Software Defect Prediction Using Static Code Metrics Underestimates Defect-Proneness

David Gray, David Bowes, Neil Davey, Yi Sun and Bruce Christianson

Abstract— **Many studies have been carried out to predict the presence of software code defects using static code metrics. Such studies typically report how a classifier performs with real world data, but usually no analysis of the predictions is carried out. An analysis of this kind may be worthwhile as it can illuminate the motivation behind the predictions and the severity of the misclassifications. This investigation involves a manual analysis of the predictions made by Support Vector Machine classifiers using data from the NASA Metrics Data Program repository. The findings show that the predictions are generally well motivated and that the classifiers were, on average, more 'confident' in the predictions they made which were correct.**

I. Introduction

A growing number of studies have been carried out on the subject of automated software defect prediction using static code metrics ([1], [2], [3], [4] and [5] for example). Such studies are motivated by the tremendous cost of software defects (see [6]) and typically involve observing the performance achieved by classifiers in labelling software modules (functions, procedures or methods) as being either defective or otherwise. Although such a binary labelling toward module defectiveness is clearly a simplification of the real world, it is hoped that such a classification system could be an effective aid at determining which modules require further attention during testing. An accurate software defect prediction system would thus result in higher quality, more dependable software that could be produced more swiftly than was previously possible.

The predictions that are made in typical defect prediction studies are usually assessed using confusion matrix related performance measures, such as recall and precision. Rarely in the research literature are these predictions mapped back to the original corresponding modules for further analysis. An examination of this kind may be worthwhile as it can illuminate the motivation behind the predictions and the severity of the misclassifications. For example, a module that is incorrectly predicted as defective (a *false positive*) which consists of 1000 lines of code (LOC) may be a far more logical (and forgivable) mistake for a classifier to make than a false positive which consists of 5 LOC. The former misclassification may even be desirable, as a module with highly defect-prone characteristics should probably be subjected to some kind of further inspection in the interests of code quality.

All authors are with the Computer Science Department at the University of Hertfordshire, UK. Respective email addresses are: {d.gray, d.h.bowes, n.davey, y.2.sun, b.christianson}@herts.ac.uk

In this study a defect prediction experiment was carried out to allow a manual analysis of the classifications made in terms of each module's: original metrics, corresponding classification result (one of either: true positive, true negative, false positive or false negative) and corresponding *decision value;* a value output by the classifier which can be interpreted as its certainty of prediction for that particular module. The experiment carried out involved assessing the performance of Support Vector Machine (SVM) classifiers against the same data with which they were trained. The purpose of this was to gain insight into how the classifiers were separating the training data, and to see whether this separation appeared consistent with current software engineering beliefs (i.e. that larger, more complex modules are more likely to be defective). Additionally it was interesting to examine the modules that were misclassified in the experiment, to try and see why the classifiers associated these modules with the opposing class.

The data used in this study was taken from the NASA Metrics Data Program (MDP) repository[1], which currently contains 13 data sets intended for software metrics research. Each of these data sets contains the static code metrics and corresponding fault data for each comprising module. One of the data sets (namely, PC2) was the main focus of this study as it contained the fewest modules (post data preprocessing) and was therefore the least labour intensive to manually examine. The remaining 12 data sets were all used in the experiment but were not subjected to the same level of scrutiny.

Initial analysis of data set PC2 involved the collection and examination of basic statistics for each of the metrics in each class (the class labels were: {defective, non-defective}) to observe the distribution of values amongst classes. Principal Components Analysis was then used as a data visualisation tool to see how the classes were distributed within the feature space, and if any patterns emerged. This enabled the detection of *outliers;* data points which substantially differ from the rest of their class. These outliers were later cross-examined to see how they correlated with the SVMs predictions.

The findings from this study are that the predictions for the modules comprising data set PC2 were generally well motivated; they seemed logical given current software engineering beliefs. Also, each of the classifiers for all 13 NASA MDP data sets had higher average decision values for the defective predictions they made which were correct (the

[1] http://mdp.ivv.nasa.gov/

true positives) than were incorrect (the *false positives*). This information could be exploited in a real world defect prediction system where the predicted modules could be inspected in decreasing order of their decision values. The findings in this study indicate that defect prediction systems may be doing far better at predicting module defect-proneness than they are at predicting actual defectiveness. This highlights one of the fundamental issues with current defect prediction experiments - the assumption that all modules predicted as having defect-prone characteristics are in fact defective.

The rest of this paper is presented as follows: Section II begins with a brief introduction to static code metrics, followed by a description of the data used in this study and then an overview of our chosen classification method, Support Vector Machines. Section III describes the data preprocessing carried out and the experimental design. The findings are shown in Section IV in two parts, firstly the initial data analysis is presented and then the classification analysis. The conclusions are given in Section V.

II. Background

A. Static Code Metrics

Static code metrics are measurements of software features that may potentially relate to defect-proneness, and thus to quality. Examples of such features and how they are often measured include: size, via LOC counts; readability, via operand and operator counts (as proposed by [7]) and complexity, via linearly independent path counts (also known as the cyclomatic complexity [8]).

Consider the C program shown in Figure 1. Here there is a single function called *main*. The number of lines of code this function contains (from opening to closing bracket) is 11, the number of arguments it takes is 2, the number of linearly independent paths through the function is 3. These are just a few examples of the many metrics that can be statically computed from source code.

```
#include <stdio.h>

int main(int argc, char* argv[])
{
int return_code = 0;
if (argc < 2) {
  printf("No Arguments Given\n");
  return_code = -1;
}
int x;
for(x = 1; x < argc; x++)
  printf("'%s'\n", argv[x]);
return return_code;
}
```

Fig. 1. An example C program.

Because static code metrics are calculated through the parsing of source code, their collection can be automated. Thus it is computationally feasible to calculate the metrics of entire software systems, irrespective of their size. Sommerville points out that such collections of metrics can be used in the following contexts [9]:

- **To make general predictions about a system as a whole.** For example, has a system reached a required quality threshold?
- **To identify anomalous components.** Of all the modules within a software system, which ones exhibit characteristics that deviate from the overall average? Modules thus highlighted can then be used as pointers to where developers should be focusing their efforts. This is common practice amongst several large US government contractors [2].

B. Data

The data used in this study was obtained from the NASA MDP repository. This repository currently contains 13 data sets intended for software metrics research. All 13 of these data sets were used in this study: brief details of them are shown in Table I. Each of the MDP data sets represents a NASA software system/subsystem and contains the static code metrics and corresponding fault data for each comprising module. Note that "module" can refer to a function, procedure or method. Between 23 to 42 metrics and a unique module identifier comprise each data set, a subset of the metrics are shown in Table II.

All non-fault related metrics within each of the data sets were generated using McCabeIQ 7.1; a commercial tool for the automated collection of static code metrics. The *error count* metric was calculated by the number of error reports issued for each module via a bug tracking system. It is unclear precisely how the error reports were mapped back to the software modules, however the MDP homepage states

Name	Language	Total KLOC	No. of Modules	% Defective Modules
CM1	C	20	505	10
JM1	C	315	10878	19
KC1	C++	43	2107	15
KC3	Java	18	458	9
KC4	Perl	25	125	49
MC1	C & C++	63	9466	0.7
MC2	C	6	161	32
MW1	C	8	403	8
PC1	C	40	1107	7
PC2		26	5589	0.4
PC3		40	1563	10
PC4		36	1458	12
PC5	C++	164	17186	3

TABLE I
BRIEF DETAILS OF THE 13 NASA MDP DATA SETS. NOTE THAT KLOC REFERS TO THOUSAND LINES OF CODE.

Metric Name	Metric Definition
Cyclomatic Complexity	# of linearly independant paths (see [8])
Essential Complexity	Related to the # of unstructured constructs
No. Operators	# of operators (‘==’, ‘!=’, keywords, etc)
No. Operands	# of operands (variables, literals, etc)
No. Unique Operators	# of unique operators
No. Unique Operands	# of unique operands
LOC Blank	# of blank lines
LOC Comments	# of lines containing only comments
LOC Executable	# of lines containing only executable code
LOC Code & Comments	# of code and comments on the same line

TABLE II
A SMALL SUBSET OF THE METRICS CONTAINED WITHIN EACH OF THE NASA MDP DATA SETS.

that “if a module is changed due to an error report (as opposed to a change request), then it receives a one up count”.

The NASA MDP data has been used extensively by the software engineering research community. There are currently more than 20 published studies that have used data which first originated from this repository. The motivation for using these data sets is often due to the difficulty in obtaining real world fault data. Using the NASA MDP data can be problematic however as access to the original source code is not possible, making the validation of data integrity difficult. This is especially problematic as the NASA MDP data sets appear to have quality issues with regard to their accuracy (briefly described in Section III-A). These issues may not have been taken into account during previous fault prediction studies based on this data.

C. Support Vector Machines

Support Vector Machines (SVMs) are a set of closely related and highly sophisticated machine learning algorithms that can be used for both classification and regression [10]. Their high level of sophistication made them the classification method of choice for this study, although they have been used previously within the software engineering community (see [1], [3] and [5]).

SVMs are *maximum-margin classifiers,* they construct a separating hyperplane between two classes subject to zero or more slack variables. The hyperplane is constructed such that the distance between the classes is maximised. This is intended to lower the generalisation error when the classifier is tested. Note that SVMs can also be used to classify any number of classes via recursive application.

Although originally only suitable for linear classification problems, SVMs can now also be used successfully for non-linear classification by replacing each dot product with a kernel function. A kernel function is used to implicitly map the data points into a higher-dimensional feature space, and to take the inner-product in that feature space. The benefit of using a kernel function is that the data is more likely to be linearly separable in the higher feature space. Importantly, the actual explicit mapping to the higher-dimensional space is never needed.

There are a number of different kinds of kernel functions (any continuous symmetric positive semi-definite function will suffice) including: linear, polynomial, Gaussian and sigmoidal. Each has varying characteristics and is suitable for different problem domains. The one used here is the *Gaussian radial basis function* (RBF), as it can handle non-linear problems and requires fewer hyperparameters than the remaining aforementioned non-linear kernels [11]. In fact, this kernel implicitly maps the data into an infinite dimensional feature space whereby any finite data set will be linearly separable.

When SVMs are used with a Gaussian RBF kernel there are two user-specified hyperparameters, C and γ. C is the error cost hyperparameter - a variable that determines the trade-off between minimising the training error and maximising the margin. γ is a kernel hyperparameter and controls the width (or radius) of the Gaussian RBF. The performance of SVMs is largely dependant on these hyperparameters, and the optimal values; the pair of values which yield best performance while avoiding both underfitting and overfitting, should ideally be determined *for each training set* via a systematic search.

The biggest potential drawback of SVMs is that their classification models are black box, making it very difficult to work out precisely why the classifier makes the predictions it does. This is very different to white box classification algorithms such as Bayesian networks and decision trees, where the classification model is easy to interpret.

Although SVMs are black box algorithms, classification analysis can still be carried out upon them by analysing their performance on individual instances. This involves mapping the predictions made back to each instance. An analysis can then take place according to each instance’s: original module metrics, achieved classification result, and corresponding *decision value*. The decision value is a real number output by the SVM which corresponds to the instance’s distance from the separating hyperplane in the feature space. The decision value can be interpreted as the SVM’s certainty of prediction for a particular instance.

As this studies main focus is on classification analysis rather than classification performance, it was decided to classify the training data rather than having some form of tester set. The instances misclassified in the experiment would thus be outliers, as they were not placed with their corresponding class post hyperparameter optimisation. These instances occur mainly because of the SVM’s cost hyperparameter, as they are deemed too costly to place on the correct side of the decision hyperplane. Thus it is of interest to see why these instances are more similar to those in the opposing class. For an investigation into the performance of SVMs for software defect prediction see [1].

III. METHOD

A. Data Pre-processing

1) Initial Data Set Modifications: Each of the data sets had their module identifier and *error density* attributes removed, as well as all attributes with zero variance. The

error count attribute was then converted into a binary target attribute for each instance by assigning all values greater than zero to defective, non-defective otherwise. Note that this is the same as was carried out in [1], [2], [3] and [5].

2) Removing Repeated and Inconsistent Instances: Instances appearing more than once in a data set are known as repeated (or redundant) instances. Inconsistent instances are similar to repeated instances, however the class labels differ. So in this domain inconsistent instances would occur where identical metrics were used to describe (for example) two different modules, of which one had been reported as faulty and the other had not.

The effect of being trained using data containing repeated and/or inconsistent instances is classification algorithm dependant. SVMs for example can be affected by repeated data points because the cost value for those data points may be being over-represented. More importantly however (and independent of classification algorithm) is that when dividing a data set into training and testing sets, a data set containing repeated data points may end up with instances common to both sets. This can lead to a classifier achieving unrealistically high performance.

Analysis of the MDP data sets showed that some of them (namely MC1, PC2 and PC5) contained an overwhelmingly high number of repeating instances (79%, 75% and 89% respectively). Although no explanation has yet been found for these high numbers of repeated instances, it appears highly unlikely that this is a true representation of the software, i.e. that 75% of the modules within a system/subsystem could possibly have the same number of: lines, operands, operators, unique operands, unique operators, linearly independent paths, etc. Although in this experiment the data is never divided into training and testing sets, it was still decided to remove these instances in order to defend against the potential SVM difficulties previously described. The pre-processing stage involved the removal of all repeated instances so they were only represented once, and then the complete removal of all inconsistent pairs.

3) Missing Values: Missing values are those that are unintentionally or otherwise absent for a particular attribute in a particular instance of a data set. The only missing values within the data sets used in this study were within the *decision density* attribute of data sets CM1, KC3, MC2, MW1, PC1, PC2 and PC3. Decision density is calculated as: condition count ÷ decision count and each instance with a missing value had a value for both of these attributes of zero, therefore all missing decision density values were replaced with zero.

4) Balancing the Data: All the data sets used within this study with the exception of KC4 contain a much larger amount of one class (namely, non-defective) than they do the other (see Table I). When such *imbalanced* data is used with a machine learning algorithm the classifier will typically be expected to overpredict the majority class. This is because the classifier will have seen more examples of this class during training.

There are various techniques that can be used to deal with imbalanced data (see [12] and [13]). The approach taken here is very simple however and involves randomly undersampling the majority class until it becomes equal in size to that of the minority class. The number of instances that were removed during this undersampling process varied amongst the data sets, however for data set PC2 it was a total of 97%. Removing such a large proportion of the data does threaten the validity of this study as the instances randomly chosen to remain in the non-defective class may not be representative of that class. To defend against this problem the experiment was repeated several times with different versions of the balanced data sets. The result of this showed that although the predictive accuracy changed for each different sample of the data, the concluding statements that are made in Section 5 remained unchallenged.

5) Normalisation: All values within the data sets used in this study are numeric. To prevent attributes with a large range dominating the classification model, all values were normalised between -1 and 1.

B. Experimental Design

Section II-C described how SVMs require the selection of optimal hyperparameter values in order to balance the trade-off between underfitting and overfitting. The two hyperparameters required in this study (C and γ) were chosen for each data set using a *grid search;* a process that uses n-fold (here n = 5) stratified cross-validation and a wide range of possible hyperparameter values in a systematic fashion (see [11] for more details). The pair of values that yields the highest average accuracy across all 5-folds is taken as the optimal hyperparameters and used when generating the final model for classification.

After the optimal hyperparameters had been found for each data set, an SVM was trained and classified using that same data. This enabled the production of 13 spreadsheets (one for each data set) containing the original module metrics for each instance as well as the following additional columns:

- *Classification Result:* Either a true positive (TP), false positive (FP), true negative (TN) or false negative (FN).
- *Decision Value:* As described in Section II-C. Negative values are instances predicted as non-defective while positive values are instances predicted as defective.

For each spreadsheet the rows were ranked by their decision value. The averages for each of the original metrics were then calculated for the instances predicted as defective and the instances predicted as non-defective. The average decision values were also computed, but this time for each of the TP, TN, FP and FN instances respectively. For data set PC2, a thorough manual examination of each of the rows in the spreadsheet was carried out. For all other data sets, the decision value averages were examined to see if any patterns emerged.

IV. Findings

A. Raw Data Analysis

After pre-processing, the raw statistics for data set PC2 were examined (see Fig. 2). This process revealed that the

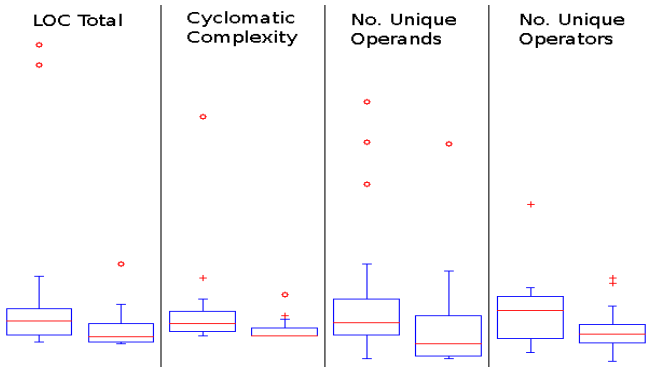

Fig. 2. A box plot showing basic statistics for data set PC2. Boxes on the left in each column are the defective instances while boxes on the right are the non-defective instances.

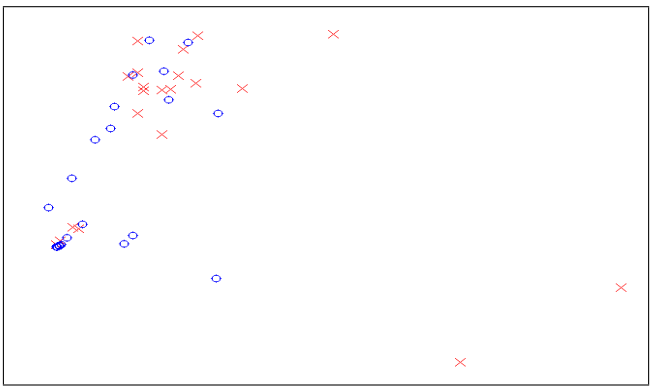

Fig. 3. Principal Components Analysis on data set PC2. Crosses represent modules labelled as defective while circles represent modules labelled as non-defective. Observe the extreme outliers belonging to the defective class in the bottom right corner.

modules labelled as defective have higher average values across all 36 attributes other than: cyclomatic density, design density, maintenance severity, Halstead level and normalised cyclomatic complexity. However, the only attributes which were statistically significant (to .95 confidence) between the two classes were: design density, branch count and percent comments. Definitions of these metrics can be found at the NASA MDP website. These findings show that the data is highly intermingled between classes.

Principal Components Analysis (PCA) is a popular dimensionality reduction / data visualisation tool which transforms data into a lower dimensional space whilst maximising the variance. For data set PC2, 36 features were mapped down to 2 whilst keeping 65% of the variance. The plot generated from this process is shown in Fig. 3 and clearly shows that the data is highly intermingled, but has two extreme outliers amongst the defective instances (bottom right corner). Locating these two instances revealed that they were the two largest modules (in terms of LOC total) within the data set and that they also had the two highest no. unique operands, no. unique operators and cyclomatic complexity attribute values (amongst others).

B. Classification Analysis

The classification result and corresponding decision value for each of the 42 instances which comprise data set PC2 are shown in Fig. 4. Examination of these values revealed that the SVM had a higher average confidence in the instances predicted as defective which were correct (the TPs with an average decision value of 0.86) than were incorrect (the FPs with an average of 0.60). The average decision value for the instances predicted as non-defective were very similar, but the incorrectly classified modules (the FNs) were being predicted with slightly more confidence (an average of -0.88 as opposed to -0.81).

Examining the averages computed for the remaining 12 data sets showed the SVMs again had more confidence in the TPs than the FPs, by an average of 49%. Unlike data set PC2 however, the remaining data sets also had more confidence in the TNs than the FNs, by an average of 51%.

Table III contains a subset of the metrics for PC2 which comprise the modules that are labelled in Fig. 4, as well as their corresponding classification result and decision value. Note that data set PC2 contains 36 of these metrics but only 4 are shown here due to space limitations. Module 1 (as labelled in Fig. 4) is the module that is furthest from the decision hyperplane. The high level of confidence associated with this defective prediction does seem logical however, due to there being 76 unique operands and 15 linearly independent paths within only 68 LOC. Modules 2 and 3 are the outliers identified during PCA (see Section IV-A). These modules were predicted with above average decision values (for the TPs), which is reassuring as they appear to be very large and may benefit from decomposing. It may be surprising that these modules were not predicted with the two highest decision values, however this shows that the SVM in this case is not being dominated entirely by size related metrics. This is reassuring as it suggests there is worth in the other metrics.

At first sight when looking at the FPs it appears that module 4 is on the wrong side of the separating hyperplane as it is only comprised of 10 LOC. This looks suspicious as the classifier has so much confidence in the prediction. On closer inspection however this module had an essential complexity value (which relates to the number of unstructured constructs within a module) of 3; and in the 42 instances passed to the classifier 78% of modules with an essential complexity greater than 1 were defective. Unstructured constructs have been known to be problematic with regard to code quality for over 40 years [14], and the SVM predicted that 89% of modules with an essential complexity greater than 1 (the metrics minimum) were defective. Module 5 is the module that is closest to the separating hyperplane, it was predicted as defective but only by a very small margin. This classification appears more immediately understandable than module 4, as although the module contains only 7 LOC it contains 16 unique operands.

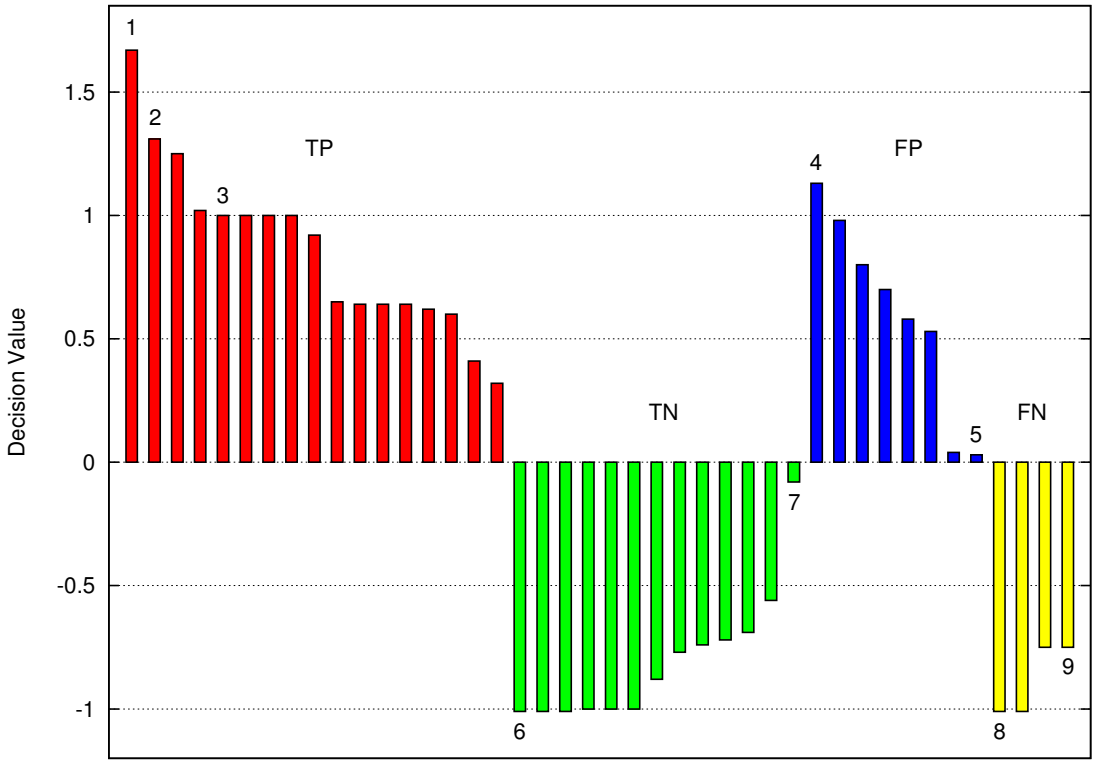

Fig. 4. The decision value and classification result for each of the 42 modules in data set PC2.

Module ID No.	LOC Total	v(G)	No. Unique Operands	No. Unique Operators	Prediction Result	Decision Value
1	68	15	76	16	TP	1.67
2	316	54	111	37	TP	1.31
3	294	84	94	88	TP	1.00
4	10	3	9	11	FP	1.13
5	7	2	16	9	FP	0.03
6	2	1	5	9	TN	-1.00
7	3	2	2	8	TN	-0.08
8	2	1	5	8	FN	-1.00
9	6	1	5	5	FN	-0.75

TABLE III

A SUBSET OF THE METRICS FOR THE MODULES LABELLED IN FIG. 4. NOTE THAT V(G) IS MCCABE'S CYCLOMATIC COMPLEXITY [8].

All of the modules predicted as non-defective (6 to 9) contain very low values for the four metrics shown in Table III, and (from what can be deduced from the metrics) appear very small and simple. This highlights the difficulty of this classification domain. None of the four modules had defect-prone characteristics, yet two of them did indeed turn out to be defective. This is problematic when data mining with static code metrics as they can provide only a limited insight into software defect-proneness, not actual defectiveness. It is a fair assumption that the majority of defective modules within a software system will exhibit defect-prone characteristics however, be them difficult to define precisely and programming language specific. This is the primary reason that software defect prediction is worthy of the growing research surrounding it. The findings in this study seem to suggest a limiting factor in the performance achievable by such defect prediction systems however. This limitation appears to be in the proportion of defective modules containing defect-prone characteristics.

The misclassified instances labelled in Fig. 4 have already been discussed. The remaining misclassified instances also all appeared to be well motivated, with the FPs generally having defect-prone metrics and the FNs not so. This shows that there is a low severity for the misclassifications in this data set, i.e. that a further examination of module 4 may in fact be worthwhile and that modules 8 and 9 would be very difficult to correctly predict, as they do not possess defect-prone characteristics.

V. Conclusion

In this study SVM classifiers were found to consistently have more confidence in the defective predictions they made which were correct than were incorrect, as the average decision value for the TP predictions was significantly greater than that of the FP predictions for all 13 of the NASA MDP data sets. These findings could be exploited in a real world classification system, where the predicted modules could be ranked in decreasing order of their decision values. Code inspections could then be prioritised around this ordering. Note that taking the decision values into account as well as the binary classifications also helps to alleviate the conceptual problems with using a binary classifier in this problem domain, where the defectiveness of a module would be more of a fuzzy value than a binary one.

A more in depth manual examination of the predictions made for one of the NASA data sets (namely, PC2) showed that the classifications were generally well motivated; that the SVM was separating the data according to current software engineering beliefs. Moreover it appeared that the classifiers were doing far better at predicting defect-proneness than they were at predicting actual defectiveness. Because it is easily possible for a module without defect-prone characteristics to contain a defect (a programmer typing a '==' instead of a '!=' in a single line module for example), the proportion of defective modules containing defect-prone characteristics may be the biggest limiting factor on the performance of defect prediction systems.

References

[1] D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson, "Using the support vector machine as a classification method for software defect prediction with static code metrics," in *EANN 2009*, 2009, pp. 223–234.

[2] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *Software Engineering, IEEE Transactions on*, vol. 33, no. 1, pp. 2–13, Jan. 2007.

[3] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings," *Software Engineering, IEEE Transactions on*, vol. 34, no. 4, pp. 485–496, 2008.

[4] F. Xing, P. Guo, and M. R. Lyu, "A novel method for early software quality prediction based on support vector machine," in *ISSRE '05: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 213–222.

[5] K. O. Elish and M. O. Elish, "Predicting defect-prone software modules using support vector machines," *J. Syst. Softw.*, vol. 81, no. 5, pp. 649–660, 2008.

[6] M. Levinson, "Lets stop wasting $78 billion per year." *CIO Magazine*, 2001.

[7] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. New York, NY, USA: Elsevier Science Inc., 1977.

[8] T. J. McCabe, "A complexity measure," in *ICSE '76: Proceedings of the 2nd international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, p. 407.

[9] I. Sommerville, *Software Engineering: (8th Edition) (International Computer Science Series)*. Addison Wesley, 2006.

[10] B. Schölkopf and A. J. Smola, *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond (Adaptive Computation and Machine Learning)*. The MIT Press, 2001.

[11] C. W. Hsu, C. C. Chang, and C. J. Lin, "A practical guide to support vector classification," Taipei, Tech. Rep., 2003.

[12] N. V. Chawla, N. Japkowicz, and A. Kolcz, "Special issue on learning from imbalanced datasets."

[13] G. Wu and E. Y. Chang, "Class-boundary alignment for imbalanced dataset learning," in *ICML 2003 Workshop on Learning from Imbalanced Data Sets*, 2003, pp. 49–56.

[14] E. Dijkstra, "Go to statement considered harmful," *Comm. ACM*, vol. 11, pp. 27–33, 1979.