Analysing Ferret XML Reports to Estimate the Density of Copied Code

Pam Green
Peter C.R. Lane
Austen Rainer
Sven-Bodo Scholz

April 2010

Technical Report No. 501

School of Computer Science
University of Hertfordshire

Contents

List of Figures

List of Tables

Abstract

This document explains a method for identifying dense blocks of copied text in pairs of files. The files are compared using Ferret, a copy-detection tool which computes a similarity score based on trigrams. This similarity score cannot determine the arrangement of copied text in the file; two files with the same similarity to another file may have different distributions of matched trigrams in the file. For example, in one file the matched trigrams may be in a large block, while they are scattered throughout the other file. However, Ferret produces an XML report which relates matched and unmatched trigrams back to the original text. This report can be analysed to find identical or densely copied blocks in the files. We address the problems of defining and locating the blocks, and of representing the blocks found as a meaningful feature vector, regardless of copy pattern. We provide a step-by-step example to explain our method for finding dense blocks. A set of artificial files, built to mimic different copy patterns, is used to explore a set of features which profile the dense blocks in a file. A range of density parameters is used to construct features which show that the copy patterns in the artificial files can be separated.

1 Introduction

The Ferret copy-detection tool computes the similarity between two or more files either of text or of program code. The similarity measure is based on trigrams of words or tokens. The tool also produces a number of reports which can be further analysed. In some applications it is useful to find blocks of text which have been copied but also undergone minor alterations, interrupting an otherwise unbroken sequence of copied tokens or words. We use the term "dense blocks" to describe sections of a file which contain a mixture of matched and unmatched trigrams of words or tokens, the majority of which are matched. Dense blocks usually result from a copy-paste-edit sequence. This document explains a method, which we call density analysis, for using the XML report output by Ferret to locate these blocks.

Ferret is described in Section 2, density analysis in Section 3, and the features we construct based on this density information for input to a machine learning system in Section 4.3. Experiments with constructed files are described in Section 4, and a summary follows in Section 5.

2 Background

The Ferret copy-detection tool is described in Section 2.1; two example files are used to explain how the tool works, and to provide an illustration of the Ferret XML output. Copy patterns are introduced in Section 2.2.

2.1 The Ferret copy-detection tool

Ferret [2, 3, 4, 6] was developed by the UH Plagiarism Detection Group as an efficient method for measuring duplicated text. Using distinct trigrams (three consecutive words), a similarity measure between two documents is calculated using the Jaccard coefficient or resemblance measure [5, 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 indicates no matching trigrams and therefore unmatched text, while a score of 1 shows that all of the trigrams in one text also occur in the other. Although, in theory, a score of 1 does not necessarily mean that the two texts are

File-copy, version c.	File-copy, version x.

`#include <stdio.h>`
 	
`main()`
`{`
`   int c;`
 	
`   while((c = getchar()) != EOF);`
`      putchar(c);`
`}`
	`#include <stdio.h>`
	
	`main()`
	`{`
	`   int x;`
	
	`   while((x = getchar()) != EOF)`
	`      putchar(x);`
	`}`

Table 1: Two sample files which are identical except for the identifier name change, from ‘c’ to ‘x’.

identical, because trigram frequency is not measured; in practice two texts with a similarity score of 1 are unlikely to be different.

Ferret has also been adapted for use with program code, with lexical tokens used in place of words. It takes source code files as input, the only preprocessing being the tokenisation of the text, where whitespace and formatting become irrelevant. Efficiency is achieved by storing file identity numbers against matched trigrams, with new trigrams added as they are encountered.

A study by Rainer et al. [6] shows that although some language specific trigrams (such as “ `) ; }` “ or ” `i = 1` ”) occur frequently, the frequency of trigrams in program code follows a similar Zipfian, or negative exponential, distribution to that of natural language sequences [6, Figure 1]. This distribution of trigrams in code indicates that Ferret is likely to be as effective in matching program texts as it is in matching document texts.

Two small example files are given in Table 1 to provide a basis for explanation of the similarity measurement. The files are identical, except that the identifier name is changed from ‘c’ to ‘x’. To illustrate how Ferret computes a similarity score for program code, the trigrams for version c of the file copy program are shown in Table 2. There are 30 trigrams, 21 are shared by the two versions of the code, versions c and x. The trigrams which appear in both files have their numbers highlighted in bold. The similarity score between the two files is $\frac{21}{39} = 0.538$.

Ferret provides several choices of output which include: a list of pairwise similarity scores, with summary trigram information for the files; the trigram list; and an XML report which separates blocks of code covered by trigrams which occur in the other file, from blocks with no matching trigrams. Each of the outputs can be analysed to describe an aspect of the relationship between the files being compared.

The simplest output from Ferret is a list of each pair of files compared, giving the file names, the number of trigrams in each file and shared by the files, and the similarity score. Another output, the trigram report, lists every trigram together with the identity numbers of the files in which it appears.

Neither the trigram analysis nor the basic statistics output by Ferret provide information about the distribution of copied code in a file. Two files can have the same similarity to another file, but one of these files may have one or more large blocks of matched trigrams, while the other file has the matched trigrams scattered throughout the file, (see Figures 2(b) and 2(e), p.7). Large blocks are more likely to be the result of deliberate copying, while scattered trigrams are likely to be due to incidental similarity. The XML report of similarity between two files relates matched and non-matched trigrams back to the source files. It shows the alternating blocks of code covered by trigrams present in the other file, and blocks with no matching trigrams in the other file. These blocks can be used to provide a profile of the matched code distribution in a file. In the remainder of this report, the tokens or code covered by matched trigrams are referred to as copied, and unmatched tokens or code as non-copied.

1	#	include	<	11	{	int	c	**21**	(	)	)
2	include	<	stdio	12	int	c	;	**22**	)	)	!=
3	<	stdio	.	13	c	;	while	**23**	)	!=	EOF
4	stdio	.	h	**14**	;	while	(	**24**	!=	EOF	)
5	.	h	>	**15**	while	(	(	**25**	EOF	)	putchar
6	h	>	main	16	(	(	c	**26**	)	putchar	(
7	>	main	(	17	(	c	=	27	putchar	(	c
8	main	(	)	18	c	=	getchar	28	(	c	)
9	(	)	{	**19**	=	getchar	(	29	c	)	;
10	)	{	int	**20**	getchar	(	)	**30**	)	;	}

Table 2: Trigrams for file-copy code, see Table 1.

Analysis of the alternating sections of copied and non-copied code may give a distorted picture of the distribution in a file of copied code which has undergone small changes. Where several sections of copied code are interspersed with a few tokens of non-copied code, each section will be accounted for separately. This would be the case where identifiers have been renamed, such as in the example shown in Table 1, where the four copied sections are each separated by one token. The sections of copied and non-copied tokens are shown in Table 3.

Another way to represent the distribution of the copied code is to find consecutive sections of copied code broken by short sections of non-copied code, which may be the result of minor editing. Our method for finding groups of sections which may belong together, by analysing the density of copied code in the sequences taken from the XML report, is described in the next section.

Figure 1 shows the Ferret XML report produced by comparing the two files shown in Table 1. The pattern of copied and non-copied (tagged as 'normal') code is extracted from this report. The report used as an example here is the output from comparisons between a pair of C code files, but Ferret can be used with other programming languages and with text. The method described can be used with any suitable alternating sequence of numbers.

The XML report can be analysed to give the number of units in each copied or non-copied section of the files. The pattern of copied and non-copied code can be represented in a number of ways. For clarity, in this explanation, the number of units in the sequence of copied and non-copied blocks, (n1, n2, n3, n4, ...), are paired with the labels c (copied) or n (not), ((c n1)(n n2)(c n3)(n n4) ...). The copy pattern for the example files is ((c 12) (n 1) (c 4) (n 1) (c 11) (n 1) (c 3)), see Table 3. As these files are almost identical, the copy pattern is the same for both files; this is not usually the case when files are compared. In this pattern the units are tokens; however, lines of code, words or characters might be used in different applications.

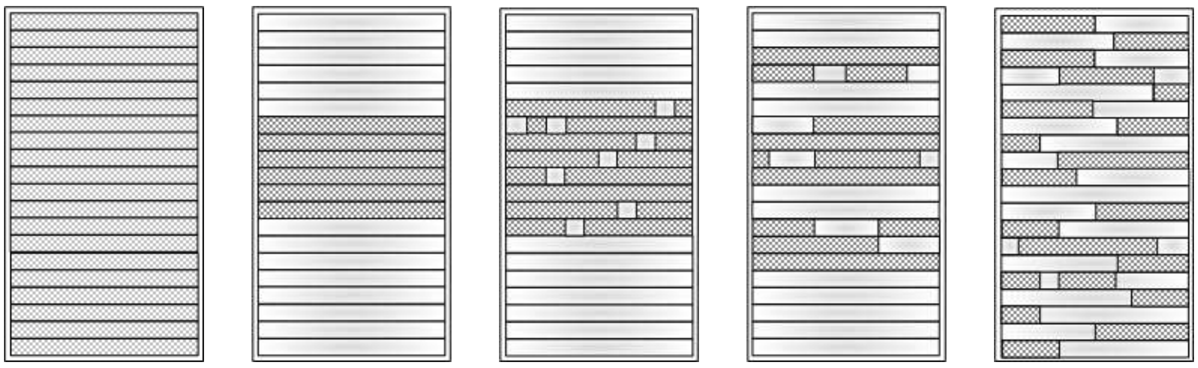
No.of	Copied	Tokens in section											
tokens	or not	1	2	3	4	5	6	7	8	9	10	11	12
12	c	#	include	<	stdio	.	h	>	main	(	)	{	int
1	n	c											
4	c	;	while	(	(								
1	n	c											
11	c	=	getchar	(	)	)	!	=	EOF	)	putchar	(	
1	n	c											
3	c	)	;	}									

Table 3: The tokens in each section of copied or non-copied code from the XML report (see Figure 1)

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="uhferret.xsl" ?>
<uhferret>
<common-trigrams>21</common-trigrams>
<similarity>0.538462</similarity>

<document>
<source>/home/pam/file-copy-vc.c</source>
<num-trigrams>30</num-trigrams>

<text>
<block text="copied"><![CDATA[#include <stdio.h>

main()
{
    int ]]></block>
<block text="normal"><![CDATA[c]]></block>
<block text="copied"><![CDATA[;
while ((]]></block>
<block text="normal"><![CDATA[c]]></block>
<block text="copied"><![CDATA[= getchar()) ! = EOF)
putchar (]]></block>
<block text="normal"><![CDATA[c]]></block>
<block text="copied"><![CDATA[);
}]]></block>

</text>
</document>

<document>
<source>/home/pam/file-copy-vx.c</source>
<num-trigrams>30</num-trigrams>

<text>
<block text="copied"><![CDATA[#include <stdio.h>

main()
{
    int ]]></block>
<block text="normal"><![CDATA[x]]></block>
<block text="copied"><![CDATA[;
while ((]]></block>
<block text="normal"><![CDATA[x]]></block>
<block text="copied"><![CDATA[= getchar()) ! = EOF)
putchar (]]></block>
<block text="normal"><![CDATA[x]]></block>
<block text="copied"><![CDATA[);
}]]></block>

</text>
</document>
</uhferret>
```

Figure 1: Ferret XML output for a comparison between the two sample files shown in Table 1 (p. 4)

2.2 Copy patterns



(a) Original file (b) Copied block (c) Block with new identifiers (d) Copy and edit (e) Incidentally similar

Figure 2: Figure (a) represents an original file, the other four figures, (b)-(e), represent different copy patterns, with the copied code shaded the same as the original file. The four files with copied code have approximately the same similarity to the original file.

The idea that different distributions of copied code may have the same similarity score was introduced in Section 1. Examples of four different copy patterns are given in Figure 2. The file on the left, 2(a), is the original file. Each of the other files have some code in common with the original file, which is shaded with a dark cross-hatched pattern. The four different types of copy pattern are shown in the four files in Figures 2(b)-2(e). Assuming that there are few repeated trigrams within the files, comparisons between each of the files and the original file will result in approximately the same similarity score.

Figure 2(b) represents a block of code copied directly to another file, for example, when a function is reused or moved from one file to another. Figure 2(c) depicts similar copying, but shows small changes to the copied code. For example, when the code appears in another file, identifiers may need to be renamed to suit the new location. Figure 2(d) shows the type of copy pattern which may result from a copy, paste and edit, perhaps where a function has been moved or used as a template for a new function, and then amended. The last figure, 2(e), represents a possible 'copy' pattern which results from incidental similarity, where commonly-used programming constructs are repeated, and where identifiers and function calls are similar because the two files belong to the same project.

In one application [1], we combine similarity measures from copy-detection tools to trace changing files. Dense blocks provide one source of similarity information. Finding these blocks in code is useful when comparing files to find sections which have been moved and edited. The aim is to characterise similar code in a file, in particular for machine learning. There are two problems here: first, to find a way to measure the density and distribution of dense blocks, and second, to represent the measurement(s) as a meaningful and preferably compact vector, regardless of file size or copy pattern.

3 Density analysis

The terms used in the rest of the document are defined in Section 3.1. Following this, two methods for finding dense blocks are described; a simple bottom-up scan in Section 3.2, and a top-down method based on decreasing sliding windows in Section 3.3. In section 3.4, the ideas introduced in this section are related to the copy patterns shown in Figure 2.

3.1 Terms

The density of 'copied code' can be measured in different units. For example, characters, tokens or lines may be used for program code text; while for other text, characters, words or sentences may be chosen. To generalise the description, the term 'unit' is used. Other terms used to describe the copy patterns in a file are depicted in Figure 3.

A *sequence* is the representation of the copied and non-copied code (or text) in a file, for example, ((c 6)(n 3)(c 15)(n 25)(c 20)(n 1)(c 10)). The word *section* is used to describe each uninterrupted sub-sequence of copied or non-copied units, for example, (c 6) which could be a statement such as " a = b * 5 ; ", with six tokens which have matched trigrams in the other file. A *block* is a part of a sequence, used here when describing a dense subsequence, for example, the blocks ((c 6)(n 3)(c 15)) or ((c 20)(n 1)(c 10)).

To determine which blocks of code are dense enough to be considered the result of deliberate copying, the required density has to be defined. The simplest density measure for a block of code is the proportion of copied to total units in the block. For example, the block ((c 6)(n 3)(c 15)) has density $\frac{6+15}{6+3+15} = \frac{21}{24} = 0.875$. In the examples given here, a block with a density of at least 0.8 is considered to be dense enough to be of interest.

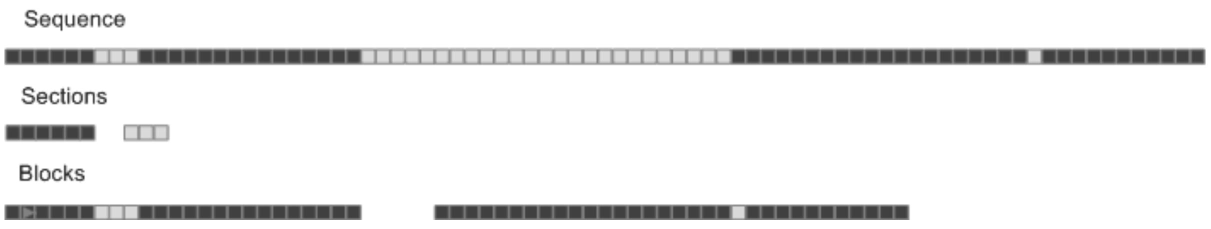


Figure 3: The dark squares represent copied units and the pale squares, non-copied units. The *sequence* ((c 6)(n 3)(c 15)(n 25)(c 20)(n 1)(c 10)) is depicted at the top of the diagram. In the middle there are two *sections* from the sequence, (c 6) and (n 3), and, at the bottom of the diagram, two *blocks* from the sequence, ((c 6)(n 3)(c 15)) and ((c 20)(n 1)(c 10)).

3.2 Method 1: Bottom-up scan

One simple way to find dense blocks is to scan a sequence starting at the first copied section. As all of the code in this single section block is copied, the block density is 1.0. Further pairs of non-copied and copied sections can be added to the sequence if this does not reduce the density below the given threshold, otherwise a new block starts at the next copied section. For example, if the sequence ((c 6)(n 3)(c 15)... continued ...(n 2)(c 20)), the new density would be $\frac{41}{46} = 0.89$, and the two sections would be added to the block; however, as the sequence continues ...(n 25)(c 20)), the new density is $\frac{41}{70} = 0.586$, therefore the first block ends and a new block starts at (c 20).

There are three problems with this simple density measure: First, density alone does not provide any indication of the amount of copied code in a block. If a block has just one section in it then the density is 1.0. For example, blocks such as (c 10) are unlikely to represent potentially interesting blocks if the unit is tokens, but may be interesting if the unit is lines of code. A parameter providing a required minimum number of units in a block will eliminate small blocks and can be set to a value suited to the unit.

Second, large sections of non-copied code may be absorbed into a sequence which has large copied sections, when the non-copied code should be excluded. For example, the block ((c 130)(n 5)(c 100)(n 2)(c 175)(n 3)(c 200)(n 5)(c 125)(n 5)(c 250)) which has density $\frac{980}{1000} = 0.98$ may be followed by the two sections (n 480)(c 1020), which gives a new density $\frac{2020}{2500} = 0.808$ and is therefore dense enough. A more natural interpretation of this sequence is as two blocks of copied code, one of ((c 130)(n 5)(c 100)(n 2)(c 175)(n 3)(c 200)(n 5)(c 125)(n 5)(c 250)) and one of a single copied section of 1020 units. This can be achieved by setting a maximum number of non-copied units allowed in one section within a block, so that the block ends wherever a non-copied section exceeds the maximum number of units permitted.

Lastly, it is difficult to determine from density alone whether a fragmented block, such as ((c 4)(n 1)(c 5)(n 1)(c 3)(n 2)(c 5)(n 1)(c 4)(n 1)(c 3)) which has a density of $\frac{24}{30} = 0.8$, represents copied code with multiple small edits, or is the result of incidental similarity. One answer to this question is to assume that copied and edited code is likely to result in a larger block than code which is incidentally similar. The number of copied units within the block can be totalled, and blocks with fewer units than a specified minimum disregarded. This is similar to the solution to the first problem.

To address the problems outlined above, two parameters can be added to the simple density measure:

- the maximum units in a non-copied section and
- the minimum units in a block.

Maximum non-copied section size can be applied with the density parameter while scanning for blocks. Once built, the blocks which contain insufficient copied units can be removed.

One flaw in this method is that the forward search may miss parts of a sequence which could form the beginning of a block. For example, in the sequence ((c 20)(n 8)(c 10)(n 7)(c 30)), the (c 20) will be in one block because $\frac{30}{38} = 0.79$ and ((c 10)(n 7)(c 30)) will form another block. However, the density of the entire sequence is $\frac{60}{75} = 0.8$, so the two blocks could be joined. This problem could be resolved by using some form of bi-directional search.

Density analysis could be tackled using other bottom up methods. For example, starting with individual copied sections, adding pairs of copied and non-copied sections from one or other side, depending on the cumulative density. It may be more difficult to identify the 'best' blocks in a sequence using a bottom-up method, rather than a top-down method. Best can mean the most dense blocks, or those with the most copied code, or with the most code overall.

3.3 Method 2: Top-down sliding windows

While a bottom-up approach is possibly more suitable for a file with little copied code, a top-down approach may be more efficient for finding blocks of code in files which have a high proportion of dense blocks. We chose to develop a top-down method over a bottom-up method because we thought it would be simpler to find the densest blocks this way.

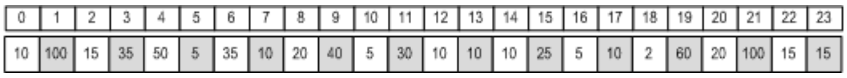
Our method is based on reducing sliding windows. First the sequence is broken into subsequences between large non-copied sections, which are removed. If these subsequences are dense enough, then no more processing is required. If not dense enough, then each subsequence is repeatedly broken into a series of windows of decreasing size until one or more of these is dense enough. The dense blocks are removed from the sequence, the windows which are overlapped by the sequences are fragmented, and the process is repeated with the remaining subsequences until all of the code is broken into dense blocks. The last stage of the method for finding dense blocks filters out those blocks with little content. Once the blocks are found, features can be built to form a profile of the number, size and density of the blocks and their relationship to the whole file.

In Figure 4 the method is outlined as a flowchart. An example of the way the algorithm works is shown in Table 5, and the steps are outlined below. Each step is described, illustrated, and numbered to correspond with the example shown in Table 5. The arbitrary parameters used in the worked example are:

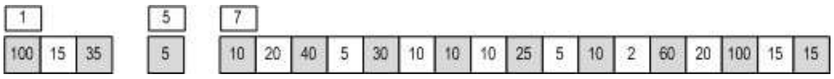
- the largest block of non-copied code permitted in a sequence – 20 units,
- the minimum density of the blocks – 0.9, and
- the minimum units of copied code in a sequence – 50 units.

In the step-by-step explanation, windows are referenced by their starting section number, for example, the window starting at section 7 is known as window 7. In the diagrams, copied sections are shown with a pale grey background and the non-copied sections with white, section numbers are displayed in the smaller white rectangles above. Blocks with dense enough copying are indicated by darker shading. Selected section numbers are shown at each step to aid identification of the windows referred to in the text.

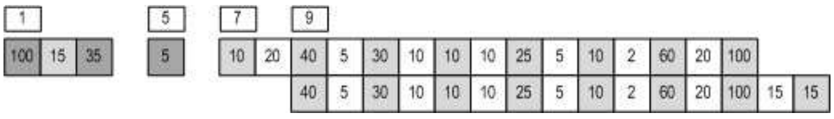
Once dense blocks are identified they are shown on the left of the diagram. The block is removed from the sequence, leaving either a sequence which is at least as long as the new window size, or smaller fragments, which are shown on the right of the diagram.



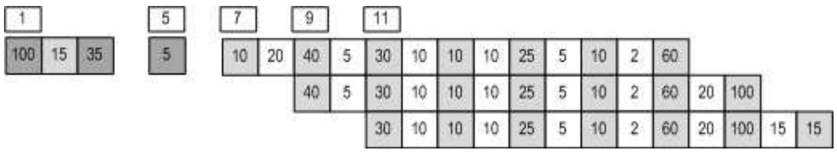
Step 1. The XML output is analysed to give the pattern of copied and non-copied sections.



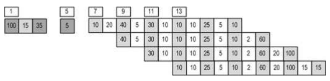
Step2. The pattern is split into subsequences where non-copied sections are larger than the given maximum size. In the example there are 2 such sections, the fourth (50) and sixth (35). Non-copied sections at either end of a sequence will not be part of a block, and are therefore dropped.

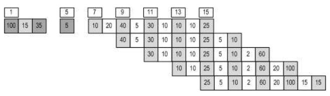


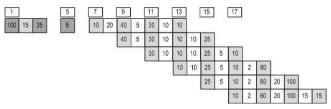
Step 3. If the subsequence density is above the minimum, it is stored. If not, 2 windows, 2 sections shorter, are formed. In the example, the first 2 subsequences are dense enough, the last is not. The 17 section window is divided into two 15 section sliding windows.



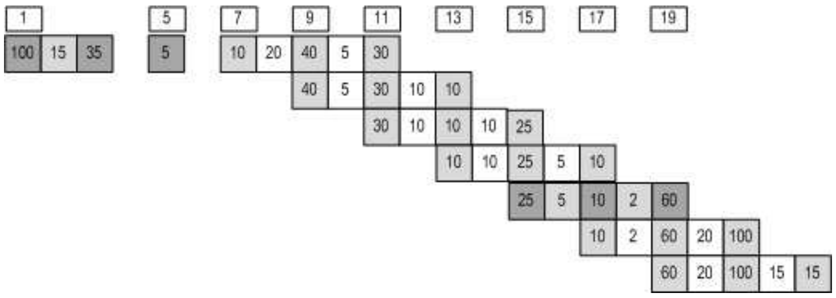
Step 4. If the windows are not dense enough, the sequence is further subdivided.
In this example, there are now 3 windows, each of 13 sections.



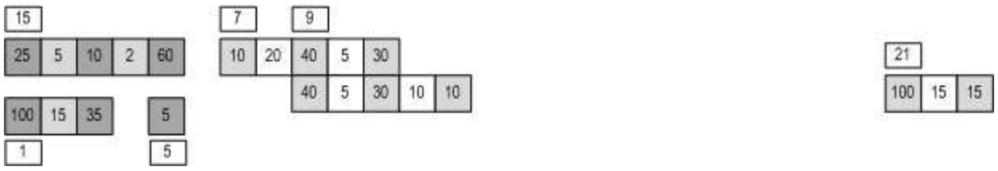




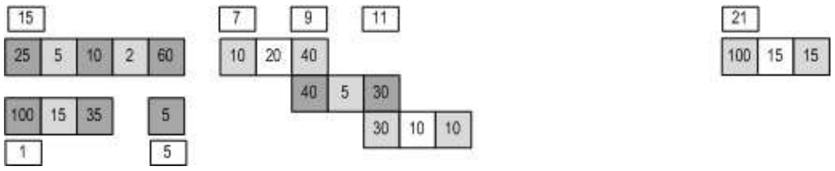
Step 5. The process is repeated ... Step 6. ... again ... Step 7. ... and again ...



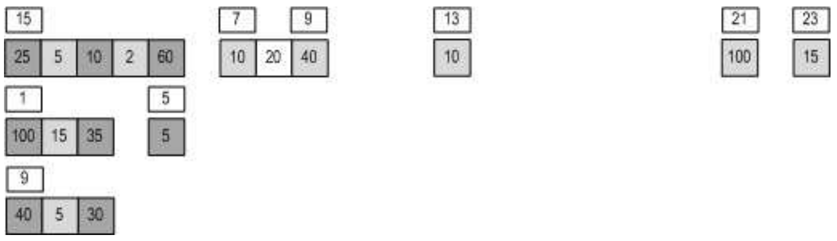
Step 8. ... until one or more windows are dense enough, as window 15 in the example, where 95 of the 102 tokens are copied, $\frac{95}{102} = 0.93$.



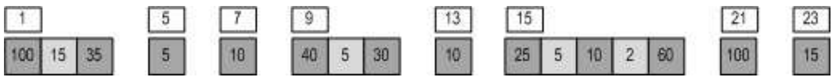
Step 9. Dense windows are stored, and their sections removed from the remaining windows. This results in some fragmented windows. In the example, window 19 shares a section with the removed window and the rest of the pattern is not contained by another block. Window 17 is not fragmented as the remaining code is covered by the new fragment 21. Similarly windows 11 and 13 have their remaining patterns covered by window 9.



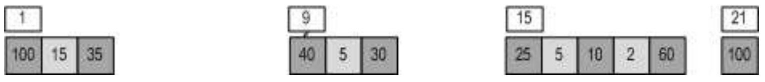
Step 10. The window size is reduced again and fragments of the same size added.
Here, all of the fragments happen to be the size of the next reduced window.
Smaller fragments remain in the pool until the same length as current windows.



Steps 11, 12. Window 9 is now dense enough, so the remaining windows are reduced or fragmented.



Step 13. The remaining windows now have only one section, and therefore density 1.0.



Step 14. Lastly, each block is checked to ensure that it contains enough copied code, resulting in four blocks, comprising sections 1-3, 9-11, 15-19 and 21.

There are other methods for tackling this problem. Small variations to the method described above can be made by changing the selection criterion. Rather than selecting from the dense windows on density, the dense window with most tokens in total, or with most copied tokens could be selected. An alternative top-down method is to remove from one or other side of a block the pair of sections which most increases the density of the remaining pattern.

The top-down method described here was designed for use with pairs of files with a moderate to high similarity score. It may be more efficient to use a bottom-up method for finding dense blocks when the similarity between the files is lower. This is a subject for future work.

Ref	Description	Len	Workings
1	Pattern		((n 10)(c 100)(n 15)(c 35)(n 50)(c 5)(n 35)(c 10)(n 20)(c 40)(n 5)(c 30)(n 10)(c 10)(n 10)(c 25)(n 5)(c 10)(n 2)(c 60)(n 20)(c 100)(n 15)(c 15))
2	3 blocks	17	(1 ((c 100)(n 15)(c 35)) 0.9) (5 ((c 5)) 1.0) (7 ((c 10)(n 20)(c 40)(n 5)(c 30)(n 10)(c 10)(n 10)(c 25)(n 5)(c 10)(n 2)(c 60)(n 20)(c 100)(n 15)(c 15)) 0.756)
3	Dense sequences New windows	15	(1 ((c 100)(n 15)(c 35)) 0.9) (5 ((c 5)) 1.0) (9 ((c 40)(n 5)(c 30)(n 10)(c 10)(n 10)(c 25)(n 5)(c 10)(n 2)(c 60)(n 20)(c 100)(n 15)(c 15)) 0.812) (7 ((c 10)(n 20)(c 40)(n 5)(c 30)(n 10)(c 10)(n 10)(c 25)(n 5)(c 10)(n 2)(c 60)(n 20)(c 100)) 0.798)
4	Reduced in size	13	(9 ((c 40) (n 5) (c 30) (n 10) (c 10) (n 10) (c 25) (n 5) (c 10) (n 2) (c 60) (n 20) (c 100)) 0.841) (11 ((c 30) (n 10) (c 10) (n 10) (c 25) (n 5) (c 10) (n 2) (c 60) (n 20) (c 100) (n 15) (c 15)) 0.801) (7 ((c 10) (n 20) (c 40) (n 5) (c 30) (n 10) (c 10) (n 10) (c 25) (n 5) (c 10) (n 2) (c 60)) 0.781)
5	again	11	(9 ((c 40) (n 5) (c 30) (n 10) (c 10) (n 10) (c 25) (n 5) (c 10) (n 2) (c 60)) 0.845) (11 ((c 30) (n 10) (c 10) (n 10) (c 25) (n 5) (c 10) (n 2) (c 60) (n 20) (c 100)) 0.833) (13 ((c 10) (n 10) (c 25) (n 5) (c 10) (n 2) (c 60) (n 20) (c 100) (n 15) (c 15)) 0.809) (7 ((c 10) (n 20) (c 40) (n 5) (c 30) (n 10) (c 10) (n 10) (c 25) (n 5) (c 10)) 0.714)
6	and again	9	(13 ((c 10) (n 10) (c 25) (n 5) (c 10) (n 2) (c 60) (n 20) (c 100)) 0.847) (11 ((c 30) (n 10) (c 10) (n 10) (c 25) (n 5) (c 10) (n 2) (c 60)) 0.833) (15 ((c 25) (n 5) (c 10) (n 2) (c 60) (n 20) (c 100) (n 15) (c 15)) 0.833) (9 ((c 40) (n 5) (c 30) (n 10) (c 10) (n 10) (c 25) (n 5) (c 10)) 0.793) (7 ((c 10) (n 20) (c 40) (n 5) (c 30) (n 10) (c 10) (n 10) (c 25)) 0.719)
7	and again	7	(15 ((c 25) (n 5) (c 10) (n 2) (c 60) (n 20) (c 100)) 0.878) (13 ((c 10) (n 10) (c 25) (n 5) (c 10) (n 2) (c 60)) 0.861) (17 ((c 10) (n 2) (c 60) (n 20) (c 100) (n 15) (c 15)) 0.833) (9 ((c 40) (n 5) (c 30) (n 10) (c 10) (n 10) (c 25)) 0.808) (11 ((c 30) (n 10) (c 10) (n 10) (c 25) (n 5) (c 10)) 0.75) (7 ((c 10) (n 20) (c 40) (n 5) (c 30) (n 10) (c 10)) 0.72)
8	and again	5	(15 ((c 25) (n 5) (c 10) (n 2) (c 60)) 0.931) (17 ((c 10) (n 2) (c 60) (n 20) (c 100)) 0.885) (9 ((c 40) (n 5) (c 30) (n 10) (c 10)) 0.842) (19 ((c 60) (n 20) (c 100) (n 15) (c 15)) 0.833) (11 ((c 30) (n 10) (c 10) (n 10) (c 25)) 0.765) (7 ((c 10) (n 20) (c 40) (n 5) (c 30)) 0.762) (13 ((c 10) (n 10) (c 25) (n 5) (c 10)) 0.75)
9	Windows Dense-windows Fragments		(9 ((c 40) (n 5) (c 30) (n 10) (c 10)) 0.842) (7 ((c 10) (n 20) (c 40) (n 5) (c 30)) 0.762) (15 ((c 25) (n 5) (c 10) (n 2) (c 60)) 0.931) (1 ((c 100)(n 15)(c 35)) 0.9) (5 ((c 5)) 1.0) (21 ((c 100) (n 15) (c 15)) 0.885)
10	Windows Dense-windows Fragments	3	(9 ((c 40) (n 5) (c 30)) 0.933) (21 ((c 100) (n 15) (c 15)) 0.885) (11 ((c 30) (n 10) (c 10)) 0.8) (7 ((c 10) (n 20) (c 40)) 0.714) (15 ((c 25) (n 5) (c 10) (n 2) (c 60)) 0.931) (1 ((c 100)(n 15)(c 35)) 0.9) (5 ((c 5)) 1.0) ( )
11	Windows Dense-windows Fragments		(21 ((c 100) (n 15) (c 15)) 0.885) (9 ((c 40) (n 5) (c 30)) 0.933) (15 ((c 25) (n 5) (c 10) (n 2) (c 60)) 0.931) (1 ((c 100)(n 15)(c 35)) 0.9) (5 ((c 5)) 1.0) (13 ((c 10)) 1.0) (7 ((c 10)) 1.0)
12	Windows Dense-windows Fragments	1	(13 ((c 10)) 1.0) (7 ((c 10)) 1.0) (21 ((c 100)) 1.0) (23 ((c 15)) 1.0) (9 ((c 40) (n 5) (c 30)) 0.933) (15 ((c 25) (n 5) (c 10) (n 2) (c 60)) 0.931) (1 ((c 100)(n 15)(c 35)) 0.9) (5 ((c 5)) 1.0) ( )
13	Windows Dense-windows Fragments	0	( ) (13 ((c 10)) 1.0) (7 ((c 10)) 1.0) (21 ((c 100)) 1.0) (23 ((c 15)) 1.0) (9 ((c 40) (n 5) (c 30)) 0.933) (15 ((c 25) (n 5) (c 10) (n 2) (c 60)) 0.931) (1 ((c 100)(n 15)(c 35)) 0.9) (5 ((c 5)) 1.0) ( )
14	Final windows		(1 ((c 100)(n 15)(c 35)) 0.9) (9 ((c 40) (n 5) (c 30)) 0.933) (15 ((c 25) (n 5) (c 10) (n 2) (c 60)) 0.931) (21 ((c 100)) 1.0)

Table 5: Finding the densest blocks of code: a step-by-step example. The step number in the first column relates to the steps in the illustrated example on pages 10-12. Column 3 shows the window size at each step.

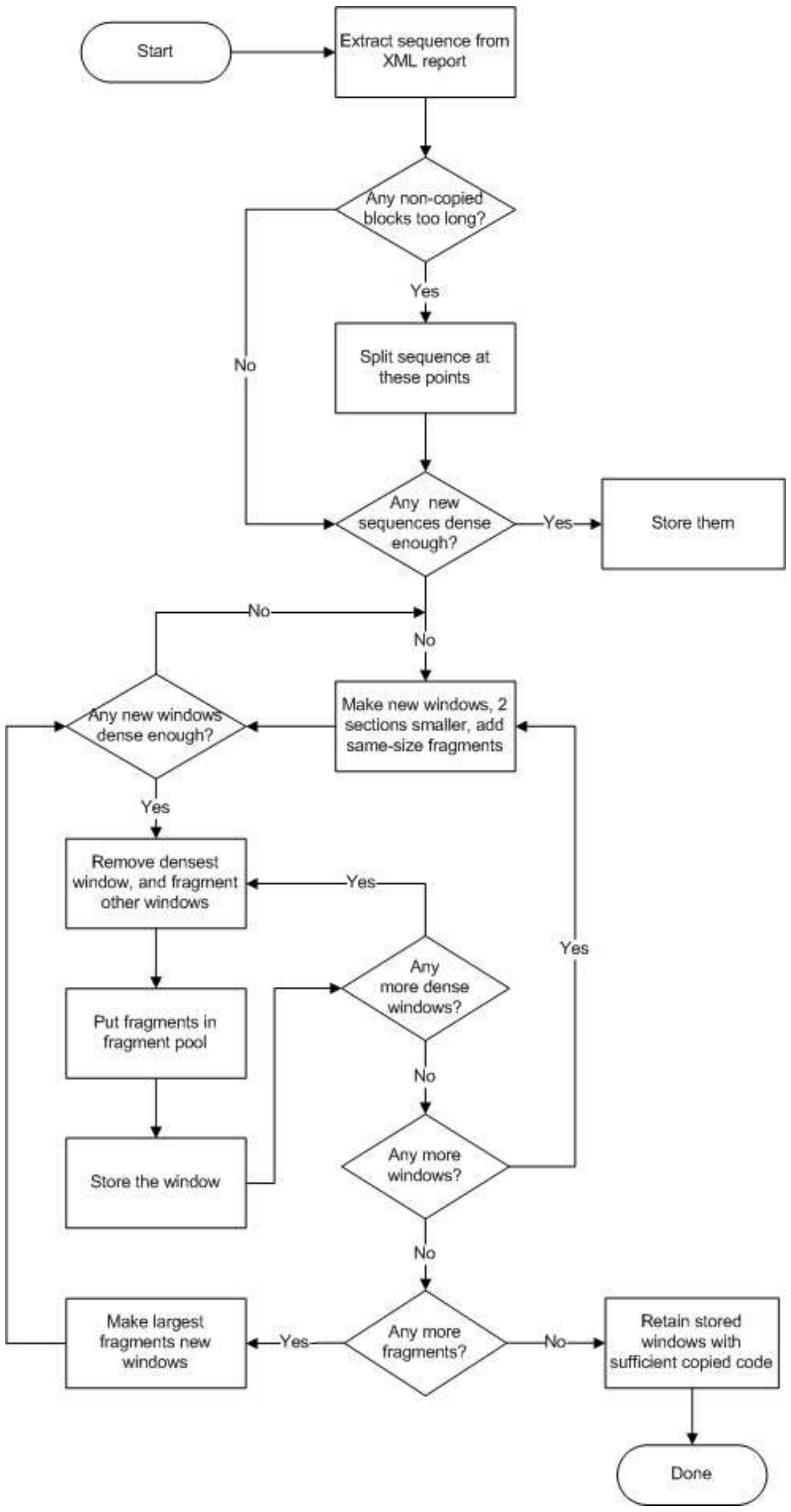

Start
Extract sequence from XML report
Any non-copied blocks too long?
Yes
No
Split sequence at these points
Any new sequences dense enough?
Yes
Store them
No
No
Make new windows, 2 sections smaller, add same-size fragments
Any new windows dense enough?
Yes
Remove densest window, and fragment other windows
Yes
Put fragments in fragment pool
Any more dense windows?
Yes
No
Store the window
Any more windows?
No
Make largest fragments new windows
Yes
Any more fragments?
No
Retain stored windows with sufficient copied code
Done

Figure 4: Flowchart illustrating extraction of dense blocks from a Ferret XML report.

3.4 Copy pattern files and dense blocks

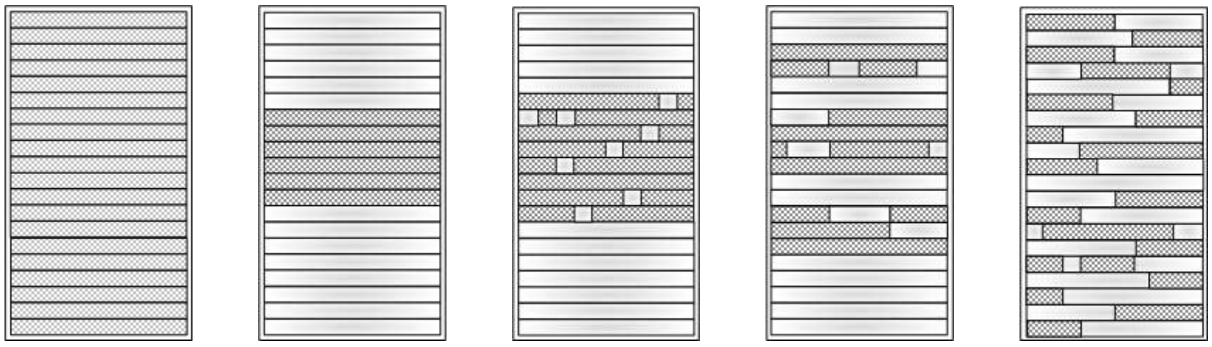

(a) Original file (b) Copied block (c) Block with new identifiers (d) Copy and edit (e) Incidentally similar

Figure 5: The copy patterns introduced in Section 2.2, repeated here for reference.

Looking at the copy patterns in the files introduced in Section 2.2 and repeated in Figure 5; if we assume that there are 10 tokens per line, that the copy sequence is based on tokens, and that none of the trigrams in a file are repeated; then the similarity of each pattern to file a is approximately 0.17. The copy sequences for the files shown in 5 (b)-(e) are:

b (n 60)(c 60)(n 80)

c (n 50)(c 8)(n 1)(c 1)(n 1)(c 1)(n 1)(c 14)(n 1)(c 7)(n 1)(c 6)(n 1)(c 23)(n 1)(c 6)(n 1)(c 6)(n 70)

d (n 20)(c 13)(n 2)(c 3)(n 25)(c 18)(n 2)(c 6)(n 1)(c 10)(n 20)(c 4)(n 3)(c 10)(n 3)(c 10)(n 50)

e (c 5)(n 11)(c 9)(n 8)(c 5)(n 10)(c 7)(n 11)(c 6)(n 11)(c 11)(n 21)(c 8)(n 8)(c 7)(n 8)(c 6)
(n 1)(c 3)(n 11)(c 5)(n 13)(c 8)(n 7)

Given the (arbitrary) parameters: minimum density 0.8, maximum non-copied section size 10 units, minimum number of tokens per block 15, the copied blocks in the files are:

b 1 block, tokens 61–120, (c 60)

c 1 block, tokens 51–130,
(c 8)(n 1)(c 1)(n 1)(c 1)(n 1)(c 4)(n 1)(c 7)(n 1)(c 6)(n 1)(c 23)(n 1)(c 6)(n 1)(c 6)

d 3 blocks, tokens 21–38, 64–100, 121–150,
(c 13)(n 2)(c 3), (c 18)(n 2)(c 6)(n 1)(c 10) and (c 4)(n 3)(c 10)(n 1)(c 10)

e no blocks

	File b	File c	File d			File e
			1	2	3	
Blocks	1	1	3			0
Density	1.00	0.89	0.90	0.92	0.85	0.00
Total copied tokens	60	72	16	34	24	0
Total all tokens	60	80	18	37	30	0
Mean tokens per copied section	60	8	8	11.33	8	0

Table 6: Features of the dense blocks in the example copy pattern files shown in Figure 2 with parameters: minimum density 0.8, maximum non-copied section 10, minimum copied tokens per block 15.

		Copied tokens in blocks		
		High	Med	Low
Mean copied	High	b		
tokens	Med	c	d	
per section	Low			e

Table 7: The relationship between the number of tokens in a block and the mean number of copied units per section for the four copy patterns.

Table 6 gives information about the blocks formed with the given parameters. Table 7 shows that the patterns are differentiated by the relationship between the number of tokens in a block and the mean number of copied units per copied section.

4 Experiments with constructed examples

To investigate the use of the density information extracted from Ferret XML reports, we considered whether is it possible to discriminate between different copy patterns using density information. Files with different copy patterns were constructed, compared to a 'source' file and the output analysed.

After an explanation of the construction of the artificial data, this section continues with a brief description of the experiments in Section 4.2, then discusses representation of density information and goes on to give results in Section 4.4. The results are analysed in Section 4.5, with a conclusion in Section 4.6.

4.1 Constructing artificial files

To test whether the four types of copy pattern depicted in Figure 2 can be identified from file density information, eight files of 1000 tokens were constructed. Two sequences of 1000 tokens were randomly generated from a pool of 26 tokens (a-z). One of these is the 'source' sequence to which the constructed files are compared. The other is used to provide the 'non-copied' tokens. The files were constructed by combining a number of consective tokens, n, from the source sequence and a number of consecutive tokens, (1000-n), from the uncopied sequence. These tokens were combined in different ways to produce files with different 'copy patterns'. The files were made so that each had approximately the same similarity, around 0.14, to the 'source' sequence.

Each of the files has some incidental similarity, which can be clearly seen in Figure 7, where tokens in matched trigrams are shown as asterisks. File 1 has a copied block of 200 tokens at the end of the file, the asterisks scattered through the rest of the file show the incidental similarity. File 2 has a block of 230 tokens of which 4% are randomly replaced. File 3 has a block of 300 tokens of which 10% are randomly replaced.

The remaining files were constructed by interleaving tokens from the source sequence and the non-copied sequence, where m consecutive tokens are selected from the sequence of 'non-copied' tokens followed by n tokens from the source sequence, see Figure 6. The values for m and n are determined at each step by randomising the chosen maximum chunk size for each source. The subsequences are alternated until the required file size is reached. If one sequence is exhausted, tokens from the remaining sequence complete the file.

File 4 has 250 tokens which are selected sequentially from the source, in chunks of up to 25 tokens, each source chunk is followed by a non-copied chunk of up to 15 tokens. File 5 also has 250 tokens from the source sequence, selected in chunks of up to 20 tokens, with the non-copied

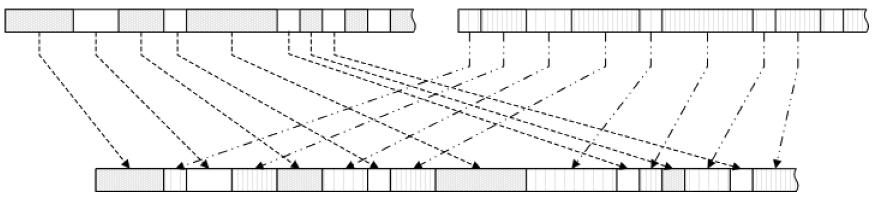


Figure 6: Illustration of the interleaving of two files to make the artificial files. The number of tokens selected from each file at each step is determined by randomising n, the maximum chunk size. A subsequence of between 1 and n tokens is removed from the head of the first sequence and placed in the new file, followed by a similarly selected subsequence from the second sequence. The process is repeated until the constructed sequence reaches the required size. If one sequence is exhausted, the file is completed with tokens from the remaining sequence.

tokens in chunks of up to 10 tokens. File 6, made in the same way has 350 source tokens, has up to 8 of these interleaved with up to 3 non-copied tokens. Files 7 and 8 were similarly constructed with the number of source tokens 300 and 350 respectively; the maximum size of source chunk 11 and 8 respectively and the maximum size of non-copied chunk 20 and 13 respectively. Chunk sizes were driven by the aim of building files with approximately the same similarity. Table 8 outlines the file construction methods and Figure 7 shows a graphical representation of the file copy profiles.

These constructed files relate to those shown in Figure 2. File 1 has a single copied block as Figure 2(b), files 2 and 3 have a copied block with some token replacement to represent identifier name changes, as Figure 2(c). Files 4 and 5 are examples of the type of copy pattern found when a file has some copied code which has been edited, similar to Figure 2(d). Files 7 and 8 have varying amounts and patterns of copied code which may be found in files within the same project, as shown in Figure 2(e). File 6 falls somewhere between files 4 and 5, and files 7 and 8. It can be seen either as representing copy and heavy edit, or as having clustered incidental similarity, or an element of each of these characteristics.

File	Method	Source token content	Non-source token token content	Maximum source chunk size	Maximum non-source chunk size	Similarity
1	Copy	200	800	N/A	N/A	0.1397
2	Copy/Replace 4%	230	770	N/A	N/A	0.1397
3	Copy/Replace 10%	300	700	N/A	N/A	0.1382
4	Interleave	250	750	25	15	0.1429
5	Interleave	250	750	20	10	0.1398
6	Interleave	350	650	8	3	0.1349
7	Interleave	300	700	11	20	0.1383
8	Interleave	350	650	8	13	0.1362

Table 8: Artificial file construction data

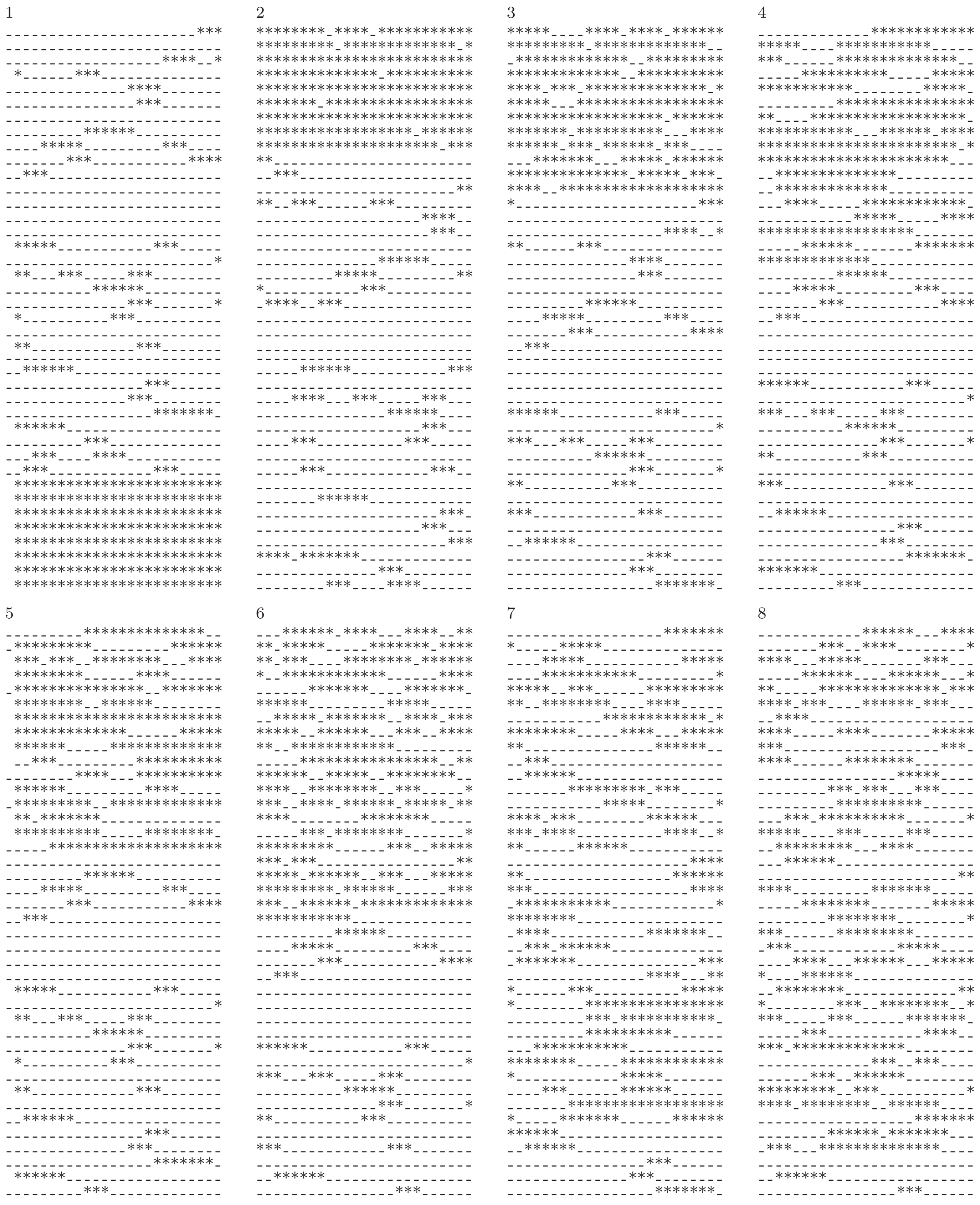


Figure 7: The eight artificial files described in Section 4.1 laid out in lines containing 25 tokens. Asterisks show tokens occurring in matched trigrams and dashes, tokens in non-copied trigrams. File 1 has a single block of copied code, files 2 and 3 have 4% and 10% of the tokens in the copied block replaced randomly. Files 4 and 5 represent code which has been copied and edited. Files 7 and 8 have copied code scattered in different patterns. File 6 is on the spectrum between files 4 and 5, and files 7 and 8.

4.2 Experiments

Each of these densities are combined with the parameter pairs on the right of the table	Minimium copied tokens per block	Maximum non-copied tokens per section
0.9 0.8 0.7 0.6	100	50
	50	50
	50	20
	20	20
	20	10

Table 9: Parameter sets used with the eight constructed files to test whether the density data is able to discriminate between copy patterns. Each of the four densities on the left of the table are combined with the five parameter pairs on the right of the table forming twenty different sets.

Each constructed file was analysed at four different densities with five different combinations of minimum copied tokens per block and maximum non-copied section size. The twenty parameter sets are shown in Table 9.

4.3 Representing density information for machine learning

Finding a meaningful, compact and uniform way to represent the blocks of densely copied code found in a file, regardless of file size and copy pattern, is a challenge. There are several measures which can be used, both absolute and relative to file size. Absolute measures include the number of blocks, the number of tokens within the blocks, the mean size of a block or the maximum block size. Relative measures include the proportion of the file which is contained within dense blocks or the proportion of the copied tokens in the file contained in dense blocks.

The parameters used when finding dense blocks affect the outcome. A higher density will tend to produce smaller blocks than a lower density. A low minimum number of copied units per block will include more blocks than a higher number. If large sections of non-copied code are permitted then the blocks may be less fragmented than when a smaller limit is stipulated. It is possible that combining the data output using several different parameter sets would provide more useful information than that output by one set alone.

Features which are proportional to file size can be useful when the sets of files being compared vary in size, however, they are not relevant to the artificial files here because the files are the same size. The measurements taken for the artificial files were those shown in Table 10.

There are many other measurements which could have been used. For example, the mean number of tokens in the copied sections of the XML report which gives an idea of how large the raw blocks are, this measure is not used with the artificially constructed files because the construction process dictates the possible range of values. Another possible measure is the proportion of tokens in blocks which are at least, say, one tenth of the file size, giving an idea of the proportion of the copied code which is in dense blocks of significant size. In an application where file sizes differ, a range of proportional measures can be added.

4.4 Results

The measures which were recorded about the dense blocks resulting from the analyses are shown in Table 10. Results for features from this set are shown in the graphs in Figure 8 and 9. Each graph shows the features recorded at one density. The five features recorded are the number of copied tokens in dense blocks (blue left-hand columns), the mean number of copied tokens per block (lilac second columns), maximum number of copied tokens in a block (maroon centre columns), the total number of tokens in dense blocks (yellow fourth columns) and mean number

Features
The number of dense blocks found in the file
The number of copied tokens found in the dense blocks
The mean number of copied tokens per block
The maximum number of copied tokens in one block
The total of all tokens in dense blocks (includes non-copied tokens)
The mean number of all tokens per block

Table 10: The information recorded from each comparison between two files.

of tokens per block (green right-hand columns). The number of dense blocks is not shown on the graph, but is apparent from the relationship between total tokens and mean tokens per block. For each of the five combinations of minimum block size and maximum gap size, the five features are shown for each of the eight files. The labels on the x-axis are the density, the minimum number of copied tokens in a block, the maximum non-copied section allowed within a block, and the file reference (1-8).

4.5 Analysis

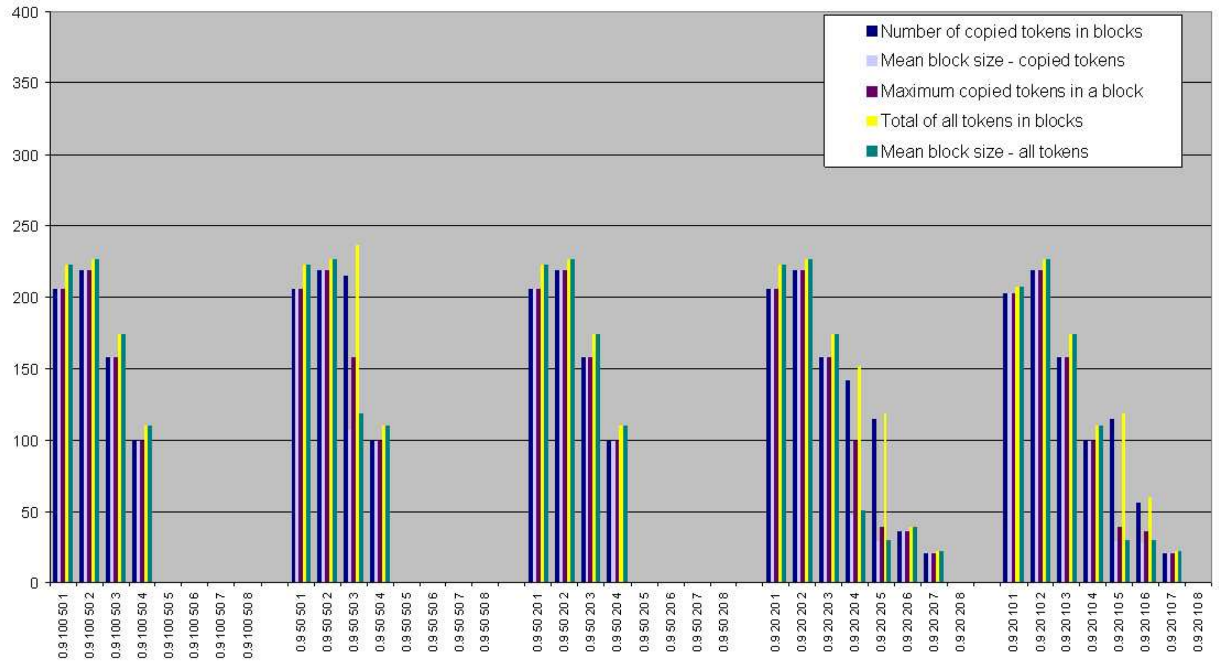
The files form three main groups: files 1-3 have a block of code which is copied into the file, either without change or with some replacement; files 4-5 mimic copy and edit operations and files 7-8 have scattered copying, with file 6 being between the latter two groups.

The density information should reflect both the three groups and, possibly, the differences between the files within groups. At 0.9 density and minimum block size of 50 or 100, there are no blocks reported for files 5-8. When the minimum block size is reduced to 20, the groups are more apparent; files 1-3 have all of their copied code in one block, files 4 and 5 have a number of smaller blocks, there is little copied code detected for file 6, less for file 7 and none for file 8.

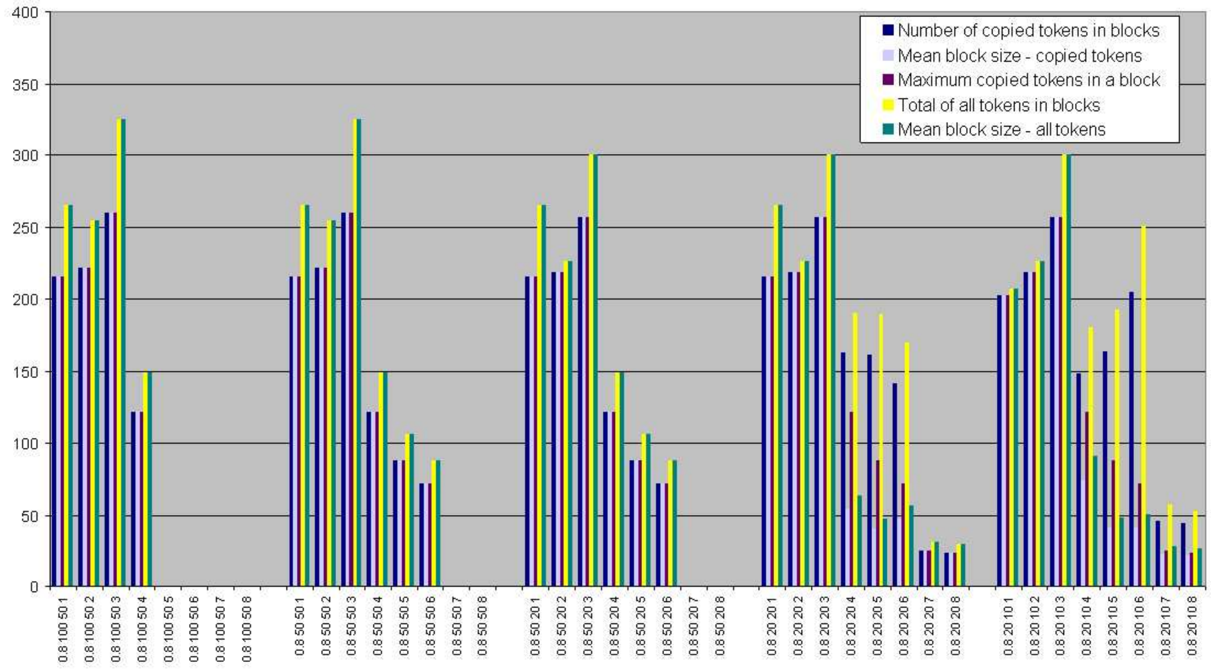
At 0.8 density, file 6 shares more characteristics with files 4 and 5 than with files 7 and 8, while files 1-3 also form a clear group. Files 1-3 each have one block which includes some non-copied tokens. Files 4 to 8 have several blocks of dense code, with the size both of the mean block size in copied tokens and of the maximum block decreasing from file 4 to file 8.

When the density parameter is 0.7, and the other parameters are 100 and 50, there is little difference between the first five files and none between the last three. When the minimum block size is reduced to 50, the groups appear to be files 1-4, with one copied block, files 5 and 6, with more than one copied block, and files 7 and 8. However when the minimum block size is 20, and the maximum gap size is also 20, the first three files form a group, files 4 and 5 another, files 7 and 8 a third group, with file 6 between the two.

The more relaxed minimum density requirement of 0.6 does a poor job of differentiating between either files or groups, except when used with the smallest of the other parameters tested, when three groups are identifiable.

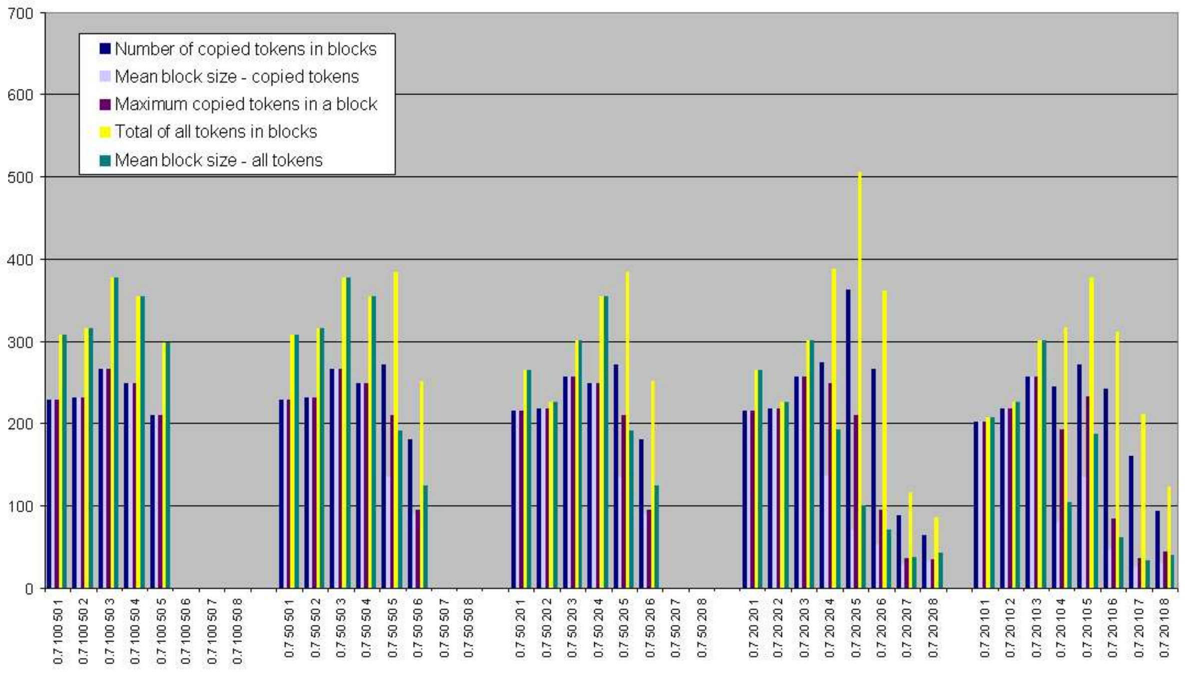


(a) 0.9 density

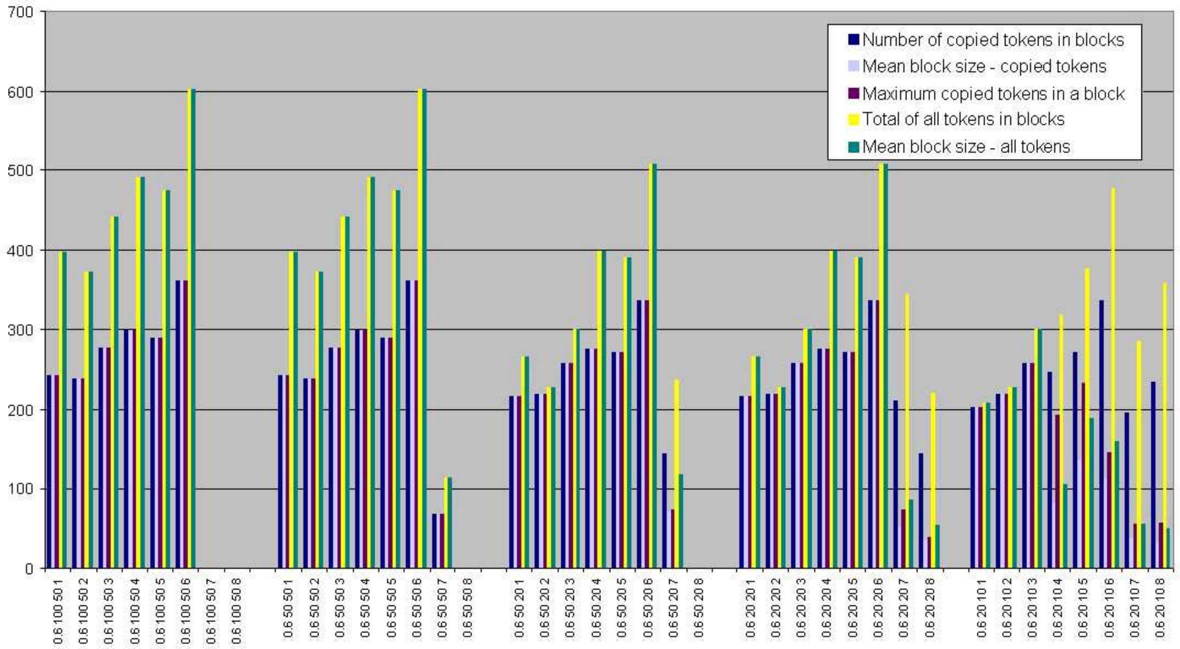


(b) 0.8 density

Figure 8: Each graph shows the results for five parameter sets at one density. Within each parameter set group, information is shown for each of the eight artificial files. Each file's information consists of five features: the number of copied tokens in dense blocks (blue left-hand columns), the mean number of copied tokens per block (lilac second columns), maximum number of copied tokens in a block (maroon centre columns), the total number of tokens in dense blocks (yellow fourth columns) and mean number of tokens per block (green right-hand columns) for the 8 test files at densities 0.9 and 0.8. The labels on the x-axis are the density, the minimum number of copied tokens in a block, the maximum non-copied section allowed within a block, and the file reference (1-8).



(a) 0.7 density



(b) 0.6 density

Figure 9: Each graph shows the results for five parameter sets at one density. Within each parameter set group, information is shown for each of the eight artificial files. Each file's information consists of five features: the number of copied tokens in dense blocks (blue left-hand columns), the mean number of copied tokens per block (lilac second columns), maximum number of copied tokens in a block (maroon centre columns), the total number of tokens in dense blocks (yellow fourth columns) and mean number of tokens per block (green right-hand columns) for the 8 test files at densities 0.7 and 0.6. The labels on the x-axis are the density, the minimum number of copied tokens in a block, the maximum non-copied section allowed within a block, and the file reference (1-8).

4.6 Conclusion

Table 11 shows whether each parameter set separates the groups and whether expected differences between the files within the groups are identified. Generally, the smaller block and gap sizes separate the groups and files better than the larger sizes. The densities 0.8 and 0.7 are best for discriminating between the groups. The two parameter sets which can determine both the groups and the files in the groups are 0.9, 20, 20 and 0.8, 20, 10, marked in bold. However there are many sets which do the same with the exception of an unexpected profile for one file. An exception is marked where the files are differentiated but not in the expected manner. For example, the last entry in the table shows that at density 0.6 with the other parameters 20 and 10, file 5 has some features (both mean block sizes and maximum block size) which appear to be out of sequence with files 4 and 6. The patterns are different for different parameter sets and it is possible that combining two or three sets may provide better information about the density of copied code in a file than one set.

Parameter set	Group differences	Intra-group differences	Exceptions	Parameter set	Group differences	Intra-group differences	Exceptions
0.9 100 50	x	x		0.7 100 50	✓	x	
0.9 50 50	x	x		0.7 50 50	✓	x	
0.9 50 20	x	x		0.7 50 20	✓	x	
0.9 20 20	✓	✓		0.7 20 20	✓	✓	file 1
0.9 20 10	✓	✓	file 4	0.7 20 10	✓	✓	file 5
0.8 100 50	x	x		0.6 100 50	x	x	
0.8 50 50	✓	✓	file 1	0.6 50 50	x	x	
0.8 50 20	✓	✓	file 1	0.6 50 20	x	x	
0.8 20 20	✓	✓	file 1	0.6 20 20	x	x	
0.8 20 10	✓	✓		0.6 20 10	✓	✓	file 5

Table 11: The parameter sets tested with the artificial files and their use in separating both the file groups [1-3, 4-5(6), and (6)7-8], and the file differences. A tick means that the groups or files are separated, with at most one exception.

5 Summary

Density measures extracted from the XML report output by Ferret provide useful information about the copied code in a file. Tests with artificial data show that the differences between varied copy patterns are characterised by the density information collected.

Density analysis is useful in comparing text as well as program code files. The measure can pinpoint blocks of text which have been edited, for example, by automatic word replacement; and differentiate areas in a file which are accidentally similar from those which are deliberately copied and edited.

In general, any pair of files with a large similarity score clearly contain copied text or code. Two files with an extremely low score are unlikely to share anything significant. Without a visual check of the copy pattern, it is more difficult to discern whether small similarity scores are the result of scattered incidental copying or whether there is a similar block of text or code shared by the files. Density analysis can detect these blocks automatically. The parameters for level of density, block size and gap size mean that the user has reasonable control over the type of block found.

A tool implementing this method can be downloaded at http://homepages.stca.herts.ac.uk/~gp2ag/density.html.

References

[1] P. Green, P. C. R. Lane, A. Rainer, and S.-B. Scholz. Building classifiers to identify split files. In P. Perner, editor, *MLDM Posters*, pages 1–8. IBaI Publishing, 2009.

[2] P. C. R. Lane, C. M. Lyon, and J. A. Malcolm. Demonstration of the Ferret plagiarism detector. In *2nd International Plagiarism Conference*, 2006.

[3] C. M. Lyon, R. Barrett, and J. 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.

[4] C. M. Lyon, J. A. Malcolm, and R. G. 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.

[5] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, Cambridge, MA, USA, 2001.

[6] A. W. Rainer, P. C. R. Lane, J. A. Malcolm, and S.-B. Scholz. Using n-grams to rapidly characterise the evolution of software code. In *The Fourth International ERCIM Workshop on Software Evolution and Evolvability*, 2008.