures on frequency vectors of features in the code, such as words [6], tokens [82], n-grams [9], keywords [58], tree nodes [116], and metrics [136, 170], are also used to measure similarity. Table A.3 shows 3 examples of similarity measures on vectors, based on the frequency of the letters in the words ‘similar’ and ‘measure’. A simple distance measure for 2 vectors is the Manhattan or City Block distance, which is the sum of the absolute differences between each pair of elements in the vectors. Another measure is the Euclidean distance, the root of the sum of the squared differences between the pairs of elements. The cosine measure has a value between 0, dissimilar, and 1, identical, when the terms in the vectors are positive [6].

Weighting is used in some vector similarity measures, reflecting the relative frequency with which each term appears in the document set, the query document and the document to which it is compared. Burrows et al. [34] and Arwin and Tahaghoghi [9] test several such measures, both finding the BM25 ranking function [118, 202] most effective in plagiarism detection in the tests.

Measures on trees: As previously discussed, trees are often transformed into sequences or vectors. However, two approaches match trees more

directly. Baxter et al. [16] first try to find isomorphic subtrees, and then similar subtrees, using a similarity measure which is like the Jaccard coefficient, where the shared nodes in two trees are divided by the total number of nodes. Chawathe's minimum conforming edit script [38] is based on an idea like the weighted edit distance for sequences, but the operations are insert, delete, move, update and align. Fluri et al.'s minimal tree edit script [77], is an adaptation of Chawathe's algorithm, to make it more suitable for code.

Measures on graphs: Graphs are matched in similar ways to trees, either matching subgraphs directly, or by transforming the graphs into another representation. Krinke [139] matches program dependency subgraphs for clone detection, as do Liu et al. [151] to detect plagiarism. Gabel et al. [87] transform a program dependency graph into vectors for matching.

Appendix B

Classifiers

This appendix provides further information about classifier ensembles, in particular those with the best results for the datasets in this research. The motivation for combining classifiers is explained, along with methods of combining them, and the ways in which diversity is introduced into classifier ensembles are analysed.

B.1 Ensembles

Dietterich [62, p.3] states that *“Learning algorithms that output only a single hypothesis suffer from three problems that can be partly overcome by ensemble methods: the statistical problem, the computational problem, and the representational problem.”* These problems are caused respectively by difficulty in selecting which of many equally suitable hypotheses are appropriate; difficulty in finding the best hypothesis where local minima exist; and the lack of a suitable function within the hypothesis space. In each case, a combination of hypotheses can improve the outcome of the learning system. Kuncheva [141] notes that although an ensemble may not improve on the performance of a single classifier, experiments with ensembles have proved their worth.

There are several ways to combine the classifiers in an ensemble. The simplest is a vote-based system, using, for example, majority, weighted or

Algorithm	Reference	Data	Features	Parameters	Classifiers
AdaBoost	[84]	✓			
Bagging	[28]	✓			
Dagging	[234]	✓			
Decorate	[166]	✓			
EnsembleSelection	[36]				✓
Grading	[212]				✓
LogitBoost	[85]	✓			
MultiBoostAB	[243]	✓			
RacedIncrementalLogitBoost	[80]	✓			
Random Forest	[29]	✓	✓		
RandomCommittee	-			✓	
RandomSubSpace	[109]		✓		
Rotation Forest	[203]	✓	✓		
Stacking	[211, 235, 248]				✓
Voting	[132, 141]				✓

Table B.1: Methods for introducing diversity in ensemble learners.

thresholded votes [132, 141]. These may be applied to absolute or probable classifications from the ensemble members. Another way to combine the classifiers is to use a learning algorithm to combine the outcomes of the individual classifiers to best effect, for example, by Stacking [211, 235, 248]. The learning algorithms may combine the outputs from all of the classifiers, or select one or more which are most effective in the region of the newly presented instance by Grading [212].

Without diversity among the classifiers in an ensemble, there is unlikely to be any advantage over a single classifier. Diversity can be introduced in several ways: by combining a selection of different classifiers (heterogenous), or, when building ensembles from one type of classifier (homogenous), the training data, the features or the parameters can be varied. Table B.1 notes the aspect(s) varied in 15 classifier ensemble algorithms.

Other meta-learners provided by Weka fall into two main groups; those which seek to optimise in some respect and those which manipulate data. The first group aim to optimise either a specific measure, for example f-measure; or take account of a user-defined cost function which can be useful, for example, in medical diagnosis, where it is important not to miss a positive case. The second group allows data to be used with classifiers which would not otherwise be suitable, for example, multiclass problems can be structured to use binary class classifiers.

B.2 Selected algorithms

Three classifiers have proved more successful than others in this research: Rotation Forest [203], Random Forest [29] and Simple Logistic [145, 227].

Random Forest [29] uses bagged data sets to build an ensemble of random decision trees, which are combined by voting. Bagging [28] creates varied datasets, usually of the same size as the full dataset, by randomly sampling with replacement. Random Trees are constructed by considering a random selection of features at each node.

The Rotation Forest [203] algorithm randomly selects a subset of the features, performing Principal Component Analysis (PCA) to give linear combinations of the selected features rotated around the original axes. The principal components are then used to build a tree and the process repeated to create a forest.

Simple Logistic [227] models are built step by step. At each step, LogitBoost is used to optimise a simple logistic regression function for each of the features, and the feature giving the smallest squared error is added. To reduce complexity and prevent overfitting of the models, features are added only if the performance on unseen examples in cross-validation is improved. This method naturally results in feature selection.

Appendix C

Machine learning in software engineering

Machine learning appears not to have been used in origin analysis before, although, as discussed in Chapter 3, S.Kim et al. [130] use a statistical approach, along with exhaustive search, to combine measures in their system. This section provides a brief review of the development of machine learning in the more general area of software engineering. A selection of software engineering applications using machine learning, in particular those related to software evolution, are analysed in Table C.1 with the aim of ascertaining whether there are pointers to suitable features, matching techniques, or machine learning algorithms for use with software code.

In 2005, Zhang and Tsai published a survey on the use of machine learning in software engineering [255] covering the period from the mid-1980s to 2002. This survey shows that most applications at the time were aimed at predicting either aspects of development, such as cost and productivity; maintenance effort; or aspects of quality, such as defect prediction or testability. These areas of research were in large part due to the public availability of the NASA software defect and effort datasets [190], which are now available at the Promise repository [169], along with other datasets contributed by researchers in the field.

Automated search in software engineering, known as search-based soft-

ware engineering (SBSE), is a growing area of research. This encompasses applications using evolutionary methods of machine learning in many areas of software engineering. A list of SBSE applications are held at the repository maintained by the research group at University College London [53].

Open source software repositories have become accessible in recent years, providing a large amount of data for analysis. Kagdi et al. [123] surveyed work in mining software repositories to discover patterns in evolving software (up to August 2006). Their survey identifies ten areas of research, in many of which machine learning techniques have been used [p.15].

The 2011 survey by Halkidi et al. [100] places software engineering tasks which use data mining and machine learning into six categories: requirements elicitation and tracing, development analysis, testing, debugging, maintenance, and software reuse, showing that machine learning is now used widely in software engineering.

Table C.1, on page 365, analyses a selection of work in the area. The selection includes applications concerned with changes to code, but mainly focuses on classification tasks, particularly those using multiple versions of evolving software. These examples are based on static analysis of code, or of associated artefacts. There also exists a large body of work, not included here, based on dynamic analysis, especially in the areas of testing and usage patterns.

The first three columns in the table identify the paper by author, reference and date. The upper part of the table shows work based on evolving software, a C in the fourth column means that the intervals between versions studied are either commit or transaction level, R means release level. Work based on a single version of a project is in the lower part of the table and is marked “-”. The entries are sorted by date within the upper and lower parts of the table.

Halkidi et al. [100] categorises machine learning and data mining tasks in software engineering into: classification(C), clustering(G) and pattern mining(P). The research in the table has been labelled with these categories in the column headed “C/G/P”. Brief descriptions of the algorithm(s), the

features, and the purpose of the investigation are provided in columns 5-7.

Change related tasks include predicting future refactorings from the change history of a project [197], categorising change types, finding patterns of change types by clustering [76, 216], and finding co-change patterns [88, 217, 254, 257, 260]. Another popular class of applications mines patterns in software to find exceptions, and so to alert the user to inconsistencies when making changes [1, 149, 150, 153, 195, 196, 232, 233]. Many classification tasks relate to predicting defects. For example, predicting post-inspection defects [185], or defect density [133], investigating whether a module's complexity metrics can be used to predict post-release defects [178, 186], assessing a bug's severity from its report wording [54, 142], and classifying bug-introducing changes [10, 129, 218, 219, 259].

In summary, in around half of the classification tasks shown, a number of different algorithms are applied to the data to find that best suited to the task. Also, the large majority of approaches to classification surveyed use a broad range of features taken from more than one source.

Author	Ref.	Yr.	C/R	Algorithm	C/G/P	Features	Purpose	
Bouktif et al.	[26]	02	R	DT (C4.5), AdaBoost, GA	C	O-O software metrics	Predict the stability of classes	
Ying et al.	[254]	04	C	Frequent pattern mining	P	Files in a transaction	Find co-changing files, esp. without obvious link	
Knab et al.	[133]	06	R	DT (J48)	C	Counts of LOC, variables, calls, references, functions	Predict defect density in files	
Pan et al.	[186]	06	C	Bayes Network	C	Slice-based metrics and standard software metrics	Compare metrics in buggy file & function prediction	
Zimmermann et al.	[257]	06	C	Frequent itemset mining	P	Individual lines of code which change	Find co-changed line groups co-changed lines	
Aversano et al.	[10]	07	C	SVM, C4.5, KNN(10), AdaBoost, Simple Logistic	C	Vector of tf-idf of changes to alphanumericers in source text	Predict bug-introducing changes	
Gürba et al.	[88]	07	C,R	Formal concept analysis	P	Change in software metrics (eg complexity, no.methods)	Find co-change patterns	
Ratzinger et al.	[197]	07	C	J48, LMT, JRip, Ninge	C	Numerous, mixed sources	Predict refactoring activity from past	
Buse and Weimer	[35]	08	R	Several inc. Bayes, VF1, NN	C	Text-based e.g. line length, identifier frequency, branches	Develop a readability metric, relate to bugs, changes	
Fluri et al.	[76]	08	C	Clustering on cosine distance	G	Change types e.g. statement insert, parameter delete	Find change patterns, link to development activities	
S.Kim et al.	[129]	08	C	SVM (Weka)	C	Numerous, mixed sources	Find bug-introducing changes	
Shivaji et al. (based on [129])	[218]	09	C	SVM, Naive Bayes, J48, JRip	C	Numerous, mixed sources	Using feature selection to improve bug prediction from changes	
Table C.1.	Machine learning and data mining in software engineering tasks							Continued on next page

Author	Ref.	Yr.	C/R	Algorithm	C/G/P	Features	Purpose
Hindle et al.	[108]	09	C	J48, NB, Kstar, SMO, IBk, JRip	C	Metadata: words in commit messages, authors, file types	Categorise changes eg defective, corrective
Kosker et al.	[138]	09	R	Weighted Naïve Bayes	C	Object-oriented software metrics	Find classes suitable for refactoring
Zimmermann et al.	[258]	09	R	Logistic regression, decision tree	C	Metrics for defect prediction, other for project similarity	Investigate cross-project defect prediction
Shi et al.	[216]	11	C	Clustering - heuristic rules	G	Words from modified lines in API docs	Categorise types of change

Regolin et al.	[198]	03	-	GP and NN	C	Function points and component sizes	Estimate the number of lines of code
Shirabad et al.	[217]	03	-	Decision tree	C	Bag of words, shared calls, references	Find co-updated files
Nagappan et al.	[178]	06	-	Regression on PCA	C	Metrics	Predict post-release failure of modules
Tan et al.	[230]	07	-	Rule mining, clustering, decision tree	CGP	NLP on comments, topic-groups	Tie comments to code, find rules and exceptions
Lamkarfi et al.	[142]	10	-	Naïve Bayes	C	Words in bug reports	Predict bug severity from report content

Table C.1: A selection of software evolution related tasks in which machine and data mining are used.

Column 4 shows intervals between source code versions, commit level (C) or release level (R), or from a single version (-).

A brief description of the algorithm(s), features, and purpose of the task are in columns 5, 7 & 8.

Column 6 (C/G/P) shows whether the task is one of classification (C), clustering (G) or pattern mining (P).

Appendix D

Additional file comparison visualisations

This appendix supplements Chapter 5, showing visualisations produced by a selection of other tools referenced in this dissertation. First, the similarity between files output by three clone detection tools, in particular, Code Clone Finder which is used in creating features for the machine learning application described in Chapter 10. Second, examples of visualisations of the movement of code in a system found by other origin analysis tools.

D.1 Clone detection tools

Clone detection tools are not set up to find relative similarity between files but rather to find fragments of code which recur throughout a file or a system. Their initial reporting therefore differs from plagiarism detection tool output. Rather than the similarity between files, the tools report on clone classes. A small sample of their outputs are displayed in Figure D.1, the two on the left giving information about more than two files, one as a list and one graphically.

Figures D.1a and D.1b show clone classes reported by NiCad [50] and CloneDetective [121] respectively. NiCad provides HTML output and shows the location and content of clone classes. CloneDetective shows

D.2 Origin analysis tools

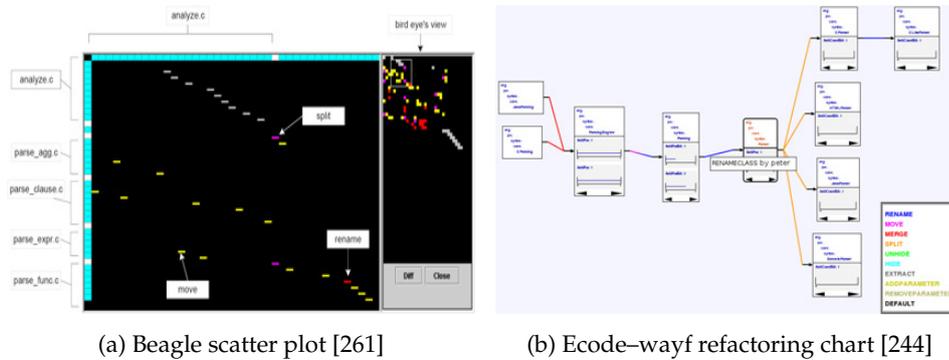


Figure D.2: Beagle scatter plot (left), where the dashes represent functions, the colours of which show whether they have been moved, merged or split. Weißgerber’s Ecode-wayf (right) which shows the evolution of a class, colouring the connectors to show types of refactoring.

Two examples of origin analysis tools which colour-code changes to software entities are included here. The University of Waterloo’s tool Beagle [261] has a scatter plot to show the changes to functions between versions (Figure D.2a). The dashes on the plot are coloured to show whether they have been moved, renamed, split or merged. Weißgerber and Schafer’s Ecode-wayf [244] hierarchy graph in Figure D.2b, shows the evolution of a Java class, with the connectors coloured to indicate the type of refactoring undertaken between transactions.

Appendix E

Ferret: example calculation

To illustrate how Ferret computes a similarity score, the files `fact.c` and `power.c`, shown together in Table E.1, are compared. The trigrams for the two files are presented in Table E.2. There are 36 trigrams in total in `fact.c`, one of these, “= 1 ;”, labelled 10, is duplicated, making 35 distinct trigrams in the file. `Power.c` has 39 trigrams, with one duplicate, making 38 distinct trigrams. The files share 29 trigrams, so that 6 in `fact.c` and 9 in `power.c` are not shared. There are therefore $29 + 6 + 9 = 44$ distinct trigrams in the two files. The Jaccard coefficient, or similarity score, for the two files is $\frac{29}{44} = 0.659$.

<pre>//fact.c long factorial (int n) { long result = 1; int i; for (i = 1; i ≤ n; i++) result *= i; return (result); }</pre>	<pre>// power.c long power (int base, int n) { long result = 1; int i; for (i = 1; i ≤ n; i++) result *= base; return (result); }</pre>
--	---

Table E.1: Code for `fact.c` and `power.c`.

1	long	factorial	(13	int	i	;	24	;	i	++
2	factorial	(int	14	i	;	for	25	i	++)
3	(int	n	15	;	for	(26	++)	result
4	int	n)	16	for	(i	27)	result	*=
5	n)	{	17	(i	=	28	result	*=	i
6)	{	long	18	i	=	1	29	*=	i	;
7	{	long	result	19	=	1	;	30	i	;	return
8	long	result	=	20	1	;	i	31	;	return	(
9	result	=	1	21	;	i	≤	32	return	(result
10	=	1	;	22	i	≤	n	33	(result)
11	1	;	int	23	≤	n	;	34	result)	;
12	;	int	i	23	n	;	i	35)	;	}
i	long	power	(11	1	;	int	23	n	;	i
ii	power	(int	12	;	int	i	24	;	i	++
iii	(int	base	13	int	i	;	25	i	++)
iv	int	base	,	14	i	;	for	26	++)	result
v	base	,	int	15	;	for	(27)	result	*=
vi	,	int	n	16	for	(i	vii	result	*=	base
4	int	n)	17	(i	=	viii	*=	base	;
5	n)	{	18	i	=	1	ix	base	;	return
6)	{	long	19	=	1	;	31	;	return	(
7	{	long	result	20	1	;	i	32	return	(result
8	long	result	=	21	;	i	≤	33	(result)
9	result	=	1	22	i	≤	n	34	result)	;
10	=	1	;	22	≤	n	;	35)	;	}

Table E.2: The 35 distinct trigrams in fact.c (top) and 38 in power.c. In each file “= 1 ;”, no. 10, is duplicated. Power.c trigrams matching those in fact.c have the same numbers, the remaining 9 trigrams have Roman numerals.

Appendix F

Density tool prototype

This appendix shows the input and output screens of the density tool introduced in Section 6.6.¹ The input screen, see Figure F.1, allows the user to choose the input XML file (previously output from a Ferret comparison between two files), the density, the minimum number of matched units in a dense block, the gap size, whether to base the calculations on tokens, words, or characters and how to prioritise the blocks - by density, matched units, or total units. Output from the tool is in two forms: either the interface exemplified in Figures F.2 and F.3, or the XML output shown in Figure F.4.

In Figure F.2 the dense blocks have been found in the comparison between two very large text files (a quarter, and a third of a million words) taken from the database of source and suspicious documents created for the PAN'09 (Plagiarism ANalysis 2009) competition.² In the upper part of the screen, statistics for the two files, such as size in words, the number of blocks found, and the matched and unmatched words, are displayed on either side. In this example there are 71 blocks in file 1 and 25 in file 2. The parameters chosen for the analysis are shown in the centre. In the lower part of the screen, the blocks and their individual statistics, location, density, number of words, both matched and unmatched, are displayed on

¹A prototype of the tool is available from <http://homepages.stca.herts.ac.uk/~gp2ag/density.html>

²<http://www.uni-weimar.de/medien/webis/research/workshopseries/pan-09/competition.html#corpus>

either side. The centre panel shows the text in the block selected from one of the side panels, with matched trigrams highlighted in blue text against the unmatched text in grey. Figure F.3 shows detail from a similar comparison between two other files.

Figure F.4 shows the XML output containing information about the dense blocks found in two other large files from the PAN'09 collection. This example has been chosen because the output is short and therefore fits into a screenshot. In this case, the files have around a quarter of a million words each and share over 9200 trigrams, however, there is only one small dense block of 37 words. The matched trigrams are highlighted in bold purple text, with unmatched text in grey.

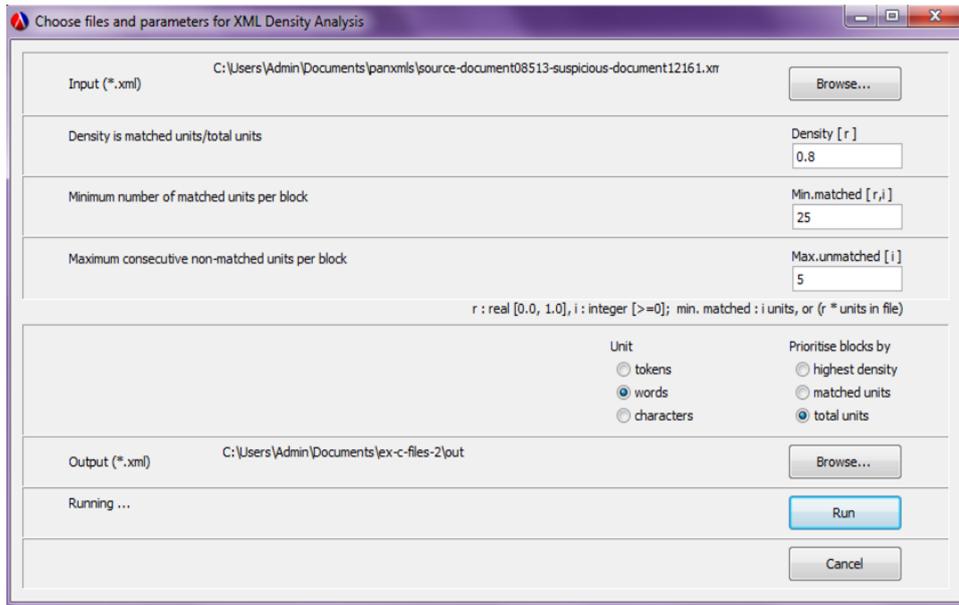


Figure F.1: Density tool input screen, which allows the user to choose the minimum block density; the minimum number of matched units in a dense block, e.g. 25 (or can be expressed as a proportion of file size, e.g. 0.1); the maximum number of consecutive unmatched units in the block; choice of unit; and how to sort the blocks.

Density analysis of UH-Ferret file: C:\Users\Admin\Documents\panwin\source-document\08513-suspicious-document\2161.xml

File 1:-

Units (tokens/words/characters) in file: 242213
 Matched and unmatched sections in file: 36660
 Number of dense blocks in file: 71
 Total units in dense blocks: 105633
 Covered by matched trigrams: 99337
 Not covered by matched trigrams: 6296

Parameters :-

Units: Words
 Density: 0.8
 Min. matched units per block: 100
 Maximum unmatched gap: 10
 Block selection on most: dense block

SEARCH, ANALYSIS OF COMPANIES, SPONSORED BY SMART CORPORATION, 2011
 IN: ANALYSIS OF COMPANIES, SPONSORED BY SMART CORPORATION, 2011.

File 2:-

Units (tokens/words/characters) in file: 353720
 Matched and unmatched sections in file: 37169
 Number of dense blocks in file: 23
 Total units in dense blocks: 91893
 Covered by matched trigrams: 88091
 Not covered by matched trigrams: 3802

[Change parameters](#)

File 1 blocks: click to see text.

Block: 40

Starting at unit: 99110
 Block density: 0.94
 Total units in block: 1889
 Matched units in block: 1760
 Unmatched units in block: 99

Block: 41

Starting at unit: 100884
 Block density: 0.94
 Total units in block: 1488
 Matched units in block: 1429
 Unmatched units in block: 59

Block: 42

Starting at unit: 100797
 Block density: 0.915
 Total units in block: 732
 Matched units in block: 183

Block: 43

Starting at unit: 102827
 Block density: 0.97
 Total units in block: 3824
 Matched units in block: 93

Block: 44

At seven o'clock the calm was succeeded by a gale breeze from N.E. as far as we could see; so that we began to reckon what time we should reach the Sound the next day, but at nine the wind shifted to its old quarter N.W., and blew a fresh gale, with which we stretched to the S.W. under angle-reefed topsails and courses, with the Adventure in company. She was seen until midnight, at which time she was two or three miles astern, and presently after disappeared. We supposed she had tacked and stood to the N.E., by which manœuvre we lost sight of her.

We continued to stretch to the westward with the wind at N.N.W., which increased in such a manner as to bring us under our two courses, after spitting a new main-top-sail. At noon Cape Horn was sighted, and the Adventure tacked to the northward, and the Adventure began to abate and to veer more to the north, so that we fell in with the land, under the Snowy Mountains, about four or five leagues to windward of the Lookers on, where there was the appearance of a large bay. I now regretted the loss of the Adventure; for had she been with me, I should have given up all thoughts of going to Queen Charlotte's Sound; in wood and water, and have sought for a place to these islands farther south, as the wind was now westerly, and the Adventure's manœuvre made it necessary for me to repair to the Sound, that being the place of rendezvous.

As we approached the land, we saw smoke in several places along the shore; a sure sign that the coast was inhabited. Our soundings were from forty-seven to twenty-five fathoms, that is, from the distance of one mile, where we tacked, and stood to the eastward, under the two courses and close-reefed top-sails, but the latter we could not carry long before we were obliged to hand them. We continued to stand to the eastward all night, in hopes of meeting with the Adventure in the morning.

Seeing nothing of her then, we wore and brought to, under the fore-sail and main-stay-sail the wind having increased to a perfect storm; but we had not been long in this situation before it abated, so as to permit us to carry the two courses, under which we stood to the west; and at noon the Snowy Mountains bore W.N.W., distant twelve or fourteen leagues. At six o'clock after it began to blow with redoubled force, and obliged us to lie under the main-stay-sail, in which situation we continued till midnight, when the storm lessened and two hours after it fell calm.

On the 1st of November at four o'clock in the morning, the calm was succeeded by a breeze from the west. The Adventure was not seen, and we were obliged to stand to the westward, which gave us hopes that the N.W. winds were done; for it must be observed, that they were attended with clear and fair weather. We were not wanting in taking

File 2 blocks: click to see text.

Block: 1

Starting at unit: 54374
 Block density: 0.89
 Total units in block: 4723
 Matched units in block: 4206
 Unmatched units in block: 517

Block: 2

Starting at unit: 154014
 Block density: 0.92
 Total units in block: 3897
 Matched units in block: 3548
 Unmatched units in block: 349

Block: 3

Starting at unit: 132230
 Block density: 0.92
 Total units in block: 3979
 Matched units in block: 3694
 Unmatched units in block: 285

Block: 4

Starting at unit: 137082
 Block density: 0.99
 Total units in block: 2849
 Matched units in block: 25

Block: 5

Figure F.2: Density tool output screen

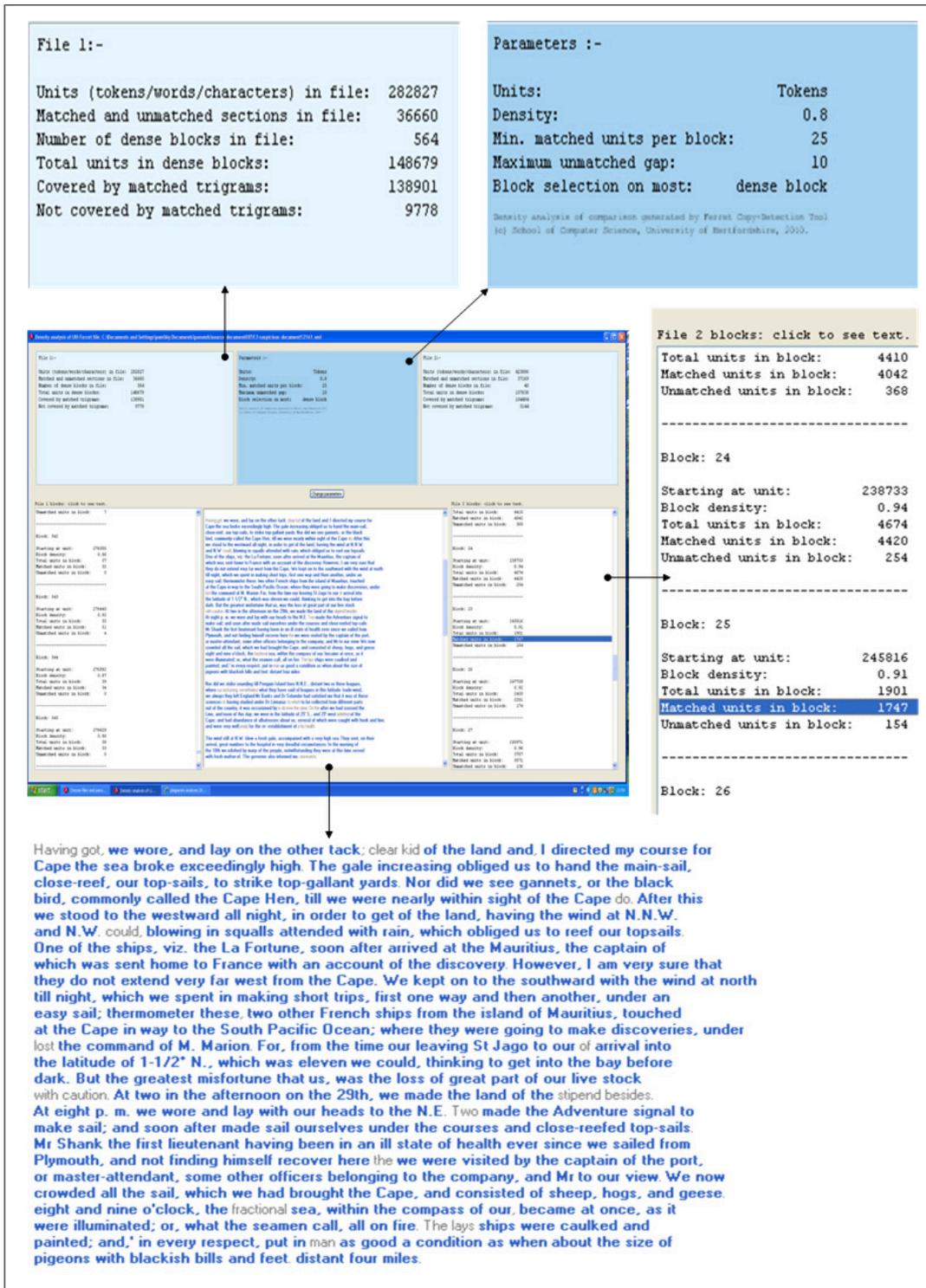


Figure F.3: The density analysis tool screen and detail from it, clockwise from top left: file details; parameters used; a list of the dense blocks found in file 2; and text from the selected block, with matched trigrams in blue. File 2 statistics and the file 1 block list mirror those of the other file.

UH-Ferret: Density Analysis of ".../source08513-suspicious07113.xml"	
Parameters:-	
Units	Tokens
Density	0.8
Minimum copied units in a block	30
Maximum non-copied gap	10
Density analysis of comparison generated by the Ferret Copy-Detection Tool, (c) School of Computer Science, University of Hertfordshire, 2010.	
File 1 :	
Total units in file:	282808
Total sections in file:	37674
Number of dense blocks in file:	1
Units in dense blocks:	
Covered by matched trigrams	30
Not covered	7
Total	37
Block starting at unit: 126763	
Starting at section:	16597
Start unit:	126763
Units covered by matched trigrams:	30
Total units in the block:	37
Block density:	0.81
Text (matched text is purple):	
<pre> from him: and then he put it off with a laugh, acting his part with so much address, that it was hardly possible for me to be angry with him: so that we </pre>	
File 2 :	
Total units in file:	221057
Total sections in file:	23854
Number of dense blocks in file:	0
Units in dense blocks:	
Covered by matched trigrams	0
Not covered	0
Total	0

Figure F.4: Example of the XML file output by the Ferret density analysis tool. This shows the results of analysing a comparison between two very large text files, where only one small block of 2 lines is found.

Appendix G

File pair ranks from Chapter 9

	Ratios				Counts			Ratios				Counts	
	Fer ret	Ex. P	Ex. O	Ex. PO	Wt. 2	Wt. 4		Fer ret	Ex. P	Ex. O	Ex. PO	Wt. 2	Wt. 4
3, 7		17				19	19, 27			5	5		
3, 15	3	3	17	19		13	19, 36			3			
3, 20			18	16			19, 43			1	1		
3, 25	2	4	11	8	11	7	19, 46	8					
3, 56		20					19, 54	1					
4, 44					20		21, 28			14	14		
5, 13					8	9	22, 61	7					
5, 52					4	2	23, 41			20	17	9	14
7, 15		19					24, 30		16	6	3	6	5
7, 25		12	16	13		11	24, 40	6	11	8	6	10	10
8, 17		14	2	2	1	1	26, 52					19	
8, 23					16		27, 28	20	10	9			
8, 25					18		27, 53	17					
8, 63				20	5	4	27, 61	11					
10, 29		9					29, 42	18	13				
10, 42	16	7					29, 58		8				
10, 58		2	13	9			30, 40	19		15	11		16
13, 49					7	8	32, 33			4		3	6
13, 52					14		34, 53	13					
13, 56		18					36, 43	15					
14, 22							41, 58			19			
14, 53	14						42, 58		6		12		
14, 61	10						43, 46			10	7		
15, 25	5	5				12	47, 48			12	10	12	18
15, 56	4	1					49, 52					17	20
17, 34				15			51, 63				18	2	3
17, 63					15	15	53, 61	12					
							57, 60	9	15	7	4	13	17

Table G.1: The top 20 similarities determined by the different measures shown in Table 9.1. Column 1 gives the pairs of student projects which have high similarity scores under one or more measures, the remaining columns indicate for which measure(s) this is so, ranked from 1–20, top down.

Appendix H

Details of the 89 projects

The eighty-nine projects from which the split and disappearing file datasets were collected are introduced in Chapter 12. The tables in this appendix give details of these projects. The first, Table H.1, shows the project names and purposes, while Table H.2 shows statistics relating to the size of the projects, which are, ordered by column:

1. The number of releases in the project.
2. The size in Mb of all of the code in the project.
3. The total number of files in the project.
4. The amount of C code in KLOC in the project.
5. The minimum number of KLOC of C code,
6. and the maximum in one release.
7. The size of the stripped C code in the project in Mb.,
8. and the number of C code files.

Project	Purpose
acidblood	an IRC robot
aqtwo-tng	Action Quake2 variant
artoolkit	captures images from video sources, optically tracks markers in the images
barcode	barcode printer
beecrypt	cryptography libraries
biew	file viewer with built-in editor for binary, hexadecimal and disassembler modes
bitcollider	generates bitprints and metadata tags from files for Bitzi metadata project
cipe-linux	encrypts IP over UDP tunneling, can be used to build a range of VPN solutions
dbacl	general purpose digramic Bayesian text classifier
diald	link management tool able to control dial-on-demand network connections
drivel	GNOME client for working with blogs
dt3155a	version 1.8 of the Linux DT3155 device driver, for kernels 2.2, 2.4 and 2.6
dynamics	scalable, dynamical, and hierarchical mobile IP software for Linux
effectv	real-time video effector
etherape	graphical network monitor like etherman, displaying network activity graphically
extace	3D audio visualization tool
felt	finite element analysis, primarily for mechanical problems
fidogate	Fido-Internet gateway, provides Fido utilities
fobbit	uses the Creative Labs USB VOIP Blaster on NetBSD, Linux, and Windows
ganc	gnome2 based algebraic calculator
gema	general purpose text processing utility based on pattern matching
giw	gtk widgets for scientific/instrumentation and visualization
gnochm	CHM file viewer for Gnome2, using a set of Python wrappers around libchm
gpmudmon	battery monitor for Linux PPC similar to Batmon but running as a Gnome applet
gpsbabel	reads, writes, and manipulates GPS waypoints in a variety of formats
gwyddion	multiplatform scanning probe microscope, data visualization and analysis tool
hardsid	driver for the HardSID cards for Linux and other free operating systems
hatari	Atari ST emulator for Linux, designed for running old ST games and demos
heme	portable console hex editor for unix operating systems
hptalx	program to communicate with an HP calculator, uses GTK for GUI
interest	financial management system for personal investments
ipcop	Linux firewall distro, geared towards home and SOHO users
jack-rack	LADSPA effects rack for the JACK low latency audio API
judy	general purpose dynamic array implemented as a C callable library
lde	disk editor for linux, originally written to help recover deleted files
lejos	for lego robot programming in java for LEGO Mindstorms RCX and NXT bricks
lgeneral	turn-based strategy game inspired by the classic Panzer General
libbt	C implementation of the BitTorrent core protocols
liflines	genealogy software
lirc	supports receiving and sending IR signals of common IR remote controls
log4c	for flexible logging to files and elsewhere in time-space critical environments
mfstools	set of utilities for a TiVo
mjs	console MP3 Jukebox System
mkdrec	CD-ROM recovery tool, makes a bootable (El Torito) disaster recovery image
continued ...	

The open source projects used in this research. Page 1 of 2.

Project	Purpose
... continued	
motif-pstree	(xps) dynamically displays Unix processes as a tree or forest
mplayerplug-in	plugin for Mozilla that uses mplayer to play embedded media
mpop	POP3 client
msmtp	SMTP client
multignometermin	version of gnome-terminal, with new features and extensions
nano	GNU nano is a GPLed clone of the Pico text editor
nap	text-based Napster client for Linux
newstar	transfers Usenet articles between local and remote servers using NNTP transport
noffle	news server optimized for low speed dial-up connections to the Internet
nptltraceool	mechanism to unobtrusively trace the NPTL Library in dynamics
nvrwakeup	reads and writes the WakeUp time in the BIOS
oww	one-wire weather, for Dallas Semiconductor / AAG 1-wire weather station kits
pam-mysql	allows PAM aware applications to authenticate users through MySQL database
pbbuttons	supports laptop functions, eg power management, hotkeys, battery supervisor
pidgin-hotkeys	pidgin plugin for global hotkey assignment in list and message management
pio	emulation of the board game The Settlers of Catan
poptop	PPTP server
ppcboot	Embedded PowerPC Bootloader Project, esp. for Embedded PowerPC boards
premake	project configuration scripting tool
pxlib	enables reading and writing of Paradox database and primary index files
rcalc	symbolic calculator for the GNOME desktop environment
rembowiz	network based PC disk image management, based on Rembo Toolkit
rio500	utilities for Diamonds Rio 500 digital audio player under Linux
rsyslog	syslog support utilities
rtnet	real-time network protocol stack for Linux extensions Xenomai and RTAI
seti-applet	displays the status of any seti@home client in a small GNOME panel applet
sf-xpaint	paint program for X, suitable for producing simple graphics
sonasound	sonogram and waveform display program displays musical features
sortmail	processes incoming email, classifying and processing accordingly
sphinx2	speech recognition system
sphinx3	real-time speech recognition system
suparun	(OBLISK) system for GNU/Linux to run binary packages on most systems
toxine	scriptable, interactive text UI program using xine-lib
tulip	Linux 2.4.x kernel driver for the Tulip series of ethernet chips
tuxnes	emulator for the 8-bit NES that runs under Linux and FreeBSD
vacm	virtual camera - creates a virtual representation of recorded scenes
wmweatherplus	will download the National Weather Service METAR bulletins
wx200d	data collector and server daemon for weather station hardware
xastir	real-time tracking of stations via radio/internet APRS data streams
xawdecode	XdTV allows you to watch, record and stream TV
xbae	Xbae set consists of the XbaeMatrix, Caption and XbaeInput widgets
xmp	extended module player for Unix-like systems playing over 80 module formats
yplot	scientific graphics plotting package implemented as a Yorick interface to PLplot
ysmv7	ICQ client based on v7/v8 protocol version, no external libraries required
zimg	generates png / jpeg images from arbitrary formatted 2-D ascii or binary data

Table H.1: The open source projects used in this research. Page 2 of 2

	All code			Stripped C code				
	No.of Rel's.	Size Mb	No.of files	Project KLOC	Min KLOC	Max KLOC	Size Mb	No.of files
acidblood	4	1.0	341	19	3	5	0.5	100
aqtwo-tng	7	31.2	2649	322	35	42	9.8	529
artoolkit	9	42.9	5127	256	20	28	8.3	1247
barcode	9	420.3	4980	28	1	4	0.6	126
beecrypt	4	7.4	967	2321	86	205	1.7	532
biew	8	1500.0	7842	499	48	54	14.5	2361
bitcollider	6	9.0	683	50	5	11	1.3	291
cipe-linux	12	4.5	961	70	4	7	2.0	397
dbacl	10	217.1	3013	190	12	23	4.3	287
diald	9	7.0	1662	82	8	10	4.5	313
drivel	6	5.7	499	31	1	8	1.0	115
dt3155a	4	1.5	309	20	4	4	0.6	114
dynamics	25	44.4	8778	878	20	47	23.6	3326
effectv	17	6.0	1579	175	3	14	3.4	857
etherape	88	115.9	13858	618	1	17	17.3	2497
extace	8	15.4	489	75	6	13	2.2	432
felt	3	32.7	2895	380	84	100	10.6	1247
fidogate	35	249.5	38540	854	21	26	37.2	3238
fobbit	3	1.4	220	30	7	12	0.7	72
ganc	6	1.9	361	18	2	3	0.5	256
gema	4	1.7	304	32	6	7	0.7	85
giw	3	11.3	730	40	10	10	2.8	178
gnochm	14	15.3	1101	1	<1	<1	0.1	26
gpmudmon	10	2.0	627	16	<1	1	0.6	57
gpsbabel	8	119.7	11980	862	82	113	68.1	2996
gwyddion	29	153.9	23892	2486	3	130	94.6	8546
hardsid	10	1.5	324	10	1	2	0.5	25
hatari	9	19.4	2663	398	29	50	10.6	1709
heme	3	0.3	64	7	1	2	0.2	36
hptalx	4	6.0	286	29	6	6	2.2	105
interest	56	37.8	7132	709	9	27	20.4	3975
ipcop	22	203.2	24294	402	5	19	20.6	1801
jack-rack	7	4.5	730	51	4	7	2.2	302
judy	5	33.1	645	134	27	27	9.0	244
lde	11	5.6	829	82	3	11	1.9	413
lejos	9	23.6	7270	77	3	12	5.3	588
lgeneral	11	44.3	4312	322	22	33	14.8	1274
libbt	7	6.1	796	54	5	8	1.3	261
lifelines	15	99.7	7495	767	24	62	19.3	3153
lirc	47	86.9	23619	1664	9	47	37.4	4562
log4c	8	5.3	1453	73	4	12	2.5	621
mfstools	9	2.7	723	59	2	9	1.7	204
mjs	9	1.8	521	38	4	5	1.0	300
mksdrec	24	123.5	17306	1482	1	75	31.8	6319

	All code			Stripped C code				
	No.of Rel's.	Size Mb	No.of files	Project KLOC	Min KLOC	Max KLOC	Size Mb	No.of files
motif-pstree	3	4.7	854	31	8	8	1.0	288
mplayerplug-in	12	36.7	4340	68	1	2	4.6	405
mpop	37	26.0	3940	311	14	20	5.1	1368
msmtp	18	17.8	2686	195	12	19	7.2	823
multignometerm	17	46.4	7621	378	13	31	33.1	458
nano	38	42.6	3545	344	7	14	8.5	1138
nap	13	11.0	1024	232	15	19	5.3	544
newstar	10	9.9	1606	90	8	8	2.4	648
noffle	7	4.5	847	71	8	11	3.5	398
nptltraceool	2	8.1	967	44	5	12	3.1	463
nvrnm-wakeup	10	3.7	969	30	<1	6	2.6	117
oww	29	69.9	4812	515	9	24	14.7	2997
pam-mysql	13	1.4	415	22	<1	2	0.7	113
pbbUTTONS	10	13.7	2747	125	7	20	4.1	683
pidgin-hotkeys	8	1.4	467	12	1	5	0.4	57
pio	9	29.9	4245	272	17	30	12.8	1094
poptop	8	8.9	1222	51	4	6	1.8	268
ppcboot	9	89.5	11920	2321	7	200	49.4	6770
premake	4	3.5	689	56	8	13	1.6	324
pxlib	38	26.9	4760	78	1	7	3.5	658
rcalc	14	14.2	2726	61	2	5	1.8	587
rembowiz	10	7.1	1055	99	4	12	5.8	243
rio500	4	2.7	591	35	6	8	1.0	131
rsyslog	52	79.6	11730	974	11	29	25.7	3915
rtnet	9	32.7	3833	352	21	41	11.0	1418
seti-applet	9	14.4	1667	56	4	7	1.8	317
sf-xpaint	12	21.4	3114	487	27	64	12.8	1499
sonasound	31	31.3	3981	275	5	10	7.8	1738
sortmail	5	1.5	232	36	6	6	0.8	138
sphinx2	6	166.3	2380	1341	204	224	11.8	1021
sphinx3	6	262.4	3532	481	28	163	10.7	1123
suparun	10	5.1	675	11	1	2	0.4	127
toxine	6	7.0	938	142	10	25	3.7	446
tulip	10	3.0	409	66	5	6	1.9	130
tuxnes	4	3.9	330	80	13	18	2.5	198
vacm	9	119.9	6309	559	51	64	16.9	1933
wmweatherplus	12	4.1	1256	88	6	8	2.2	716
wx200d	11	2.5	463	22	1	4	0.6	173
xastir	140	931.2	48347	11899	47	111	476.8	16976
xawdecode	19	123.4	6221	742	20	57	40.8	2706
xbae	55	103.4	15900	1179	18	27	41.5	2835
xmp	9	30.1	4075	517	30	55	11.2	2238
yplot	5	2.2	389	6	1	1	0.2	14
ysmv7	8	13.7	946	166	11	22	4.9	416
zimg	47	48.7	2798	272	3	8	7.5	806
Grand Total	1405	5792.5	418422	41900			2023.4	117515

Table H.2: Information about project sizes. Page 2 of 2

Appendix I

Density test results

The range of density parameters tested to determine a suitable set for classifying the data were all possible combinations of those listed in Table I.1. These parameter sets are listed in Table I.2, with their classification accuracy on the split file dataset, as dd-bb-gg, where dd is the density, bb is the minimum block size and gg is the largest gap permitted. Of those tested, the parameter set which achieves the highest accuracy over the 23 algorithms (see Table 14.2, p.226) is 95-40-3, a minimum density of 95%, a minimum block size of 40 tokens, and a maximum gap size of 3 tokens. These are therefore the parameters used in this research. The ranks (from 1–80) are summed for each component of the parameters to find their contribution to the results, and are listed in Table I.3. The components of the top performing combination 0.95-40-3 are top of their respective groups. The lower densities, largest block size, and bigger gap are the worst for this data.

Minimum Density	Minimum block	Maximum gap
60	10	3
65	20	10
70	40	
75	80	
80	160	
85		
90		
95		

Table I.1: Density test parameters

Density set	Avg % correct	Density set	Avg % correct	Density set	Avg % correct
0.95-40-3	82.44	0.70-80-3	79.93	0.65-20-3	79.13
0.95-10-3	82.04	0.75-80-3	79.92	0.80-40-10	79.04
0.95-40-10	81.92	0.95-80-3	79.88	0.80-10-10	78.99
0.90-40-3	81.69	0.85-80-3	79.87	0.75-40-10	78.96
0.90-40-10	81.57	0.60-80-3	79.86	0.85-160-10	78.92
0.95-20-3	81.41	0.65-80-3	79.86	0.90-160-10	78.92
0.90-10-10	81.27	0.75-10-3	79.86	0.70-10-10	78.89
0.95-10-10	81.19	0.85-80-10	79.78	0.70-80-10	78.88
0.90-10-3	81.16	0.85-40-10	79.76	0.70-20-10	78.87
0.85-40-3	81.10	0.95-80-10	79.74	0.90-160-3	78.80
0.90-20-10	80.98	0.80-10-3	79.71	0.75-80-10	78.77
0.95-20-10	80.75	0.70-10-3	79.69	0.65-20-10	78.76
0.90-80-10	80.69	0.75-10-10	79.64	0.80-80-10	78.63
0.80-40-3	80.61	0.80-20-3	79.56	0.65-10-10	78.61
0.75-40-3	80.60	0.75-20-10	79.54	0.65-80-10	78.59
0.90-20-3	80.56	0.95-160-3	79.49	0.60-80-10	78.49
0.70-40-3	80.52	0.95-160-10	79.33	0.60-10-10	78.39
0.90-80-3	80.49	0.80-20-10	79.30	0.60-20-10	78.33
0.85-10-3	80.30	0.75-160-3	79.26	0.70-40-10	78.17
0.65-40-3	80.27	0.70-160-3	79.26	0.80-160-10	77.73
0.60-40-3	80.17	0.65-160-3	79.26	0.65-40-10	77.67
0.85-10-10	80.15	0.60-160-3	79.26	0.60-40-10	77.52
0.85-20-10	80.12	0.70-20-3	79.25	0.70-160-10	77.51
0.85-20-3	80.02	0.60-20-3	79.25	0.75-160-10	77.44
0.80-80-3	80.01	0.75-20-3	79.24	0.60-160-10	77.34
0.60-10-3	80.00	0.85-160-3	79.23	0.65-160-10	77.20
0.65-10-3	79.96	0.80-160-3	79.20		

Table I.2: Density test: mean classification rate for each parameter set over all algorithms. The parameter set names show minimum density-minimum block size-maximum gap size.

Density		Block		Gap	
0.95	186	40	484	3	1221
0.9	207	10	528	10	2019
0.85	312	20	629		
0.75	459	80	644		
0.8	471	160	955		
0.7	517				
0.65	541				
0.6	547				

Table I.3: Sum of ranks for each component of dd-bb-gg parameters

Appendix J

SVM grid search results

The classification accuracies achieved by the Weka classifiers were compared with those of support vector machines, one with a radial basis kernel and one with a linear kernel. The libsvm¹ implementation used was that in Chicken Scheme.² The data was split into 100 different training and test sets, to echo the Weka experiments. The mean results of the best of these, with the trigram-based “tris-singles” and “tris” feature sets, and a radial basis kernel, are plotted over the grid search region, in Figures J.1 and J.2. The maximum accuracy on this coarse grid search was 88.2% for “tris-singles” and 86.8% for the “tris” set. These sets, which differ slightly from those reported in Chapter 14 had results of 88.5% and 87.8% respectively with the Rotation Forest algorithm. As the SVM did not outperform the Weka based algorithms, and would not be expected to gain much in a finer-grained search, the use of the SVMs was not pursued further.

¹<http://www.csie.ntu.edu.tw/~cjlin/libsvm/>

²<http://wiki.call-cc.org/eggref/4/libsvm>

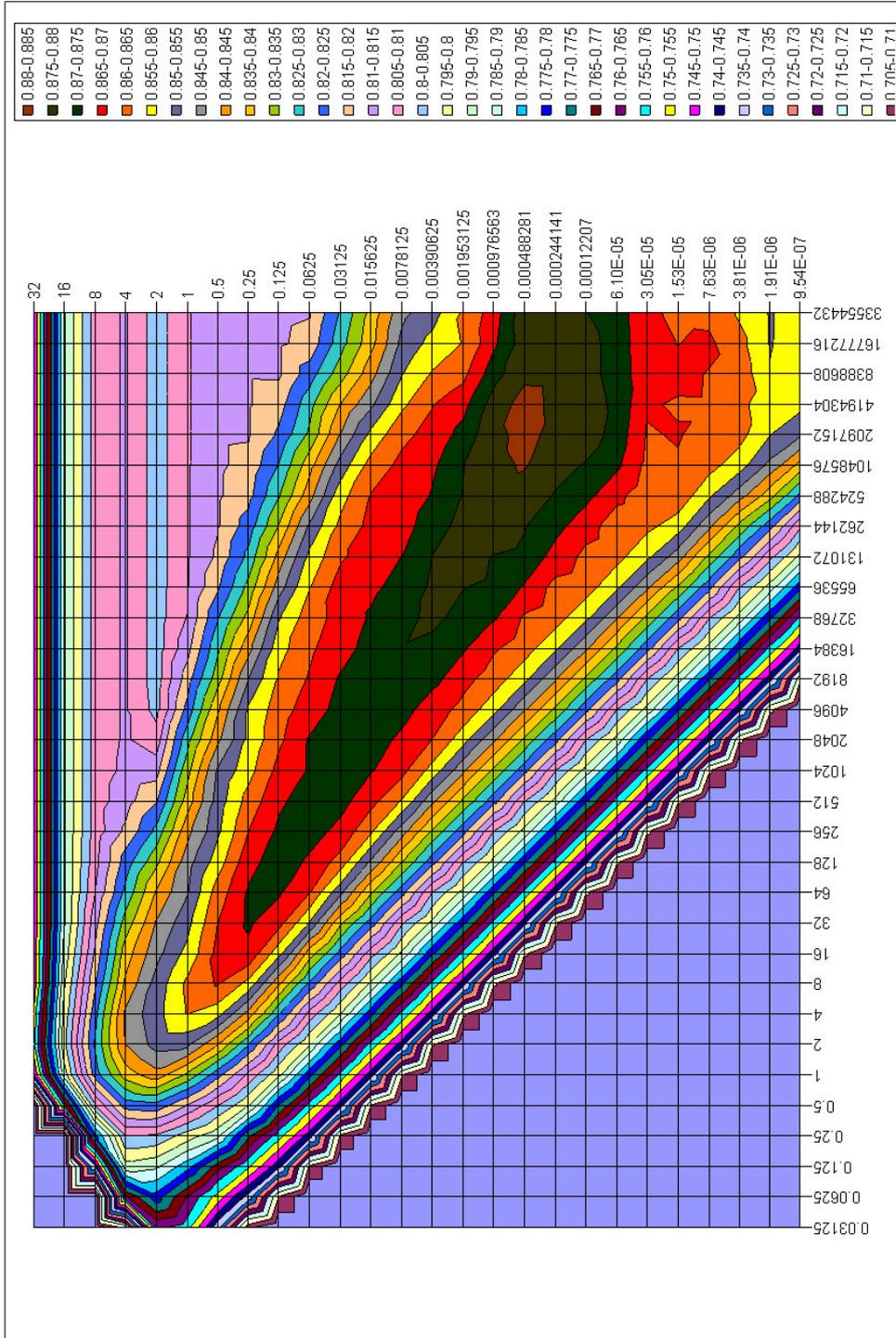


Figure J.1: SVM grid search results: the mean for 100 test runs with the feature set "tris-singles".

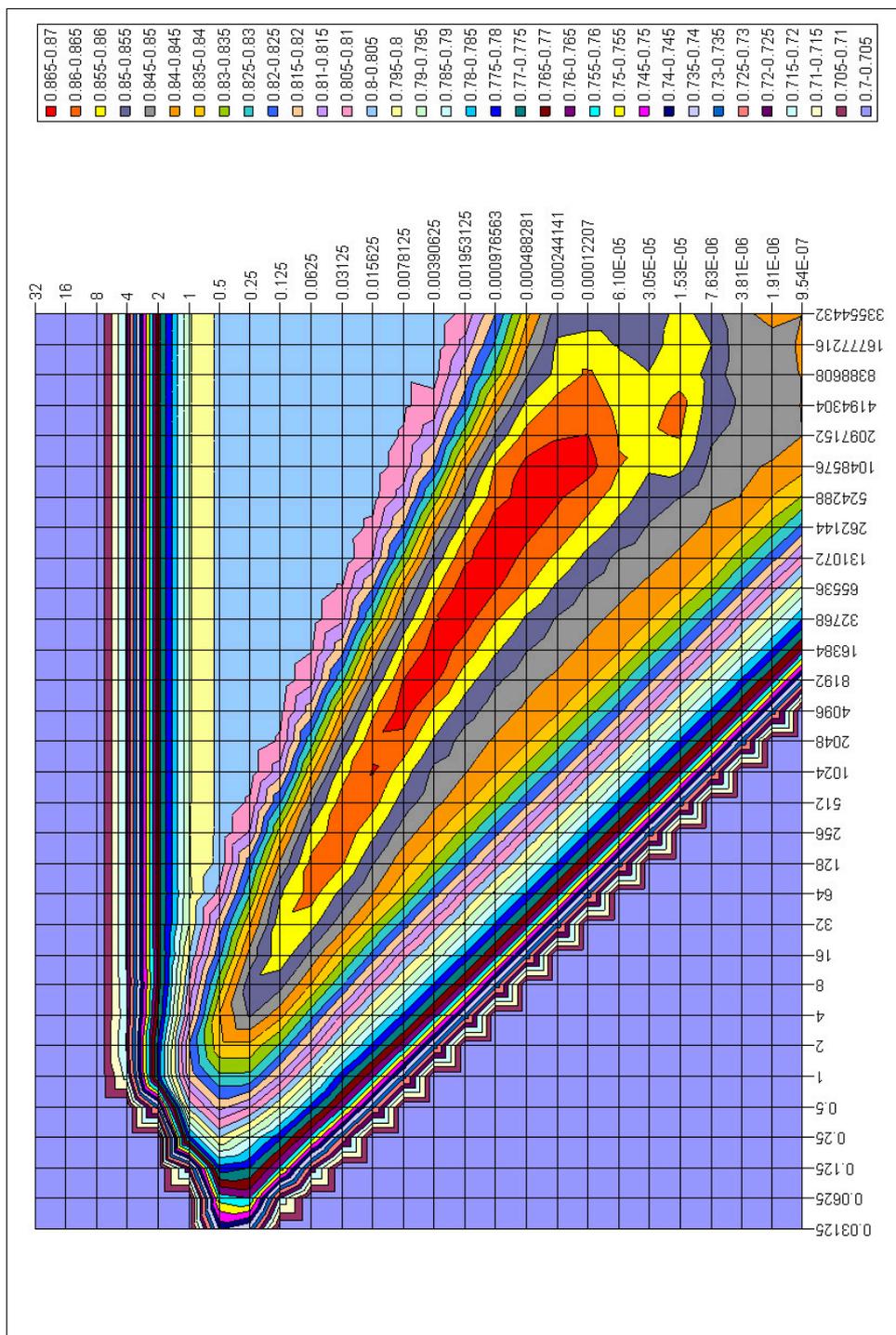


Figure J.2: SVM grid search results: the mean for 100 test runs with the feature set "tris".

Appendix K

Supplementary results for Chapter 14

This appendix reports on two sets of experiments which explore the effect on classification of further feature combinations, and of classifier combination. None of the combinations reported here improve on the results of simpler models on the split file dataset reported in Chapter 14. Also in this appendix is a listing of the DNSjava file MX_KXRecord, referenced in Chapter 14.

K.1 Feature sets - further combinations

Three-, four- or five-way feature set combinations

The top 5 singles sets, fb, fc, ft, pdp and tris, were combined in groups of 3, 4, and 5. The results in Table K.1 show that these combinations do not

Feature set	Mean % correct	Feature set	Mean % correct
fc+fb+tris+pdp-singles	91.70	fc+ft+fb+pdp-singles	91.40
fc+tris+pdp-singles	91.65	ft+tris+pdp-singles	91.36
fb+tris+pdp-singles	91.61	ft+fb+pdp-singles	91.20
fc+ft+fb+tris+pdp-singles	91.53	fc+ft+pdp-singles	91.12
fc+fb+pdp-singles	91.51	ft+fb+tris-singles	91.05
fc+ft+tris+pdp-singles	91.51	fc+ft+fb+tris-singles	91.04
ft+fb+tris+pdp-singles	91.47	fc+ft+tris-singles	91.02
fc+fb+tris-singles	91.43	fc+ft+fb-singles	90.48

Table K.1: Mean classification of 3-, 4- and 5-way combinations of fb, fc, ft, pdp and tris singles sets, over the 23 algorithms.

Feature set	Mean % correct	Feature set	Mean % correct
all-fb+tris-singles	91.79	all-fb+all-pdp	91.03
all-fb+all-tris	91.58	all-fb+ft-singles	90.79
all-fb+pdp-singles	91.45	all-fb+all-fc	90.60
all-fb+fc-singles	91.27	all-fb+all-ft	90.32

Table K.2: Mean classification for selected feature sets in combination with the full fb set over the 23 algorithms.

improve on the classification rate of the best of each group on its own.

The full Ferret basic (“fb”) set combined with other feature sets

The full “fb” set with the SMO algorithm has the best performance for an individual feature set/algorithm combination. To explore the “fb” set’s performance in combination with other sets, it was paired with each of the full and singles features of the other four best sets: “tris”, “fc”, “ft” and “pdp”. The results are shown in Table K.2. Small improvements over the mean performance of 91.27% are found by adding “tris”, “tris-singles” or “pdp-singles”. However, the best individual performance of fb/SMO of $94.29 \pm 1.89\%$ is not improved on.

The full trigram (“tris”) set combined with other feature sets

The other full set which performs well is the “tris” set. This was also combined with the better of the other sets, “fb”, “fc”, “ft” and “pdp”, both as singles and full sets. The results are in Table K.3. None of these combinations improve on the full “tris” set results (91.91%).

Feature set	Mean % correct	Feature set	Mean % correct
all-tris+fb-singles	91.85	all-tris+ft-singles	91.23
all-tris+fc-singles	91.62	all-tris+all-pdp	91.18
all-tris+pdp-singles	91.59	all-tris+all-fc	91.00
all-fb+all-tris	91.58	all-tris+all-ft	90.71

Table K.3: Mean classification for selected feature sets in combination with the full “tris” set over the 23 algorithms.

K.2 Heterogenous meta-classifiers

Heterogenous meta-classifiers combine a number of algorithms with the aim of finding complementary information which will improve on classification accuracy. Previous experiments on earlier versions of this data showed that the best combination of algorithms are those which do best individually. This strategy was repeated with the current dataset. First, the accuracy of the better performing algorithms on the better performing feature sets are noted. Then the results of combining the top 3 algorithms for each set with each of 5 meta-classifiers are listed. In general the results are no better than for the best single algorithm.

How do the better base algorithms perform with the better feature sets?

Table K.4 shows the details of the classification accuracy of each of the 13 selected feature sets with the top 7 of the 11 selected algorithms over these sets. The results are ranked by the mean over each set and each algorithm. The best result for each algorithm is in bold text, and the best for each

Feature set	SMO	Simple Logistic	Rotation Forest	FT	Random Forest	SGD	Dagging	Mean
tris-singles	94.16	93.42	93.20	93.11	92.61	94.01	* 93.78	93.36
fb+tris-singles	94.03	93.47	93.11	93.12	92.54	93.87	93.74	93.41
all-fb+tris-singles	94.25	* 93.49	93.17	92.98	92.66	93.68	93.65	93.41
tris	94.14	93.37	93.15	92.95	93.05	93.96	93.13	93.39
fb	* 94.29	93.68	92.71	93.27	91.84	* 94.17	93.00	93.28
all-fb+all-tris	94.15	93.24	93.30	92.44	92.60	93.68	92.58	93.14
fb-singles	93.46	93.36	92.44	* 93.37	91.57	94.15	90.72	92.72
fc+fb+tris-singles	93.14	93.04	* 93.49	92.56	93.35	91.30	91.65	92.65
fc+tris-singles	92.93	93.04	93.27	92.57	93.37	91.11	91.02	92.47
all-fb+fc-singles	92.66	92.67	* 93.37	92.03	93.33	91.38	91.73	92.45
pdp+tris-singles	92.80	93.23	92.90	92.69	92.60	91.04	91.28	92.36
all-singles	91.25	92.94	92.94	92.35	92.99	91.53	91.42	92.20
all-fb+pdp-singles	91.67	92.76	92.38	92.18	92.57	91.18	91.35	92.01
Mean	93.23	93.21	93.03	92.74	92.70	92.70	92.24	92.84

Table K.4: Classification rates of 13 top feature sets with 7 selected algorithms

feature set is marked by an asterisk. SMO gives the best result with 6 of the 13 feature sets, but there is no clear “winner” among the feature sets.

Does combining algorithms improve performance?

The aim of this experiment was to discover whether combining algorithms using a heterogeneous meta-classifier could improve these results. The meta-classifiers Grading, Stacking, MultiScheme, majority vote and mean vote were used to combine the algorithms. The results for a selection of 4 feature sets are shown in Table K.5. In each case, 3 algorithms which perform best with the set are combined, excluding those of similar type, for example, SGD is not included if SMO is in the set, as both are based on SVMs. The 3 sets are noted on the row labelled ‘Combination’. These are SMO, Simple Logistic (SL), Rotation Forest (ROT), Random Forest (RAND) and Functional Trees (FT). The best result with a single classifier is given in the next row, followed by the results with heterogenous meta-classifiers which combine the three algorithms selected for the feature set.

These results indicate that there is no room for improvement over the single best classifier on any feature set by combining algorithms. As this is also the case when combining feature sets, it seems that the limit of classification accuracy has been reached for this data.

Feature set	tris-singles	fb	all-fb-all-tris	fc+tris
Combination	SMO, SL, ROT	SMO, SL, FT	SMO, SL, ROT	SMO, SL, RAND
Best result	94.16	94.29	94.15	93.37
Majority vote	94.14	94.05	93.96	93.65
Mean vote	93.93	94.14	94.20	93.52
Grading	94.01	93.98	93.96	93.50
MultiScheme	93.41	93.23	93.27	93.40
Stacking	93.36	93.47	93.10	92.82

Table K.5: Heterogeneous meta-classifier results for a selection of feature sets. The highest accuracy obtained by a single classifier with each feature set is given, with the mean results for each of the meta-classifiers below. The algorithms combined are shown, SMO, SL (Simple Logistic), ROT (Rotation Forest), RAND (Random Forest) or FT (functional tree)

Package and import statements are as MXRecord and KXRecord, but removed here to fit to page

```

public class MX_KXRecord extends Record {
private short priority;
private Name target;
protected
MX_KXRecord() {}
public
MX_KXRecord(Name _name, short _type, short _dclass, int _ttl, int _priority,
             Name _target)
{
    super(_name, _type, _dclass, _ttl);
    priority = (short) _priority;
    target = _target;
}
protected
MX_KXRecord(Name _name, short _type, short _dclass, int _ttl,
             int length, DataByteInputStream in, Compression c)
throws IOException
{
    super(_name, _type, _dclass, _ttl);
    if (in == null)
        return;
    priority = (short) in.readUnsignedShort();
    target = new Name(in, c);
}
protected
MX_KXRecord(Name _name, short _type, short _dclass, int _ttl,
             MyStringTokenizer st, Name origin)
throws IOException
{
    super(_name, _type, _dclass, _ttl);
    priority = Short.parseShort(st.nextToken());
    target = new Name(st.nextToken(), origin);
}
public String
toString() {
    StringBuffer sb = toStringNoData();
    if (target != null) {
        sb.append(priority);
        sb.append(" ");
        sb.append(target);
    }
    return sb.toString();
}
public Name
getTarget() {
    return target;
}
public short
getPriority() {
    return priority;
}
void
rrToWire(DataByteOutputStream out, Compression c) throws IOException {
    if (target == null)
        return;
    out.writeShort(priority);
    target.toWire(out, null);
}
void
rrToWireCanonical(DataByteOutputStream out) throws IOException {
    if (target == null)
        return;
    out.writeShort(priority);
    target.toWireCanonical(out);
}
}

```

Figure K.1: The new file MX_KXRecord, referenced in Section 14.2.2

Appendix L

Less direct methods of filtering

This appendix explores the three less direct similarity measures for filtering suggested in Section 15.1. The measures in this category are explained in the next three sections, and each is illustrated with a small set of example files taken from the dataset. These files were selected from two projects: “hatari”, which has multi-way split files, and “gwyddion”, which has high incidental similarity among the files (a mean similarity of 0.31 between all pairs of files in one release, compared to Hatari’s 0.17).

Two files were chosen from the hatari project: `dialog.c`,¹ which is split eleven ways, and `gemdos.h`, a three-way split.² The number of unique trigrams per file in the two releases analysed for this project ranges from none to nearly four thousand, with a mean of around three hundred and thirty-five. There are 149 and 162 files in the two releases.

Three files were chosen from gwyddion: `gwyutils.c`,³ and `gwymodule-file.h`,⁴ which are two-way splits, and `gwpixmapplayer.c`,⁵ which is not split. The file `gwymodule-file.h` is interesting because the code which has been extracted from it appears in two other files, `gwymodulenums.h` and `main.c`. The same code has been factored out of a number of other files to `gwymoduleenums.h` as part of the same restructuring, however, this code

¹<http://homepages.stca.herts.ac.uk/~gp2ag/xmls/dialog-8.xml>, also see p.184

²<http://homepages.stca.herts.ac.uk/~gp2ag/xmls/gemdos.xml>, also see p183

³<http://homepages.stca.herts.ac.uk/~gp2ag/xmls/gwyutils-7.xml>

⁴<http://homepages.stca.herts.ac.uk/~gp2ag/xmls/gwymodule-file-10.xml>

⁵<http://homepages.stca.herts.ac.uk/~gp2ag/xmls/gwpixmapplayer-7.xml>

has not been moved from main.c.

There are 279, 303 and 342 files in the next release to each of these files respectively. The number of unique trigrams per file in these releases ranges from 4 to 2,500, with one outlier of 3,500, and the mean number in each release is around the same as the two hatari releases.

L.0.1 Uniquely shared trigrams (2a)

The trigrams uniquely shared by the candidate file and files in the next release are found by analysing the trigram-file index for a comparison between files in two releases. Any files which uniquely share trigrams with the candidate file are considered to be its targets.

There are advantages and disadvantages to this method of selecting target files. On one hand, it cuts out incidentally similar files. On the other hand, in projects with high incidental similarity, there may be few or no unique trigrams among the files in a release, so that no target files will be selected.

Table L.1 shows the results of the investigation. Candidate (in bold) and target file names are in the second column of the table. The next two columns show the number of unique trigrams each file has within the release, and the number of these which are shared with the candidate file.

Files with fewer than five uniquely shared trigrams are excluded from the table. This method selects the same true target files as the combination conditions for gemdos.c. Two of the targets selected by combination conditions for dialog.c are not true targets. These two files have few uniquely shared trigrams: dlgDevice.c has 8, and dlgFileSelect.c has 2.

One of the two split files in the gwyddion project, gwyutils.c, is matched to gwyenum.c, to which four functions are moved. However, the other split file, gwymodule-file.h, has no target files selected because the code in the target file is shared by another file in the system. No target files are selected for the final file, gwypixmapplayer.c, which is as expected, because it is not a split file. In the similarity combination and shared trigram change sets, 12 and 22 targets respectively are selected for this file, because of the high

Project	Filename	Trigrams unique among files in the same release	Trigrams uniquely shared with the candidate file	
hatari	src/dialog.c	212	198	
	src/gui-sdl/dlgDisc.c	311	286	
	src/gui-sdl/dlgScreen.c	234	228	
	src/gui-sdl/dlgMain.c	232	216	
	src/gui-sdl/dlgTosGem.c	178	172	
	src/gui-sdl/dlgAbout.c	162	143	
	src/gui-sdl/dlgKeyboard.c	128	123	
	src/gui-sdl/dlgSound.c	176	114	
	src/gui-sdl/dlgSystem.c	111	106	
	src/gui-sdl/dlgJoystick.c	85	79	
	src/gui-sdl/dlgMemory.c	147	77	
	src/gui-sdl/dlgDevice.c	73	8	

		gemdos.h	44	40
		gemdos_defines.h	172	166
	gemdos.c	2261	53	
gwyddion	gwyutils.c	708	515	
	gwyenum.c	181	58	

	gwymodule-file.h	42	42	

	gwpixmapplayer	227	203	

Table L.1: Target files found by uniquely shared trigrams for the test cases (<5, not reported). Filenames are listed with the number of unique trigrams in the file and the number of these shared with the candidate file.

level of incidental similarity in the project.

L.0.2 Weighted trigram count (2b)

One way to try to address the problem of target files not being selected by uniquely shared trigram count (because the code is in more than one file) is to use a weighted trigram count. As explained in Chapter 9, to compute this value, each file is compared to all of the files in the next release. Trigrams shared by the candidate and another file are weighted by the inverse of the number of files in release $n+1$ which share the trigram, and are added to the count.

Table L.2 shows the weighted trigram counts for the five example files. The same target files are found as by the count of uniquely shared trigrams for the files dialog, gemdos and gwyutils. Where a value is given as “< n ” in the table, it means that the remaining values are greater than zero and

Candidate	True Targets	Other files	No. other files
dialog	330-87	<36	50
gemdos	171, 67	<36	28
gwyutils	111	32, <13	99
gwymodule-file	12	18, 12, <3	12
gwyxmaplayer	-	22, <13	29

Table L.2: Weighted trigram count for the example files

graduate between zero and the upper value. When the difference between the values is more clear cut, the values are noted. For example, the weighted counts for gwyutils are 111 (the true target), 32, 12.83, 12.25, 11.83, 11.08 The number of other files sharing trigrams with the candidate file is in the last column. The correct target file for gwymodule-file.h is ranked second equal under this measure, but the value is small (12), because little code is moved between the files and it is shared by other files.

L.0.3 Trigrams shared with the candidate difference set (2c)

The candidate difference set is the name given to the set of trigrams which are in the candidate file, but not in the amended file. The difference set for each potential candidate file is compared to the trigrams in each of the files in release n+1. Files containing significant numbers of these trigrams are added to the target group.

Table L.3 shows the number of trigrams which true targets and other files share with the difference set. There are a large number of files which share difference set trigrams in both projects. The true targets for gemdos and gwyutils share noticeably more trigrams with the difference set than other files. There is one non-target file which shares more trigrams than do some of the target files for dialog.c. As with the weighted trigram count, the true target file for gwymodule-file is one of three with marginally higher scores than the rest in the release.

Candidate	True Targets	Other files	No. other files
dialog	576-180	215, <100	125
gemdos	198, 70	<23	86
gwutils	206	<51	259
gwymodule-file	29	30, 10, <4	125
gwpixmapplayer	-	<20	316

Table L.3: Trigrams shared by the example files with the candidate difference set

L.0.4 Discussion

None of these three less direct methods, which require more processing, perform as well as the simpler methods in selecting target files for the more difficult gwymodule-file.h. However, they do reduce the number of incidentally similar files selected as targets for other files. Selection of target files by uniquely shared trigrams can exclude true targets because of duplication in the code. This duplication may be due to incidental similarity or to parallel subsystems. For example, the high incidental similarity of the project gwyddion, or of "ppcboot", which has around 70 subsystems for different motherboards. Weighted similarity goes some way to overcoming the exclusion problem where uniquely shared trigrams fail, but also introduces target files which share candidate code with few others but are not true targets. In general, it is also a less useful measure for projects with high incidental similarity. Finding the number of trigrams shared with the candidate difference set does not improve on the results with uniquely shared trigrams or weighted counts for the problem file tested here.

Appendix M

Extracts from the Lifelines change log

This appendix contains edited entries from the Lifelines project change log. Entries were initially selected by the keywords: factor, refactor, split, move, remove, and while scanning the log visually, the words pull, spun and combine were added. The log was also searched by filename to try to find matches for split files found by filtering and not initially found in the log. Those judged to relate to file splitting are listed, and each split given a number. If the entry is matched by a candidate file, '+++' is added to the entry (usually before the date), if not, '- - -' is added. Other notes are in square brackets.

Release 3

--- 2007-05-19

*NEW src/lifelines/lines_usage.c

1. Move show_usage to be shared by both llines and llexec.

Release 4

+++ 2006-09-16

* ChangeLog src/interp/interp.c src/interp/progerr.c

2. Move declaration of prog_var_error_zstr to correct module (progerr.c).

[* 3, 4, 5: three changes all to one file, so are one split]

* +++ 2006-09-04

406 APPENDIX M. EXTRACTS FROM THE LIFELINES CHANGE LOG

```
* ChangeLog src/gedlib/messages.c src/lifelines/interact.c
src/lifelines/linesi.h src/lifelines/screen.c
src/lifelines/screeni.h src/lifelines/searchui.c
```

3. Move search menu & window painting into searchui.c
Affected: invoke_search_menu, invoke_fullscan_menu,
repaint_fullscan_menu, repaint_search_menu

* +++ 2006-09-02

```
* ChangeLog build/msvc6/lfiles/lfilesprj.dsp src/hdrs/screen.h
src/lifelines/Makefile.am src/lifelines/listui.c
src/lifelines/lfilesi.h src/lifelines/screen.c
*NEW src/lifelines/interact.c
*NEW src/lifelines/screeni.h
*NEW src/lifelines/searchui.c
```

Split interact calls into interact_screen_menu and
interact_choice_string, and renamed interact to interact_worker.
4. Moved interact_screen_menu, interact_choice_string,
interact_worker, translate_hardware_key, translate_control_key
to new file interact.c. Created new header screeni.h.

* +++ 2006-07-25

```
* ChangeLog build/msvc6/lfiles/lfilesprj.dsp src/gedlib/messages.c
src/hdrs/screen.h src/lifelines/Makefile.am src/lifelines/screen.c
src/lifelines/show.c src/lifelines/listui.c src/lifelines/listui.h
```

5. Move popup and browse list code out of screen.c into listui.c.

--- 2006-07-24

```
* ChangeLog src/gedlib/messages.c src/hdrs/screen.h src/lifelines/browse.c
src/lifelines/brwsmenu.c src/lifelines/dynmenu.c src/lifelines/main.c
src/lifelines/menuset.c src/lifelines/screen.c win32/mycurses.c
```

6. Moved platform_postcurses_init from main.c into screen.c.

+++ 2006-06-04

```
* ChangeLog build/msvc6/lfiles/lfilesprj.dsp
po/POTFILES.in src/interp/Makefile.am
src/interp/interp.c src/interp/interpi.h src/interp/rassa.c
*NEW src/interp/progerr.c
```

7. Move lifelines report error routines to new src/interp/progerr.c.

--- 2005-11-26

```
* ChangeLog src/interp/Makefile.am src/interp/more.c
*NEW src/interp/rptsort.c
```

8. Move sort & rsort implementations from end of more.c into new
file rptsort.c.

+++ 2005-11-15

```
* ChangeLog build/msvc6/dbverify/dbverifyCmd.dsp
build/msvc6/llexec/llexec.dsp build/msvc6/lfiles/lfilesprj.dsp
src/hdrs/llstdlib.h src/lifelines/error.c src/stdlib/signals.c
src/tools/dbverify.c
*NEW src/stdlib/errlog.c [from error.c]
*NEW src/stdlib/llabort.c [from signals.c]
```

9. Factor out crashlog reporting into new file src/stdlib/errlog.c,

and implement crashlog for dbverify as well. Also implement
 10. optional abort for dbverify (by factoring optional abort into
 new file src/stdlib/llabort.c).

+++ 2005-10-05

* ChangeLog build/msvc6/llexec/llexec.dsp build/msvc6/lfiles/lfilesprj.dsp
 src/hdrs/liflines.h src/liflines/Makefile.am src/liflines/ask.c
 src/liflines/llexec.c src/liflines/main.c
 *NEW src/liflines/selectdb.c

Combine two copies of open_or_create_database code from
 11. main.c &
 12. llexec.c into one copy in new selectdb.c.

* ChangeLog build/msvc6/dbverify/dbverifyCmd.dsp
 build/msvc6/llexec/llexec.dsp build/msvc6/lfiles/lfilesprj.dsp
 src/hdrs/Makefile.am src/hdrs/gedcom.h

13. Move macros from gedcom.h to new gedcom_macros.h.

+++ 2005-10-01

* ChangeLog src/gedlib/gedcomi.h src/gedlib/init.c
 build/msvc6/lfiles/lfilesprj.dsp
 *NEW src/gedlib/llgettext.c

14. Pull gettext code out of init.c into new file llgettext.c.

+++ 2005-09-25

* ChangeLog build/msvc6/lfiles/lfilesprj.dsp src/gedlib/datei.c
 src/gedlib/datei.h src/gedlib/dateparse.c
 src/gedlib/dateprint.c src/gedlib/date.c

15. Split code for parsing dates, and code for printing dates,
 into different files.

+++ * ChangeLog src/gedlib/datei.c src/gedlib/datei.h
 src/hdrs/date.h src/interp/builtin.c

16. Move a bunch of date structures & enums from date.h to private datei.h.
 Add accessors needed for this move (date_get_day, date_get_month, etc).

 Release 5

+++ 2005-02-27

* ChangeLog src/gedlib/init.c src/hdrs/gedcom.h

17. Move dblist functions out of init.c into new dblist.c

--- 2005-02-19

* build/msvc6/lfiles/lfilesprj.dsp src/gedlib/date.c
 src/gedlib/editvtab.c src/gedlib/gengedc.c src/gedlib/indiseq.c
 src/gedlib/init.c src/gedlib/keytonod.c src/gedlib/lloptions.c
 src/gedlib/misc.c src/gedlib/names.c src/gedlib/node.c
 src/gedlib/valtable.c src/gedlib/xlat.c src/hdrs/table.h
 src/interp/alloc.c src/interp/builtin.c src/interp/interp.c
 src/interp/pvalalloc.c src/interp/pvalue.c src/interp/symtab.c
 src/liflines/import.c src/liflines/llexec.c src/liflines/main.c
 src/liflines/screen.c src/liflines/valgdcom.c
 src/stdlib/memalloc.c src/stdlib/proptbls.c src/stdlib/table.c
 *NEW src/hdrs/hashtab.h

408 APPENDIX M. EXTRACTS FROM THE LIFELINES CHANGE LOG

*NEW src/stdlib/hashtab.c

18. Move hash table into new hashtab.h (use it from table.c).

--- 2005-02-06

* ChangeLog src/btree/btreei.h src/gedlib/gedcomi.h src/hdrs/btree.h

19. Move some btree internal calls to btreei.h.

* ChangeLog src/gedlib/gengedc.c src/gedlib/xlat.c
src/hdrs/table.h src/stdlib/table.c

20. Move gengedc.c local table_incr_item to table.c table_incr_int.
Fix next_table_ptr to use const key.

--- 2005-02-03

* ChangeLog src/btree/btrec.c src/gedlib/gstrings.c src/gedlib/node.c
src/gedlib/spltjoin.c src/hdrs/gedcom.h src/interp/write.c
src/lifelines/advedit.c src/lifelines/browse.c src/tools/dbverify.c

21. Move normalize_indi from private browse.c to public spltjoin.c.

--- 2005-01-30

22. Move SORTEL (indiseq element) structure from indiseq.h to indiseq.c.

+++ 2005-01-25

* ChangeLog build/msvc6/lfiles/lfilesprj.dsp src/btree/Makefile.am
src/gedlib/Makefile.am src/gedlib/gedcomi.h src/gedlib/indiseq.c
src/gedlib/intrface.c src/gedlib/keytonod.c src/gedlib/node.c
src/hdrs/ src/hdrs/btree.h src/hdrs/gedcom.h src/hdrs/pvalue.h
src/interp/builtin.c src/interp/more.c src/interp/pvalue.c
src/lifelines/add.c src/stdlib/list.c
*NEW src/btree/btrec.c
*emptied src/btree/record.c
*NEW src/gedlib/record.c

23. Move record struct & all associated functions into new file record.c.
[from node.c]

+++ 2003-11-12

* .linesrc ChangeLog lines.cfg build/msvc6/lfiles/lfilesprj.dsp
src/gedlib/init.c src/hdrs/Makefile.am src/hdrs/mystring.h
src/lifelines/llexec.c src/lifelines/main.c src/stdlib/Makefile.am
src/stdlib/mystring.c src/stdlib/stdstrng.c src/tools/btedit.c
src/tools/dbverify.c
*NEW src/hdrs/mychar.h
*NEW src/stdlib/mychar_funcs.c
24. *NEW src/stdlib/mychar_tables.c

+++ 2003-10-07

* ChangeLog build/msvc6/lfiles/lfilesprj.dsp
src/gedlib/keytonod.c src/gedlib/node.c src/gedlib/nodeio.c
src/hdrs/Makefile.am src/hdrs/cache.h src/hdrs/gedcom.h
src/hdrs/standard.h src/stdlib/Makefile.am

25. Move two functions taking CACHE into keytonod.c statics.

Release 7

+++ 2003-07-01

* ChangeLog src/hdrs/interp.h src/hdrs/pvalue.h src/interp/interpi.h

Header file restructuring.

26. Move most of interp.h into interpi.h

27. Move pvalue-related stuff into pvalue.h

Release 9

+++ 2003-02-06

* ChangeLog build/msvc6/lfiles/lfilesprj.dsp
src/gedlib/indiseq.c src/gedlib/node.c src/hdrs/interp.h
src/interp/builtin.c src/interp/interpi.h src/interp/pvalue.c
src/stdlib/list.c src/stdlib/table.c
*NEW src/interp/pvalalloc.c

28. Move pvalue memory code into pvalalloc.c.

Release 10

2003-02-04

* ChangeLog build/msvc6/lfiles/lfilesprj.dsp
src/gedlib/Makefile.am src/gedlib/gedcomi.h src/gedlib/node.c
src/hdrs/gedcom.h src/hdrs/liflines.h src/interp/write.c
src/liflines/add.c src/liflines/edit.c
*NEW src/gedlib/nodeio.c

29.

+++ Factored out GEDCOM I/O code from node.c into nodeio.c.

* ChangeLog src/gedlib/nodeio.c src/liflines/add.c
src/liflines/edit.c

30.

+++ Move gedcom output routines from edit.c to nodeio.c.

[Note: Looks as if some has gone to nodeio.c, and some to replace.c.]

+++ 2002-11-25

* ChangeLog build/msvc6/lfiles/lfilesprj.dsp
src/hdrs/Makefile.am src/hdrs/standard.h src/hdrs/version.h
src/interp/builtin.c src/liflines/screen.c
*NEW src/hdrs/list.h
*NEW src/stdlib/list.c
*DELETED src/stdlib/double.c:

31. and moved list declarations from standard.h to list.h.

2002-10-11

* ChangeLog build/msvc6/lfiles/lfilesprj.dsp src/hdrs/Makefile.am
src/hdrs/feedback.h src/hdrs/gedcom.h src/hdrs/liflines.h
src/hdrs/win32/curses.h src/interp/alloc.c src/interp/builtin.c
src/liflines/add.c src/liflines/advedit.c src/liflines/ask.c
src/liflines/askprogram.c src/liflines/browse.c src/liflines/delete.c
src/liflines/export.c src/liflines/lfilesi.h

32. src/liflines/merge.c src/liflines/newrecs.c src/liflines/screen.c

+++ NEW: src/hdrs/uiprompts.h: [from gedcom.h]

Convert some use of NODE to RECORD. Work on ui separation

* ChangeLog src/btree/traverse.c src/hdrs/btree.h
src/hdrs/impfeed.h src/hdrs/version.h src/interp/builtin.c
src/liflines/export.c src/liflines/import.c src/liflines/lfilesi.h

410 APPENDIX M. EXTRACTS FROM THE LIFELINES CHANGE LOG

src/lifelines/loadsave.c src/lifelines/newrecs.c src/lifelines/screen.c:

33.

+++ Pull curses UI out of export.c (into loadsave.c).

--- 2002-07-14

* ChangeLog build/msvc6/lfiles/lfilesprj.dsp hdrs/interp.h
hdrs/llstdlib.h interp/Makefile.am interp/alloc.c
interp/interp.c interp/lex.c interp/yacc.h interp/yacc.y
lifelines/pedigree.c stdlib/path.c
NEW: interp/parse.c [NB this file is deleted 2002-07-18]
NEW: interp/parse.h:

34. Move more parse globals into parse context.

--- 2002-07-07

* ChangeLog stdlib/llstrcmp.c stdlib/memalloc.c stdlib/strcvt.c:

35. Moved string widening code in strcmp into
new function makewide in strcvt.c.

--- 2002-07-04

* ChangeLog gedlib/locales.c hdrs/Makefile.am:

36. Move language & country arrays from gedlib/locales.c into
hdrs/isolang.h.

Release 12

--- 2002-07-02

* ChangeLog arch/vsnprintf.c build/msvc6/lfiles/config.h
build/msvc6/lfiles/lfiles.rc build/msvc6/lfiles/lfilesprj.dsp
gedlib/Makefile.am gedlib/gengedc.c gedlib/indiseq.c
gedlib/lloptions.c gedlib/translat.c hdrs/Makefile.am
hdrs/bfs.h hdrs/feedback.h hdrs/gedcom.h hdrs/impfeed.h
hdrs/llnls.h hdrs/lloptions.h hdrs/metadata.h hdrs/warehouse.h
hdrs/win32/iconvshim.h lifelines/main.c lifelines/menuitem.c
lifelines/pedigree.c stdlib/Makefile.am stdlib/bfs.c
stdlib/llstrcmp.c stdlib/strutf8.c
NEW: gedlib/locales.c
NEW: hdrs/icvt.h
NEW: stdlib/icvt.c

37. Moved locale code from translat.c to locales.c.

+++ 2002-06-27

* ChangeLog build/msvc6/lfiles/lfilesprj.dsp gedlib/charmaps.c
gedlib/init.c gedlib/names.c hdrs/llstdlib.h hdrs/mystring.h
interp/interp.c interp/rassa.c lifelines/ask.c
lifelines/askprogram.c lifelines/export.c lifelines/loadsave.c
lifelines/main.c lifelines/screen.c stdlib/Makefile.am
stdlib/mystring.c stdlib/path.c stdlib/stdstrng.c
NEW: stdlib/appendstr.c stdlib/sprintpic.c stdlib/stralloc.c
NEW: stdlib/strapp.c stdlib/strcvt.c stdlib/strutf8.c
NEW: stdlib/strwhite.c:

38. Split string functions by type.

--- 2002-06-14

* ChangeLog configure.in gedlib/init.c hdrs/Makefile.am

```
hdrs/llstdlib.h hdrs/standard.h stdlib/Makefile.am
stdlib/double.c stdlib/memalloc.c
NEW: hdrs/llnls.h stdlib/llnls.c:
```

39. Move NLS stuff from standard.h into llnls.h.

Release 13

+++ 2002-06-08

```
* ChangeLog Makefile.am gedlib/init.c gedlib/messages.c
hdrs/Makefile.am hdrs/gedcheck.h hdrs/gedcom.h hdrs/screen.h
lifelines/Makefile.am lifelines/import.c lifelines/llinesi.h
lifelines/main.c lifelines/screen.c lifelines/valgdcom.c
m4/Makefile.am po/POTFILES.in po/de.po po/el.po po/it.po
po/sv.po stdlib/double.c win32/msvc6/libintl/config.h
win32/msvc6/libintl/libintlvc6.dsp win32/msvc6/llines/config.h
win32/msvc6/llines/llines.rc win32/msvc6/llines/llinesprj.dsp
NEW: hdrs/impfeed.h lifelines/loadsave.c:
```

40. Remove curses code from import.c via interface functions.
[to loadsave.c (& export.c?)]

+++ 2002-02-18

```
* ChangeLog gedlib/indiseq.c gedlib/intrface.c hdrs/Makefile.am
hdrs/interp.h hdrs/llstdlib.h interp/Makefile.am
interp/builtin.c interp/date.c interp/eval.c interp/interp.c
interp/intrpseq.c interp/more.c interp/pvalue.c interp/rassa.c
lifelines/Makefile.am lifelines/browse.c lifelines/main.c
lifelines/screen.c lifelines/show.c stdlib/double.c
win32/msvc6/llines/llinesprj.dsp
NEW: hdrs/date.h:
```

41. Moved date function declarations out of interp.h into new date.h.

+++ 2002-02-17

```
* ChangeLog gedlib/names.c hdrs/interp.h hdrs/llstdlib.h
hdrs/mystring.h hdrs/screen.h hdrs/table.h interp/Makefile.am
interp/alloc.c interp/builtin.c interp/date.c interp/eval.c
interp/heapused.c interp/interp.c interp/intrpseq.c
interp/more.c interp/pvalue.c interp/rassa.c interp/yacc.y
lifelines/browse.c lifelines/main.c lifelines/screen.c
reports/exercise.ll stdlib/mystring.c stdlib/stdstrng.c
stdlib/table.c win32/mycurses.c win32/msvc6/llines/llines.rc
win32/msvc6/llines/llinesprj.dsp:
```

42. Spun off pvalmath.c & symtab.c from pvalue.c.

--- 2002-01-01

```
* ChangeLog hdrs/screen.h lifelines/llinesi.h lifelines/screen.c
lifelines/show.c:
```

43. Renamed show_list to show_big_list, and moved to screen.c
(preparation for improving full list screen).

--- 2001-12-31

```
* ChangeLog docs/Install.LifeLines.Windows.txt
docs/Run.LifeLines.Windows.txt gedlib/messages.c hdrs/interp.h
hdrs/llstdlib.h interp/builtin.c interp/date.c
stdlib/stdstrng.c:
```

44. Moved all month names into messages.c.

412 APPENDIX M. EXTRACTS FROM THE LIFELINES CHANGE LOG

--- 2001-12-24

* ChangeLog gedlib/messages.c lifelines/show.c:

45. Moved remaining English strings in show.c into messages.c.

* ChangeLog gedlib/editmap.c gedlib/messages.c hdrs/gedcom.h
hdrs/interp.h hdrs/lifelines.h interp/builtin.c interp/date.c
interp/eval.c interp/functab.c interp/interpi.h lifelines/ask.c
lifelines/screen.c reports/exercise.ll:

46. Moved some English strings from builtin.c to messages.c.

--- 2001-11-20

* .linesrc ChangeLog lines.cfg docs/lifelines.sgml
gedlib/messages.c gedlib/node.c hdrs/cache.h hdrs/lloptions.h
interp/date.c lifelines/export.c lifelines/show.c
stdlib/lloptions.c tools/dbverify.c:

47. (Internat.) Moved some indi strings ("born"...) into messages.c.

Release 14

++?? 2001-11-08

48. Moved ui code from lifelines/remove.c into lifelines/delete.c,
and then moved non-ui remove.c to gedlib/.
%[looks like code has moved from lifelines/delete.c to gedlib/remove.c]

Appendix N

Additional results 1: Chapter 16

In this appendix, detailed results of filtering the two projects PostgreSQL and DNSjava to find files which have disappeared from the system are provided. For each project, there are two tables where unmatched and matched files are listed. Unmatched means files where there is no file with a similarity of at least 0.05 in the next release, these files are considered to have been deleted. Matched means files with a target file of at least 0.85 similarity in the next release, these files are considered to be renamed, moved or both. Uncertain files fall between the two and are the files whose classification is determined using the machine learning model.

N.1 PostgreSQL

In the PostgreSQL tables, the information is split into two sections, with files from the backend subsystem studied by Zou [261] separated from the rest. The first table (N.1) shows the matched files, with their similarity, and the second table (N.2) lists unmatched files. Figure N.1 shows `geqo_paths.c`, referred to in Chapter 16, not classified as merged. The target file, `prune.c`, is not selected because the number of trigrams shared by the files falls from 231 to 119.

backend subsystem		not backend subsystem	
Ref.	Filename	Ref.	Filename
2	/backend/access/heap/stats.c	2	/include/catalog/pg_inheritproc.h
2	/backend/access/transam/transsup.c	2	/include/catalog/pg_ipl.h
2	/backend/lib/hasht.c	2	/include/catalog/pg_log.h
2	/backend/libpq/pqpacket.c	2	/include/catalog/pg_variable.h
2	/backend/utills/cache/rel.c	2	/include/lib/hasht.h
2	/backend/utills/mb/palloc.c	2	/include/storage/pagenum.h
2	/backend/utills/mb/sjstest.c		
2	/backend/utills/mb/utftest.c	4	/bin/pg_version/pg_version.c
		4	/include/access/funcindex.h
4	/backend/lib/fstack.c	4	/include/lib/fstack.h
4	/backend/libpq/be_dumpdata.c	4	/include/optimizer/internal.h
4	/backend/libpq/be_pqexec.c	4	/include/regex/cdefs.h
4	/backend/libpq/portal.c	4	/include/regex/regexp.h
4	/backend/libpq/portalbuf.c	4	/include/utills/lztext.h
4	/backend/nodes/freelfuncs.c	4	/include/utills/module.h
4	/backend/optimizer/geqo/geqo_params.c	4	/include/utills/trace.h
4	/backend/optimizer/util/indexnode.c	4	/install-sh
4	/backend/port/hpux/fixade.h	4	/interfaces/odbc/acconfig.h
4	/backend/storage/lmgr/multi.c	4	/interfaces/odbc/install-sh
4	/backend/storage/lmgr/single.c	4	/utills/version.c
4	/backend/utills/adt/chunk.c	4	/win32/endian.h
4	/backend/utills/adt/filename.c	4	/win32/tcp.h
4	/backend/utills/init/enbl.c	4	/win32/un.h
4	/backend/utills/mmgr/oset.c		
4	/backend/utills/mmgr/palloc.c	6	/bin/psql/psqlHelp.h
		6	/include/lib/qsrt.h
6	/backend/utills/sort/lselect.c	6	/include/optimizer/ordering.h
6	/backend/utills/sort/psort.c	6	/include/utills/lselect.h
		6	/include/utills/psort.h
		10	/include/catalog/pg_parg.h
10	/backend/access/common/heapvalid.c	10	/include/optimizer/geqo_paths.h
		10	/interfaces/libpq++/pgenv.h
		10	/pl/plpgsql/gram.c
		10	/pl/plpgsql/y.tab.h
11	/backend/port/linuxalpha/machine.h	11	/include/utills/oidcompos.h
11	/backend/utills/adt/oidint2.c	11	/include/version.h
11	/backend/utills/adt/oidint4.c	11	/interfaces/ecpg/test/Ptest1.c
11	/backend/utills/adt/oidname.c	11	/interfaces/ecpg/test/test1.c
12	/backend/lib/qsrt.c	12	/include/catalog/pg_defaults.h
12	/backend/optimizer/prep/archive.c	12	/include/catalog/pg_demon.h
12	/backend/parser/sysfunc.c	12	/include/catalog/pg_hosts.h
12	/backend/port/univel/frontend-port-protos.h	12	/include/catalog/pg_magic.h
		12	/include/catalog/pg_server.h
		12	/include/catalog/pg_time.h
		12	/include/parser/sysfunc.h

Table N.2: PostgreSQL files which have disappeared from the system and for which there is no file of at least 0.05 similarity in the next release

Document: /home/.../postgresql-n/src/backend/optimizer/geqo/geqo_paths.c

Blue file: /home/.../postgresql-n/src/backend/optimizer/path/prune.c O

Red file: /home/.../postgresql-n+1/src/backend/optimizer/path/prune.c O

Yellow file: No third file

```

#include "postgres.h"
#include "nodes/pg_list.h"
#include "nodes/relation.h"
#include "nodes/primnodes.h"
#include "utils/balloc.h"
#include "utils/elog.h"
#include "optimizer/internal.h"
#include "optimizer/paths.h"
#include "optimizer/pathnode.h"
#include "optimizer/clauses.h"
#include "optimizer/cost.h"
#include "optimizer/geqo_paths.h"
static List *geqo_prune_rel (RelOptInfo * rel, List * other_rels);
static Path *set_paths (RelOptInfo * rel, Path * unorderedpath);
List *
geqo_prune_rels (List * rel_list)
{
    List *temp_list = NIL;
    if (rel_list != NIL)
    {
        temp_list =
            lcons (lfirst (rel_list),
                geqo_prune_rels (geqo_prune_rel
                    (RelOptInfo *) lfirst (rel_list),
                    lnext (rel_list)));
    }
    return temp_list;
}
static List *
geqo_prune_rel (RelOptInfo * rel, List * other_rels)
{
    List *i = NIL;
    List *t_list = NIL;
    List *temp_node = NIL;
    RelOptInfo *other_rel = (RelOptInfo *) NULL;
    foreach (i, other_rels)
    {
        other_rel = (RelOptInfo *) lfirst (i);
        if (same (rel->relids, other_rel->relids))
        {
            rel->pathlist = add_pathlist (rel, rel->pathlist, other_rel->pathlist);
            t_list = nconc (t_list, NIL);
        }
        else
        {
            temp_node = lcons (other_rel, NIL);
            t_list = nconc (t_list, temp_node);
        }
    }
    return t_list;
}
void
geqo_rel_paths (RelOptInfo * rel)
{
    List *y = NIL;
    Path *path = (Path *) NULL;
    JoinPath *cheapest = (JoinPath *) NULL;
    rel->size = 0;
    foreach (y, rel->pathlist)
    {
        path = (Path *) lfirst (y);
        if (!path->p_ordering.ord.sortop)
            break;
    }
    cheapest = (JoinPath *) set_paths (rel, path);
    if (IsA (JoinPath (cheapest)))
        rel->size = compute_joinrel_size (cheapest);
}
static Path *
set_paths (RelOptInfo * rel, Path * unorderedpath)
{
    Path *cheapest = set_cheapest (rel, rel->pathlist);
    if (unorderedpath != cheapest && rel->pruneable)
    {
        rel->unorderedpath = (Path *) NULL;
        rel->pathlist = lremove (unorderedpath, rel->pathlist);
    }
    else
        rel->unorderedpath = (Path *) unorderedpath;
    return cheapest;
}

```

Figure N.1: The disappearing file `geqo_paths.c` compared to `prune.c`, the most similar file in the `optimizer/path` subdirectory, in both the same release (blue file), and in the next release (red file). The trigrams shared by the two files are reduced from 231 (in blue and purple) to 119 (purple), making it unlikely that any text-based analysis will find this relationship. Beagle's combination of matching techniques finds this merge.

N.2 DNSjava

There are also two tables for DNSjava, and the files are split into two groups, those before release 39, studied by Antoniol et al., and those from release 40 onwards. Otherwise the Tables N.3 and N.4 echo the PostgreSQL tables.

N.3 Reporting

As each project is run, a report is output detailing the unmatched files; the matched files, and which file they are matched to; the uncertain files, the similarity to the main target, and the main target; and lastly the number of files which are thought to have changed directory, based on a threshold which can be selected at run time (see example in Figure N.2).

Ref.	V'n.	File
2	56	/org/xbill/DNS/Verifier.java
2	56	/org/xbill/DNS/security/CERTConverter.java
2	56	/org/xbill/DNS/security/DHPubKey.java
2	56	/org/xbill/DNS/security/DNSSECVerifier.java
2	56	/org/xbill/DNS/security/DSAPubKey.java
2	56	/org/xbill/DNS/security/DSASignature.java
2	56	/org/xbill/DNS/security/RSAPubKey.java
10	48	/org/xbill/DNS/TypedObject.java
12	46	/org/xbill/DNS/Utils/md5.java
18	40	/org/xbill/DNS/Utils/StringValueTable.java

25	33	/org/xbill/Task/WorkerThread.java
27	31	/org/xbill/DNS/TypeMap.java
28	30	/org/xbill/DNS/BitString.java
38	20	/org/xbill/DNS/TypeClass.java

Table N.3: Unmatched disappearing DNSjava files

Ref.	From	To	No.files
45	dnsjava-13/DNS/Utils/	dnsjava-14/org/xbill/DNS/Utils/	8
45	dnsjava-13/DNS/	dnsjava-14/org/xbill/Task/	1
45	dnsjava-13/DNS/	dnsjava-14/org/xbill/DNS/	47
47	dnsjava-11/DNS/	dnsjava-12/DNS/Utils/	1

Figure N.2: Extract from an example report on the number of “disappearing” DNS-java files which move from one directory to another

Ref.	Rel.	File	Renamed file	Sim.
45	13	/DNS/CNAMERecord.java	/org/xbill/DNS/CNAMERecord.java	0.87
45	13	/DNS/NSRecord.java	/org/xbill/DNS/NSRecord.java	0.87
45	13	/DNS/PTRRecord.java	/org/xbill/DNS/PTRRecord.java	0.87
45	13	/DNS/SimpleResolver.java	/org/xbill/DNS/SimpleResolver.java	0.89
45	13	/DNS/Credibility.java	/org/xbill/DNS/Credibility.java	0.92
45	13	/DNS/NS_CNAME_PTRRecord.java	/org/xbill/DNS/NS_CNAME_PTRRecord.java	0.93
45	13	/DNS/Opcode.java	/org/xbill/DNS/Opcode.java	0.94
45	13	/DNS/Record.java	/org/xbill/DNS/Record.java	0.94
45	13	/DNS/HINFORecord.java	/org/xbill/DNS/HINFORecord.java	0.94
45	13	/DNS/UNKRecord.java	/org/xbill/DNS/UNKRecord.java	0.94
45	13	/DNS/TXTRecord.java	/org/xbill/DNS/TXTRecord.java	0.94
45	13	/DNS/DClass.java	/org/xbill/DNS/DClass.java	0.94
45	13	/DNS/MXRecord.java	/org/xbill/DNS/MXRecord.java	0.95
45	13	/DNS/Compression.java	/org/xbill/DNS/Compression.java	0.95
45	13	/DNS/ARecord.java	/org/xbill/DNS/ARecord.java	0.95
45	13	/DNS/Utils/StringValueTable.java	/org/xbill/DNS/Utils/StringValueTable.java	0.95
45	13	/DNS/SRVRecord.java	/org/xbill/DNS/SRVRecord.java	0.95
45	13	/DNS/TypeClass.java	/org/xbill/DNS/TypeClass.java	0.95
45	13	/DNS/Flags.java	/org/xbill/DNS/Flags.java	0.95
45	13	/DNS/CERTRecord.java	/org/xbill/DNS/CERTRecord.java	0.96
45	13	/DNS/Section.java	/org/xbill/DNS/Section.java	0.96
45	13	/DNS/OPTRecord.java	/org/xbill/DNS/OPTRecord.java	0.96
45	13	/DNS/Rcode.java	/org/xbill/DNS/Rcode.java	0.96
45	13	/DNS/KEYRecord.java	/org/xbill/DNS/KEYRecord.java	0.96
45	13	/DNS/Address.java	/org/xbill/DNS/Address.java	0.96
45	13	/DNS/NXTRecord.java	/org/xbill/DNS/NXTRecord.java	0.96
45	13	/DNS/NameSet.java	/org/xbill/DNS/NameSet.java	0.96
45	13	/DNS/SOARRecord.java	/org/xbill/DNS/SOARRecord.java	0.97
45	13	/DNS/SetResponse.java	/org/xbill/DNS/SetResponse.java	0.97
45	13	/DNS/dns.java	/org/xbill/DNS/dns.java	0.97
45	13	/DNS/Utils/base16.java	/org/xbill/DNS/Utils/base16.java	0.97
45	13	/DNS/Utils/DataByteOutputStream.java	/org/xbill/DNS/Utils/DataByteOutputStream.java	0.97
45	13	/DNS/Utils/DataByteInputStream.java	/org/xbill/DNS/Utils/DataByteInputStream.java	0.98
45	13	/DNS/TSIGRecord.java	/org/xbill/DNS/TSIGRecord.java	0.98
45	13	/DNS/TTL.java	/org/xbill/DNS/TTL.java	0.98
45	13	/DNS/Master.java	/org/xbill/DNS/Master.java	0.98
45	13	/DNS/TSIG.java	/org/xbill/DNS/TSIG.java	0.98
45	13	/DNS/Header.java	/org/xbill/DNS/Header.java	0.98
45	13	/DNS/SIGRecord.java	/org/xbill/DNS/SIGRecord.java	0.98
45	13	/DNS/Type.java	/org/xbill/DNS/Type.java	0.98
45	13	/DNS/Utils/hmacSigner.java	/org/xbill/DNS/Utils/hmacSigner.java	0.98
45	13	/DNS/Message.java	/org/xbill/DNS/Message.java	0.98
45	13	/DNS/Name.java	/org/xbill/DNS/Name.java	0.99
45	13	/DNS/FindServer.java	/org/xbill/DNS/FindServer.java	0.99
45	13	/DNS/Zone.java	/org/xbill/DNS/Zone.java	0.99
45	13	/DNS/Utils/base64.java	/org/xbill/DNS/Utils/base64.java	0.99
45	13	/DNS/Utils/MyStringTokenizer.java	/org/xbill/DNS/Utils/MyStringTokenizer.java	0.99
45	13	/DNS/Cache.java	/org/xbill/DNS/Cache.java	0.99
45	13	/DNS/Utils/md5.java	/org/xbill/DNS/Utils/md5.java	0.99
47	11	/DNS/MyStringTokenizer.java	/DNS/Utils/MyStringTokenizer.java	0.99

Table N.4: Disappearing DNSjava files with a file of at least 0.85 similarity in the next release. All of these files are moved from one directory to another.

Appendix O

Additional results 2: Chapter 16

The dataset of disappearing files with two or more target files is imbalanced, with a ratio of 45:45:10 unrelated:renamed/moved:split files. In this appendix, the results of classifying the data using over- and under-sampling and cost-based classification are reported. Each of these methods aims to improve classification of the minority class, the first two by artificially balancing the data, and the last by applying a penalty to the misclassification of a member of the minority class.

O.1 Disappearing files: classifying imbalanced data

Table O.1 repeats Table 16.18 for reference, and shows the geometric means of the top ten models, ranked by classification accuracy on this dataset.

Three of the feature sets in these top ten combinations are the larger ones: “all-feats”, “fall” and “all-cats” (all, all Ferret, and all concatenated features, respectively). However, the geometric means of classification accuracy for these three sets are not as high as those of the smaller sets. The seven other sets were used to investigate the different methods of dealing with imbalanced data discussed in Chapter 4: over-sampling, under-sampling, and cost-based algorithms.

The data was over-sampled using the SMOTE [39] algorithm with three different random seeds, to bring the minority class to 258 instances, as the

Feature set	Algorithm	Unrelated	Renamed	Split	Geo.mean
fl+tris-singles	ROT	0.929	0.885	0.729	0.843
all-fb+all-pdp	SL	0.906	0.897	0.712	0.833
all-tris+all-fl	ROT	0.922	0.904	0.661	0.820
all-tris+fc-singles	ROT	0.890	0.908	0.678	0.818
all-tris+all-fc	ROT	0.898	0.893	0.678	0.816
tris	ROT	0.906	0.877	0.661	0.807
fall-cats	ROT	0.910	0.893	0.627	0.799
fall	ROT	0.902	0.897	0.627	0.797
all-feats	ROT	0.918	0.889	0.576	0.778
all-cats	ROT	0.929	0.893	0.508	0.750

Abbreviations: ROT - Rotation Forest, SL - Simple Logistic

Table O.1: Geometric means of accuracy for the top 10 results from Table 16.15

majority classes have 255 and 261 instances. Each of the sets were run with the nine algorithms listed in Table 16.17. Random Forest is one of the two top-performing algorithms on this data and so was used in the comparison reported in Table O.2. The results over the 100 sets on the 89 project dataset are given in column 2. Columns 3-5 have results for the PostgreSQL and DNSjava data. Column 3 has the classification accuracy using the model trained on the imbalanced 89 project dataset, with the mean of the results over the three SMOTE sets in column 4, and the difference between the two in column 5. The mean difference between the two sets of results is insignificant at approximately 0.1%. However, in the unseen data, in which there are 14 examples of split disappearing files, on average one more of these examples are correctly classified by the SMOTE sets. As the overall

Feature set	Imbalanced mean over 100 sets	PostgreSQL & DNSjava		
		Imbal.	SMOTE	Diff.
all-tris+all-fl	86.40	85.30	85.00	-0.30
all-tris+all-fc	86.57	87.70	88.25	0.55
all-fb+all-pdp	85.38	85.25	86.07	0.82
all-fb+all-fc	85.19	88.50	88.50	0.00
all-fb+all-tris	85.43	89.30	88.52	-0.78
tris	85.53	86.70	86.07	-0.63
fall-cats	85.16	86.89	87.34	0.45
fl+tris-singles	84.29	86.07	86.89	0.82
fc+tris-singles	83.84	88.52	88.52	0.00

Table O.2: The effect on classification of balancing the dataset by over-sampling with SMOTE. The mean classification rate over the 100 test sets for the imbalanced set with Random Forest (25) is in column 2. The rate for the unseen data is in column 3, with the mean over the 3 SMOTE sets in the 4th column, and the difference between them in the 5th column.

accuracy is unchanged, this means that an instance from one of the other classes will be misclassified instead.

The data was also under-sampled using the Weka SpreadSubSample filter, so that the datasets consisted of 59 instances from each class, small samples given the variation in the data. Classification on the unseen data with the three sets tested was around 3% less accurate than with other sets, and is therefore not reported in detail.

Two cost-based wrappers are provided in Weka 3.7.3, Cost Sensitive and Meta Cost, for which the user provides the costs to be associated with incorrect classification. Splits are under-represented in this data, therefore costs were set to 1 for misclassifying each of the unrelated and renamed classes, and to 2 for the split class. Table O.3 reports on the results of applying costs to the Rotation Forest, Random Forest, Simple Logistic, and Random Committee algorithms to the 89 project dataset. There are small differences between classifying with, and without, the costs. For example, the mean difference between Rotation Forest with costs, and without, is around a quarter of a percent over the feature sets in the table, and between the geometric means is 0.005.

Tests on the unseen data with the “all-tris+all-fl” (trigram and line count) feature set, and costs of both 2 and 5, given in Table O.4, show that for these feature set/algorithm combinations, introducing costs does not improve the number of correctly classified members of the minority class.

Strategies for dealing with imbalance in the classes: over-sampling and under-sampling the data, and using cost-based algorithms, do not give significant changes in most of the tests reported. The Random Forest models built using data over-sampled by the SMOTE algorithm tend to correctly classify one more instance of the minority class. However, this effect is not repeated in the better models, such as the “all-tris+all-fl”/SimpleLogistic model, which already classifies 11 of the 14 instances correctly. In summary, none of the strategies for dealing with the imbalance in the classes improves on the better models built with the raw data.

Meta	'Base'	all-tris+all-fl	tris	all-tris+all-fc	all-fb+all-tris	all-fb+all-pdp	fall-cats	all-fb+all-fc	fl+tris-singles	fc+tris-singles	Mean
-	ROT	88.13	87.22	87.31	87.13	86.41	87.17	86.47	87.20	86.60	87.07
CS	ROT	88.37	87.60	87.51	87.50	86.70	87.34	86.66	87.73	86.56	87.33
MC	ROT	88.19	87.43	87.58	87.44	86.46	87.25	86.47	87.49	86.82	87.24
CS	RAN	87.07	87.38	87.17	87.05	86.26	85.97	86.50	86.61	86.34	86.71
MC	RAN	86.57	87.07	86.54	86.97	85.73	85.44	85.90	86.15	86.08	86.27
CS	RC	85.99	86.39	85.97	86.14	85.35	84.88	85.33	85.48	85.48	85.67
MC	RC	85.91	86.25	85.69	85.99	85.17	85.12	85.53	85.16	85.30	85.57
MC	SL	86.41	85.06	85.54	84.98	87.28	87.24	85.79	83.91	82.66	85.43
CS	SL	86.74	84.85	86.06	84.55	87.20	86.54	86.63	83.94	82.76	85.47
-	ROT	0.82	0.81	0.82	0.80	0.83	0.80	0.79	0.84	0.84	0.82
CS	ROT	0.80	0.83	0.83	0.82	0.82	0.83	0.82	0.83	0.81	0.82
MC	ROT	0.82	0.84	0.83	0.83	0.80	0.82	0.79	0.81	0.80	0.82
CS	SL	0.80	0.82	0.82	0.82	0.81	0.77	0.82	0.75	0.80	0.80
MC	SL	0.81	0.80	0.80	0.80	0.81	0.74	0.80	0.77	0.80	0.79
CS	RAN	0.76	0.82	0.81	0.81	0.80	0.82	0.78	0.80	0.74	0.79
MC	RAN	0.76	0.81	0.81	0.81	0.82	0.79	0.75	0.81	0.73	0.78
CS	RC	0.72	0.80	0.80	0.81	0.76	0.79	0.77	0.74	0.70	0.77
MC	RC	0.72	0.79	0.80	0.81	0.75	0.79	0.75	0.75	0.68	0.77

Table O.3: Classification using Cost Sensitive (CS) and Meta Cost (MC) wrappers with Rotation Forest (ROT), Random Forest (RAN), Random Committe (RC) and Simple Logistic (SL). Mean classification accuracies over 100 sets at the top, and geometric means at the bottom.

Base	Std.	MC-2	CS-2	MC-5	CS-5
Simple Logistic	11	11	10	11	10
Rotation Forest	8	7	9	7	8
Random Forest	7	7	7	7	7
Random Committe	7	7	7	8	7

Table O.4: The number of minority class (disappearing and split) from the 14 examples in the unseen dataset correctly classified with the base algorithm and the all-tris+all-fl feature set, with Cost Sensitive (CS) and Meta Cost (MC) wrappers, with misclassification costs of 2 and 5.

Appendix P

Dig et al.: Struts results

This appendix gives the refactorings Dig et al. [64] expected to find in the project Struts between release 1.1 and 1.2.4 on pages 425 and 426.¹ The list on page 425 has been rearranged to group the impact of the changes at file level, for comparison with the findings of the research in this dissertation. The refactorings which were found by Dig et al.'s Refactoring Crawler are ticked.

The changes to methods listed here mean that the following changes were made at file level: RequestUtils was split three ways to ModuleUtils and TagUtils, ResponseUtils is also split to TagUtils, and ActionMapping is split to ActionConfig. Two of the split files, ActionMapping.java (left) and ResponseUtils.java (right) and are shown in Figure P.1, the other file is online at <http://homepages.stca.herts.ac.uk/~gp2ag/xmls/RequestUtils.xml>.

¹<http://netfiles.uiuc.edu/dig/RefactoringCrawler>



Figure P.1: One method moved from ActionMapping to ActionConfig (left) and most of the code in the file ResponseUtils (right) moved to TagUtils (right) between releases 1.1 and 1.2.4 of the project Struts

Moved from file RequestUtils to file ModuleUtils

- ✓ MovedMethods, org.apache.struts.util.RequestUtils.selectModule, org.apache.struts.util.ModuleUtils.selectModule
- ✓ MovedMethods, org.apache.struts.util.RequestUtils.selectModule, org.apache.struts.util.ModuleUtils.selectModule
- ✓ MovedMethods, org.apache.struts.util.RequestUtils.getModuleName, org.apache.struts.util.ModuleUtils.getModuleName
- ✓ MovedMethods, org.apache.struts.util.RequestUtils.getModuleName, org.apache.struts.util.ModuleUtils.getModuleName
- ✓ MovedMethods, org.apache.struts.util.RequestUtils.getModuleConfig, org.apache.struts.util.ModuleUtils.getModuleConfig
- ✓ MovedMethods, org.apache.struts.util.RequestUtils.getModulePrefixes, org.apache.struts.util.ModuleUtils.getModulePrefixes

Moved from file RequestUtils to file TagUtils

- ✓ MovedMethods, org.apache.struts.util.RequestUtils.computeParameters, org.apache.struts.taglib.TagUtils.computeParameters
- ✓ MovedMethods, org.apache.struts.util.RequestUtils.computeURL, org.apache.struts.taglib.TagUtils.computeURLWithCharEncoding
- ✓ MovedMethods, org.apache.struts.util.RequestUtils.computeURL, org.apache.struts.taglib.TagUtils.computeURL
- ✓ MovedMethods, org.apache.struts.util.RequestUtils.computeURL, org.apache.struts.taglib.TagUtils.computeURL
- ✓ MovedMethods, org.apache.struts.util.RequestUtils.getActionMappingName, org.apache.struts.taglib.TagUtils.getActionMappingName
- ✓ MovedMethods, org.apache.struts.util.RequestUtils.getActionMappingURL, org.apache.struts.taglib.TagUtils.getActionMappingURL
- ✓ MovedMethods, org.apache.struts.util.RequestUtils.lookup, org.apache.struts.taglib.TagUtils.lookup
- ✓ MovedMethods, org.apache.struts.util.RequestUtils.getScope, org.apache.struts.taglib.TagUtils.getScope
- ✓ MovedMethods, org.apache.struts.util.RequestUtils.lookup, org.apache.struts.taglib.TagUtils.lookup
- ✓ MovedMethods, org.apache.struts.util.RequestUtils.message, org.apache.struts.taglib.TagUtils.message
- ✓ MovedMethods, org.apache.struts.util.RequestUtils.message, org.apache.struts.taglib.TagUtils.message
- ✓ MovedMethods, org.apache.struts.util.RequestUtils.present, org.apache.struts.taglib.TagUtils.present
- ✓ MovedMethods, org.apache.struts.util.RequestUtils.pageURL, org.apache.struts.taglib.TagUtils.pageURL
- ✓ MovedMethods, org.apache.struts.util.RequestUtils.saveException, org.apache.struts.taglib.TagUtils.saveException
- ✓ MovedMethods, org.apache.struts.util.RequestUtils.getModuleConfig, org.apache.struts.taglib.TagUtils.getModuleConfig
- ✓ MovedMethods, org.apache.struts.util.RequestUtils.getActionMessages, org.apache.struts.taglib.TagUtils.getActionMessages
- ✓ MovedMethods, org.apache.struts.util.RequestUtils.encodeURL, org.apache.struts.taglib.TagUtils.encodeURL
- ✓ MovedMethods, org.apache.struts.util.RequestUtils.isXhtml, org.apache.struts.taglib.TagUtils.isXhtml

Moved from file ResponseUtils to file TagUtils

- ✓ MovedMethods, org.apache.struts.util.ResponseUtils.write, org.apache.struts.taglib.TagUtils.write
- ✓ MovedMethods, org.apache.struts.util.ResponseUtils.filter, org.apache.struts.taglib.TagUtils.filter
- ✓ MovedMethods, org.apache.struts.util.ResponseUtils.writePrevious, org.apache.struts.taglib.TagUtils.writePrevious

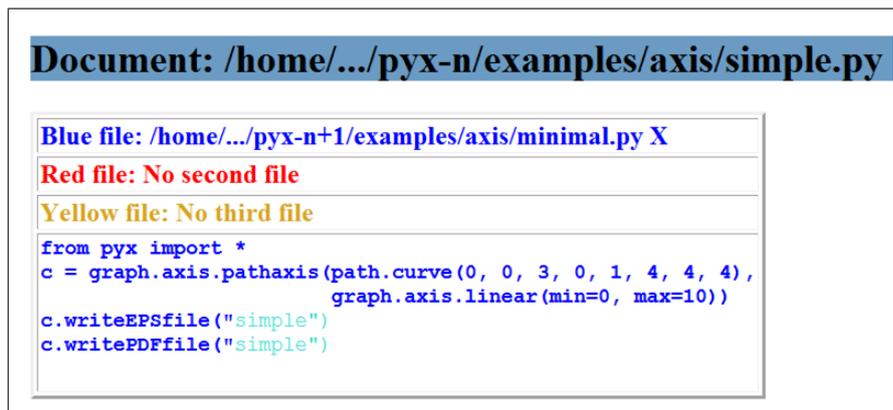
Moved from file ActionMapping to file ActionConfig

- ✓ PulledUpMethods, org.apache.struts.action.ActionMapping.findException, org.apache.struts.config.ActionConfig.findException

Appendix Q

PyX: matched and unmatched disappearing files

In this appendix the matched and unmatched files from the PyX project are listed, in the same format as those for PostgreSQL and DNSjava in Appendix N. As an example of a file which is matched to another with similarity close to the threshold, the comparison between `minimal.py` and `simple.py` is shown in Figure Q.1.



Document: /home/.../pyx-n/examples/axis/simple.py

Blue file: /home/.../pyx-n+1/examples/axis/minimal.py X

Red file: No second file

Yellow file: No third file

```
from pyx import *
c = graph.axis.pathaxis(path.curve(0, 0, 3, 0, 1, 4, 4, 4),
                        graph.axis.linear(min=0, max=10))
c.writeEPSfile("simple")
c.writePDFfile("simple")
```

Figure Q.1: The disappearing file `minimal.c` matched to `simple.py`, similarity 0.89

428 APPENDIX Q. PYX: MATCHED AND UNMATCHED DISAPPEARING FILES

Ref.	File	Match	Sim.
	Matched files		
3	/pyx/lfs/createlfs.c	/pyx/data/lfs/createlfs.c	1.00
4	/manual/palettename.c	/manual/gradientname.c	0.94
5†	/examples/axis/simple.c	/examples/axis/minimal.c	0.89
5	/examples/bitmap/julia.c	/gallery/misc/julia.c	1.00
5	/examples/graphs/arrows.c	/gallery/graphs/arrows.c	1.00
5	/examples/graphs/errorbar.c	/examples/graphstyles/errorbar.c	1.00
5	/examples/graphs/histogram.c	/examples/graphstyles/histogram.c	0.93
5	/examples/graphs/inset.c	/gallery/graphs/inset.c	1.00
5	/examples/graphs/integral.c	/gallery/graphs/integral.c	1.00
5	/examples/graphs/link.c	/examples/axis/link.c	0.92
5	/examples/graphs/mandel.c	/gallery/graphs/mandel.c	1.00
5	/examples/graphs/manyaxes.c	/gallery/graphs/manyaxes.c	1.00
5	/examples/graphs/partialfill.c	/gallery/graphs/partialfill.c	1.00
5	/examples/graphs/piaxis.c	/gallery/graphs/piaxis.c	1.00
5	/examples/graphs/symbolline.c	/gallery/graphs/symbolline.c	1.00
5	/examples/graphs/washboard.c	/gallery/graphs/washboard.c	1.00
5	/examples/misc/box.c	/gallery/misc/box.c	1.00
5	/examples/misc/connect.c	/gallery/misc/connect.c	0.91
5	/examples/misc/pattern.c	/gallery/misc/pattern.c	1.00
5	/examples/misc/vector.c	/gallery/misc/vector.c	1.00
5	/examples/path/circles.c	/gallery/path/circles.c	0.96
5	/examples/path/sierpinski.c	/gallery/path/sierpinski.c	1.00
5	/examples/path/tree.c	/gallery/path/tree.c	1.00
5	/examples/splitgraphs/shift.c	/gallery/graphs/shift.c	1.00
6	/examples/misc/bitmap.c	/manual/bitmap.c	0.87
6	/examples/misc/latex.c	/examples/text/textrunner.c	0.90
7	/examples/box.c	/examples/misc/box.c	0.91
7	/examples/connect.c	/examples/misc/connect.c	1.00
7	/examples/latex.c	/examples/misc/latex.c	1.00
7	/examples/pattern.c	/examples/misc/pattern.c	1.00
7	/examples/sierpinski.c	/examples/path/sierpinski.c	1.00
7	/examples/tree.c	/examples/path/tree.c	1.00
7	/examples/valign.c	/examples/misc/valign.c	1.00
7	/examples/vector.c	/examples/misc/vector.c	1.00
	Unmatched files		
3	pyx/pykpathsea/_init...c		
3	pyx/siteconfig.c		
5	pyx/helper.c		
5	pyx/t1strip/_init...c		
5	pyx/t1strip/fullfont.c		
6	examples/examples.c		
6	pyx/base.c		
6	pyx/tex.c		

Table Q.1: Matched and unmatched disappearing files: PyX project.
†See Figure Q.1.

Appendix R

Comparison between Moss and 3CO

In Chapter 2, Moss's [3] matching technique is discussed. As previously noted, Moss' exact matching algorithm is unknown. However, looking Figure R.1, Moss appears to exclude header information from its calculations, and to match code using relaxed parameters, on a "first come, first served" basis, like a greedy parameterised clone detection tool, when used on this code.

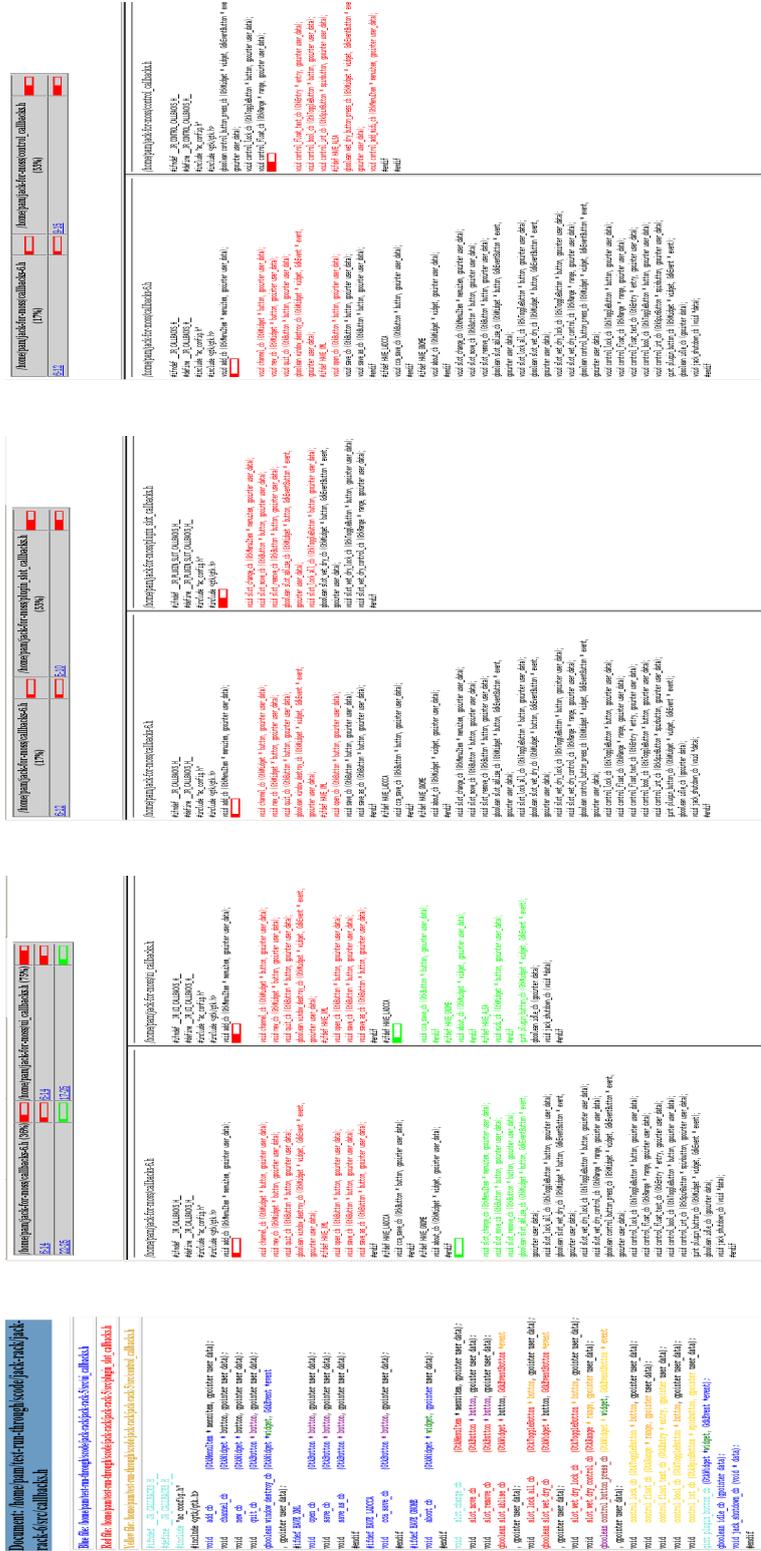


Figure R.1: 3CO and Moss comparisons between callbacks.h (blue file), plugin_slot_callbacks.h (red file) and ctrl_callbacks.h (yellow file). The Moss comparisons are each between callbacks.h and one other file, noted in the caption by the colour it is given by 3CO. Moss' matching algorithm gives different results to the textual matching of Ferret.

Appendix S

Selected Features

The features selected by the Simple Logistic algorithm from the fc+tris-singles feature set are listed in Table S.1, and those from the fl+tris-singles set in Table S.2. There are five selections for each set: for split files, for the three classes for disappearing files with 2 or more target files, and for the disappearing files with one target. In the second table, the features which are common to the two feature sets are highlighted with coloured backgrounds.

The F1 and F2 columns in the table show the 2 (or 3) files compared. The third column, headed 'To' shows the file in which the measure is taken or the file to which the measure is proportional. For example, in Table S.2, looking at the first two XML based features: the first takes the blocks of code shared by files 1 and 2 and measures the mean number of lines in the blocks in file 1; and the second feature is the ratio of the lines shared by files 1 and 2, to the number of lines in file 1.

Although around two-thirds of the features are proportional, a wide range of features is selected. All of the block-based measures, laid out in Table 13.8 (p.217), are represented in the fc+tris set except for the counts of the blocks (item 1) and of the number of units in the blocks (item 2).

F1	F2	To	P1	Description	Spl 0,1	2+ Dis 0	1	2	1 Dis 0,1
Trigrams									
1	2		-	Trigrams shared by files		1			
1	3		-	Trigrams shared by files	1				
1	2,3	1	✓	Trigrams shared by $1 \wedge (2 \text{ XOR } 3)$ to total	1	1			
1	2,3	1	✓	Trigrams unique to the file to total		1			
2	1,3	2	✓	Trigrams unique to the file to total		1			
1	2	1	✓	Shared trigrams to total (containment)				1	
1	3	1	✓	Shared trigrams to total (containment)	1				
1	3	3	✓	Shared trigrams to total (containment)	1	1		1	
2	3	3	✓	Shared trigrams to total (containment)	1				
2	1,3	2	✓	Trigrams shared by all 3 files to total	1		1		
1	2,3	1	✓	Trigrams shared by other files to total			1		
Sundries									
				File type (applies to C code)	1				
				Number of target files selected	1			1	
XML block characters									
1			-	Number of characters in the file			1		
3			-	Number of characters in the file	1				
1	2	1	✓	Characters in largest shared block to total	1				
1	2	1	✓	Mean shared characters in blocks to total		1			
1	2	1	✓	Shared characters to total (containment)					1
1	2	1	✓	Characters in blocks ≥ 40 characters to all characters			1		1
1	2	1	✓	Characters in blocks ≥ 80 characters to all characters		1			
1	2	1	-	Characters in blocks $\geq \frac{1}{32}$ file size				1	
1	2	1	✓	Characters in blocks $\geq \frac{1}{16}$ file size to all characters	1				
1	2	1	-	Characters in blocks $\geq \frac{1}{4}$ file size				1	
1	2	2	✓	Mean shared characters in blocks to total			1		
1	2	2	-	Mean shared characters in blocks	1				
1	2	2	✓	Shared characters to total (containment)					1
1	2	2	✓	Characters in blocks ≥ 40 characters to all characters				1	
1	2	2	✓	Characters in blocks ≥ 640 characters to all characters	1				
1	3	1	✓	Characters in largest shared block to total			1		
1	3	1	✓	Characters in blocks ≥ 40 characters to all characters				1	
1	3	1	✓	Characters in blocks ≥ 80 characters to all characters	1				
1	3	1	✓	Characters in blocks ≥ 160 characters to all characters	1				
1	3	1	✓	Characters in blocks ≥ 640 characters to all characters			1	1	
1	3	1	-	Characters in blocks $\geq \frac{1}{64}$ file size		1			
1	3	1	-	Characters in blocks $\geq \frac{1}{16}$ file size	1				
1	3	3	✓	Mean shared characters in blocks to total	1				
1	3	3	✓	Shared characters to spread		1			
1	3	3	-	Number of characters in shared blocks ≥ 160 characters	1		1		
1	3	3	✓	Characters in blocks ≥ 40 characters to all characters				1	
2	3	2	✓	Shared characters to spread	1				
2	3	2	✓	Characters in blocks ≥ 320 characters to all characters	1				
2	3	2	-	Characters in blocks $\geq \frac{1}{4}$ file size			1		
2	3	3	-	Size of the largest shared block	1				
2	3	3	✓	Shared characters to spread	1				
2	3	3	-	Characters in blocks $\geq \frac{1}{8}$ file size	1				
2	3	3	✓	Characters in blocks $\geq \frac{1}{32}$ to all characters	1				
2	3	3	✓	Characters in blocks $\geq \frac{1}{16}$ to all characters		1	1		

Table S.1: fc+tris-singles features selected by Simple Logistic. The first 2 columns show the files in the comparison, the next the base file for the measures, which, if proportional, have a tick in the next column. The last 4 columns show in which model the features are used: split, disappearing files with 2 or more targets, or 1 target.

F1	F2	To	P'1	Description	Spl 0,1	2+ Dis 0	1	2	1 Dis 0,1
Trigrams									
1	2		-	Trigrams shared by files		1			
1	3		-	Trigrams shared by files	1				
1	2,3	1	✓	Trigrams shared by 1 ^ (2 XOR 3) to total	1				
1	2,3	1	✓	Trigrams unique to the file to total		1			1
2	1,3	2	✓	Trigrams unique to the file to total					1
1	2	2	✓	Shared trigrams to total (containment)				1	
1	3	1	✓	Shared trigrams to total (containment)	1				
1	3	3	✓	Shared trigrams to total (containment)	1	1		1	
1	2,3	1	✓	Trigrams shared by other files to total			1		
Sundries									
			-	File type (applies to C code)	1	1			
			-	Number of target files selected	1				
XML blocks line-based									
1	2	1	-	Mean shared lines in blocks		1			
1	2	1	✓	Shared lines to total (containment)			1		
1	2	1	✓	Lines in blocks ≥ 4 lines to all lines		1			
1	2	1	✓	Lines in blocks $\geq \frac{1}{8}$ to all lines	1				
1	2	1	-	Lines in blocks $\geq \frac{1}{32}$ file size				1	
1	2	1	-	Lines in blocks $\geq \frac{1}{16}$ file size				1	
1	2	2	-	Lines in blocks $\geq \frac{1}{16}$ file size			1		
1	2	2	✓	Shared lines to spread				1	
1	2	2	✓	Lines in blocks ≥ 2 lines to all lines		1			
1	2	2	✓	Lines in blocks ≥ 8 lines to all lines				1	
1	3	1	-	Lines in blocks $\geq \frac{1}{16}$ file size		1			
1	3	1	✓	Lines in blocks ≥ 4 lines to all lines	1				
1	3	1	✓	Lines in blocks ≥ 8 lines to all lines	1				
1	3	1	✓	Lines in blocks ≥ 32 lines to all lines			1		
1	3	1	✓	Lines in blocks $\geq \frac{1}{16}$ to total lines			1		
1	3	3	-	Lines in shared blocks to total	1				
1	3	3	✓	Lines in blocks ≥ 32 lines to all lines			1		
2	3	2	-	Lines in blocks $\geq \frac{1}{8}$ file size	1				
2	3	3	✓	Shared lines to spread	1				
2	3	3	✓	Shared lines to total (containment)			1		
2	3	3	✓	Lines in blocks $\geq \frac{1}{16}$ to total lines	1				

Table S.2: fl+tris-singles features selected by Simple Logistic. The first 2 columns show the files in the comparison, the next the base file for the measures, which, if proportional, have a tick in the next column. The last 4 columns show in which model the features are used: split, disappearing files with 2 or more targets, or 1 target. Features marked by a coloured background appear in both this selection, and that from fc+tris-singles.

Appendix T

Comparing related feature sets

The feature sets based on P-Duplo (pdp) and the line-based XML analysis (fl) should be similar, as each looks at the similarity between lines of code. One difference is that P-Duplo requires the whole line to match, whereas line-based analysis requires only that trigrams in a line appear elsewhere in the file. Also, the way that the lines are counted differs (see Chapters 6 and 11).

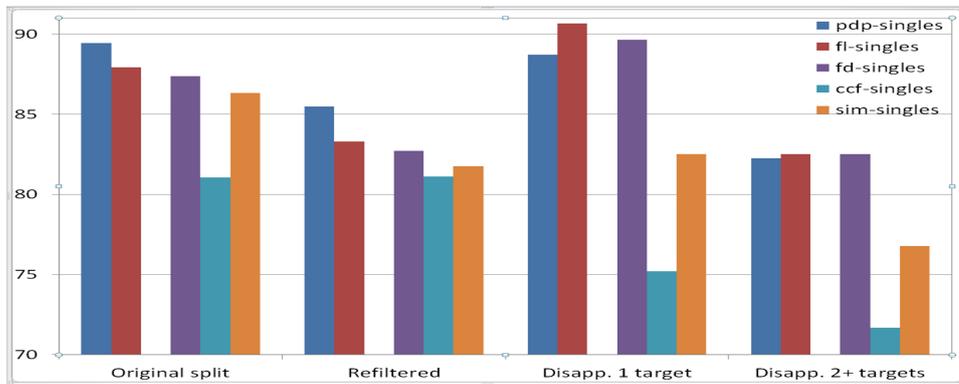
To some extent, the XML density analysis (fd) should find similar blocks of code to Code Clone Finder (ccf) and Simian (sim), in that all find “gapped” copies. The differences are that CCFinder and Simian exclude parts of the code and that the minimum sizes of matching section of code vary.

Table T.1 gives the classification rates for each of these feature sets, with the minimum and maximum for all feature sets in the last two columns. The graphs in Figure T.1 show the classification rates for each of the 5 feature sets, both for the singles (or block-singles) sets and for cat (or block-cats) sets, as these sets all have features derived from matched blocks in the same way, varying only in the method of comparing the files to find these blocks.

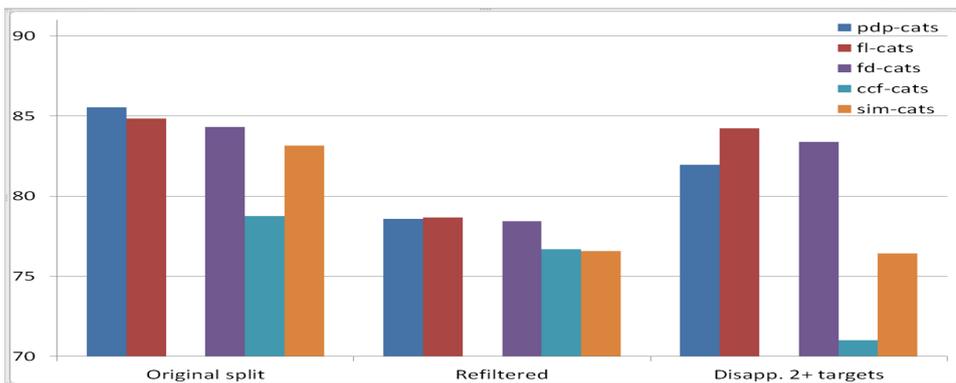
The graphs show that P-Duplo and line-based XML analysis give similar results. The other three sets vary, with the density analysis being more like the P-Duplo and line-based results than the CCFinder and Simian results. These direct comparisons show that with these features the clone detection tools are less suited to classifying restructured files than the other two tools.

	pdp	fl	fd	ccf	sim	Min	Max
Singles							
Original split	89.43	87.92	87.39	81.05	86.33	75.49	92.11
Refiltered	85.49	83.30	82.73	81.13	81.77	72.63	88.70
Disapp. 1 target	88.72	90.66	89.66	75.21	82.52	72.33	93.89
Disapp. 2+ targets	82.25	82.51	82.51	71.67	76.79	68.43	86.02
Concatenations							
Original split	85.55	84.83	84.31	78.75	83.14	75.49	92.11
Refiltered	78.58	78.66	78.43	76.69	76.57	72.63	88.70
Disapp. 2+ targets	81.97	84.24	83.40	71.02	76.42	68.43	86.02

Table T.1: Comparison of classification accuracy of related feature sets



(a) 3-way single file comparisons



(b) Concatenated file comparisons

Figure T.1: Classification with related feature sets: P-Duplo and line-based; density, CCFinder and Simian. The features are block based: the top graph based on single file, and the bottom on concatenated file, comparisons.