Software Defect Prediction Using Static Code Metrics: Formulating a Methodology

David Philip Harry Gray

December 2012

Submitted to the University of Hertfordshire in partial fulfilment of the requirements of the degree of

Doctor of Philosophy

in Computer Science

Acknowledgements

First and foremost, I would like to thank my parents. Thank you for an amazing upbringing, for constant love and support, and for the wisdom I have learnt from you both. Without your support in so many aspects of my life, I would never have reached where I am today. Just think, if only you had chosen to stick to the original plan of naming me Philip Harry David Gray, I could of had two PhD's to my name! However, I'm proud to be DPHG PhD, I hope you're proud too.

I would now like to thank my supervisors. Firstly, to David Bowes, for believing in me. Had it not been for your advice and encouragement, I would never have undertaken a PhD. Thank you for all you have taught me. I sincerely hope your first PhD student was a memorable one, and that you have many more to come. Next I would like to thank Neil Davey, whose years of supervisory experience helped me to the end in one piece. Thanks Neil, I'm sure I'll think of you whenever I watch an episode of Red Dwarf! Next I would like to thank Yi Sun, for all the help with machine learning, and for being a great office companion. I would also like to thank Yi for when she, in the final year of my PhD, granted me use of the finest office in the whole of the STRI; I'm really going to miss that air-conditioned comfort! To the final member of my supervision team: Bruce Christianson, I would like to say thank you for sharing your profound knowledge with me, and especially for helping me get the first draft of this dissertation into shape. Lastly, I would like to say thank you to the whole of the Computer Science Department at the University of Hertfordshire. Someone I feel compelled to name explicitly is the legendary Bob Dickerson, for all the help he has given me and for opening my eyes to the wonder of Linux.

Lastly, I would like to thank my dearest, the radiant Claire Jennifer Beals. Thank you for all of your love, support, encouragement, kindness and perseverance. Thank you for all of the ways in which you help me, and the joy that you provide. Thank you for always believing in me, regardless of whether times are good or bad. Thank you for waiting for me Claire, I hope it was worth the wait.

Abstract:

Software defect prediction is motivated by the huge costs incurred as a result of software failures. In an effort to reduce these costs, researchers have been utilising software metrics to try and build predictive models capable of locating the most defect-prone parts of a system. These areas can then be subject to some form of further analysis, such as a manual code review. It is hoped that such defect predictors will enable software to be produced more cost effectively, and/or be of higher quality.

In this dissertation I identify many data quality and methodological issues in previous defect prediction studies. The main data source is the NASA Metrics Data Program Repository. The issues discovered with these well-utilised data sets include many examples of seemingly impossible values, and much redundant data. The redundant, or repeated data points are shown to be the cause of potentially serious data mining problems. Other methodological issues discovered include the violation of basic data mining principles, and the misleading reporting of classifier predictive performance.

The issues discovered lead to a new proposed methodology for software defect prediction. The methodology is focused around data analysis, as this appears to have been overlooked in many prior studies. The aim of the methodology is to be able to obtain a realistic estimate of potential real-world predictive performance, and also to have simple performance baselines with which to compare against the actual performance achieved. This is important as quantifying predictive performance appropriately is a difficult task.

The findings of this dissertation raise questions about the current defect prediction body of knowledge. So many data-related and/or methodological errors have previously occurred that it may now be time to revisit the fundamental aspects of this research area, to determine what we really know, and how we should proceed.

Contents

Chapter 1

Introduction

Contents

Software defect prediction involves the use of algorithms to predict whereabouts in a software system non-syntactic implementational errors are most likely to occur. These predictions are made based on software product and/or process metrics. In this dissertation both metric types are used and discussed. The product metrics used are *static code metrics;* these are measures of physical software source code that can be extracted at compile time. The process metrics used are *fault measurements,* which relate to past faults and are used as an indicator of software quality. Both these metric types are explained and formally defined in Chapter 2.

Defect prediction is motivated by the huge costs that are caused by software failures [Levinson 2001]. Although the growing complexity of software means that fault-free systems are typically impossible to guarantee, the objective of software developers is often to produce high quality code in a timely fashion. However, there is a trade-off between quality and timeliness. Traditional methods of improving software quality, such as independent quality reviews, are a good example of this. Sommerville states that: “Quality reviews are expensive and time-consuming and inevitably delay the completion of a software system. Ideally, it would be possible to accelerate the review process by using tools to process the software design or program and make some automated assessments of the software quality. These assessments could check that the software had reached the required quality threshold and, where this has not been achieved, highlight areas of the software where the review should focus.” [Sommerville 2006] This is the task of software defect predictors, tools that are built for the purpose of automatically prioritising which parts of a software system should be subject to further examination before release. In this dissertation the focus is on using static code metrics in conjunction with historic fault data to gain insight into past software quality. From here predictions can be made regarding the quality of future code units using only their static code metrics, which can be collected cheaply, quickly and easily. It is hoped that such tools will enable software to be produced more cost effectively, and/or be of higher quality.

1.1 Overall Summary

In this dissertation many data quality and methodological issues are discovered in the current defect prediction literature. The primary data source in my experiments is the NASA Metrics Data Program Repository, which is introduced in Chapter 4. These data sets have been heavily used in prior studies; however, they contain many data quality issues, some of which have received little attention from the defect prediction community. At the most fundamental level, there is so little information available regarding how the data was constructed that it is impossible to know what the primary error data actually describes. More practically, there are many data points containing seemingly impossible values, and also many data points that are redundant (repeated). As shall be demonstrated in Chapter 6, repeated data points can be the cause of serious problems in classification experiments. Obtaining accurate data that is suitable for defect prediction experiments is very difficult (see Chapter 7); this has led to the prosperity of public-domain fault data repositories. However, many of the data sets in these repositories, prior to the work described in this dissertation, seemed to be lacking a substantial analysis. The findings of this dissertation indicate that the blind faith in the quality of these data sets may have been a substantial error.

In addition to data quality issues, problems relating to machine learning methodology are also discovered. On many occasions fundamental machine learning assumptions have been broken, often relating to models being built with prior knowledge of the data they will later be assessed on. Another problematic area is that of quantifying predictive performance; as will be shown in Chapter 6, the characteristics of many fault data sets make this task potentially more difficult than may be initially perceived.

All of these issues lead to a new proposed methodology for software defect prediction, and the realisation that the findings from much prior work may be compromised. The aim of the methodology is to be able to obtain a realistic estimate of potential real-world predictive performance, and also to have simple performance baselines with which to compare against the actual performance achieved. The methodology includes approaches that can be used to address many of the issues previously described; however, serious questions regarding the quality and suitability of the data remain.

1.2 Dissertation Outline

The following is an overview of each subsequent chapter in this dissertation:

- **Chapter 2** gives an overview of software metrics and defects, including an introduction on how to go about collecting data that is suitable for defect prediction experiments.

- **Chapter 3** gives an overview of machine learning, with particular focus on supervised learning, where classifiers learn from examples of previously labelled data.

- **Chapter 4** presents a literature review of the studies most relevant to this dissertation. In this literature review I begin detailing the shortcomings of many well-cited studies, as one of my major contributions is highlighting the extent to which methodological errors have been made in prior work.

- **Chapter 5** describes the first two major experiments carried out during my PhD study. Both these experiments made use of support vector machine classifiers, which feature heavily throughout the dissertation. The first experiment aimed to provide an estimate of current, state-of-the-art predictive performance, while the second aimed to analyse the inner workings of these state-of-the-art classifiers. This chapter contains details of methodological shortcomings made in my own early work.

- **Chapter 6** contains details of the data quality issues that I have discovered regarding the NASA Metrics Data Program data sets, which have been heavily used in prior studies. I present a novel data cleansing algorithm to address these issues, and also show how one of the issues in particular (namely: the issue of repeated data points) may have compromised much prior research. I then go on to contribute to the current discussion regarding the use of various classifier performance measures, showing that the exclusion of one measure in particular (namely: precision) may lead to misleading results. This chapter ends with a repeat of the classification experiment described in Chapter 5, where many of the major issues described thus far are addressed.

- **Chapter 7** gives a description of a study to try and obtain new fault data suitable for defect prediction from an open-source system. The motivation for this study came from the data quality issues found with the NASA data sets described in Chapter 6. The findings from this experiment highlight how extremely difficult it is to obtain accurate software fault data, and how the lack of documentation available for public-domain fault data sets is a big issue.

- **Chapter 8** presents a new methodology for software defect prediction, which is mainly a synthesis of points raised earlier in the dissertation. This methodology includes a novel approach to dealing with genuine repeated data points in classification experiments.

- **Chapter 9** concludes this dissertation, reviews my contributions to knowledge, and highlights potential avenues of future work that should now be explored.

1.3 Contributions

The main contributions made in this dissertation include highlighting the issues with much prior work in this area, especially with studies that have made use of the NASA Metrics Data Program data sets. Many issues with these data sets are documented, and approaches to deal with them explained. The methodology proposed in Chapter 8 contains guidelines that can be used in future studies irrespective of the data used. It is hoped that this methodology will increase the rigour with which future studies are performed. A more detailed list of the contributions made in this dissertation is given in Section 9.2.

Chapter 2

Software Metrics & Defects

Contents

Software metrics are direct or indirect measurements of either a software artefact or a software development process. The two main types of software metrics are product metrics and process metrics. Product metrics, such as *static code metrics,* are based on a software artefact. Process metrics, such as *fault measurements,* are based on a software development process. In this dissertation both metric types are used and discussed.

This chapter begins with a discussion of the most widely used product source metrics, *static code metrics.* These can be extracted from source code at compile time. The motivation to introduce these metrics is that they typically comprise the independent variables used during defect prediction. Following on from this is a discussion regarding software defects and their measurement. This is because such measures typically comprise the dependent variable used during defect prediction.

2.1 Static Code Metrics

Static code metrics are direct measurements of source code that can be used in an attempt to quantify various software properties. These are properties that may potentially relate to code quality, and therefore to defect-proneness. Measurement in software engineering has been motivated by many factors, including that "you cannot control what you cannot measure" [DeMarco 1986], and that precise and frequently used metrics are commonplace in many other scientific domains [Halstead 1977].

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
        int return_code = 0;

        if (argc < 2) {
                printf("No Arguments Given\n");
                return_code = 1; }

        // print each command line argument
        int x;
        for(x = 1; x < argc; x++)
                printf("'%s'\n", argv[x]);

        return return_code;
}
```

Figure 2.1: An example C program.

Perhaps the most well-known static code metrics are based on lines of code (LOC) counts, and give an indication of software size. Consider the C program shown in Figure 2.1. Here there is a single function called *main*. This function occupies 15 consecutive lines of the containing file; however: 3 of the lines are empty, 2 of the lines contain only a single bracket, and 1 of the lines contains only comments. For these reasons, there are various types of LOC-counts; examples of these include: *LOC total, LOC executable, LOC comments* and *LOC non-blank*. Unfortunately, details on precisely what has been measured are often omitted, leading to ambiguity when LOC-count values are given [Jones 2008]. Problems with traditional LOC-counts also include that they are sensitive to coding style. Looking at Figure 2.1, one or more of the brackets used in the *if* statement could have been placed on their own line, potentially affecting LOC-count-based measurements. However, such differences may not be of practical importance so long as coding style remains consistent [Rosenberg 1997]. In practice this often leads to code-indentation tools being used prior to metrics collection. Despite the issues with LOC-counts, they are the most frequently used static code metrics, and if measured consistently can provide some insight into software size [Fenton 1998].

While LOC-count-based measures aim to provide insight into software size, another set of metrics, proposed by Maurice Halstead in 1977, also aim to provide insight into code complexity and developer effort [Halstead 1977]. The Halstead metrics are based on the following four base measures:

- The number of unique operators: n_1
- The number of unique operands: n_2
- The total number of operators: N_1
- The total number of operands: N_2

Using these base measures the following derived measures can be calculated:

- *Halstead Length:* $N = N_1 + N_2$
- *Halstead Vocabulary:* $n = n_1 + n_2$
- *Halstead Volume:* $V = N * \log 2(n)$
- *Halstead Difficulty:* $D = (n_1 \ / \ 2) * (N_2 \ / \ n_2)$
- *Halstead Level:* $L = 1 \ / \ D$
- *Halstead Effort:* $E = D * V$
- *Halstead Content:* $C = L * V$
- *Halstead Error Estimate (number of validation bugs):* $B = V \ / \ 3000$
- *Halstead Programming Time (seconds):* $T = E \ / \ 18$

The length metric is the sum of all operators and operands, and is an alternate size measure to those based around LOC-counts. The vocabulary metric is the sum of all unique operators and operands; code with a high vocabulary is thought to be hard to read and therefore difficult to maintain. The volume metric describes information content in bits, and is another size related measure. The difficulty metric was claimed to measure how difficult the code was to write, and therefore how error-prone it is likely to be. The complement of this is the level metric, with a lower level thought to indicate less error-prone code. The effort metric is used to measure the effort to comprehend and therefore maintain code, while the content metric was claimed to be a language independent complexity measure. Perhaps the most unjustified of these measures are the error estimate and time to program ones, as both include unfounded constants. For the error estimate measure, the proportion of defects within all systems is assumed to be constant, whereas for the time to program measure, the number of programmer elementary mental decisions per second is assumed to be constant. For reasons such as these, the Halstead metrics have been repeatedly and heavily scrutinised [Hamer 1982, Shen 1983, Fenton 1998].

Metrics concerned solely with code complexity were proposed by Thomas McCabe in 1976 [McCabe 1976]. The most well-used of these is known as the *cyclomatic complexity,* and is based on program control flow. This metric measures *linearly independent paths,* and is equal to the upper bound of required unit tests for *basis path coverage.* Basis path coverage is a white-box testing technique where it is ensured that all executable statements have been run at least once. This is a good starting point when building automated test suites, meaning that cyclomatic complexity has practical worth. Although cyclomatic complexity has direct practical implications, its use as a complexity measure has been widely questioned [Shepperd 1988, Fenton 1998]. The main arguments are that it is based on poor theoretical foundations, and that it does not fully capture what is intuitively perceived as complexity.

Other metric suites have been proposed, such as those specifically designed for the object-oriented paradigm [Sommerville 2006, Chidamber 1994]. Such metrics are typically based around degrees of coupling and cohesion, as well as the depth and suitability of inheritance trees. Although defect prediction studies have been carried out using these metrics [Catal 2007], they are not used within this dissertation.

When analysing static code metrics, it is important to know the level of granularity at which they were captured. Common granularities include the file and package level, as well as the module level. In this dissertation the term *module* is used as a generic term to refer to the function or method level. However, it is worth noting that the term *module* has different interpretations between authors, which can result in ambiguity during meta-analysis of studies [Hall 2012]. An additional necessity when analysing metrics is to know the programming language of the measured code. This can be more problematic than initially perceived, as many systems comprise more than one language. The varied abstraction levels of programming languages mean that metrics are typically language dependent; for instance, a line of a high-level language (such as Python) will typically achieve more than a line of a low-level language (such as C).

Because static code metrics are calculated through the parsing of source code, their collection can be automated. Thus it is computationally and resourcefully 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 [Sommerville 2006]:

- **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.

2.2 Software Defects

Throughout this dissertation the terms: *defect, fault* and *bug* are all used interchangeably. These terms refer to the manifestation of an *error* in source code, where an error is an erroneous action made by a developer. Faults can be the cause of *failures,* which occur when users experience undesirable system behaviour at a specific point in time. In general, *syntax errors* are not regarded as faults; this is because they can be found with far more efficiency by parsers and/or basis path testing. These definitions are based on those given in [iee 1990], although it is worth noting that they are not universally agreed upon [Fenton 1999].

Recording the number of faults found within a unit of software provides a simplistic method of partly assessing its quality. However, there are many different types of fault measurement. During software development, the number of bugs found during the testing phase can be recorded. This is an example of pre-release fault measurement; however, post-release (or latent) fault measurement is also possible. The most prevalent form of post-release fault measurement is to record how many failures were experienced up until a specific point in time after a release. It is worth pointing out that such measures are to be used with caution, as the total number of faults within a unit of software ($discovered\ defects + residual\ defects$) is typically never known. This is because of the computational time and complexity required to determine such a value. Therefore, it is folly to assume that a unit of software with many defects removed will be less fault-prone than another with few defects removed, or the converse. Additional problems with such fault counts are that fault severity is not always considered. Although distinguishing fault severity is a challenging task, it is dangerous to have no distinction between trivial and life-threatening faults.

2.2.1 Software Defects & Defect Prediction

In defect prediction experiments, a single fault variable, such as *'the number of faults discovered during system testing',* is typically the dependent variable. In regression experiments, such measurements are often used as is, and the number of faults within each software unit predicted. Note that it is not necessary for the predicted number of faults within each unit to be highly accurate in order for the classifier to be of practical worth. This is because the numbers of predicted faults can be ranked in descending order, producing a ranked list of the seemingly most defect-prone components. Code inspections can then be prioritised around this ordering. It is also possible to normalise the predicted numbers of faults by a size estimator such as LOC total, and then rank the software units in descending order of predicted defect density.

As well as regression experiments, classification experiments are also increasingly popular. Classification experiments involve software units being grouped into categories or *classes*. Although there is theoretically no limit to the number of classes used in such experiments, two class (or binary) classification experiments are most common. There is typically a 'defective class' and a 'non-defective' class. Because of this, a pre-processing transformation of the fault count measurements is often required. The most common transformation is to map all software units with no reported faults to the non-defective class, and map all others to the defective class. Such a mapping has been carried out in many studies, including: [Menzies 2007b, Lessmann 2008, Elish 2008]. Note that as well as the issue regarding fault severity mentioned previously, there is now additionally the issue of fault quantity information being lost. This is clearly a radical simplification, and one that is made solely for the benefit of the learning technique. Although classification experiments produce categorical predictions, it is often still possible to produce a defect-proneness ranked list of software units, as is the case with regression experiments. This is achieved by extracting some internal classifier confidence value associated with each prediction, and then ranking the software units accordingly. There will be more discussion on this in Section 5.2.

2.2.2 Defect Measurement

As opposed to static code metrics that can be accurately, quickly and easily extracted, fault measurements are more challenging to obtain. This is because fault measurements are often indirect, rather than direct measurements. A user of a system may be a developer, a tester, or an end user. When such a (human or automated) user experiences a software failure, the occurrence can be recorded. While this typically provides an overall measure of error for the system or subsystem in question, it is often more desirable to obtain fault measurements at a lower level of granularity. This involves the failure inducing fault(s) being isolated and localised, in order to find the offending code unit(s). The level of localisation required depends on the level of granularity required. As with static code metrics, common granularities include the package, file, class and module level. Note that it is often possible to aggregate low-level granularity measures to obtain higher level ones.

The most common methods for localising faults involve analysis of the system history recorded in revision control systems. Revision control systems, such as *concurrent versions system (CVS)*, record who changed what, when, how, and (ideally) why [Śliwerski 2005]. Such systems enable users to submit a textual message with each commit (set of file changes). The purpose of this message is to describe the changes being made, although there are no guarantees regarding its accuracy. Despite this, in cases where such messages are consistently entered and of reasonable quality, it is often possible to use them to determine when and where supposed fault-fixes have occurred. This has been carried out in previous studies using simple text-based searches [Mockus 2000, Śliwerski 2005]. After identifying a supposed fault-fix, an assumption can be made that the previous revision to the one in question contains faulty code. A more advanced approach than simple text-based searching is known as the *SZZ algorithm,* and involves progressively working backwards through each previous revision. A search is carried out for the most recent revision(s) to have changed the code being 'fixed' in the current revision. If any such revision is found, it is known as a *fix-inducing change:* a change that caused a 'problem' that later needed to be 'fixed'. The SZZ algorithm was first proposed in [Śliwerski 2005] and later extended in [Kim 2006] & [Williams 2008].

Rather than using revision control log messages alone, the SZZ algorithm was originally proposed to additionally make use of bug reports. Such reports are often maintained throughout the lifetime of a project using purpose built database-backed bug-tracking software. An example of such a software system is *Bugzilla*[1]. For projects that have utilised such a system, the process for localising faults described above can be extended. Searching for all bug reports that have been assigned the 'fixed' status results in a set of unique bug identifiers. As it is common practice for developers to include a bug report number in the commit message of a 'bug fix' [Śliwerski 2005], commit messages can be searched for these identifiers, in addition to other indicators of a fault-fix (such as the word: 'fixed'). This is claimed to increase the precision achieved when using log messages alone [Śliwerski 2005], resulting in more confidence as to whether or not a commit was an intended fault-fix.

An alternative method for identifying fault-fixing revisions was proposed in [Ostrand 2005], following a recommendation by developers. This simple approach classifies revisions using information regarding the number of files changed in each commit. Commits containing one or two changed files are classified as fault-fixing, all others are considered not so. Although this method is very simplistic, it was claimed to work well with a sample of the proposing authors' data [Ostrand 2005]. The main advantage of this method is that it can be used in cases where commit messages have not been entered appropriately.

Perhaps the most accurate way to determine whether or not a commit is an intended fault-fix is to manually classify the textual difference between it and its previous revisions. This method requires an expert and is labour intensive, often making it infeasible even for small systems. The automated methods previously described were motivated by this frequently encountered infeasibility. Although manual classification is likely the most accurate way to classify commits, "it is very difficult to reliably extract fault-fixing data from change repositories" [Hall 2010]. This is because of the expertise required to do so, the subjective nature of manually classifying fault-fixes, and also because the historical data may be lacking in quality.

The issues involved in extracting fault data from change repositories are revisited in Chapter 7, which contains a description of the work undertaken to extract fault data from the *Barcode* open-source system.

[1] `http://www.bugzilla.org/`

Chapter 3

Machine Learning

Contents

Machine learning, which is closely related to data mining, is a broad area of computer science regarding algorithms that enable computers to *learn* [Segaran 2007]. This often involves "the extraction of implicit, previously unknown, and potentially useful information from data" [Witten 2005]. Perhaps the most popular area of machine learning is *supervised learning,* where predictions are made regarding future events based on knowledge of past events. Another popular area of machine learning is *unsupervised learning,* which is concerned with finding hidden structure within data. Machine learning is possible because almost all non-random data contains underlying patterns [Segaran 2007]. When these patterns are discovered they can then be exploited.

This chapter begins with an introduction to supervised learning, or learning by example. Basic supervised learning techniques are then described, those most relevant to this dissertation. Methods to assess the predictive performance of classifiers are discussed in Section 3.3, where the main difficulties are introduced. Section 3.4 leads on from this with a brief introduction to *the class imbalance problem.* The process of classification model optimisation is detailed in Section 3.5, a process key to successful utilisation of many classification methods. A family of such classification methods are described in Section 3.6, namely, *support vector machines.* These highly sophisticated classifiers have been heavily used and are extensively referred to throughout this dissertation.

3.1 Supervised Learning

Supervised learning is a sub-discipline of machine learning, which is in turn a sub-discipline of artificial intelligence. Supervised learning algorithms are trained using labelled data. Such data ($\boldsymbol{x}$) comprises p features (or attributes) and q feature vectors (vectors hereafter). Additionally, there are q class labels (labels hereafter), each one corresponding to a single vector. The concatenation of a vector with its label is known as an instance or data point. Although it is possible for a label to comprise multiple values, this is not considered in this dissertation. If the labels within a data set form a set of nominal values, then this is considered a *classification* problem, whereas if the labels form a set of continuous values, this is considered a *regression* problem. In this dissertation the main focus is on classification problems where the cardinality of the set of labels is two. This is known accordingly as binary classification. Example training data suitable for a binary classifier is shown in Table 3.1. This data comprises: two features (lines of code and cyclomatic complexity), six feature vectors, and their six corresponding labels. Each data point describes two quantitative features of a software unit (the *independent variables*), as well as whether or not that software unit caused any operational failures (the *dependent variable*). This data is suitable for a two class (binary) classifier, as the set of labels consists of two elements.

Lines of Code	Cyclomatic Complexity	Defects?
7	1	No
500	31	Yes
5	2	No
12	3	Yes
84	6	No
90	13	No

Table 3.1: Example data suitable for a supervised learning algorithm.

The aim of supervised classification learning methods is to use labelled data to generate a mapping function $f(\boldsymbol{x})$ such that when a vector is presented, its correct label is returned. Minimising the error of this process is often the most important learning algorithm criteria. The data used in this process should be a representative sample from the specific problem domain. However, only a very small proportion of all possible instance combinations (the *input space*) is typically contained within the available data. Therefore, learning methods are required to generalise in order to successfully predict *unseen data,* data that was not used while constructing the classifier, but that will almost certainly be found in the real world. Generalisation is therefore very often the key to successful data mining.

3.2 Basic Learning Techniques

3.2.1 Instance-Based Learning

Instance-based (or case-based) learning requires each and every training instance to be stored verbatim [Witten 2005]. Such algorithms are said to be lazy, as rather than building a classification model during initial training from which all predictions are based, the original training data must instead be consulted for each prediction. The main advantage of this is that training data can be added, removed or modified dynamically. Disadvantages are that large data sets can result in slow algorithm operation, and that there is no classification model from which new knowledge can be easily extracted (as can be the case with decision trees (Section 3.2.3), for example).

For all but the most simple forms of instance-based learning, generalisations can occur. This is via a search for the most similar vector(s) to the one being classified. Similarity is typically determined by a distance function, with the corresponding label(s) of the nearest vector(s) to the one being classified used to determine the prediction made. The assumption is that instances of the same class will be relatively nearby in the feature space. The distance function used depends on the type of features contained within the data. For numeric features (such as those used throughout this dissertation), the most common distance function is the Euclidean distance. This is defined in Equation 3.1, where the distance between two vectors (x_i and x_j) is calculated using all p features.

$$Euclidean\_Distance(\boldsymbol{x}_i, \boldsymbol{x}_j) = \sqrt{\sum_{d=1}^{p} \left(\boldsymbol{x}_i^d - \boldsymbol{x}_j^d\right)^2}. \tag{3.1}$$

3.2.1.1 Rote Learning

Rote learning, or learning by memorisation, is one of the simplest forms of supervised learning [Noyes 1992]. Rote learning involves the storage of training data into some form of database. When required this data can then be recalled, potentially with great efficiency. The limitation of rote learning is that only vectors included in the training data can be classified. This is because the training data has only been stored, no attempt to infer or generalise has been made. For this reason rote learning systems are mainly used where the correct label for every possible combination of features is known beforehand. This situation is not stereotypical of machine learning problem domains. However, rote learning systems can additionally be used as components of ensemble learning methods, perhaps to efficiently classify parts of the input space where the correct labels are known.

3.2.1.2 Nearest-Neighbour-Based Learning

An extended and far more robust classification method than rote learning is nearest-neighbour-based learning. The most basic nearest-neighbour-based learning algorithm is *one-nearest-neighbour*. This algorithm performs the same outward function as rote learning for cases where the test vector is included in the training data. When this is not the case, the training data is searched for the instance whose distance to the test vector is minimal. The test vector is predicted as belonging to the same class as the instance found, its nearest-neighbour (see Figure 3.1).

The more general form of this concept is *k-nearest-neighbour*, where $k \geq 1$ nearest training instances are found, and a majority vote is taken on the class of the test vector. Accordingly, k should be defined as an odd number for binary classification problems, to prevent ties during voting. The potential benefit of using a higher value of k is that it results in a clearer picture of whereabouts in the data space the test vector lies. This is because more information regarding the test vector's surrounding instances is taken into account (see Figure 3.2). The optimal value of k is data specific, and typically cannot be known a priori. However, a model optimisation phase (see Section 3.5) may be used to aid its estimation.

Although intuitive and easily comprehensible, nearest-neighbour-based learning methods are very sensitive to both noise (incorrect or erroneous data) and irrelevant attributes (those that are uncorrelated with the class). This is because all features typically have an equal weight when similarity is determined (see Equation 3.1). Because of this, data cleansing is often a prerequisite to successfully using such techniques, and cleansing methods specific to instance-based learners have been proposed [Segata 2010, Segata 2009]. An additional problem with nearest-neighbour-based learners is that they are very sensitive to the class distribution of

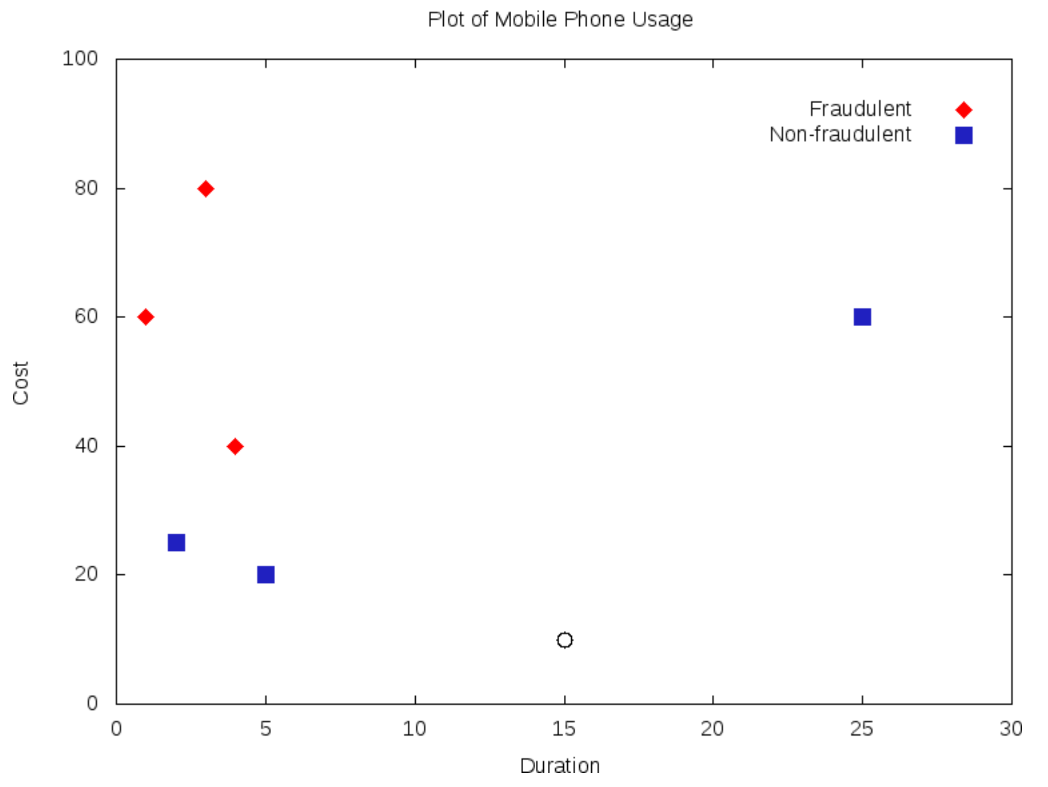


Figure 3.1: An example of the (one) nearest-neighbour algorithm. Here is a plot of genuine and fraudulent mobile phone usage. The two features are call cost and call duration. Fraudulent calls are represented by diamonds whereas non-fraudulent calls are represented by squares. The test data (represented by a circle) is predicted as belonging to the non-fraudulent class, as this is the class of its nearest data point.

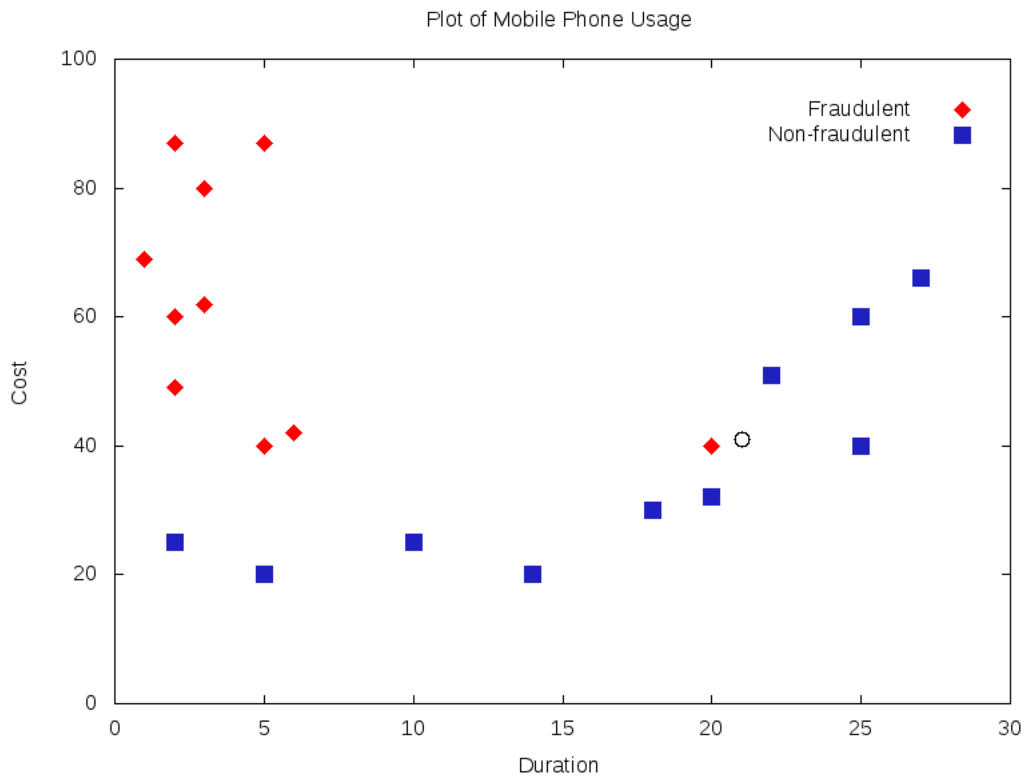


Figure 3.2: Another k-nearest-neighbour example. Although the test vector (represented by a circle) appears to be more similar to the non-fraudulent class, k will need to be 3 or more for this to be the prediction. Note that the fraudulent example nearest to the test vector is an *outlier*, as it is not representative of its class. It may be that this data point has an incorrect label, perhaps due to a data quality issue.

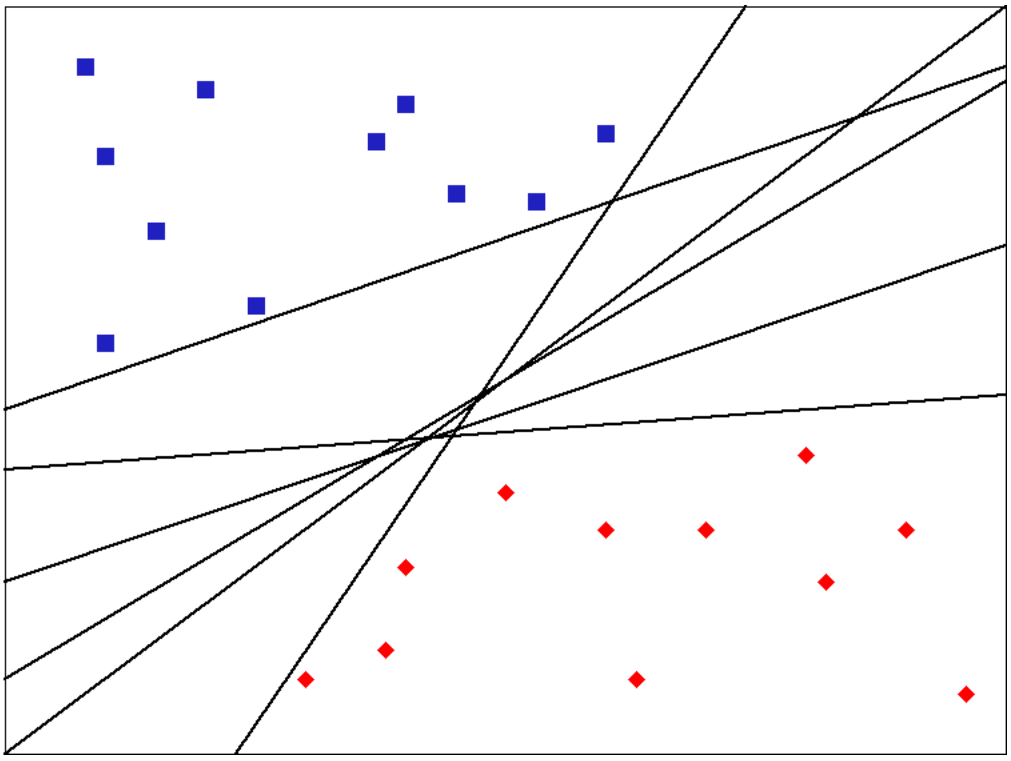


Figure 3.3: Some of the possible separating lines of a linear separator.

the training data. If the majority class (the class that appears most often in the training data) contains 90% of instances, the probability of having a majority class nearest-neighbour is higher than the probability of having a minority class nearest-neighbour. In such cases, the majority class will usually be over-predicted, meaning that potentially all test vectors will be predicted as belonging to the majority class.

3.2.2 Linear Separators

Linear separation techniques usually involve using linear combinations of numeric features to separate classes. An example of this is shown in Figure 3.3. In this figure, several possible linear separators are shown, all of which separate both classes without error. An example of an algorithm that can generate such separators is *the perceptron* [Rosenblatt 1958]. It follows that this is a non-deterministic algorithm, as there are many possible solutions shown in Figure 3.3. In fact, there are an infinite number of solutions to this problem.

Although each of the separators shown in Figure 3.3 separate both classes without error, we ideally wish to maximise performance on unseen data, not simply the training data. The most intuitive way to try and ensure best performance on unseen data is to have a separator with a large *margin,* a large distance between both classes. Having a large-margin separator, as opposed to a small-margin separator, generally results in improved performance on unseen data.

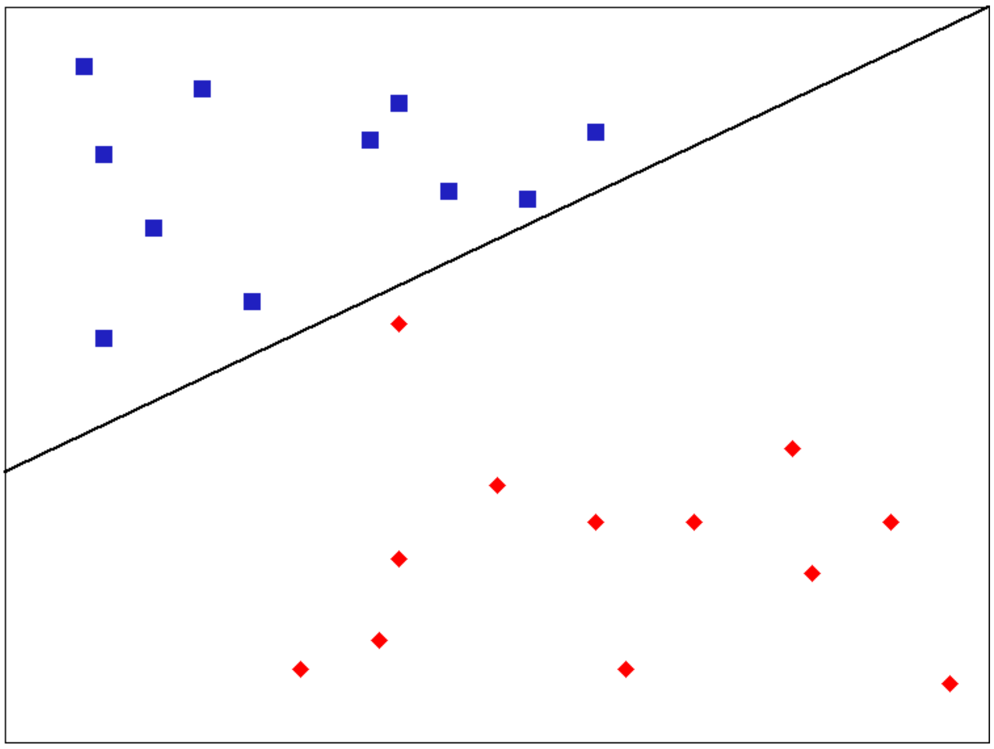
Because margin size is so influential to generalisation ability, it may be beneficial for predicting unseen data to have a non-perfect separator. This is a separator where there are training instances which fall on the wrong side, the side that is not shared with the majority of their class. Methods which do not allow this to occur produce *hard-margin* separators, whereas methods which do allow this to occur produce *soft-margin* separators. The difference between these two types of separators is shown in Figure 3.4.

Hard-margin separators generally do not involve parameters; however, soft-margin separators typically require a slack parameter to determine the trade-off between minimising the training error and maximising the size of the margin. This is known as the *cost* or c parameter, and is tunable to user requirements. A low cost value generally results in many training errors and a large margin, whereas a high cost value generally results in few training errors and a small margin. In fact, cases where $c = \infty$ are conceptually identical to hard-margin separators. The optimal value of c is data dependent, and is often estimated using a systematic search (see Section 3.5).

The main disadvantage of linear separators is that they are limited to linearly separable problems. However, many problems are linearly separable, especially when there are a large number of features (resulting in a high-dimensional space). The main advantages of linear separators are that they are easily comprehensible and computationally efficient, in that they are suitable for huge data sets with large numbers of features.

3.2.3 Tree-Based Learning

Tree-based (or decision tree) learning algorithms produce a structured classification model during initial training on which all predictions are based. Such methods are said to be eager, as all computation regarding future predictions is made in the training period (the raw training data is not revisited at classification time). A recursive, top-down method is usually employed to construct trees. This is known as the *divide-and-conquer* approach [Witten 2005], and can be solved recursively. The top (or root) node branches (or splits) at the feature which best separates the data into homogeneous subsets with the same label. Various methods exist for determining this feature; however, the most well-known decision tree algorithms use either entropy [Quinlan 1993] or Gini index [Ceriani 2011] based measures. An example graphical representation of a decision tree is shown in Figure 3.5. This figure shows a possible decision tree that sailors could use to determine whether or not it is safe to sail under certain conditions. In this figure it can be seen that *Wind Speed (mph)* is the feature at the root node, and that its test condition is whether or not it is greater than 22. If it is, a prediction of *Don't Sail* is made. If it is not, the test condition at the following node is evaluated. This process continues until a prediction is made. Therefore, each prediction can be seen as the conjunction of one or more simple test conditions.


(a) Hard margin. All training data is correctly classified, but with a small margin.

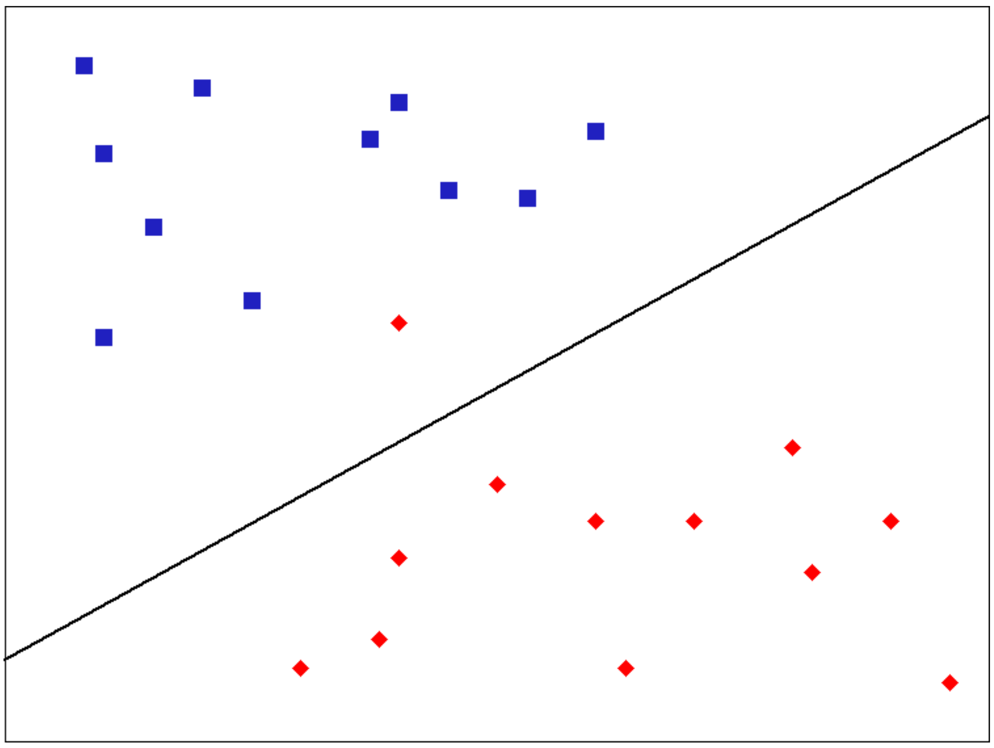

(b) Soft margin. By allowing training error, the size of the margin is increased.

Figure 3.4: Hard versus soft-margin linear separators.

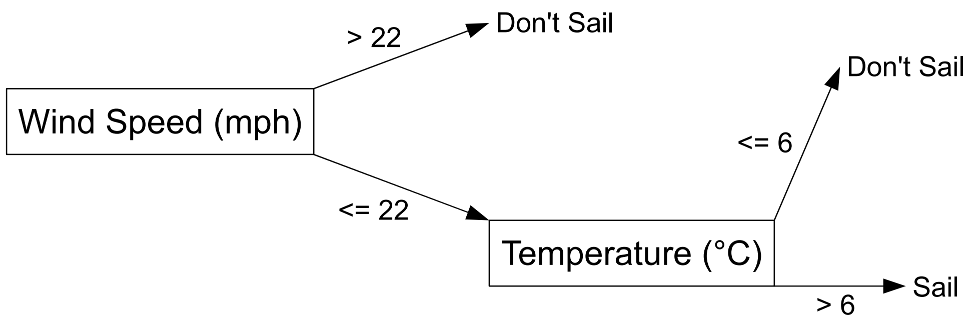


Figure 3.5: An example decision tree.

3.2.3.1 Random Forests

Random forest classifiers are strictly an ensemble classification method, but they are introduced in this section as they are comprised solely of multiple decision trees. A random forest classifier comprises two or more CARTs (classification and regression trees) and utilises a *bootstrap aggregating* (or *bagging*) ensemble approach [Breiman 2001]. Additionally, each of the trees are said to be *randomised,* as they only train on a random subset of $a < p$ features (where p is the total number of features available). The mode classification across all individual classifiers is taken as the final prediction for each test vector.

Bagging is an ensemble approach whereby each individual node (or tree in this case) is equally weighted in the final prediction [Breiman 1996]. Other ensemble approaches, such as *boosting,* differ from this as they weight each node judging by past performance. To encourage variation in the results of nodes, the bagging approach provides each node with a sampled version of the training data. This *bootstrap* sample is both uniform and of the same size as the original sample. It is also made *by replacement,* resulting in many training instances being repeated, and some not making it into the new sample. Separate to bagging, but also designed to encourage variation in results, is the method of passing a random subset of training features to each node. This technique is known as randomisation. Combining the power of multiple decision trees by using random forests "often produces excellent predictors" [Witten 2005].

3.2.4 Bayesian Classifiers

A Bayesian (or Bayes) classifier is a linear classification technique based on Thomas Bayes' theorem. This theorem (presented in Equation 3.2) is concerned with probability theory, and provides a method to determine *conditional probabilities.*

$$\Pr(hypothesis|evidence) = \frac{\Pr(evidence|hypothesis) \times \Pr(hypothesis)}{\Pr(evidence)} \tag{3.2}$$

Perhaps the most popular Bayes classifier is the naïve Bayes algorithm, so-called because of the simple but often false assumption of *conditional independence,* where every feature is assumed to be fully independent. Despite this assumption, naïve Bayes classifiers have been reported to perform competitively with far more sophisticated techniques [Huang 2003]. Similar to instance-based learners, naïve Bayes classifiers can be updated over time, allowing model evolution. Unlike instance-based learners however, naïve Bayes models are independent of the training data (it does not require storage). Because of this, and low computational demands, naïve Bayes is a suitable method for potentially huge, constantly evolving domains, where timely execution may be required. An example of such a domain is that of spam email classification, where naïve Bayes classifiers have been extensively used [Vangelis Metsis 2006].

During training, naïve Bayes classifiers use basic statistical properties of each feature to determine their probabilities of association with each class. The advantage of this is that models can be easily interpreted, as the probabilities for each feature are stored explicitly. This ease of human interpretation of the inner workings of the model makes naïve Bayes a *white-box* classification method. This is similar to both tree-based and instance-based learners, amongst others. White-box methods are typically preferable to black-box methods, as it is always possible to discover the reasoning for each prediction made. Additionally, new knowledge can potentially be discovered via model analysis. In the case of naïve Bayes, this would involve examining the probabilities associated with each feature.

Naïve Bayes classifiers determine, for each test vector, a probability of association with each class. Note that these are not strictly probabilities, but can be conceptualised as them nonetheless. In the simplest of implementations, the class with the highest probability will be the predicted class. However, many implementations include a user-tunable *decision threshold* parameter. For binary classification tasks, this is described as follows. If the probability of association with the class of most interest is greater than the decision threshold, then this will be the predicted class. Otherwise, a prediction of the opposing class is made. Therefore, the decision threshold effectively controls how eager (or not) the classifier is to make predictions of a certain class.

3.3 Assessing Predictive Performance

Assessing the predictive performance of a classifier is more complicated than may be initially perceived. For rote learning methods where no generalisation takes place, the quality of predictions is clearly a direct result of the quality of the training data. For this reason, algorithm speed is often the performance criteria assessed, rather than predictive ability. As this is not a focus of this dissertation, it is not discussed further. This section instead describes ways in which the predictive performance of a classifier can be quantified.

3.3.1 Training & Testing Sets

When assessing the performance of a classifier, the most common aim is to acquire an estimate of how well it could potentially perform in the real world. Because (as mentioned at the start of this chapter) only a very small proportion of the input space is typically contained within the available data, it is often entirely unrealistic to assume that a classifier deployed in the real world would be required to classify vectors included in the training data. For this reason, classifiers are required to be tested on *unseen data:* data that was not used *in any way* during classifier construction. Failure to adhere to this principle often results in an optimistic approximation of potential real-world performance [Witten 2005]. This is because the classifier has already learnt from the 'correct' label of each *seen* test vectors, and therefore has an unfair and often unrealistic advantage.

A classifier deployed in the real world would usually be constructed using all available data for the specific problem domain. Each prediction made by the system would then have to be (often manually) validated, which is typically a resource intensive task. To avoid this resource intensive process while classifier feasibility is being explored, an estimate of potential real-world performance is obtained instead, using only the data already available. A simple method of doing this is known as a *holdout,* where $x\%$ of available data is used for constructing a classifier, while the remaining $(100-x)\%$ is kept separate during classifier construction, and used to test how well the classifier can generalise. Thus, these subsets of data become known as a training set and a testing set, respectively (note that these are not mathematical sets). For both these sets to be fully independent of each other, and contextually valid, they must share no common instances. The classifier is trained on the training set, and performance on the independent test set is used as the estimate of potential real-world performance. Some of the various methods of measuring performance will be explained in the next few sections. In order to defend against statistical bias, the holdout process is often repeated many times, with different training and testing set samples pseudo-randomly chosen. The average performance for each holdout experiment is then reported as the final, overall performance.

A more sophisticated method than a simple holdout is known as a *stratified* holdout, where the training and testing sets both maintain an approximately equal class distribution to the one in the original sample. Imagine there are 1000 labelled data examples available in the domain of jet engine failure prediction. There are only 100 examples where the engines failed, all others are examples of non-failures. With a two-thirds training, one-third testing holdout, it is possible (as an extreme example) that the 667 training instances contain none of the engine failure examples. Supervised learners, by definition, require labelled examples in order to learn. Therefore, a classifier trained on such data would have little chance at predicting any engine failures, and would instead be expected to predict every test vector as belonging to the majority class (non-failures). A stratified holdout would be much more effective in such an *imbalanced* domain. With a stratified holdout, two thirds of each class would be contained in the training data, and one third of each class in the testing data. As long as the original data sample is representative of the problem domain, this is a much more effective way of obtaining estimates of potential real-world performance.

Although repeating the stratified holdout process many times alleviates the bias issue, potential problems remain. Repeating the process too few times may result in some instances never being included in a testing set. Additionally, testing sets will overlap, meaning that classifiers will be presented with some test vectors more times than others. Both of these issues can introduce bias. To address these issues, *k-fold cross-validation* can be used. In k-fold cross-validation, $k > 1$ approximately equal sized and mutually exclusive sub-samples (or folds) are created from the original sample [Kohavi 1995]. Then, k holdout-style experiments are carried out, where one of the folds is the testing set, and all others combined are the training set. Each fold has one turn as the testing set, and final performance is reported as the average across all k folds (see [Forman 2010] for important details on how this average should be calculated). *Stratified k-fold cross-validation* is an extension to this method, where each fold maintains a similar class distribution to the original sample, as previously described. To further reduce bias, the stratified cross-validation process can be repeated many times, with different sub-samples pseudo-randomly chosen. Final performance is then calculated as the average of all individual cross-validation averages. Although cross-validation is computationally expensive, it is fairly stable and unbiased for adequately sized data sets, where k is reasonably large (10 folds has been recommended [Kohavi 1995]).

3.3.2 Categorising & Quantifying Predictions

Because the testing sets in such classification experiments are artificial (they are not real unknown vectors, but simulations of them), the label for each test vector is already known. Clearly, to make the task of a classifier worthwhile, these test labels are not used during the classification process. After this process is complete however, they are used to assess how well the classifier has performed, or in other words, how the predictions made by the classifier correspond with the 'correct' labels.

Perhaps the most intuitive way to quantify predictive performance is to determine the proportion of correct classifications. The pseudo-code for this is shown in Figure 3.6. This figure shows an attractive as well as intuitive solution, as it scales

```
no_correct = 0        # initialise the no. of correct classifications

for test_vector in test_set:
        if test_vector.real_label == test_vector.predicted_label:
                no_correct += 1

print no_correct / test_set.size
```

Figure 3.6: Pseudo-code for calculating the correct classification rate.

robustly regardless of the number of classes in the data. The metric describing the proportion of correct classifications is known as *accuracy,* or *correct classification rate (CCR).* Its complement is known as *error rate,* and is defined as $(1 - accuracy)$. These correct/incorrect classification rate measures are suitable when the class distribution of the test data is equal (or *balanced*), and when the *misclassification cost* for each class is also equal. An example of such a domain is optical transmission error detection, where a classifier attempts to detect bit-errors in the transmission of a digital signal. In this domain it is assumed that there will be an equal number of zeros and ones transmitted over a communication medium over time. It is also assumed to be illogical to favour the correct transmission of a zero over the correct transmission of a one. For these reasons, overall correct/incorrect classification rate measures are suitable in such a domain, and a value of approximately 0.5 is the threshold for a classifier of any practical worth. This is because when there are an equal number of test vectors originally labelled as belonging to each class, predicting all test samples solely as being in either one class or the other will result in such a performance figure. Additionally, any such performance figure (not equal to 0 or 1) provides no information toward whether there were more errors in one class than another, which is not a necessity in such a domain.

Accuracy and error rate are both measures which favour the majority class. Because of this, they are not suitable in domains where there is an imbalanced class distribution. To demonstrate, we will revisit the jet engine failure prediction scenario described earlier in this section, where there were 1000 examples of which only 100 were in the minority class. If this data set were a test set, a classifier predicting every test vector as belonging to the majority class would achieve a near-optimal accuracy (0.9), even though the classifier is of no practical worth. This scenario is also appropriate for demonstrating the other limitation of such measures. Mistakenly predicting a jet engine that is not about to fail as being about to fail will result in unnecessary servicing of the engine, which is a waste of resources. However, mistakenly predicting a jet engine which is about to fail as not being about to fail could result in an operational failure, with far more serious consequences. Accuracy-like measures give no information toward class-specific error, only overall error. For these two reasons, they would be entirely unsuitable in such a domain.

3.3.3 Measuring Class-Specific Error

To overcome the problems of accuracy-like measures, class-specific error must be measured. In a binary classification task, the two classes are commonly referred to as the positive class and the negative class. There is a symmetry between these classes; however, the positive class typically refers to the class of most interest, which is often the minority class. For each test vector predicted during a binary classification experiment, there is exactly one of four possible outcomes:

- A *true positive (TP)* occurs when a data point labelled as positive is correctly predicted as positive.
- A *true negative (TN)* occurs when a data point labelled as negative is correctly predicted as negative.
- A *false positive (FP)* occurs when a data point labelled as negative is incorrectly predicted as positive.
- A *false negative (FN)* occurs when a data point labelled as positive is incorrectly predicted as negative.

Collections of these values can be put into a *confusion matrix,* as shown in Figure 3.7. Such a confusion matrix forms the basis of how the predictive performance of binary classifiers are often quantified (see Figure 3.8). Most classifier performance measures are based on simple equations of the four comprising raw measures (TP, TN, FP and FN). In addition to determining predictive performance, these values can also provide contextual insight into the test data. Imagine a confusion matrix generated during a single holdout experiment. Using the values in the confusion matrix alone, the number of test vectors originally labelled as belonging to the positive class can be determined as $POS = TP + FN$. It follows that the number of test vectors originally labelled as belonging to the negative class is determined as $NEG = TN + FP$. Therefore, the total number of vectors in the test set is determined as $|Test| = NEG + POS = TP + TN + FP + FN$.

	labelled positive	labelled negative
predicted positive	TP	FP
predicted negative	FN	TN

Figure 3.7: A confusion matrix.

To provide a more concrete example of a confusion matrix, we will again use the jet engine failure prediction example. There are 100 positive (minority) class data points in this data set and 900 negative (majority) class data points. If this data set were a test set and we predicted everything as being in the majority class, we would obtain the confusion matrix shown in Figure 3.9. As already stated, *accuracy* is not a suitable measure for this example; however, using the confusion matrix alone, we can compute *accuracy* as $(TP+TN)/(TP+TN+FP+FN)$. Although

Predicted Class Label		Actual Class Label	
−	>>>	−	✓
+	>>>	+	✓
+	>>>	−	✗
−	>>>	+	✗
−	>>>	−	✓
+	>>>	+	✓
+	>>>	−	✗
−	>>>	−	✓
−	>>>	−	✓
+	>>>	−	✗

	labelled positive	labelled negative
predicted positive	TP = 2	FP = 3
predicted negative	FN = 1	TN = 4

Figure 3.8: The confusion matrix for a toy example.

the near-optimal accuracy of 0.9 may intuitively appear as though our classifier has performed very well, in this domain we need to examine class-specific performance to determine potential real-world performance. One such suitable measure to aid with this is known as *sensitivity* (or *recall* or *true positive rate*). This measure describes the proportion of test vectors originally labelled as belonging to the positive class that were correctly classified. To calculate sensitivity we use $TP/(TP+FN)$. In this example this yields a sensitivity of 0, showing that no minority class test vectors were correctly predicted. We can work out a similar measure for the negative class, known as *specificity* (or *true negative rate*). We determine this using $TN/(TN + FP)$. In this example the specificity is 1, showing that all majority class test vectors were correctly predicted. Perfect classification is achieved when there is a sensitivity and specificity of 1, which implies that there is also an accuracy of 1. A trade-off typically exists between sensitivity and specificity, and is explained as follows. To achieve an optimal sensitivity, all test vectors appearing to belong to the positive class must be predicted as such. However, for every prediction made, there is a risk of misclassification. Making an incorrect positive prediction means a data point originally labelled as negative has been incorrectly predicted as positive. Such a *false positive* will lower the specificity, which we are aiming to maximise along with sensitivity. Thus, trying to optimise one of these measures often compromises the other. It is worth pointing out that these measures, which originate from the medical sciences, are designed to be used together and complement each other. As shown in this example, obtaining an optimal specificity is simply a case of making only negative predictions. Similarly, an optimal sensitivity can be achieved by making only positive predictions. Both such predictors clearly have little practical worth.

	labelled positive	labelled negative
predicted positive	TP = 0	FP = 0
predicted negative	FN = 100	TN = 900

Figure 3.9: A confusion matrix with only negative-class predictions.

The trade-off between sensitivity and specificity can be visualised using a *receiver operating characteristic (ROC) graph.* ROC graphs were first used during World War II to analyse radar signals; there use in machine learning began much later in 1989 [Spackman 1989]. A ROC graph is a two-dimensional plot, with *sensitivity* on the y-axis and $(1 - specificity)$ on the x-axis. The use of $(1 - specificity)$, also known as the *false positive rate* or *type 1 error rate*, means that an optimal classifier will be shown on a ROC graph as having a sensitivity (or *true positive rate*) of 1, and a false positive rate of 0. Imagine a naïve Bayes classifier tested on the same test set three times. Each time the classifier will use a different decision threshold (see Section 3.2.4) taken from the set {0.4, 0.5, 0.6}. After this process, there will be three sets of results, all in the form of confusion matrices. From these confusion matrices, we can plot three points on a ROC graph, as shown in Figure 3.10. If we continued this process for every possible decision threshold, we would end up with a *ROC curve,* as shown in Figure 3.11. This figure demonstrates the trade-off between sensitivity and specificity for this particular classifier on this particular test set, as the decision threshold is varied. An interesting point in ROC space is point {0,0}, where the classifier makes no positive predictions, and therefore has no true or false positives. The complement of this is at point {1,1}, where the classifier makes only positive predictions, and therefore has no true or false negatives. As already stated, optimal performance is at point {0,1}. In practice, classifiers rarely achieve such performance; however, curves often bend upwards towards this ideal point. Also worthy of discussion in Figure 3.11 is the straight, $x = y$ dashed line. Any point situated below this line demonstrates performance that is worse than random [Flach 2003]. In fact, inverting the predictions made by such a classifier yields better performance, and moves the point to its corresponding position (reflection) above the dashed line.

ROC curves can be useful for classifier comparisons; if two classifiers often have similar performance in a specific domain, inspection of their ROC curves can illuminate which one performs best in specific regions of ROC space. Additionally, ROC curves can show which method is more sensitive to decision threshold setting. To compare multiple classifiers in terms of their ROC curve performance without the need for visual inspection, the area under the ROC curve (AUC-ROC, or more commonly simply AUC) can be computed. This is a value between 0 and 1, with 1 indicating optimal performance.

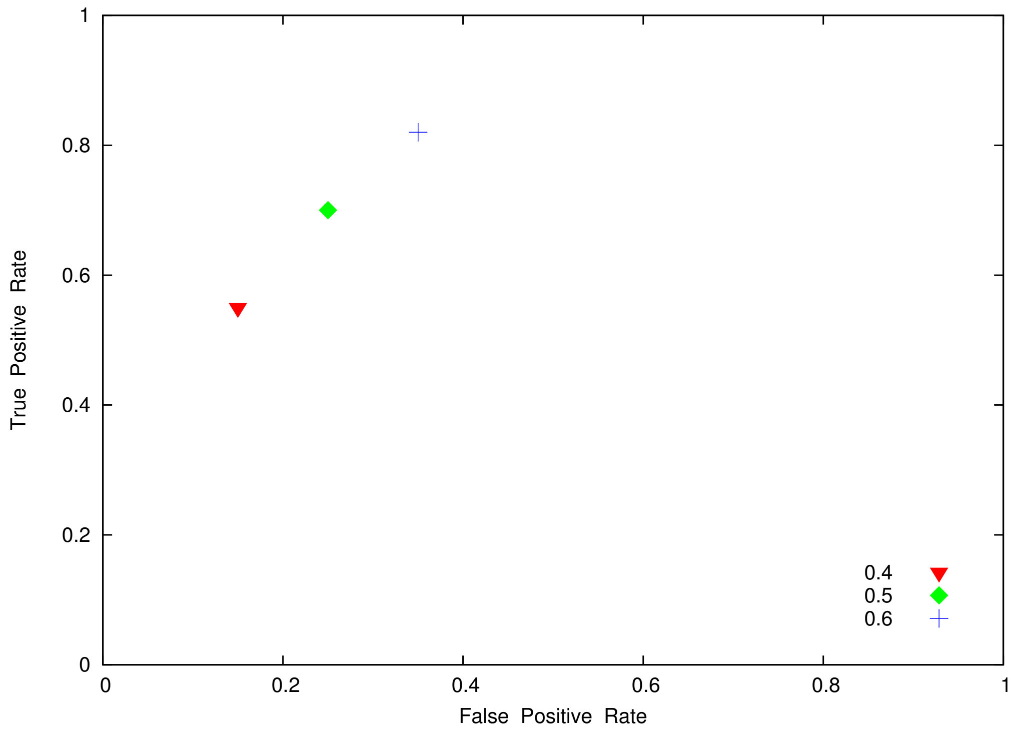


Figure 3.10: A ROC graph containing three points. The true positive rate, also known as sensitivity or recall, is on the y-axis. The false positive rate, also known as type 1 error rate or $(1 - specificity)$, is on the x-axis.

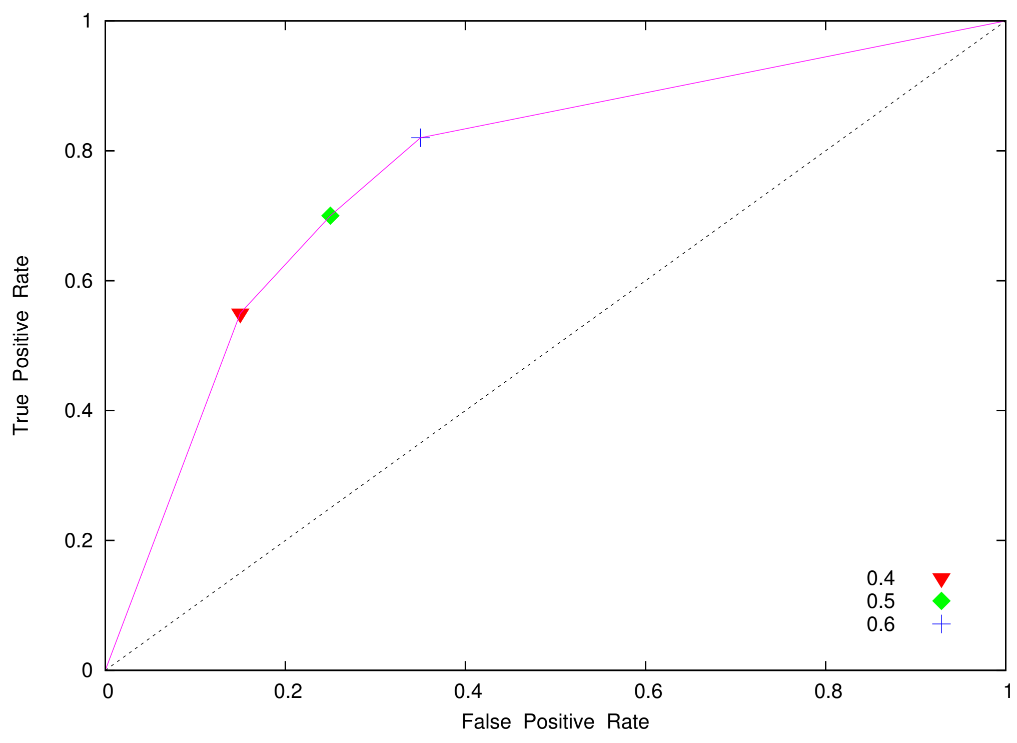


Figure 3.11: A ROC curve. The three points from Figure 3.10 are shown for reference. Each pixel comprising the solid line corresponds to the classifier's performance with a unique decision threshold. The performance for every possible decision threshold is shown on this line.

In addition to sensitivity and specificity, other commonly used performance measures, which come from the domain of information retrieval, are *recall* and *precision*. Recall is the same as sensitivity (or true positive rate), and as previously stated, describes the proportion of test vectors originally labelled as belonging to the positive class that were correctly classified. Precision is dissimilar to any of the measures discussed so far, and describes the proportion of positive predictions made that were correct. Precision is calculated as $TP/(TP + FP)$. Often reported along with these two measures is the f-measure, which is most commonly defined as the harmonic mean of the two. An f-measure of 1 would therefore describe perfect classification, where recall and precision are both equal to 1. As the motivation behind the use of these performance measures comprises a major part of this dissertation, discussion on this subject will commence again in Chapter 6. For reference to the confusion matrix derived statistics described so far (and others described later), see Table 3.2.

3.4 The Class Imbalance Problem

Class-imbalanced data, as well as introducing a requirement for more complex performance measures, also poses an extra challenge for classifiers. When there is only a sparse proportion of labelled minority class samples, many classifiers tend to over-predict the majority class, essentially ignoring the minority class [Chawla 2004]. This is known as *the class imbalance problem*, and occurs for several reasons. One of these reasons is that learners are typically designed to maximise the predictive accuracy (or CCR), which, as previously explained, favours the majority class. Therefore, when classifiers have only a small proportion of minority class samples to learn from, for example, when there is only one positive instance in every hundred training instances, the most reasonable option may be to make only negative predictions. This is because it is often the safest method of trying to ensure a low error rate on the test set.

Methods for alleviating the class imbalance problem include:

- **Modifying algorithm success criteria:** Configuring learners so that a more suitable performance metric, such as f-measure, is used instead of accuracy.

- **Modifying algorithm parameters:** Tweaking parameters to, for example, increase the cost of a minority class misclassification. More details on parameter tuning will given in Section 3.5.

- **Re-sampling the training data:** Prior to learning, the training data can be re-sampled to try and obtain a more balanced distribution. This typically involves removing majority class samples (undersampling) and/or adding minority class samples (oversampling). Discussion on this will commence again in Chapter 5.

Alias / Aliases	**Defined As**
Test Set No. Instances	$TP + TN + FP + FN$
No. Instances in Positive Class	$TP + FN$
No. Instances in Negative Class	$TN + FP$
Accuracy (+) Correct Classification Rate 1 − Error Rate	$\frac{(TP + TN)}{(TP + TN + FP + FN)}$
Error Rate (−) Incorrect Classification Rate 1 − Accuracy	$\frac{(FP + FN)}{(TP + TN + FP + FN)}$
True Positive Rate (+) Recall Sensitivity Probability of Detection (pd) 1 − False Negative Rate	$\frac{TP}{(TP + FN)}$
True Negative Rate (+) Specificity 1 − False Positive Rate	$\frac{TN}{(TN + FP)}$
False Positive Rate (−) Type 1 Error Rate Probability of False Alarm (pf) 1 − True Negative Rate	$\frac{FP}{(FP + TN)}$
False Negative Rate (−) Type 2 Error Rate 1 − True Positive Rate	$\frac{FN}{(FN + TP)}$
Precision (+)	$\frac{TP}{(TP + FP)}$
F-Measure (+) F-Score	$\frac{(2 * Recall * Precision)}{(Recall + Precision)}$
Balance (+) Distance from ROC optimal point	$1 - \frac{\sqrt{(0 - pf)^2 + (1 - pd)^2}}{\sqrt{2}}$

Table 3.2: A subset of statistics that can be derived from a confusion matrix.
Measures marked with ‘(+)’ have an optimal value of 1.
Measures marked with ‘(−)’ have an optimal value of 0.

3.5 Model Optimisation

Model optimisation is the process of tuning model parameters to try and improve predictive performance. As previously stated, we desire classifiers that generalise on labelled training data to successfully predict new, unlabelled data. However, in order for classifiers to generalise successfully, both *underfitting* and *overfitting* must be avoided. Underfitting occurs when consistently poor performance is obtained because the built model is not complex enough, or is too loosely fit to the underlying structure of the data. At the other extreme, overfitting occurs when good performance is obtained on the training data, but poor performance on unseen test data. This is because the built model is too complex, too rigidly fit to the training data. Overfitting is generally regarded as the more problematic case, as obtaining good performance on only the training data can be misleading for the inexperienced practitioner. Most classification algorithms have parameters that can affect the fit of the model (and in turn the predictive performance), examples include:

- The number of decision trees comprising a random forest.
- The soft-margin cost parameter used in many linear separators.
- The number of neighbours (value of k) in the k-nearest-neighbour algorithm.
- The number of hidden layers and nodes comprising an artificial neural network.

Additionally, note that many classifiers have parameters that do not affect the fit of the model, but can nonetheless affect classifier performance. These are parameters that are often (conceptually or otherwise) specified after the model construction process. An example is the decision (or discrimination) threshold for a naïve Bayes classifier.

Model optimisation is typically a systematic process where parameter settings are varied. The purpose of this is to determine which parameters (of the ones tested) maximise predictive performance on data that is representative of the test set. The actual test set must not be used in any part of this process however, to avoid inaccurate (and often highly optimistic) final estimates of performance. The data used in this process, instead of coming from the test set, comes from the training set. The assumption is that when building the final model, the best parameter set found during model optimisation will also be best for the real, unseen test set. Recall that both sets should be independent and representative samples from the specific problem domain.

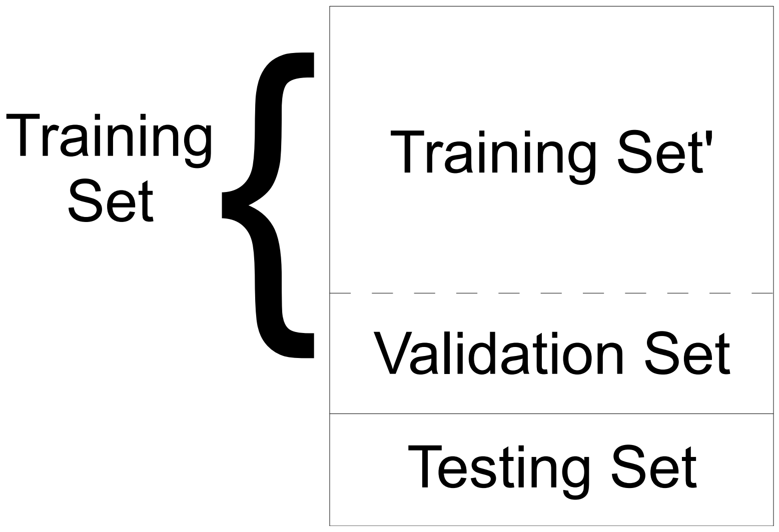


Figure 3.12: In a simple holdout experiment there is a training set and a testing set. During model optimisation, the training set is divided into a training subset and a validation set. This is to allow classification experiments to be carried out for each parameter set to evaluate. Because the testing set must not be used in any part of the optimisation process, the unseen data for the model optimisation experiments comes from the validation set.

To evaluate the suitability of a given set of parameters, a classification experiment must be carried out. This involves the data, which (as stated) comes from the training set, being re-sampled (see Figure 3.12). This can be achieved via any of the methods described in Section 3.3.1, for example by a simple holdout or by stratified cross-validation. The purpose of this is to have one or more unseen data sets to use during model optimisation. Thus, just as with the training and testing data divide, there must be a similar divide during model optimisation. The test data used during model optimisation comes from the *validation set* (see Figure 3.12), so-called to distinguish it from the final test set. The training data used during model optimisation (referred to here as the training subset) is the original training set minus the validation set. As with the training and testing set, it should be ensured that the training subset and validation set share no common instances.

After re-sampling the training data, a classification experiment is carried out for each set of parameters to try. The performance for each parameter set is recorded, and the best performing set is declared as optimal. Often it will be the best performing parameter set on average that is declared as optimal, for instance if the model optimisation process involves cross-validation (see [Forman 2010] for important details on how this average should be calculated). The pseudo-code for a typical classification experiment, involving repeated cross-validation and model optimisation (also with cross-validation), is shown in Figure 3.13.

```
M = 25          # no. of cross-validation repetitions
N = 10          # no. of cross-validation folds

DATA_SETS = ( x, y, z )                # not mathematical sets

PARAMS_TO_TRY = ( p1, p2, p3 )         # during optimisation

results = ( )                          # an empty list

for data_set in DATA_SETS:
    repeat M times:
        data_set.randomise_instance_order(get_random_number_seed())
        folds = create_stratified_cv_folds(data_set, N)
        for fold in folds:
            testing_set = fold
            training_set = data_set - testing_set
            assert testing_set.union(training_set) == {}
            params = optimise_params(training_set, N)
            model = train(training_set, params)
            results += predict(testing_set, model)

report_final_results(results)

--- --- ---

def optimise_params(opt_training_set, no_folds):

    opt_results = ( )         # an empty list

    folds = create_stratified_cv_folds(opt_training_set, no_folds)

    for param_to_try in PARAMS_TO_TRY:
        for fold in folds:
            validation_set = fold
            training_set_prime = opt_training_set - validation_set
            assert validation_set.union(training_set_prime) == {}
            model = train(training_set_prime, param_to_try)
            opt_results += predict(validation_set, model)

    return best_average_params(opt_results)
```

Figure 3.13: Pseudo-code for a typical classification experiment involving repeated, stratified cross-validation and model optimisation.

3.6 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 [Schölkopf 2001]. Their high-level of sophistication made them the classification method of choice in this dissertation. SVMs are *maximum-margin linear separators;* they construct a separating hyperplane between two classes subject to zero or more slack variables (see Section 3.2.2). The hyperplane is typically constructed deterministically, such that the distance between the classes is maximised (see Figure 3.14). This differs from the linear separators discussed in Section 3.2.2, which are non-deterministic and often produce sub-optimal separators. Ensuring a maximum-margin separator is intended to lower the generalisation error during testing. Note that SVMs can be used to classify data with any number of classes via recursive application.

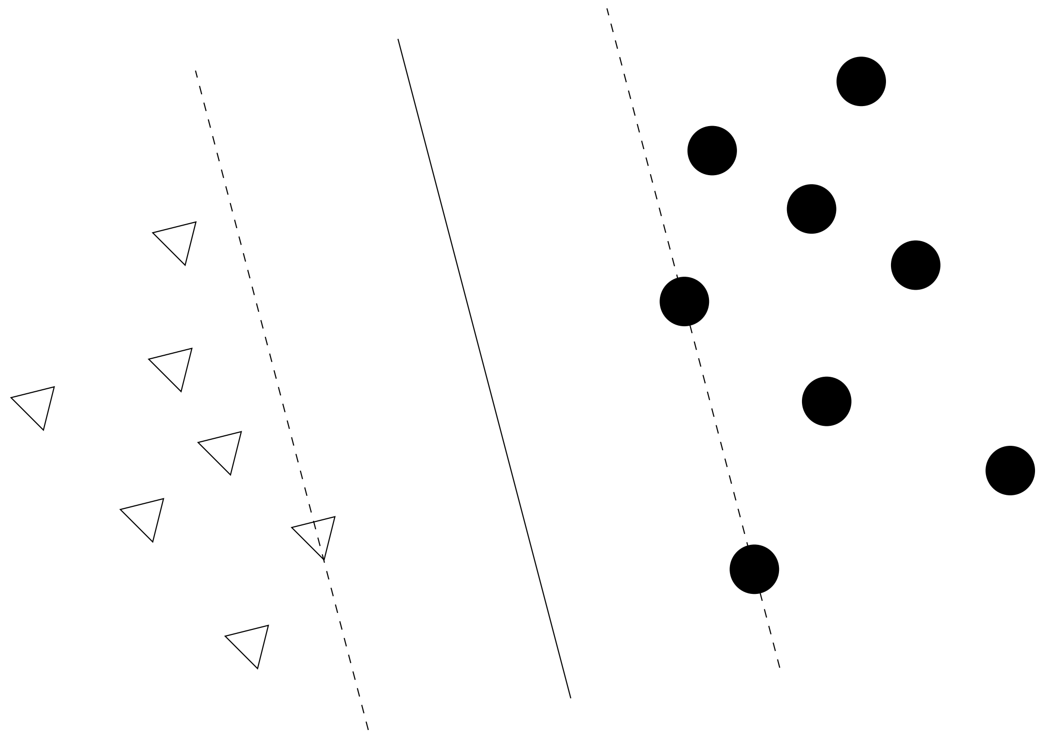


Figure 3.14: A binary classification toy problem solved by an SVM. The solid line is the separator. The distance between the dashed lines is the margin. The three data points situated upon the dashed lines are known as *support vectors.* It is these data points only that determine the separator, all others are not recorded in the model. Typically only a small proportion of data points will end up as support vectors.

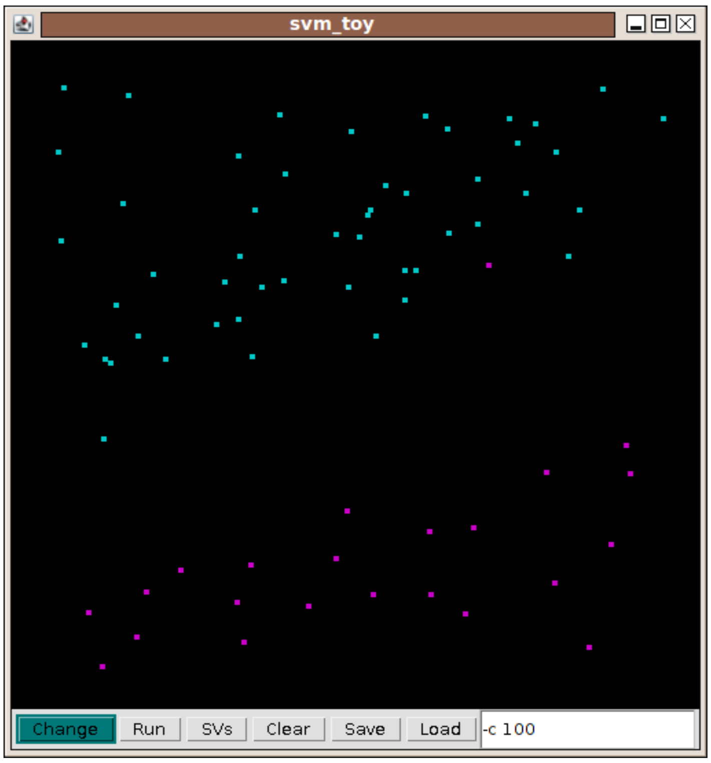
Although originally only suitable for linear classification problems [Vapnik 1963], SVMs can now also be used successfully for non-linear classification [Boser 1992]. This is achieved by use of a kernel function. A kernel function is used to implicitly map the data points into a higher-dimensional feature 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. Additionally, the explicit mapping to the higher-dimensional space is never needed.

There are several different kinds of kernel function (any continuous symmetric positive semi-definite function will suffice) including: polynomial, Gaussian and sigmoidal. Each has varying characteristics and is suitable for different problem domains. The one used throughout this dissertation is the *Gaussian radial basis function* (RBF), as it requires fewer hyperparameters than the remaining aforementioned kernels [Hsu 2003]. In fact, this kernel implicitly maps the data into an infinite-dimensional feature space whereby any consistent, finite data set will be linearly separable.

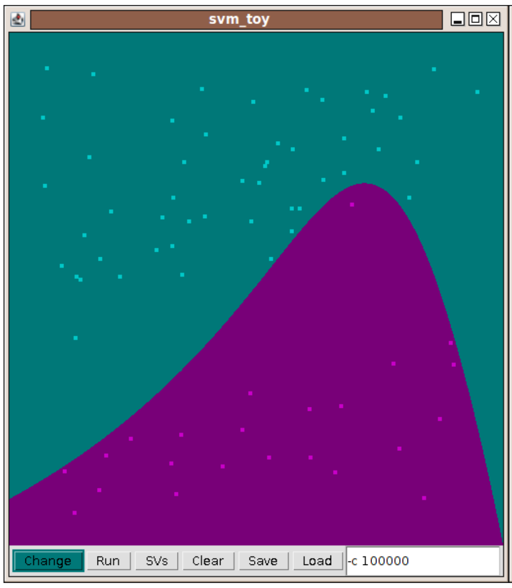
Further enhancements were made to the SVM algorithm to enable a *soft margin* [Cortes 1995], where misclassification errors are allowed in the model (see Figure 3.15 and refer back to Section 3.2.2 for a more detailed explanation). This greatly improved the generalisation ability of SVMs. It also introduced a hyperparameter, known as *cost* or *c,* to determine the trade-off between minimising the training error and maximising the margin (see Figure 3.15).

When SVMs are used with a Gaussian RBF kernel there are two user-specified hyperparameters, c and γ (gamma). As already mentioned, c is the error cost hyperparameter, and determines the trade-off between minimising the training error and maximising the margin. While c is an SVM hyperparameter, γ is a kernel hyperparameter, and controls the width (or radius) of the Gaussian RBF. The performance of SVMs is largely dependent on these hyperparameters, and the optimal values, the pair of values which yield best performance while avoiding both underfitting and overfitting, should ideally be estimated *for each training set.* The most common method for doing this is via a systematic optimisation search (see Section 3.5), known as a *grid-search.*

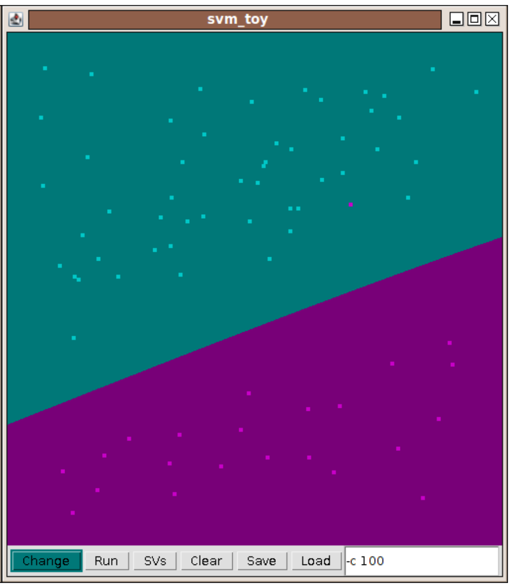
SVMs have been reported to be highly effective methods in many classification domains, including: text categorisation [Joachims 1998], DNA binding site prediction [Bradford 2005, Rezwan 2011] and face recognition [Osuna 1997, Shenoy 2011]. However, such methods do have drawbacks. The biggest potential drawback of SVMs is that their classification models are black-box, making it very difficult to work out precisely why a classifier makes the predictions it does. This is different to white-box classification algorithms such as decision trees, where the classification model is easy to interpret. Additional difficulties are that optimal hyperparameter values differ greatly depending on the data, and that the grid-search process (an exhaustive search) is computationally demanding. A related problem is that SVMs often perform poorly when appropriate hyperparameter tuning has not been carried out [Soares 2004]. This can result in inexperienced researchers obtaining poor performance.



(a) A two-dimensional plot of a data set. Note the outlier belonging to the purple class that is situated amongst the blue class.



(b) High cost value - potential overfitting.



(c) Low cost value.

Figure 3.15: In Figure 3.15a, a non-linearly separable data set is shown. Figure 3.15b shows an SVM's resulting separator with a high cost value. Figure 3.15c shows an SVM's resulting separator with a low cost value.

Chapter 4

Literature Review

Contents

This chapter is concerned with the defect prediction literature that is most relevant to this dissertation. In the sections that follow, a limited set of the most influential studies are discussed in detail. After this, the main findings from a systematic literature review (SLR) on the performance of software defect predictors are presented. I was directly involved in this SLR, which was a substantial, collaborative piece of work between Brunel University and the University of Hertfordshire.

The influential studies described in the following sections are: [Menzies 2007b], as this is a very well-cited defect prediction study; [Lessmann 2008], as this involved perhaps the largest-scale defect prediction experiment to date; [Elish 2008], as this is focused on defect prediction with SVMs; and [Liebchen 2008], as this highlights data quality awareness as an issue in modern, empirical software engineering research.

4.1 Menzies et al. 2007

The study carried out by Menzies et al. in 2007 [Menzies 2007b] is perhaps the most well-cited piece of modern defect prediction literature. It involved many defect prediction experiments, with various learners and publicly available data sets. The source of these data sets was the *NASA Metrics Data Program Repository*[1].

[1]Previously available at `http://mdp.ivv.nasa.gov/` and currently available in a more basic, less documented form at `http://filesanywhere.com/fs/v.aspx?v=896a648c5e5e6f799b`.

4.1.1 The NASA Metrics Data Program Repository

The NASA Metrics Data Program (MDP) Repository currently contains 13 module-level data sets[2] explicitly intended for software metrics research. Each data set represents a NASA software system/subsystem and contains the static code metrics and fault data for each comprising module. The static code metrics recorded include:

- **LOC-count measures** such as the number of lines of code and comments.
- **Halstead measures** such as unique operand and operator counts.
- **McCabe measures** such as the cyclomatic complexity.

All such non-fault related software 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 primary fault data in these data sets takes the form of an *error-count* metric. This metric was reportedly calculated from the number of error reports issued for each module via a bug-tracking system. From the details given at the original NASA MDP Repository, it is unclear precisely how these error reports were mapped back to the individual modules; however, it was stated that: “If a module is changed due to an error report (as opposed to a change request), then it receives a one up count. It cannot receive more than a one up for a given error report.” It was also stated that the *error-count* metric describes “the number of changes due to error.”

Because the *error-count* metric is discrete, it is often binarised by those using these data sets in classification experiments. Note that if this is not the case, there will typically need to be a class for each unique *error-count* value. This will potentially result in tiny quantities of data representing some of the classes, which may be problematic for learners (because of the class imbalance problem, see Section 3.4). The most common binarisation process used with the NASA MDP data sets (NASA data sets hereafter) was defined in [Menzies 2007b] as in Equation 4.1:

$$defective? = (error\_count \geq 1). \tag{4.1}$$

This process, despite its drawbacks (see Section 2.2.1), was carried out in each of the NASA-based studies described in this chapter. A thorough analysis of the NASA data sets is provided in Section 6.1, where many data quality issues are highlighted.

[2]Note that there is evidence to suggest the existence of a 14th data set, known as *KC2.* To the best of my knowledge this data set has never been openly hosted at the NASA MDP Repository.

4.1.2 Experiments & Findings

There were three classifiers used in [Menzies 2007b]: OneR, C4.5 (J48) and naïve Bayes. OneR (one rule) is a *decision stump,* a classification method similar to a decision tree but which can only split on a single attribute. The C4.5 algorithm is a well-known decision tree. An explanation of both this and the naïve Bayes method can be found in Section 3.2. The implementation used for each of these algorithms came from the popular data mining tool: Weka[3]. In addition to these learning techniques, this study also utilised a feature selection method. Such methods aim to reduce the number of features in a data set by locating and removing those that contain irrelevant information. The method used in this study: *InfoGain,* is concerned with information content in bits.

With 10 repetitions of a 10-fold cross-validation experiment using 8 of the NASA data sets, the individual performance of all three learners were compared. Naïve Bayes was identified as the best method, with statistically better performance than the other classifiers. With regard to the 'best' features chosen by InfoGain, the claimed results were that using only the 2 or 3 'best' features gave equal performance to that of when all 38 features were used. Furthermore, the frequencies of the 'best' features varied heavily between data sets. This led to the following two conclusions: Firstly, that there is no single set of 'best' metrics. Secondly, that defect prediction experiments should be carried out using all available metrics, and that these should then be passed on to a feature selection technique such as InfoGain.

Classifier performance in this study was measured using the two metrics used in *receiver operating characteristic (ROC) analysis,* the true positive rate and the false positive rate (see Section 3.3.3). In the journal these measures were referred to as the *probability of detection (pd)* and the *probability of false alarm (pf)*, respectively. The performance obtained by the naïve Bayes classifiers, a mean *pd* of 0.71 and a mean *pf* of 0.25, was described as being "demonstrably useful". This claim was questioned later the same year by Zhang and Zhang however, who demonstrated the choice of performance metrics to be inappropriate in the context of the class-imbalanced data [Zhang 2007]. As this subject comprises a major part of this dissertation, discussion on it will commence again in Section 6.2.

As well as the choice of performance measures used in [Menzies 2007b], aspects of the experimental design have also been called into question. A recent study by Song et al. involved a meticulous examination of the feature selection process [Song 2011]. It was found that the feature selection process used in [Menzies 2007b] violated the assumption of unseen data (see Chapter 3). This is because feature selection was carried out for each *whole data set* (not just the training set) using a method (InfoGain) which utilises class label data. Therefore, data which would not exist in the real world (the test set labels) was used in the model construction process, invaliding the experiment.

[3]`http://www.cs.waikato.ac.nz/ml/weka/`

4.2 Lessmann et al. 2008

Approximately a year and a half after [Menzies 2007b], Lessmann et al. undertook a large-scale classifier benchmarking study using 10 of the NASA data sets [Lessmann 2008]. The purpose of this study was to document and apply “a framework for comparative software defect prediction experiments”. This study was motivated by prior studies yielding inconsistent results regarding the superiority of various classifiers. In this study, 22 classifiers were tested in either the same or very similar contexts, and statistical tests were carried out. The classifiers used included:

- **Statistical classifiers:** such as naïve Bayes and logistic regression.
- **Nearest-neighbour methods:** k-nearest-neighbour and k-star.
- **Neural network approaches:** multi-layer perceptrons and radial basis function networks.
- **SVM-based classifiers:** such as least squares SVM, Lagrangian SVM and linear programming.
- **Decision tree approaches:** such as C4.5 and CART.
- **Meta-learning schemes:** logistic model tree and random forest.

For each data set and classifier, an experiment was carried out where two-thirds of the data was used for training, and the remaining one-third was used for testing. For each classifier with tunable parameters, optimal parameter values were estimated via a 10-fold cross-validation optimisation search on the training set. Performance on the final test set was assessed using the area under the ROC curve (AUC-ROC) metric. This metric was introduced in Section 3.3.3, and its suitability in this domain will be discussed in detail in Section 6.2.

The statistical approach used in this study began with a Friedman test. This particular test was chosen as it is a non-parametric alternative to ANOVA that relies on less restrictive assumptions [Demšar 2006]. The rank-based Friedman test was used to test (and in this case reject) the null hypothesis that all algorithms perform alike. Following on from this, a post-hoc pairwise Nemenyi test was carried out for all pairs of classifiers, to test the null hypothesis that their respective mean ranks are equal. The results showed that the best performing 17 classifiers (out of the original 22) had statistically indistinguishable performance ($\alpha = 0.05$).

The main conclusions from this study, which corresponded with the main conclusions from [Menzies 2007b], included that the overall level of predictive performance obtained demonstrated the worth of defect prediction systems. Additionally, the statistical results suggested “that the importance of the classification model may have been overestimated in the previous research”. The recommendation was therefore made that when selecting classification techniques, factors other than predictive performance should be highly influential. These factors include: comprehensibility, computational efficiency, and ease of use.

4.3 Elish & Elish 2008

In 2008, Elish and Elish carried out what is perhaps the currently most well-cited defect prediction study that is both based on the NASA data sets and focused on support vector machines (SVMs) [Elish 2008]. The purpose of this study was to compare the performance of SVMs with 8 other classification techniques. These techniques are similar to the ones listed previously that were used in [Lessmann 2008].

Correlation-based feature selection was carried out in this study, to reduce the number of features in each of the 4 data sets used. However, similar to the previously described problem in [Menzies 2007b] (originally reported by [Song 2011]), this was carried out for each whole data set, violating the assumption of unseen data (see Chapter 3). Additionally, unlike the experimental design reported in [Lessmann 2008] where model optimisation was carried out, a fixed value was used for each SVM-related hyperparameter. SVMs are known to be very sensitive to parameter settings, with unsuitable parameters often resulting in substandard performance [Soares 2004].

Despite these issues, the level of predictive performance reported for each classifier was substantially higher than for any classifier in any other study based on these data sets. Analysing the reported results from 100 repetitions of a 10-fold cross-validation experiment, the poorest performance was that of the Bayesian belief network for data set *KC1*. The reported *precision* was 0.92, *recall* was 0.78, and resultant *f-measure* was 0.85. In this domain this means that 78% of faulty modules were identified, and that of those modules predicted as defective, 92% turned out to be defective. To contrast these results, [Ma 2006] report best performance for data set *KC1* using a K-Star classifier with a precision of 0.53, recall of 0.51, and f-measure of 0.52.

Further analysis of the reported data statistics and performance results illuminate the reason for these ‘exceptional’ results. The first two columns of Table 4.1 show, for each data set used, the reported class distribution C (the percentage of modules belonging to the defective/minority class). The third column is calculated as $100 - C$, and is the accuracy (or correct classification rate) percentage that would be obtained by a classifier making only majority class predictions. The final column shows the reported SVM mean accuracy percentage for each data set. Observe that the values in this column and column three are very similar, with the largest difference being just 0.42. This indicates that the SVMs may be over-predicting the majority class, which is a common problem when using them with imbalanced data sets [Sun 2006]. Although this is an interesting observation, it does not explain the high figures obtained for recall and precision. The authors define these measures according to the standard definitions given in Equation 4.2:

$$recall = \frac{TP}{TP + FN} \qquad precision = \frac{TP}{TP + FP} \tag{4.2}$$

Data Set Alias	% of Defective Modules	% Accuracy if Majority Class Predicted	% SVM Mean Accuracy
CM1	9.7	90.3	90.69 ± 1.074
PC1	6.9	93.1	93.10 ± 0.968
KC1	15.4	84.6	84.59 ± 0.714
KC3	6.3	93.7	93.28 ± 1.099

Table 4.1: Data statistics and performance results from Elish & Elish 2008. The third column is the only one not taken directly from the journal. It shows the accuracy that would be obtained by making only majority class predictions, and is calculated as *100 − percentage of defective modules* (shown in the second column).

However, following on from the observation regarding accuracy rates, further analysis of the SVM results suggested that performance was in fact being reported for the majority class, using the definitions shown in Equation 4.3. Note that in this equation, *recall_majority* could also be referred to as the *true negative rate*.

$$recall\_majority = \frac{TN}{TN + FP} \qquad precision\_majority = \frac{TN}{TN + FN} \tag{4.3}$$

The evidence that these were the equations being used in the study comes from the minimum reported SVM mean 'recall' percentage being 99.4 ± 0.006. Additionally, the reported SVM mean 'precision' results are very similar to the accuracy results, with the largest difference being just 0.43%. Both these findings are what would be expected if the classifiers were almost exclusively predicting the majority class, *and* the measures shown in Equation 4.3 were being used. These findings, when taken alongside the observation regarding accuracy rates, provide conclusive evidence that the classifiers were almost exclusively making majority class predictions (also see [Bowes 2012]). Therefore, the reported results in this study are dangerously misleading, and great care should be taken when interpreting them.

4.4 Liebchen & Shepperd 2008

A study concerned solely with data quality in software engineering was carried out by Liebchen and Shepperd in 2008 [Liebchen 2008]. This study involved an SLR of empirical software engineering studies. The search criteria for this SLR specified that studies must contain both phrases: "data quality" and "software". Of the "many hundreds" of studies returned by this search, just 23 satisfied the inclusion criteria by explicitly addressing data quality. These alarming findings suggest that the software engineering community may not have been taking data quality seriously. An interesting point raised in this study is that the community "may be wasting research effort on data sets that contain such levels of noise as to prevent meaningful conclusions."

4.5 An SLR on Fault Prediction Performance

In August 2011 the IEEE Transactions on Software Engineering accepted [Hall 2012] for publication. I was a co-author of this journal, along with 3 other co-authors and the lead author. My primary role was to review many studies; this included validating their claimed results, and potentially extracting data from them that would later be used for synthesis. The main purpose of this study was to answer the following three research questions:

- **How does context affect fault prediction performance?**
- **Which independent variables should be used for fault prediction?**
- **Which data mining techniques perform best for fault prediction?**

This study involved an SLR of fault prediction studies published between January 2000 and December 2010. The *ACM Digital Library, IEEExplore* and *ISI Web of Science* search engines were used, with the search term given in Figure 4.1. In addition to identifying papers using these search engines, a manual, paper-by-paper search was carried out for 5 relevant journals and 13 relevant conferences. The works of the 5 most prolific defect prediction authors (according to DBLP[4]) were also manually searched, as were the references from all identified studies. Therefore, this was a large-scale and thorough search.

Figure 4.1: The search term used with each search engine.

```
(fault* OR bug* OR defect* OR errors OR corrections
OR corrective OR fix*) in title only
AND (software) anywhere in study
```

This process led to there being approximately 2000 studies identified, of which 208 satisfied our inclusion criteria by claiming to be empirical fault prediction studies. These included studies were subjected to a four stage, contextual quality check, to ensure that they were:

- **A prediction study:** It must have been either directly or indirectly claimed that learners were being tested against unseen data.
- **Reporting sufficient application context information:** In order to confidently interpret a fault prediction study, basic contextual information must be given. This includes the application(s): domain, maturity and programming language(s).

[4]http://www.informatik.uni-trier.de/~ley/db/

- **Reporting sufficient data mining information:** This included the learning technique(s) and variables used being clearly reported.

- **Reporting sufficient data acquisition information:** How the dependent and independent variable(s) data was collected must have been either: reported directly, referenced accordingly, or be available in the public domain.

Surprisingly, 172 (83%) of the 208 included studies failed this basic quality check. By far the most common cause of failure was studies not reporting sufficient application context information. Of the 114 such studies, 58 were based on NASA data. All exclusively NASA-based studies failed the quality check, as there is no maturity information available for these data sets in the public domain. It was therefore decided that these studies should not be brought forward for synthesis. The reasoning for this was because of our first research question being explicitly focused on the effect of context in fault prediction.

The synthesis for this SLR was therefore carried out on the 36 studies which passed the quality check. Within these 36 studies, 23 were classification-based and the remaining 13 were regression-based. Of the 23 classification studies, 19 were based on binary classification and used standard, confusion matrix derived performance measures. It is these 19 studies that comprise the main synthesis, as the other studies were not as directly comparable. Therefore, we extracted the quantitative performance data from these 19 studies, which resulted in 206 unique performance reports for various learners using a variety of data sets. The performance data extracted from these studies was, where necessary, converted to common measures. The measures we chose were: recall, precision and f-measure (see Section 3.3.3). This gave us common performance indicators with which to reason about the performance of studies and help us answer our research questions.

4.5.1 Findings

With regard to our research question concerning the effect of application context on fault prediction, we found some evidence that performance generally improves as applications grow. This was reported by studies using Eclipse[5] data, and a possible and intuitive explanation is that learners perform better when there is more training data available (as applications grow, there is more historical data to utilise). Note, however, that we had limited findings regarding context because there seems to be a gap in the current body of knowledge. Studies scarcely investigate context directly, and often fail to report even basic context details. We therefore feel that it is important for studies to report such details, which will allow for a more thorough meta-analysis in future.

[5]http://www.eclipse.org/

When analysing which independent variables were being used in studies, we frequently found claims that using LOC-count-based measures alone provided equal and if not greater performance than using complexity or object-oriented (OO) metrics. This finding corresponds with the findings in [Oram 2010], where the complexity metrics of thousands of open-source C projects were statistically analysed, and all found to be highly correlated with LOC-count. This led to the conclusion that "lines of code should be considered as the first and only metric for (defect prediction) models". Despite this, there were studies in the SLR claiming that product metrics other than LOC-count, such as various OO metrics (see [Chidamber 1994]), were useful. It is worth noting that care must be taken when comparing the results of studies using exclusively OO-based or exclusively complexity-based metrics, as they are often predicting faults at different levels of granularity (see Chapter 2). The studies in the SLR reporting best performance initially used all available metrics, but then used a feature selection technique to use only a subset of the features estimated as most useful for classification.

With regard to which learning methods are best for defect prediction, our findings are in agreement with the findings reported in [Lessmann 2008] that there is no universal 'best' method. However, an interesting observation is that simple methods, such as naïve Bayes and logistic regression, seem to be performing very competitively. Conversely, complex, state-of-the-art methods, such as SVMs, seem to be performing fairly poorly. This may be because of a lack of data mining expertise in this domain.

4.6 Summary

To summarise the key findings described in this chapter: [Menzies 2007b] claimed their naïve Bayes predictors were "demonstrably useful", with a mean *TPR* (or *pd*) of 0.71 and a mean *FPR* (or *pf*) of 0.25. These claims were swiftly questioned by [Zhang 2007] however, who demonstrated that the choice of performance measures was inappropriate in the context of the data. The experimental design in [Menzies 2007b] was also recently questioned by [Song 2011], who highlighted the flawed feature selection process.

Perhaps the largest-scale defect prediction experiment to date was carried out by [Lessmann 2008], who found the best 17 of 22 classifiers to have statistically indistinguishable performance. This led to the conclusion that choice of classification technique may not be as important as previously thought. The same year, [Elish 2008] carried out a defect prediction study intended to demonstrate the capability of SVMs for defect prediction. This study contained several major technical shortcomings. The technical shortcomings in both this study and [Menzies 2007b] indicate that defect prediction is an immature research area, which may be lacking the necessary skills. In addition to technical problems, [Liebchen 2008] have highlighted data quality awareness as an issue in modern, empirical software engineering research.

The most up-to-date SLR on defect prediction was carried out by [Hall 2012]. The focus of this SLR was predictor performance. A contextual quality check of the 208 identified studies resulted in 172 (83%) being rejected for further synthesis. The most common cause for rejection was that basic application context information had not been reported. This was problematic as investigating the effect of such context was a key research question. The main findings from this study include that:

- Simple learners tend to perform competitively, while more sophisticated techniques tend to perform poorly.
- Using LOC-count-based measures alone often yields predictors that are competitive with those built using additional metrics.
- Best performance is obtained by studies using feature selection.

Chapter 5

SVMs for Defect Prediction

Contents

In this chapter I describe two experiments where support vector machines (SVMs) were used for defect prediction. These experiments, despite their shortcomings (which were discovered after their completion and will be discussed in full), were the beginning of a learning process regarding methodological issues in fault prediction. The experience gained from these studies was substantial, and a key influence on the methodology proposed in Chapter 8.

The first experiment, which involved using SVMs to predict defects in the NASA data sets (see Section 4.1.1), led to publication in the 2009 International Conference on Engineering Applications of Neural Networks (EANN) [Gray 2009]. It was during this study that I first identified data quality issues. Details are given in Section 5.1, and the full paper can be found in Appendix A. The second experiment, concerned with classification model analysis and described in Section 5.2, led to publication in the 2010 International Joint Conference on Neural Networks (IJCNN) [Gray 2010]. The full paper can be found in Appendix B. Note that for both these experiments the SVM software used was LIBSVM [Chang 2001], an excellent, open-source library for support vector classification.

5.1 Initial Classification Experiment

To gain experience at machine learning, a classification experiment was undertaken utilising 11 of the 13 NASA data sets (described in chapters 4 & 6). These data sets were chosen as they have been heavily used in defect prediction research. The classifiers used in this study were SVMs (see Section 3.6); these were chosen as, at that time, SVMs had not been widely used for defect prediction. Additionally, SVMs are highly sophisticated classifiers, and (with suitable parameters) often work as well as, or if not better than many other learning techniques [Segaran 2007].

Analysis of the data sets prior to classification led to the discovery of many data quality issues. The most severe of these issues was that after basic pre-processing (described in Section 6.1.2.1), many of the data sets contained large proportions of non-unique, or repeated data points. Such data points can be conceptualised as repeated rows in a spreadsheet. The major potential problem caused by repeated data points is that after the data set divide into training set and testing set, it becomes possible for both sets to contain common data points. Recall that the term ‘set’ in all terms: ‘data set’, ‘training set’ and ‘testing set’ does not correspond to the mathematical notion of the term; therefore, it is possible for these collections (perhaps most suitably thought of as mathematical lists) to contain duplicate elements. Thus, when data set S contains duplicate elements and is divided (using one of the methods described in Section 3.3.1) into training set R and testing set E, it is possible for the intersection of R and E to not equal the empty set ($R \cap E \neq \emptyset$). A standard classification experiment based on R and E will therefore be compromised, as the test data contains seen data points: data points that were used in the construction of the model. The assumption of unseen data will therefore have been violated in such an experiment (more clarification on this is provided in chapters 3 & 6).

There are many more data quality issues specific to the context of machine learning with the NASA data sets: these are covered in great detail in Section 6.1. For this experiment significant quantities of suspicious/problematic data was removed from each data set during initial pre-processing. This included the removal of all data points with non-unique feature vectors, such that each unique, consistent data point was only represented once (there is more on this in Chapter 6).

After binarisation of the error-count data (see chapters 4 & 6), the NASA data sets exhibit various levels of class imbalance: there are typically many more data points labelled as ‘non-defective’ than there are that are labelled as ‘defective’. Precise details will be given in Chapter 6 during a full analysis of the data sets; however, to alleviate the class imbalance problem in this experiment, I used an undersampling technique (see Section 3.4). This simple technique involves pseudo-randomly discarding majority class data points until there is an equal number of data points representing each class. In this experiment, undersampling was carried out during initial pre-processing, prior to any separation into training and testing sets.

After pre-processing, the experiment involved 10 repetitions of 5-fold cross-validation for each data set. A further 5-fold cross-validation was carried out for each training set, for the purpose of model optimisation (see Section 3.5). This model optimisation process yielded a cost and gamma value-pair estimated as optimal for each (withheld) test set, which was subsequently used to train each corresponding final model.

Because undersampling was carried out on each whole data set during initial pre-processing, all data sets had a precisely balanced class distribution (an equal number of instances representing each class). This led to *accuracy* being the selected performance measure used to assess predictive performance (see Section 3.3.2). The results from this initial experiment are not presented here, for reasons described in the following section.

5.1.1 Issues & Shortcomings

The major shortcoming of this initial experiment was the methodology of the undersampling. Recall that undersampling was carried out during initial pre-processing, prior to any separation into training and testing sets. This meant that, indirectly, the data that would end up comprising the test sets was being modified. As test sets should be entirely unseen, this kind of pre-processing is unreasonable, and violates the assumption of unseen data (see Chapter 3). Sampling methods should typically be used on only the training data after the divide into training and test set: they should not be used on a whole data set prior to data separation. Note that this issue is similar to that described in Section 4.1.2, where a feature selection technique (which utilises class label data) was applied to whole data sets as opposed to only the training data.

A second shortcoming of this experiment is a consequence of that just described. Because undersampling was carried out on each whole data set until there was a precisely balanced class distribution, and because stratified cross-validation was used (see Section 3.3.1), this meant that each test set had a balanced class distribution. This is what led to *accuracy* being the selected performance measure; however, note the broken assumption of equal misclassification costs (see Section 3.3.2). A similar experiment utilising more suitable and sophisticated performance measures is described in Section 6.3.

Similar to the undersampling methodological issue described where majority class data points were removed during initial pre-processing, repeated data points were also removed during initial pre-processing. Again, this meant that data which would end up comprising test sets was being indirectly modified when it should have been unknown. The issue of how to deal with repeated data points is a thorny one, and is discussed in detail in chapters 6 & 8.

The results of this initial experiment are not presented here, because of the aforementioned methodological flaws which render them invalid. The interested reader can find them within the full paper in Appendix A. Section 6.3 contains details and results of an improved version of the experiment described here; these results provide more meaningful estimates of predictive performance than those obtained in this study.

5.2 Examining Predictive Models

Although there are many published defect prediction studies, few of them include an analysis of the predictive model(s) used. Such an analysis can illuminate *what* has been learnt by the classifiers, and in turn highlight the reasons *why* they make the predictions they do. Ideally, this would lead to new knowledge in the specific problem domain. Predictive model analysis is relatively simple for white-box learners where classifier internals are readily comprehensible (see Chapter 3); however, for black-box learners (such as SVMs) more complicated analysis methods are required [Núñez 2002]. One such method is described in detail in this section.

The aim of this study was to analyse the predictive models of SVMs. There were 13 models built in total, one for each of the NASA data sets. Analysis was made by observing, for each data point comprising the training data, whether or not it was placed on the 'correct' side of the separator, and its distance from the separator. Training data points placed on the 'wrong' side of the separator comprise the training error. As described in Section 3.6, SVMs allow a variable amount of training error in order to increase the size of the margin. The amount of training error to allow is determined by the *cost* (or c) parameter, and optionally any kernel parameters (in this case: *gamma*). A sufficiently high error cost will typically result in no training error (as the separator will be rigidly fit to the data); therefore, to make the experiment worthwhile, each model was optimised for best performance on unseen data. This model optimisation process was similar to the one described in the previous section and in Section 3.5. Note that to differentiate between the 'correct' and 'incorrect' placement of positive and negative data points (with respect to an SVM's implicit decision boundary), the following standard terms were used: *true positive*, *true negative*, *false positive* and *false negative* (these were described in Section 3.3.3).

In addition to observing whether or not each training data point was placed on the 'correct' side of the separator, it was also interesting to analyse the distance between each data point and the separator. This is possible using *decision values.* A decision value[1] is a scaled version of the distance between a data point and the separator (in the feature space), such that all *free* support vectors have an absolute decision value of 1 (see Figure 5.1). More specifically, a decision value is a signed real number; a positive value indicates a data point on one side of the separator, while a negative value indicates a data point on the other side of the separator. A value nearing zero indicates a data point that is very close to the separator, and can therefore be conceptualised as a difficult data point to classify (see Figure 5.1). The benefit of using decision values as opposed to raw distance values is that because of the way they are scaled, they are more comparable across data sets.

All 13 of the NASA data sets were used in this study; however, one of the data sets (namely: *PC2*) was subjected to a more thorough analysis than the others. This was because it had the fewest instances (post pre-processing), and was therefore the least labour intensive to manually analyse. The data pre-processing carried out in this study was very similar to that described in Section 5.1. More details regarding data pre-processing will be given in Chapter 6. Note that undersampling was carried out during initial pre-processing, as previously described.

[1]Note that the definition of decision value used in this dissertation differs from the formal definition (which can be found in [Bottou 2007]). The most notable difference with this version is that using it alone, all *bounded* support vectors (see Figure 5.1) may not be identifiable. This was not a problem in this study as the decision values used were primarily only for quantifying the distance between each data point and the separator.

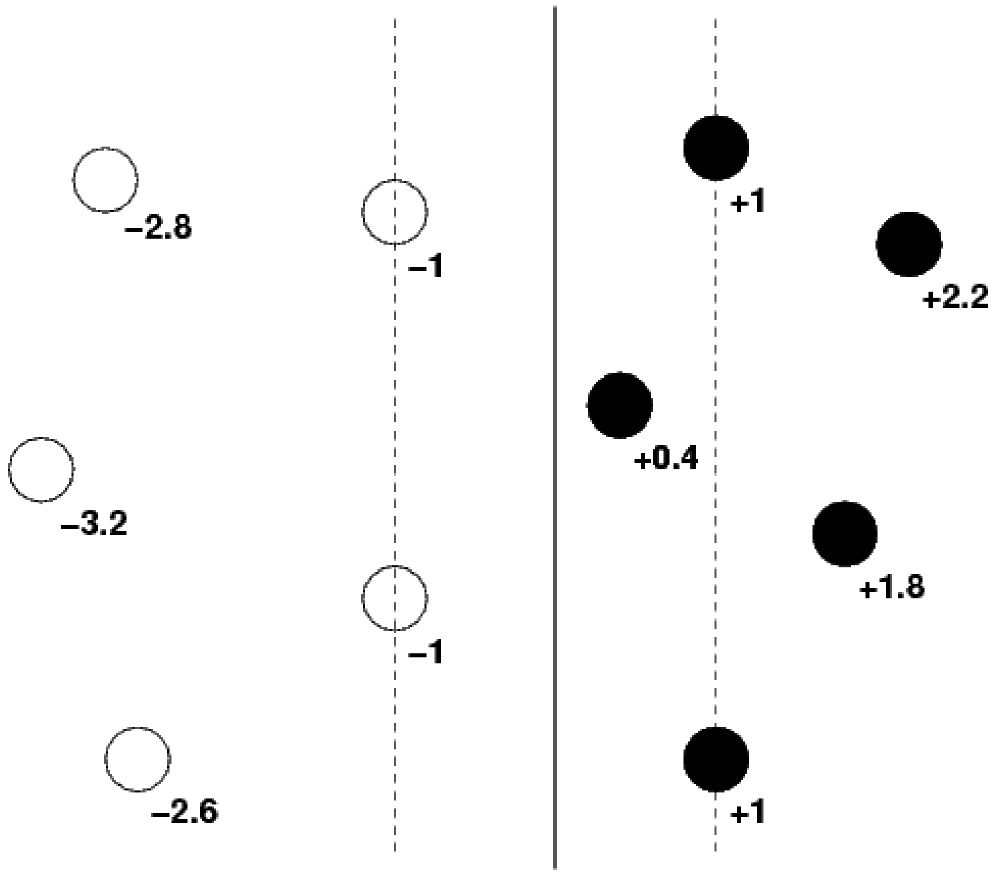


Figure 5.1: An SVM's feature space where each data point is labelled with its corresponding decision value. The final separator is shown by the solid line. The data points situated on the boundaries of the margin (shown by the two dashed lines) are those data points whose corresponding absolute decision value is equal to one. These data points are known as *free* support vectors. Data points with an absolute decision value less than one are known as *bounded* support vectors (although there may be other bounded support vectors with an absolute decision value greater than one). A decision value nearing zero indicates a data point located close to the separator, which can be conceptualised as a data point that is difficult to classify. Note that in this figure, solid and hollow dots are used only to differentiate between data points located on either side of the separator; they do not differentiate between whether or not the data points are located on the 'correct' side of separator. In fact, the data point nearest the separator (with a decision value of $+0.4$) belongs to the opposing class (those data points situated on the left of the separator), it is a low cost value that has resulted in its placement with the 'wrong' class. This is because a low cost value indicates that training error is tolerable for the sake of increasing the size of the margin.

Because data set PC2 would be the primary focus of analysis, an initial examination of this data set (post pre-processing) was undertaken. This examination involved *principal components analysis (PCA),* a dimensionality reduction technique which can be useful for data visualisation [Pearson 1901]. PCA involves an orthogonal transformation, where features are converted into a set of linearly uncorrelated features, known as *principal components.* The first principal component is the feature with the largest variance, the second principal component is the feature with the second largest variance, and so on. The number of principal components (features in the transformed data) will be less than or equal to the original number of features. Note, however, that it is possible for a significant proportion of the total variance within a data set to be contained within the first few principal components: this makes PCA a valuable data visualisation tool. For data set PC2, there was 65% of the total variance contained within the first two principal components; these are shown in Figure 5.2. This figure shows that in general, the data is highly intermingled, and as such this appears to be a very difficult classification task. There are several outliers shown, the most notable of which are the two 'defective' data points located in the bottom right corner. These were the two largest modules in terms of LOC total, and also had the two highest no. unique operators, no. unique operands, and cyclomatic complexity feature values (amongst others). Note that these outliers will be revisited later in this section during predictive model analysis.

After pre-processing, PCA, and hyperparameter tuning (see the previous section and Section 3.5), SVM models were built for each of the 13 data sets. Spreadsheets were then constructed for each data set, containing the original data, as well as the following additional columns:

- *Training Classification Result:* Whether or not each data point was placed on the 'correct' side of the separator. This field will contain one of the following values: *true positive (TP)*, *true negative (TN)*, *false positive (FP)* or *false negative (FN).*

- *Decision Value:* As previously described and shown in Figure 5.1, this is the measure used to quantify the distance between a data point and the separator. Positive values belong to data points classified as defective; negative values belong to data points classified as non-defective.

For each spreadsheet, rows were ranked by decision value. The mean decision value was also computed for each of the TP, TN, FP and FN groups. For data set PC2, a thorough manual examination of each row was carried out. For all other data sets, the decision value averages were examined to see if any patterns emerged.

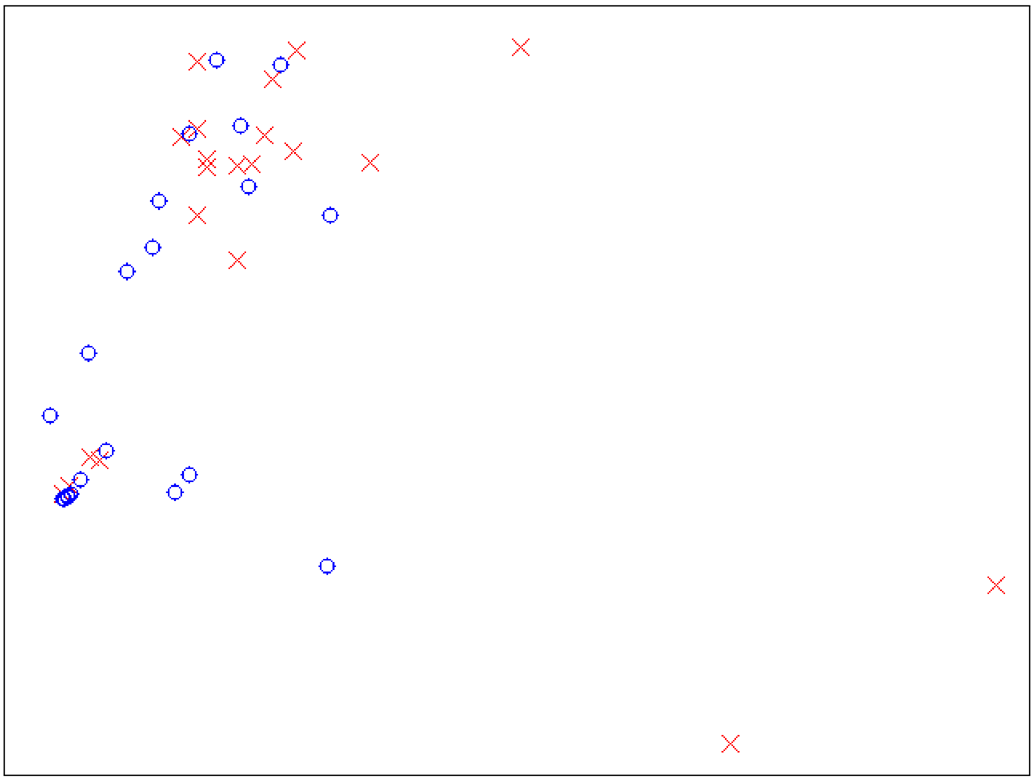


Figure 5.2: The two main principal components from data set PC2. Crosses represent modules labelled as defective while circles represent modules labelled as non-defective. Observe the highly intermingled data, and the two extreme outliers belonging to the defective class in the bottom right corner.

The training classification result and corresponding decision value for each of the 42 data points comprising data set PC2 are shown in Figure 5.3. Analysis of these values showed that the mean decision value of the TPs (0.86) was higher than the mean decision value of the FPs (0.60). This means that the TPs were generally further away from the separator than the FPs. The mean decision value of the TNs (-0.81) was similar to, but slightly greater than that of the FNs (-0.88), meaning that the FNs were generally slightly further from the separator than the TNs. Examining the averages for the remaining 12 data sets showed that the SVMs again had the TPs placed generally further away from the separator than the FPs, by an average of 49%. Unlike for data set PC2 however, the SVMs for the remaining data sets had the TNs placed generally further away from the separator than the FNs, by an average of 51%. An interesting avenue of further research would be to analyse the decision values obtained when classifying unseen test sets. This is because the TPs and TNs in such an experiment may also be located further from the separator, on average, than the FPs and FNs, respectively. Therefore, these findings would suggest that when using SVMs for defect prediction, the subsequent code inspections should be prioritised in descending order of decision value.

Table 5.1 contains a subset of the metrics which comprise the modules labelled in Figure 5.3, as well as their corresponding classification result and decision value. Note that there were 36 metrics in total but only four are shown here due to space limitations. Module 1 (as labelled in Figure 5.3) is the data point farthest from the separator. The defective-class classification made does appear fairly well-motivated however (from the limited information available from the metrics), as the module exhibits defect-prone characteristics. The most notable of these are the 76 unique operands and 15 linearly independent paths in just 68 lines of C code. Modules 2 and 3 are the outliers identified during PCA. These modules were classified with above average decision values (for the TPs), which is reassuring as they appear to be very large and may benefit from functional decomposition. It may be surprising that these two modules did not have the two largest decision values, however, this shows that the SVM in this case was not being dominated entirely by LOC-count-based measures. This suggests that there may be worth in the other metrics.

At first sight when looking at the FPs, it may appear as though the classification for module 4 is unfounded, as none of the metrics shown suggest a module that is particularly defect-prone. This looks suspicious as the instance is located far from the separator. 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 [Dijkstra 1968], and the SVM classified 89% of modules with an essential complexity greater than 1 (the metrics minimum) as defective. Module 5 is the instance that is located closest to the separator, it was classified as defective but only by a very small margin. This classification appears more immediately understandable than the classification for module 4, as although the module contains only 7 LOC it contains 16 unique operands.

All of the modules classified as non-defective (modules 6 to 9) appear very small and simple from what can be deduced from the metrics. This highlights the difficulty of this classification domain, and calls into question the suitability of using such metrics to predict defects. None of the four modules had defect-prone characteristics detectable by the metrics available, yet two of them were defective. The problem is that the metrics used can provide only a limited insight into software defect-proneness, not actual defectiveness. A defect-prone module is one that exhibits characteristics which make it likely to be defective, such as being extremely large and/or complex. It is a fair assumption that the majority of defective modules within a software system will exhibit defect-prone characteristics, although these are difficult to define precisely and programming-language specific. The findings in this study suggest a limiting factor on the performance achievable by defect predictors. This limiting factor appears to be in the correlation between defect-proneness and defectiveness. The stronger this correlation, the higher the potential predictive performance. Note that the strength of this correlation is likely to fluctuate substantially between data sets.

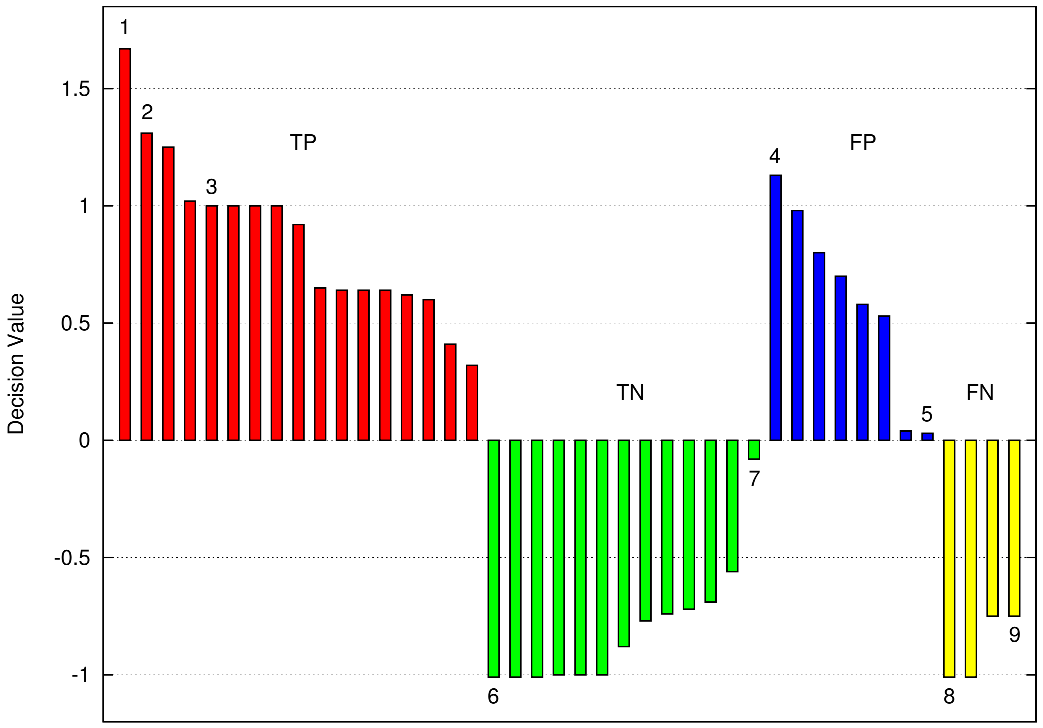


Figure 5.3: The decision value and classification result for each module in *PC2*. Note that the modules are grouped horizontally according to classification result.

Module ID No.	LOC Total	v(g)	No. Uni. Operands	No. Uni. Operators	Classification 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 5.1: A subset of the metrics for each of the modules labelled in Figure 5.3. Note that v(g) is McCabe's cyclomatic complexity [McCabe 1976].

The findings from examining all 42 of the modules comprising data set PC2 indicated that in general, the classifications made appeared consistent with current software engineering beliefs. These beliefs include that modules with metrics indicating larger, more complex code are more likely to be defective (as they have defect-prone characteristics). Moreover, it appeared that the training error in the model occurred mainly because of the data points labelled as defective which did not exhibit defect-prone characteristics (and the converse). Therefore, it appears that the model may have been doing far better at classifying defect-proneness than it was at classifying actual defectiveness. Because it is possible for a module without defect-prone characteristics to contain a defect (a programmer typing a '==' instead of a '!=' in a small single-line module for example), the biggest limiting factor on the performance of defect prediction systems may be in the strength of the correlation between defect-proneness and defectiveness. Interestingly, although it is also possible for a module with defect-prone characteristics to not contain a defect, the fact that the module exhibits such characteristics may be a sign that it is nonetheless a suitable target for refactoring [Fowler 1999]. Therefore, in this domain, it may be that FNs are generally more serious misclassifications than FPs.

5.2.1 Issues & Shortcomings

Similar to the initial classification experiment described at the beginning of this chapter, the major shortcoming of this experiment was the methodology of the undersampling. Recall that undersampling was carried out during initial pre-processing, typically resulting in large proportions of non-defective labelled data points being removed from each data set. Although there was no unseen testing phase in this experiment, and therefore no assumption of unseen data to break, there was still the bias caused by the undersampling as to which non-defective labelled data points would remain. This issue was worsened for the originally highly imbalanced data sets, where very large proportions of non-defective labelled data points were removed. Removing such quantities of data points threatens the validity of this experiment, as those chosen to remain may not comprise a representative sample. Additionally, even if the undersampled subset is representative of the original sample (for the non-defective, majority class), the new data set as a whole may still be unrepresentative in terms of the original class distribution.

Another issue caused by the premature undersampling was that, for the highly imbalanced data sets, so much data was removed that the SVMs were being trained on very few instances. For data set PC2, there were just 42 instances remaining after pre-processing, 21 in each class. Having such a small sample size introduces another source of bias. Therefore, the findings from this study cannot be extrapolated to the original (unmodified) versions of the NASA data sets, or any other software fault data set.

5.3 Conclusions

The issues highlighted in these studies and in the work of others (see Chapter 4) motivated most of the experiments described in this remainder of this dissertation. In fact, these issues culminated in a new proposed methodology for software defect prediction. The details of this methodology are presented in Chapter 8.

Chapter 6

Major Methodological Issues

Contents

Perhaps the most substantial finding of those described in the previous chapter is that of the data quality issues with the NASA data sets. This is because the NASA data sets have been the basis of much prior research; therefore the validity of this research is potentially threatened. As mentioned in Chapter 4, there is a current debate regarding the suitability of various classifier performance measures used in this domain. Effective quantification of predictive performance is clearly a key factor in determining the worth of a prediction system; therefore, I believe the use of inappropriate (and often highly misleading) performance measures to be a major methodological pitfall.

In this chapter I begin by describing the data quality issues with the NASA data sets. I do this while presenting a novel data cleansing process to address these issues. The data cleansing process has been incrementally developed over the course of my PhD study, and was published as described here in the 2011 International Conference on Evaluation and Assessment in Software Engineering (EASE) [Gray 2011b]. The full paper can be found in Appendix C. Following on from this in Section 6.2, I go into detail on the current debate regarding the suitability of various classifier performance measures. This involves an explanation of why the use of the performance measure *precision* is so important when using imbalanced data. This study also led to publication in EASE 2011 [Gray 2011a]; the full paper can be found in Appendix D. Lastly, I describe a repeated classification experiment of the one described in Section 5.1, where all of the issues described in both this chapter and the previous chapter are addressed.

6.1 Data Quality Issues

In order to carry out a software defect prediction experiment there is a requirement for reasonable quality data. However, software fault data is very difficult to obtain. Commercial software development companies often do not have a fault measurement program in place. Moreover, even if such a program is in place, it is typically undesirable from a business perspective to publicise fault data. This is particularity true for systems where quality has been a serious problem, that is, where it would be most useful to publicise such data and give researchers an opportunity to discover why this was the case.

Open-source systems are frequently used by researchers to construct their own fault data sets [Kim 2008, Schröter 2006, Shivaji 2009, Śliwerski 2005, Kim 2006]. Such systems are often developed whilst using a bug-tracking system to record the faults encountered by developers. If bug information has been correctly and consistently entered into version control commit messages, it is then possible to autonomously locate the fault-fixing revisions. From here it is possible to (fairly accurately) map fault-fixing revisions back to where the fault was first introduced (the *bug-introducing change*, see [Śliwerski 2005, Kim 2006, Williams 2008]). The major problem with constructing fault data from open-source systems is that it can be a very time consuming task to do accurately. This is because human intervention is often required to check the validity of the automated mappings. For systems of even moderate size, the quantity of these mappings can make this task infeasible.

Thus difficulty in obtaining software fault data is the major factor why public-domain fault data repositories, such as those hosted by NASA and PROMISE, have become so popular amongst researchers. These repositories host numerous data sets, which require no data analysis and little or no pre-processing, before machine learning tools such as Weka will classify them. The ease of this process can be dangerous to the inexperienced researcher. Results can be obtained without any scrutiny of the data. Furthermore, researchers may naïvely assume the NASA

Metrics Data Program (MDP) data sets are of reasonable quality for data mining. This issue is worsened by the hosting sites not indicating the main problems, and by so many researchers using these data sets inappropriately. The aim of this study is therefore to illuminate the data quality issues present in these data sets, and the problems that can arise when they are used (as they often are) in a binary classification context. It is hoped that this study will encourage researchers to take data quality seriously, and to question the results of some studies based on these data sets.

An introduction to the NASA MDP Repository was given in Section 4.1.1. Recall that there are currently 13 module-level data sets stored in the repository, each one explicitly intended for software metrics research. The originating source code for these data sets is entirely closed-source, making the validation of data integrity more difficult. A substantial amount of research based wholly or partially on these data sets has been published over the last decade, including: [Song 2011, Menzies 2004b, Menzies 2004a, Menzies 2007b, Menzies 2008, Menzies 2010, Zhang 2009, Zhang 2010, Guo 2003, Guo 2004, Jiang 2007, Jiang 2008c, Jiang 2008b, Jiang 2008a, Jiang 2009, Khoshgoftaar 2004, Zhong 2004, Seliya 2005, Liu 2010, Lessmann 2008, Turhan 2007, Turhan 2009b, Turhan 2009a, Boetticher 2006, Mende 2009, Mende 2010, Koru 2005b, Koru 2005a, Tosun 2009, Bezerra 2007, Singh 2008, Challagulla 2005, Challagulla 2006, Pelayo 2007, Kutlubay 2007, Ma 2006, Oral 2007, Rodriguez 2007, Vandecruys 2008, Mertik 2006, Cong 2010, de Carvalho 2010, Tao 2010, Li 2007, Vivanco 2010, Elish 2008, Lu 2012]. Note that in this study I focus solely on the data sets just described, because they are widely used; I do not make use of the available object-oriented metrics data set, or any other data set affiliated with NASA.

The most common usage for the NASA MDP data sets (as reported in the literature) is in binary classification experiments. Typically a classifier is trained on binary-labelled data, and then each new set of module metrics is predicted as belonging to either a 'faulty' module, or a 'non-faulty' module. This is clearly a huge simplification of the real world, for two main reasons. Firstly, fault quantity is disregarded: there is typically no distinction between a module with 1 reported fault and a module with 31 reported faults, they are both simply labelled as 'faulty'. Secondly, fault severity is disregarded: there is typically no distinction between a trivial fault and a life-threatening fault. Despite these crude simplifications, binary classification defect prediction studies continue to be very prolific.

It is widely accepted by the data mining community that in order to accurately assess the potential real-world performance of a classification model, the model must be tested against entirely different data from that upon which it was trained [Witten 2005]. This is why there is a distinction between a training set and a testing set (see Section 3.3.1). A testing set is also referred to as an independent test set, as it is intended to be independent from the training set (i.e. models should be tested against 'unseen data', see [Witten 2005]). This is very basic data mining knowledge, and is no surprise to the defect prediction community. In 2004 Menzies et al. state: "if the goal of learning is to generate models that have some useful future validity, then the learned theory should be tested on data not used to build it. Failing to do so can result in a excessive over-estimate of the learned model. . . ” [Menzies 2004b]. Despite this fact being well known, numerous studies based on the NASA MDP data sets (henceforth, NASA data sets) have potentially had high proportions of data points common to both their training and testing sets. This is because the NASA data sets contain varied quantities of repeated data points, observations of module metrics and their corresponding fault data occurring more than once. Thus, when this data is used in a classification context, the separation into training and testing sets may result in both sets containing large proportions of common data points. This can yield the aforementioned excessive estimate of performance, as classifiers can memorise rather than generalise. This is very serious, as when data mining “it is important that the test data was not used in any way to create the classifier.” [Witten 2005]

In this study I thoroughly analyse all 13 of the original NASA data sets. I am interested in data quality in terms of noise, inaccurate/incorrect data (see [Liebchen 2008]). Additionally, because these data sets are typically used in binary classification experiments, I am also interested in the issues specific to this context. Firstly, I highlight the data quality problems via my novel data cleansing process. This process is for removing noise, and for preparing the data sets for binary classification. Next, I discuss at length the potential problems caused by repeated data points when data mining, and why using lower-level metrics (such as character counts) in fault data sets may alleviate these problems, by helping to distinguish non-identical modules.

The rest of this study is presented as follows: In the next section I discuss related work, papers where problems with the NASA data sets have been documented or discussed. In Section 6.1.2 I document my novel data cleansing process in incremental stages. Section 6.1.3 contains a demonstration of the potential effects of repeated data points during classification experiments. My conclusions are presented in Section 6.1.4.

6.1.1 Related Studies

The major problem when using the NASA data sets in a classification context is that repeated data points may result in training data inadvertently being included in testing sets, potentially invalidating the experiment. This is not a new finding; however, I believe it needs spelling out to researchers, as previous studies mentioning this issue seem to have been ignored. In this section the most relevant studies surrounding this issue are discussed.

The earliest mention of repeated data points in NASA data sets that I can find was made in [Kaminsky 2004]. The authors state that they eliminated “redundant data”, but give no further explanation as to why. The data set used was NASA data set KC2, which is not available from the NASA MDP Repository. Although this data set is currently available from the PROMISE Repository[1], I did not use it in this study in an effort to use only the original, unmodified data.

In [Boetticher 2006] five NASA data sets were used in various classification experiments. The author states that “data pre-processing removes all duplicate tuples from each data set along with those tuples that have questionable values (e.g. LOC equal to 1.1).” Interestingly, it is only the PROMISE versions of the NASA data sets that contain these clearly erroneous non-integer LOC-count values. The author goes into detail on repeated data points, stating that “to avoid building artificial models, perhaps the best approach would be not to allow duplicates within datasets.” One of the experiments carried out was intended to show the effect of the repeated data points in the five NASA data sets used. This was in a 10-fold cross-validation classification experiment with a C4.5 decision tree. The claimed result was that the data sets with the repeats included achieved significantly better performance than those without. Although this result may be expected, there was an unfortunate technical shortcoming in the experimental design. When reporting the performance of classifiers on test sets with imbalanced class distributions, ‘accuracy’ (or its inverse: ‘error rate’) should not be used (see [Nickerson 2001] and Section 3.3.2). In addition to this, care is required when performing such an experiment, as the proportion of repeated data points in each class is not related to the class distribution. Therefore, post the removal of repeated data points, the data sets could have substantially different class distributions. This may boost or reduce classifier performance, because of the class imbalance problem (see [Chawla 2004, He 2009] and Section 3.4).

Classification experiments utilising probabilistic outputs were carried out in [Bezerra 2007]. Here the authors used 5 of the original NASA data sets and state that they removed both “redundant and inconsistent patterns”. Inconsistent data points are another of the problems when data mining with the NASA data sets. They occur when repeated feature vectors (module metrics) describe data points with differing class labels. Thus in this domain they occur where the same set of metrics is used to describe both a module labelled as ‘defective’ and a module labelled as ‘non-defective’. I believe the removal of such instances was first carried out in [Khoshgoftaar 2004].

The work described in the remainder of this section differs from that previously described, as it is not based on classification experiments. It is instead based on the analysis and cleansing of data. This study demonstrates: the poor quality of the NASA data sets; the extent to which repeated data points disseminate into training and testing sets; and the potential effect of testing sets containing *seen* data points during classification experiments.

[1]Formerly at `http://promisedata.org/` and now at `http://code.google.com/p/promisedata/`.

6.1.2 Method - Data Cleansing

The NASA data sets are available from the aforementioned NASA MDP and PROMISE repositories. For this study I used the original versions of the data sets from the NASA MDP Repository (see Section 4.1.1). Note, however, that the main issues also apply to the PROMISE versions of these data sets, which are for the most part simply the same data in a different format[2].

6.1.2.1 Initial Pre-Processing: Binarisation of Class Variable & Removal of Module Identifier and Extra Error-Data Attributes

In order to be suitable for binary classification, the error-count attribute is commonly reported in the literature (see [Menzies 2007b, Lessmann 2008, Elish 2008] for example) as being binarised as follows:

$$defective? = (error\_count \geq 1). \tag{6.1}$$

It is then necessary to remove the 'unique module identifier' attribute, as this gives no information toward the defectiveness of a module. Lastly, it is necessary to remove all other error-based attributes, to make the classification task worthwhile. This initial pre-processing is summarised in Figure 6.1. As the NASA data is often reportedly used post this initial pre-processing, an overview of each data set is given in Table 6.1. In this table the number of original recorded values is defined as the number of attributes (features) multiplied by the number of instances (data points). For simplicity missing values are given no special treatment. The number of recorded values metric is used to quantify how much data comprises a data set. These values will be revisited later on to help determine how much of the original data has been removed during data cleansing.

Figure 6.1: Initial pre-processing pseudo-code.

```
rmAttributes = [ MODULE, ERROR_DENSITY, ERROR_REPORT_IN_6_MON,
                   ERROR_REPORT_IN_1_YR, ERROR_REPORT_IN_2_YRS ]

for dataSet in dataSets:
    for rmAttribute in rmAttributes:
        if rmAttribute in dataSet:
            dataSet = dataSet - rmAttribute
    dataSet.binarise(ERROR_COUNT)
    dataSet.rename(ERROR_COUNT, DEFECTIVE)
```

[2]Note this is no longer the case; since the publication of [Gray 2011b] updated versions of the data sets have been uploaded at PROMISE addressing many of the issues pointed out in this work.

Name	Language	Features	Instances	Recorded Values	% Defective Instances
CM1	C	40	505	20200	10
JM1	C	21	10878	228438	19
KC1	C++	21	2107	44247	15
KC3	Java	40	458	18320	9
KC4	Perl	40	125	5000	49
MC1	C & C++	39	9466	369174	0.7
MC2	C	40	161	6440	32
MW1	C	40	403	16120	8
PC1	C	40	1107	44280	7
PC2	C	40	5589	223560	0.4
PC3	C	40	1563	62520	10
PC4	C	40	1458	58320	12
PC5	C++	39	17186	670254	3

Table 6.1: Details of the NASA data sets post initial pre-processing.

6.1.2.2 Stage 1: Removal of Constant Attributes

A numeric attribute which has a constant/fixed value throughout all instances is easily identifiable as it will have a variance of zero. Such attributes contain no information with which to discern modules apart, and are at best a waste of classifier resources. Each data set had from 0 to 10 percent of their total attributes removed during this stage, with the exception of data set KC4. This data set has 26 constant attributes out of a total of 40, thus 65 percent of available data contains no information with which to train a classifier.

This stage removes data that may be genuine, but in the context of machine learning it is of no use and is therefore discarded. Regarding data set KC4, it appears as though many of the metrics have not been collected; instead of leaving them out of the data set originally however, they were instead included with all values equal to zero.

An additional note regarding data set KC4 is that two of its attributes: 'essential complexity' and 'essential density', have two unique values each, but in each case, one of the values occurs just once. This data may be valid, but after the data divide into training and testing set, it may be that the training data contains a constant attribute. This can be problematic for some learning techniques, and is therefore something that researchers should be aware of.

6.1.2.3 Stage 2: Removal of Repeated Attributes

In addition to constant attributes, repeated attributes occur where two or more attributes have identical values for each instance. Such attributes are therefore fully correlated, which may effectively result in a single attribute being over-represented. Amongst the NASA data sets there are two repeated attributes (post stage 1), namely the 'number of lines' and 'loc total' attributes in data set KC4. The difference between these two metrics was poorly defined at the NASA MDP Repository. However, they may be identical for this data set as (according to the metrics) there are no modules with any lines either containing comments or which are empty. For this data cleansing stage I removed one of the attributes so that the values were only being represented once. I chose to keep the 'loc total' attribute label as this is common to all 13 NASA data sets.

This stage again removes data that may be genuine, because it can be problematic when data mining. It is interesting that data set KC4 has had so much data removed in these first two stages. Table 6.1 shows that KC4 is unique in that it is the only data set based on Perl code. Therefore, it may be that the metrics collection tool (McCabeIQ 7.1) was more limited in the metrics it could collect for this language.

6.1.2.4 Stage 3: Replacement of Missing Values

Missing values may or may not be problematic for learners depending on the classification method used. However, dealing with missing values within the NASA data sets is very simple. Seven of the data sets contain missing values, but all in the same single attribute: 'decision density'. This attribute is defined as 'condition count' divided by 'decision count', and for each missing value both these base attributes have a value of zero. It therefore appears as though missing values have occurred because of a division by zero error. In the remaining data set which contains all three of the aforementioned attributes but does not contain missing values, all instances with 'condition count' and 'decision count' values of zero also have a 'decision density' of zero. Because of this I replace all missing values with zero, ensuring consistency between data sets. Note that in [Bezerra 2007] all instances which contained missing values within the NASA data sets were discarded. It is more desirable to cleanse data than to remove it, as the quantity of possible information to learn from will thus be maximised.

This stage adds data via the replacement of missing values, because they are problematic for many learning techniques. Note, however, that some researchers may not wish to carry out this stage, if they are using a learning method that is resilient to missing values (such as naïve Bayes). Additionally, some researchers may wish to exclude derived features (such as 'decision density') altogether. There is more discussion on this in Section 6.1.2.7.

6.1.2.5 Stage 4: Enforce Integrity with Domain-Specific Expertise

The NASA data sets contain varied quantities of attributes derived from simple equations of other attributes, which are useful for checking data integrity. Additionally, it is possible to use domain-specific expertise to validate data integrity, by searching for theoretically impossible occurrences. The following is a non-exhaustive list of checks that can be carried out for each data point:

- Halstead's length metric (see [Halstead 1977]) is defined as: 'number of operators' + 'number of operands'.
- Each token that can increment a module's cyclomatic complexity (see [McCabe 1976]) is counted as an operator according to the original NASA MDP Repository. Therefore, the cyclomatic complexity of a module should not be greater than the number of operators + 1. Note that the minimum cyclomatic complexity is 1.
- The number of function calls within a module is recorded by the 'call pairs' metric. A function call operator is counted as an operator according to the original NASA MDP Repository, therefore the number of function calls should not exceed the number of operators.

These three simple rules are a good starting point for removing noise in the NASA data sets. Any data point which does not pass all of the checks contains noise. Because the original NASA software systems/subsystems from where the metrics are derived are not publicly available, it is impossible for us to investigate this issue of noise further. The most viable option is therefore to discard each offending instance. Note that a prerequisite of each check is that the data set must contain all of the relevant attributes (post stage 1). Six of the data sets had data removed during this stage, between 1 to 12 percent of their data points in total.

During this stage it is possible to not only remove noise (inaccurate/incorrect data), but also problematic data. A module which (reportedly) contains no lines of code and no operands and operators should be an empty module containing no code. Should such a data point be discarded? As it is impossible for us to check the validity of the metrics against the original code, this is a grey area. An empty module may still be a valid part of a system, it may just be a question of time before it is implemented. Furthermore, a module missing an implementation may still have been called by an unaware programmer (one who does not know of the missing implementation). As the module is unlikely to have carried out the task its name implies, it may also have been reported to be faulty. Despite this, researchers need to decide for themselves what to do with data that cannot be proved to be noisy, but is nonetheless strange. For example, the original data set MC1 (according to the metrics) contains 4841 modules (51% of modules in total) with no lines of code. I feel that it would therefore not be unreasonable to remove such data points, or even reject the entire data set altogether.

6.1.2.6 Stage 5: Removal of Repeated and Inconsistent Instances

The most severe issue when using the NASA data sets for classification experiments is that of repeated data points. Unfortunately, this issue is often ignored in the defect prediction literature. Repeated, redundant, or duplicate data points are data points (or instances) that appear more than once within a data set. They are either noise, most probably caused by a faulty data collection process, or, if they are genuine, they occur (in this domain) when many modules have the same values for all measured metrics; for example, when they have the same number of: lines of code, lines of comments, blank lines, operands, operators, unique operands, unique operators, function calls, and so on. Additionally, these modules have also been assigned the same class label referring to whether they are or are not ‘defective’. This situation is clearly possible in the real world; for example, in an object-oriented system, there may be many simple accessor and mutator methods that share identical metrics and have not been reported as faulty. However, such data points may be problematic in the context of machine learning, where is it imperative that classifiers are tested upon data points independent from those used during training [Witten 2005]. The issue is that when data sets containing repeated data points are split into training and testing sets (for example by an x% training, 100−x% testing split, or n-fold cross-validation), it is possible for there to be instances common to both sets. With test data included in the training data, the learning task is either simplified or reduced entirely to a task of recollection. Ultimately however, if the experiment is intended to show how well a classifier could generalise upon future, unseen data points, the results will be erroneous as the experiment is invalid. This is because the assumption of unseen data has been violated, due to the test data being contaminated with training data. Note that because of the closed-source nature of the NASA data sets, it is impossible to know whether the repeated data points are genuine or are noise.

Inconsistent (or conflicting) instances are another issue, and are very similar to repeated instances in that both occur when the same feature vectors describe multiple modules. The difference between repeated and inconsistent instances is that with the latter, the class labels differ, thus (in this domain) the same metrics would describe both a ‘defective’ and a ‘non-defective’ module. This is again possible in the real world, and while not as serious an issue as the repeated instances (in the case of the NASA data sets), inconsistent data points can be problematic during binary classification tasks. When building a classifier which outputs a predicted class set membership of either ‘defective’ or ‘non-defective’, it is illogical to train such a classifier with data instructing that the same set of features is resultant in both classes. I focus more on repeated data points than inconsistent ones in this study, as for most data sets the proportion of repeated instances is considerably larger. Note, however, that it is possible for a data point to be both repeated and inconsistent.

Adding all data points into a mathematical set is the simplest way of guaranteeing that each one is unique. This ensures that classifiers will be tested on unseen data, regardless of how the data is divided. From here it is possible to remove all inconsistent pairs of modules, to ensure that all feature vectors (data points irrespective of class label) are unique. The proportion of instances removed from each data set during this stage is shown in Figure 6.2. All data sets had instances removed during this stage, and in some cases the proportion removed was very large (90, 79 and 75 percent for data sets PC5, MC1 and PC2, respectively). Note that for most data sets the proportion of inconsistent instances removed was negligible. This is partly due to the methodology of removing all repeated instances first, and then inconsistent pairs second, as some of the inconsistent instances are also repeated ones. Looking at Figure 6.2, it appears highly unlikely that all of the repeated data points are genuine; for example, I find it extremely difficult to believe that more than 60% of modules within a large software system would share the same number of: lines of code, lines of comments, blank lines, operands, operators, unique operands, unique operators, function calls, and so on.

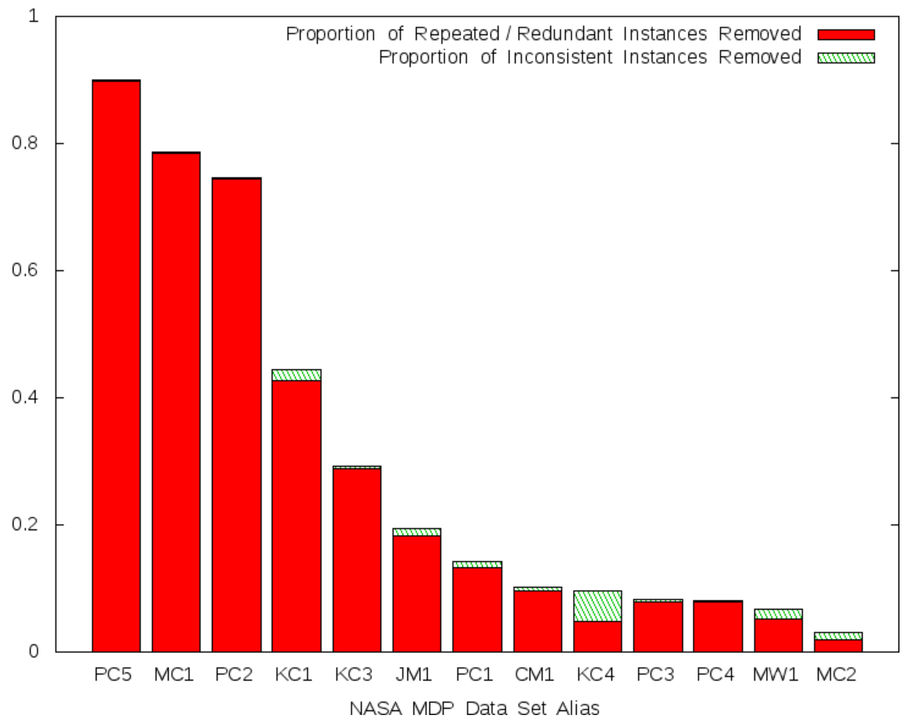


Figure 6.2: The proportion of instances removed during stage 5.

6.1.2.7 Other Issues

The most well-known issue regarding use of the NASA data sets in classification experiments is that of the varied levels of class imbalance (see Table 6.1). The table shows that data set KC4 has an almost balanced class distribution, whereas data set PC2 has only 0.4% of data points belonging to the minority class. This is an issue that researchers should be aware of. Learning from imbalanced data is an active area of research within the data mining community, I therefore refer readers to standard texts [Witten 2005, Chawla 2004, He 2009, Batista 2004]. Note, however, that defect prediction researchers need to be very careful in the way they assess the performance of their classifiers when using highly imbalanced data (see [Davis 2006, Zhang 2007, Gray 2011a] and Section 6.2).

Another issue is that, as mentioned previously, there are attributes within the NASA data sets that are simple equations of other attributes. While useful for checking data integrity, they can be problematic (or simply a waste of computational resources) depending on the learning technique used. For example, support vector machines utilising a Gaussian radial basis kernel will typically not benefit from the inclusion of such attributes, as they will be implicitly calculated. Additionally, other highly correlated attributes can be found within the data sets, which are known to harm classification performance with many learning techniques [Hall 1999, Howley 2006]. Therefore, in some contexts, researchers may wish to address these issues. This usually involves removing attributes during pre-processing and/or utilising a feature selection technique on the training data.

6.1.3 Findings

Figure 6.3 shows the proportion of recorded values removed from the 13 NASA data sets (after basic pre-processing, see Table 6.1) post the 5 stage data cleansing process just defined. Stages 1 and 2 of this process can remove attributes (features), stage 3 can replace values, and stages 4 and 5 can remove instances (data points). This was the motivation to use the number of recorded values ($attributes * instances$) metric, as it takes both attributes and instances into account. Figure 6.3 shows that between 6 to 90 percent of recorded values were removed from each data set in total.

Of the data cleansing processes with the potential to reduce the quantity of recorded values, it is the removal of repeated instances that is, by far, responsible for the largest average proportion of data removed (see Figure 6.2). This raises the following questions: *Is the complete removal of such instances really necessary? Why are there so many repeated data points and how can they be avoided in future? What proportion of seen data points could end up in testing sets when this data is used in classification experiments? What effect could having such quantities of seen data points in testing sets have on classifier performance?* Each of these questions are addressed in the sections that follow.

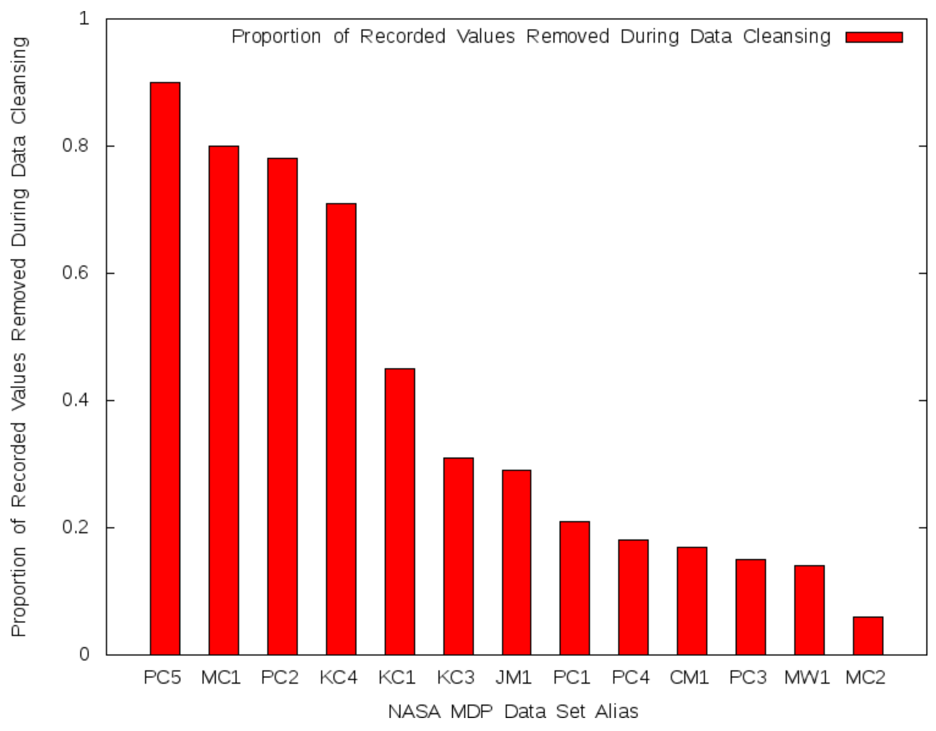


Figure 6.3: The proportion of recorded values removed during data cleansing.

6.1.3.1 Is the complete removal of repeated instances really necessary?

Removing all repeated instances during initial pre-processing is a simple way to prevent the problems they can cause. The most serious of these problems is test set contamination, the potential effects of which will be discussed in detail in Section 6.1.3.4. An additional issue with repeated data points, separate to the problem of test set contamination, may occur as a result of model construction (training and optimisation). Training a model on data containing small proportions of repeated data points is typically non-problematic. For example, a simple oversampling technique is to duplicate minority class data points in the training set(s). Using training data which contains repeated data points is reasonable, *as long as training and testing sets share no common instances*. Note, however, that the issue with excessive oversampling: overfitting, may also occur here. Overfitting can be identified when a model obtains good training performance, but poor performance on unseen test data [He 2009]. It is also possible to cause overfitting by optimising model parameters using a validation set (a withheld subset of the training set, see Section 3.5) containing duplicate data points. It is for this reason that [Kołcz 2003] recommend “tuning a trained classifier with a duplicate-free sample”. Note that this also applies when optimising using multiple validation sets, for example via n-fold cross-validation. It is also worth noting that feature selection techniques can be negatively affected by duplicates [Kołcz 2003].

Following the data cleansing process to remove the repeated instances, researchers will be able to use off-the-shelf data mining tools (such as Weka) to carry out experiments that yield meaningful results (or at least, far more meaningful results than had such instances been included blindly). However, the problems with this method (which were introduced in Chapter 5) are based around the fact that such instances can occur in the real world. Although in the case of these data sets it is impossible to tell whether or not these instances are genuine (I believe there to be mix of both genuine data and noise), the possibility that these instances are genuine cannot be ignored. This is especially true for those data sets with low rates of duplication.

If researchers believe that the repeated instances are noise, then the method of removing them during pre-processing, as described in the data cleansing process just defined, is a suitable approach. Conversely, if researchers believe that these instances are genuine, then this approach is not suitable. The main reason for this, as described in Chapter 5, is that by removing these instances during pre-processing, data that would eventually comprise the test set(s) is being removed. Thus, the assumption of unseen data is being broken. Therefore, to prevent the issues caused by repeated data points in cases where they are believed to be genuine, a different approach is required. One such approach was proposed by [Boetticher 2006], and is to add an extra attribute: *number of duplicates*. This will ensure that data points are unique, and help prevent potentially useful information being lost. Another, more sophisticated approach, is proposed in Section 8.3.1. This novel approach prevents the issues caused by repeated data points while keeping test sets unmodified, enabling more-meaningful estimates of potential real-world predictive performance to be obtained.

6.1.3.2 Why are there so many repeated data points and how can they be avoided in future?

As previously stated, the NASA data sets are based on closed-source, proprietary software, so it is impossible for us to validate whether the repeated data points are truly a representation of each software system/subsystem, or whether they are noise. Despite this, a probable factor in why the repeated (and inconsistent) data points are a part of these data sets is because of the poor differential capability of the metrics used. Intuitively, 40 metrics describing each software module seems like a large set. However, many of the metrics are simple equations of other metrics. Because of this, it may be highly beneficial in future to also record lower-level metrics, such as character counts. These will help to distinguish modules apart, particularly small modules, which statistically result in more repeated data points than large modules. Additionally, learners should be able to utilise such low-level data for helping to detect potentially troublesome modules (in terms of defect-proneness).

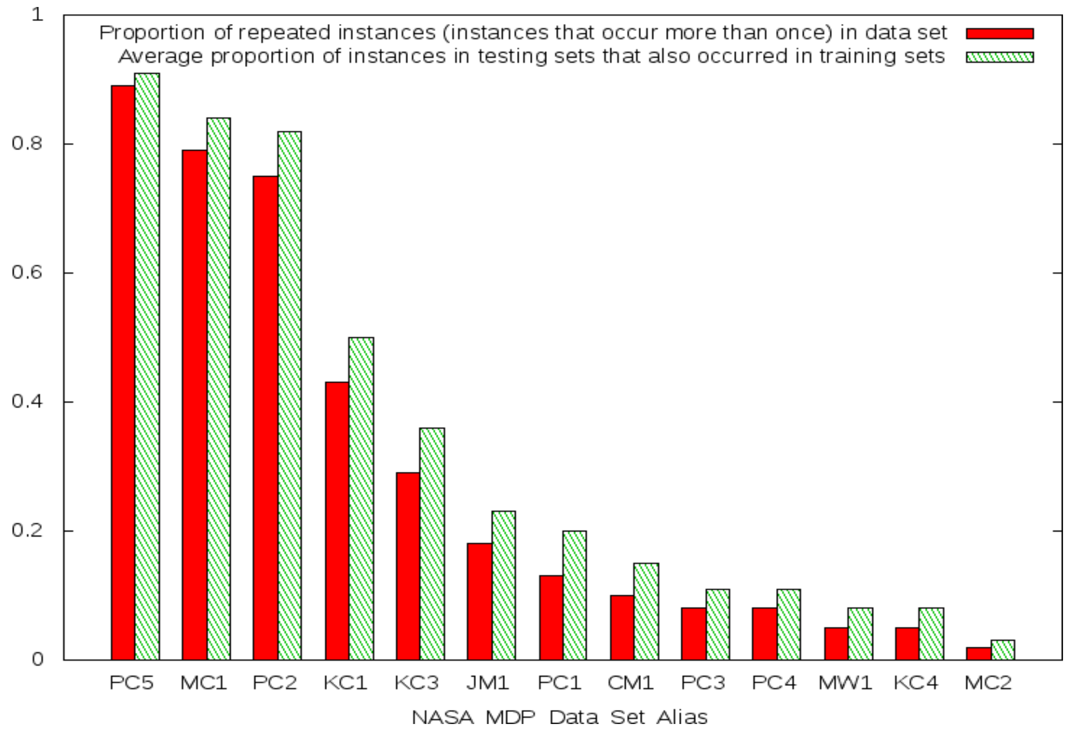
The size of the input space in these data sets and fault data sets in general has not been previously discussed in detail to the best of my knowledge. All base attributes (attributes not derived from simple equations of other attributes) in these data sets contain only discrete values ≥ 0. Therefore, the probability of a repeated data point in these data sets is much greater than, for example, a data set with real-valued measurements for the same number of attributes. This is why I believe that in future, researchers should additionally record lower-level metrics (such as character counts), to alleviate the issues with repeated data points by helping to distinguish non-identical modules. An interesting avenue of further research would be to investigate the repetition levels found in other fault data sets, to see if they are comparable with the levels of repetition found in the NASA data sets.

6.1.3.3 What proportion of seen data points could end up in testing sets when this data is used in classification experiments?

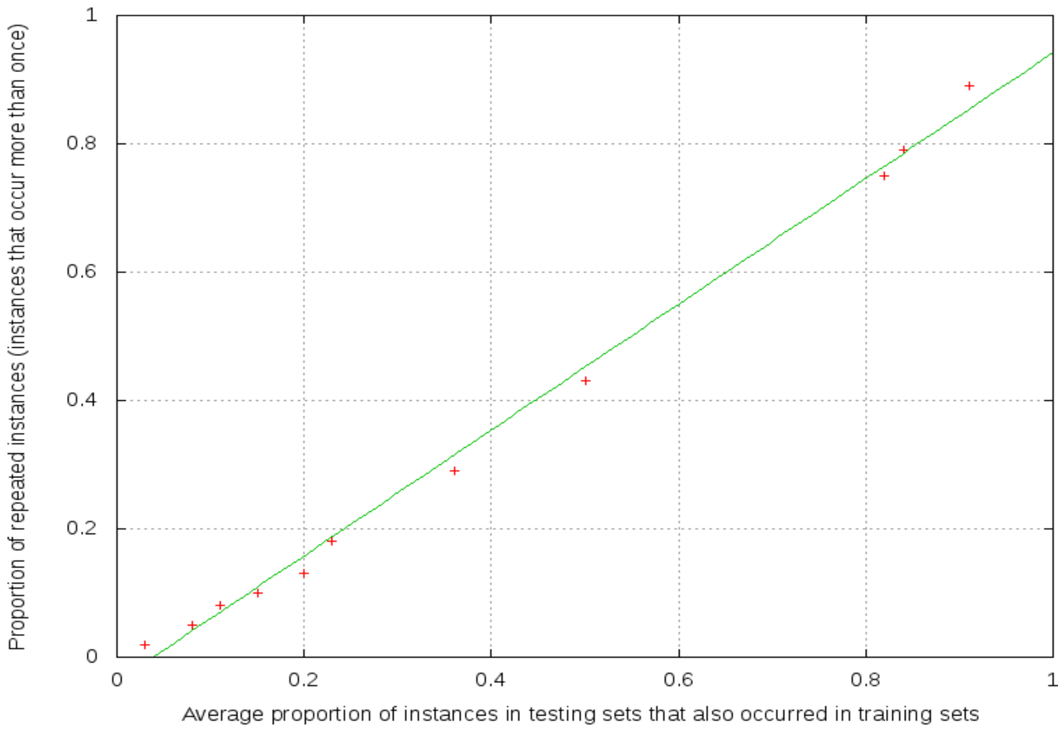
In order to find the answer to this question, a small Java program was developed utilising the Weka machine learning tool's libraries (version 3.7.5). These libraries were chosen as they have been heavily used in defect prediction research (see [Menzies 2007b, Boetticher 2006, Koru 2005b], for example). In this experiment a standard stratified 10-fold cross-validation was carried out. During each of the 10 folds, the number of instances in the testing set which were also in the training set were counted. After all 10 folds, the average number of shared instances in each testing set was calculated. This process was repeated 10000 times with different pseudo-random number seeds to defend against order effects. For this experiment I used the NASA data sets post basic pre-processing (see Table 6.1). I did this because it was as representative as possible of what will have happened in many previous studies. It is for the same reason that I chose stratified 10-fold cross-validation, the Weka default. The results from this experiment are shown in figures 6.4a & 6.4b. These figures show that for each data set, the average proportion of seen data points in the testing sets was greater than the proportion of repeated data points in total. Additionally, this relationship can be seen in Figure 6.4b to have a strong positive correlation. It is worth emphasising that in some cases the average proportion of seen data points in the testing sets was very large (91, 84 and 82 percent for data sets PC5, MC1 and PC2, respectively).

Figure 6.4: The dispersion of repeated data points in the NASA data sets.

(a) Proportions of repeated data and seen data in testing sets.



(b) Proportions of repeated data versus seen data in testing sets.



6.1.3.4 What effect could having such quantities of seen data points in testing sets have on classifier performance?

To answer this question I do not use the NASA data sets because of the point regarding class distributions mentioned in Section 6.1.1. Instead, for clarity, I construct an artificial data set, with 10 numeric features and 1000 data points. All features were generated by a uniform pseudo-random number generator and have a possible range between 0 and 1 inclusive. The data set has a balanced class distribution, with 500 data points representing each class. Note that artificially producing data is a technique that has been used in prior studies (see [Guyon 2003] for example).

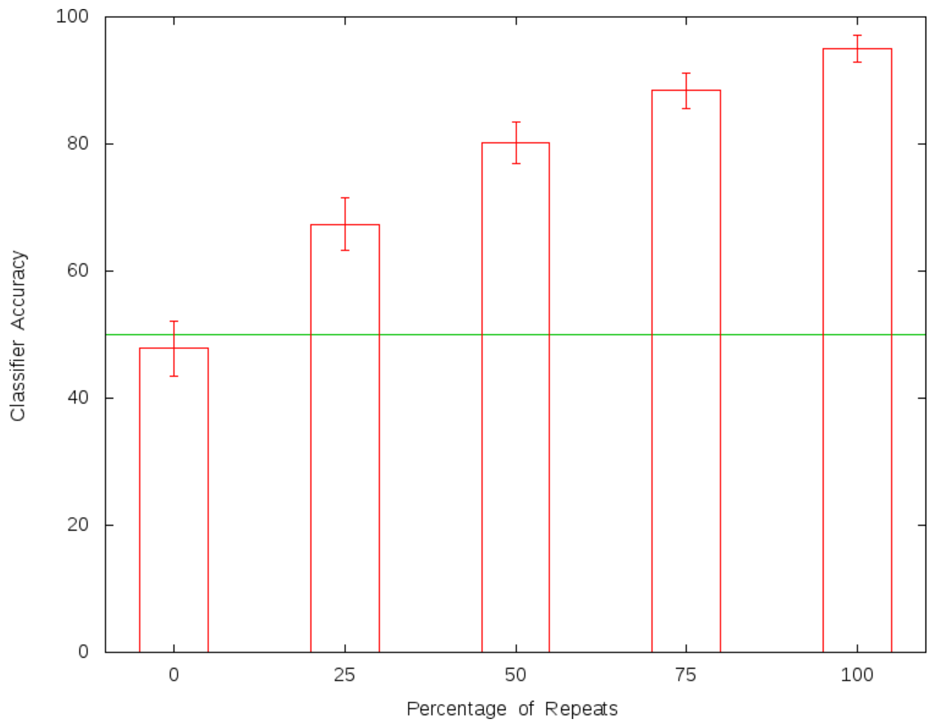
Using the Weka Experimenter, I ran 10 repetitions of a stratified 10-fold cross-validation experiment with the data set just described, and the following variations of it:

- 25% repeats from each class, an extra copy of 125 unique instances in each class, 1250 instances in total.
- 50% repeats from each class, an extra copy of 250 unique instances in each class, 1500 instances in total.
- 75% repeats from each class, an extra copy of 375 unique instances in each class, 1750 instances in total.
- 100% repeats, two copies of every instance, 2000 instances in total.

I used random forest learners for this experiment (see Section 3.2.3.1), with 500 trees and all other parameters set to the Weka defaults. I chose 500 trees as this was the number used in [Guo 2004, Jiang 2008c, Jiang 2008b]. The results of this experiment are shown in Figure 6.5. The mean accuracy levels for each data set: original, 25% repeats, 50% repeats, 75% repeats and 100% repeats were: 47.84, 67.38, 80.22, 88.37 and 95.00. This clearly shows that seen data points can have a huge influence on the performance of classifiers, even with pseudo-random data. As the percentage of duplicates increases, so does the performance of the classifiers. There are several major factors why this is the case, including that each node (or tree) comprising a random forest is:

- **Unpruned:** Meaning that the training data is essentially memorised. Therefore, in cases where the test data contains seen data points, these points are highly likely to be classified correctly, provided they are consistent in the training data.

Figure 6.5: Random Forest classifiers with repeated data points.



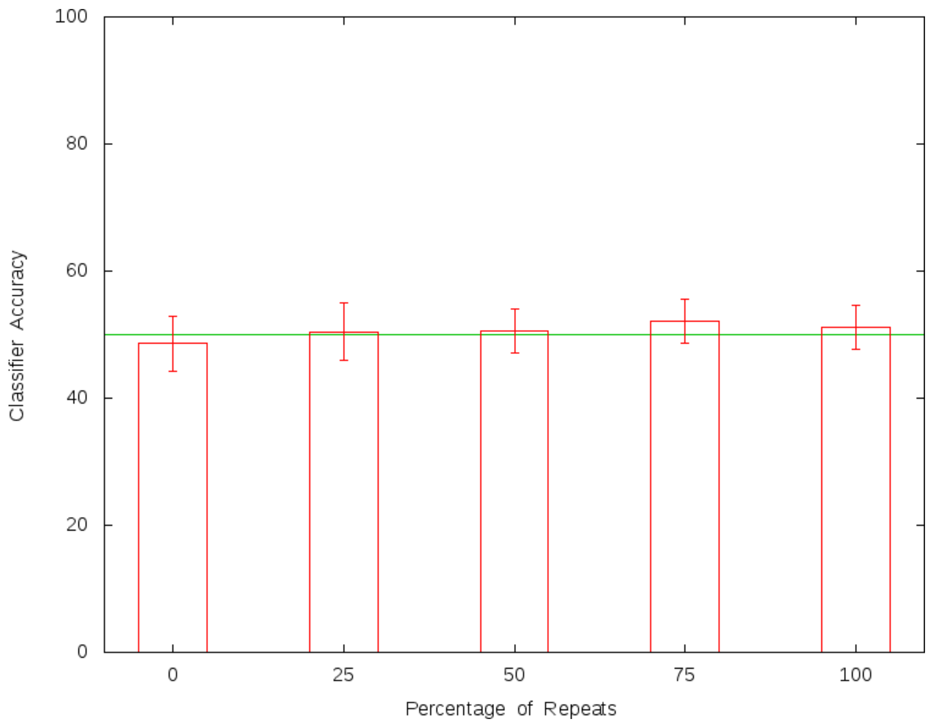
- **Trained on a bootstrap sample of the original training data**: Such samples are made with replacement, meaning that many original training instances will be repeated, and some not chosen at all. Clearly, if the original training data contains duplicates to begin with, the proportion of repeated instances trained on by each node will likely increase dramatically.

- **Trained on a subset of available features:** Meaning that, with the input space of each data point reduced due to this subset, there is a greater likelihood of repeated (and inconsistent) data.

The results from this experiment are very interesting, as random forests have been reported to work well with the NASA data sets [Guo 2004, Jiang 2007, Jiang 2008c, Jiang 2008b]. Note that random forests have also been reported to struggle with imbalanced data [Chen 2004, Segal 2004, Dudoit 2003]. This makes them not the obvious choice for use with the NASA data sets, as most of them are imbalanced (see Table 6.1). Despite this, favourable performance has been observed [Guo 2004, Jiang 2007, Jiang 2008c, Jiang 2008b], which I believe is partly due to the reasons just described. Note that I have confirmed these findings using version 5.1 of Breiman and Cutler's original Fortran code[3].

[3]http://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

Figure 6.6: Naïve Bayes classifiers with repeated data points.



It is worth pointing out that the effect of learning on repeated data points is algorithm specific. For example, naïve Bayes classifiers have been reported to be fairly resilient to duplicates [Kołcz 2003]. I confirm this by repeating the experiment just described with naïve Bayes classifiers. The results are shown in Figure 6.6. Although these results are not as striking as those for the random forests, I believe they are still noteworthy, and that in practice the effect may be significant.

6.1.4 Conclusions

Regardless of whether repeated data points are, or are not noise, it is unsuitable to have seen data in testing sets during experiments intended to show how well a classifier could potentially perform on future, unseen data. This is because seen data points can result in an excessive estimate of performance, occurring because classifiers can, to varying degrees, memorise rather than generalise. There is an important distinction between learning from and simply memorising data: only if you learn the structure underlying the data can you be expected to correctly predict unseen data.

Some researchers may argue that repeated data points should be tolerated and subject of no special treatment, as it is possible for modules with identical metrics to be contained within a software system. However, I have demonstrated that machine learning experiments based on data containing repeated data points can lead to invalid results. This is because it is possible for the test data to be (perhaps inadvertently) contaminated with training data, violating the assumption of unseen data. Even if researchers choose to ignore this fact, it is folly to report the performance of classification experiments with no distinction between performance on seen and unseen data. This is because the generalisation ability of classifiers may be far worse than the performance obtained implies.

If, in the real world, you happen to have a feature vector in your test set which is also contained in your training set, a simple lookup may be all that is required for best performance. A system could therefore be implemented where training data is stored in a hash table, and each test vector checked to see if it is contained within the hash table prior to classification. If so, the class label to predict could be looked-up directly with the classification model being unneeded (or some form of ensemble method used). A confidence interval could be derived using the lookup method, with a weight assigned to each training data point based on the number of times it occurs (due to replication) in the training data. This could also be extended to deal with inconsistent instances, perhaps by predicting the most frequently occurring of the classes. If the number of copies of the inconsistent instance in each class is equal, it may be best to report this, and/or make independent use of the classification model(s).

A simple approach to address the issues caused by repeated data points is to discard them prior to classification. A more sophisticated approach will be proposed in Chapter 8, which keeps test sets unmodified, and which may be a more realistic approach. Another possible option was proposed in [Boetticher 2006], and is to use an extra attribute: *number of duplicates*. This will help to ensure that information is not lost, and is most viable when repeats are believed to be genuine.

A possible reason why there are so many repeated (and inconsistent) data points within the NASA data sets is because of the poor differential power (and the small input space) of the metrics used. It may be highly beneficial in future to also record lower-level metrics (such as character counts), as these will help to distinguish non-identical modules, reducing the likelihood of modules sharing identical metrics. In addition to having lower-level metrics to begin with, researchers should be careful when discarding attributes, as this typically reduces the size of the input space, increasing the probability of repeated feature vectors. For example, [Menzies 2007b] used 8 of the NASA data sets with just 2 or 3 of the original 38 features. Removing over 92% of the available features in each of these data sets typically yields dramatic increases in the proportions of repeated and inconsistent instances.

"Data cleaning is a time-consuming and labor-intensive procedure but one that is absolutely necessary for successful data mining ... Time looking at your data is always well spent." [Witten 2005] I believe the data cleansing process defined in this study will increase the accuracy of the NASA data sets, and make them more suitable for machine learning. This process may also be a good starting point when using other software fault data sets. Experiments based on the NASA data sets which blindly included the repeated data points may have led to erroneous findings. This is because results are often reported as if they were based on unseen data, when in fact (to varying degrees) they were not. The impression given from the literature is that many defect prediction researchers using this data have not been aware of this issue. Future work may be required to (where possible) repeat these studies with appropriately processed data. Other areas of future work include:

- Extending the list of integrity rules described in stage 4 (Section 6.1.2.5) of the cleansing process. This will help to catch as many infeasible feature vectors, sets of metric values that can be proved to be noisy, as possible.

- Analysing other fault data sets to see whether the proportions of repeated data points in the NASA data sets are typical of fault data sets in general. This will help to determine the extent of this problem.

6.2 Performance Metric Problems

It is surprisingly difficult to characterise appropriately the performance of data mining classification algorithms from the field of machine learning. Deciding which performance measures to use involves taking several factors into account, including: the costs associated with misclassification, and the class distribution of the data. As was briefly discussed in Section 4.1.2, there has been much debate over the suitability of various performance measures used in this domain. This subject is revisited here.

In January 2007, the Menzies et al. paper *'Data Mining Static Code Attributes to Learn Defect Predictors'* was published in the IEEE Transactions on Software Engineering [Menzies 2007b]. In this study many defect prediction experiments were carried out (as described in Section 4.1). Classifier performance was reported using the two metrics used in *receiver operating characteristic (ROC)* analysis, the *true positive rate* and the *false positive rate.* In the journal these metrics were referred to as the *probability of detection (pd)* and the *probability of false alarm (pf)*, respectively. The use of these metrics motivated a comments paper by Zhang and Zhang [Zhang 2007]. They argued that the prediction "models built in [Menzies 2007b] are not satisfactory for practical use". This was because the *precision,* the proportion of modules predicted as being defective which were also originally labelled as being defective, was low for 7 out of the 8 data sets (between around 2.02 and 31.55 percent). Although the achieved precision values were omitted from [Menzies 2007b], Zhang and Zhang derived them using the available performance metrics and class distribution data. They conclude [Zhang 2007] by suggesting that, for reporting on the performance of software defect prediction models, the true positive rate be used with precision rather than with the false positive rate.

The Zhang and Zhang comments paper motivated a response by two of the original journal authors and two others [Menzies 2007a]. Here the main arguments were that "detectors learned in the domain of software engineering rarely yield high precision", and that low-precision predictors can be useful in practice. While it is true that low-precision predictors can be useful in certain contexts, and that lowering precision in order to increase the true positive rate may be desirable depending on your objectives, this is not enough to justify disregarding precision completely in such a domain.

In this section I demonstrate that when using data with a highly imbalanced class distribution, relying on true positive rates and false positive rates alone (this includes ROC analysis) may provide an overly optimistic view of classifier performance. I demonstrate this by showing that even when pairs of values for these measures appear to be near optimal, there is still considerable room for improvement in practical terms. This is not a novel finding. However, many defect prediction researchers have continued to report their classification results inappropriately since the publication of [Zhang 2007]. The contribution made here is the intuitive and easily comprehensible presentation of the examples given in Section 6.2.2.

The rest of this section is laid out as follows: Section 6.2.1 provides a background to machine learning classifier performance metrics. Section 6.2.2 describes the problem at hand, and why precision is required to describe classifier performance appropriately in highly imbalanced domains. The conclusions and advice for researchers are presented in Section 6.2.3.

6.2.1 Background

This section presents an overview of machine learning classifier performance metrics. A more thorough treatment was given in Section 3.3.3, which some readers may wish to revisit. In this study the scope is limited to that of binary classification problems, where performance is often quantified using a confusion matrix (see Figure 3.7). It is worth pointing out that in a confusion matrix, there is a symmetry between the positive and negative classes. However, the positive class typically refers to the class of most interest (*'defective'* modules), which is commonly (in this domain and many others) the minority class. Note that in this domain, a common assumption is that all modules predicted as defective will be subject to a manual code review.

Useful data statistics and commonly used classifier performance metrics can be derived from a confusion matrix; a subset of these were defined in Table 3.2. Note that in this table, the last two measures defined (*f-measure* and *balance*) are in their most commonly used form. It is however possible to weight them in order to favour either of their comprising measures [Jiang 2008a]. Additionally note that the *balance* measure was defined in [Menzies 2007b], it is a measurement of distance from a point on a ROC curve to the ideal point, which is typically defined as where the true positive rate is 1 and the false positive rate is 0.

The three measures of most interest in this study are: the true positive rate, the false positive rate, and precision. The true positive rate describes the proportion of data points labelled as positive which were correctly predicted as such; the optimal value is 1. The false positive rate describes the proportion of data points labelled as negative which were incorrectly predicted as positive; the optimal value is 0. Precision describes the proportion of data points predicted as positive which were correct (they were originally labelled as positive); the optimal value is 1.

In addition to observing classifier performance with a single, fixed set of parameters, it may also be desirable to observe how performance varies across a range of parameters. Doing so can be especially beneficial when performing classifier comparisons. ROC-curve analysis is commonly used for this task, and involves exploring the relationship between the true positive rate and the false positive rate of a classifier while (typically) varying its *decision threshold* [Witten 2005]. A trade-off commonly exists between the true positive rate and the false positive rate, and this is demonstrated by ROC analysis via a two-dimensional plot of the false positive rate on the x-axis and the true positive rate on the y-axis (see Figure 3.11). The area under the ROC curve (AUC-ROC) is commonly used to summarise a ROC curve in a single measure. The optimal AUC-ROC value is 1.

Precision and recall curves (PR curves) can be used in the same manner as ROC curves. On a *PR curve*, *recall* is on the x-axis and precision on the y-axis. A trade-off commonly exists between these two measures, and is thus shown on a PR curve. Note that recall is another alias for the true positive rate used in ROC analysis. PR curves are "an alternative to ROC curves for tasks with a large skew in the class distribution" [Davis 2006]. The area under the PR curve (AUC-PR) can be computed and used similarly to AUC-ROC. The optimal AUC-PR value is 1.

6.2.2 How Class Distribution Affects Performance Metrics

6.2.2.1 Example 1

Consider a perfectly balanced test set with 1000 data points, 500 in each class. If we achieve classification performance of *true positive rate (TPR)* = 1 and *false positive rate (FPR)* = 0.01, it appears as though our classifier has performed very well. All of the data points in the positive class have been correctly classified (TPR = 1), and only 1 percent of data points in the negative class (5 data points) have been incorrectly classified (FPR = 0.01). If we calculate the precision of such a classifier, it works out to 0.99 (to two significant figures). Therefore, approximately 99 percent of data points predicted to be in the positive class turned out to be correct. In this first example, where we have a balanced class distribution, using the TPR and FPR provided an honest and accurate representation of classifier performance, as a near-optimal pair of values likewise resulted in a near-optimal precision. A confusion matrix for this example is presented in Figure 6.7.

	labelled positive	labelled negative
predicted positive	TP = 500	FP = 5
predicted negative	FN = 0	TN = 495

Figure 6.7: TPR = 1, FPR = 0.01, Precision ≈ 0.99

6.2.2.2 Example 2

Now consider a test set with a highly imbalanced class distribution, as before with 1000 data points in total but this time with only 10 data points in the positive class (1 percent). If we again achieve TPR = 1 and FPR = 0.01, classifier performance may appear to be equal to that of the previous classifier. This is inappropriate, misleading, and exaggerates the classifier's real performance, as the precision of this classifier is 0.50 as opposed to around 0.99. Thus, *because of the imbalanced class distribution* (there being much more data in one class than the other), the same supposedly near-optimal TPR and FPR (or point on a ROC curve) represent vastly different performance. In the first example approximately 99 of every 100 positive predictions were correct, whereas in the second example, the same TPR and FPR represent a classifier for which only half of all positive predictions were correct. Note that because of rounding error, the FPR in this example was in fact 0.010102 (to six significant figures). A confusion matrix for this example is presented in Figure 6.8.

	labelled positive	labelled negative
predicted positive	TP = 10	FP = 10
predicted negative	FN = 0	TN = 980

Figure 6.8: TPR = 1, FPR ≈ 0.01, Precision = 0.50

6.2.2.3 Example 3

Turning attention toward the domain of software defect prediction, researchers often use data sets where the minority class represents less than 1 percent of data points in total (see [Menzies 2007b], [Lessmann 2008] & [Jiang 2009] for example). Thus, I now present an example using the most imbalanced of the NASA data sets: *PC2*. In original, unprocessed form this data set contains 5589 data points. Just 23 of the data points are labelled as *'defective'*, around 0.4 percent of data points in total. Thus, with a TPR of 1, all 23 data points labelled as *'defective'* were correctly classified. An FPR of 0.01 means that 1 percent of the 5566 data points labelled as *'non-defective'* were incorrectly classified. With approximately 56 false positives, a total of 79 (23 + 56) modules were predicted to require further attention. This works out to a precision of 0.29 (to two significant figures), despite the other metrics implying near-optimal performance. A confusion matrix for this example is presented in Figure 6.9.

	labelled positive	labelled negative
predicted positive	TP = 23	FP = 56
predicted negative	FN = 0	TN = 5510

Figure 6.9: TPR = 1, FPR ≈ 0.01, Precision ≈ 0.29

In the domain of software defect prediction, where the cost of false positives is typically not prohibitively large (see [Menzies 2007a]), such a classifier would, in most environments, be very attractive. However, my next observation is that, because of the highly imbalanced class distribution, small changes in the FPR have a large effect on the actual number of false positives.

6.2.2.4 Example 4

If, in the previous example, the FPR were to increase from 0.01 to 0.05, the number of false positives would increase from approximately 56 to approximately 278. The precision would subsequently drop to 0.08 (to two significant figures). In some environments examining 100 modules to find 8 of them to be defective would not be feasible. A confusion matrix for this example is presented in Figure 6.10.

	labelled positive	labelled negative
predicted positive	TP = 23	FP = 278
predicted negative	FN = 0	TN = 5288

Figure 6.10: TPR = 1, FPR ≈ 0.05, Precision ≈ 0.08

6.2.2.5 Example 5

As defect predictors do not reportedly achieve performance even close to TPR = 1 and FPR = 0.05, are these theoretical experiments valid? Zhang and Zhang [Zhang 2007] indirectly answer this question by highlighting the precision achieved in [Menzies 2007b] on data set PC2. Here, despite results that may initially appear acceptable (TPR = 0.72, FPR = 0.14), and claims of the naïve Bayes classifiers being "demonstrably useful", the precision was just 2.02 percent (to three significant figures). The classifier predicted that approximately 796 modules were in the 'defective' class, of which only approximately 17 actually were. This highlights the poor predictive performance achieved, and raises the question of whether such a classifier could be of any practical worth. An optimistic approximation of the confusion matrix for the entire data set (the results in [Menzies 2007b] were generated after 10 repeated runs of 10-fold cross-validation) is presented in Figure 6.11.

	labelled positive	labelled negative
predicted positive	TP = 17	FP = 779
predicted negative	FN = 6	TN = 4787

Figure 6.11: TPR ≈ 0.74, FPR ≈ 0.14, Precision ≈ 0.02

Note that an identical confusion matrix to the one in Figure 6.11 can be obtained simply by ranking all data points in data set PC2 by their *lines of code total* attribute (descending order), and predicting the first $n = 796$ data points as defective. It is a similar case for 2 more of the 8 data sets used in [Menzies 2007b], where n is the approximate average number of *'defective'* predictions made. This is known as *LOC module-order modelling* (see [Khoshgoftaar 2003] & [Mende 2009]), and in this case highlights both the poor actual predictive performance of the classifiers, and that research into making defect predictors 'effort aware' is worthwhile (see [Arisholm 2007] & [Mende 2010]).

6.2.2.6 Final Examples

Table 6.2 presents statistics for each of the 13 NASA data sets. The table shows class distribution details for each data set as well as the precision when TPR = 1, FPR = 0.01 and when TPR = 1, FPR = 0.05. Note that the false positives in these calculations were rounded to the nearest integer. The data sets are ranked in ascending order of the percentage of modules in the positive (minority) class. From the table it can be seen that for the three data sets with the highest class imbalance (namely: PC2, MC1 and PC5), near-optimal values of TPR and FPR result in classifiers which are far from optimal in terms of precision, and hence in practical terms. The table also shows, as was demonstrated in Example 6.2.2.1, that the TPR and FPR are suitable measures when using data with a predominantly balanced class distribution (as is the case for data set KC4).

NASA Data Set Alias	Total No. Data Points	No. Positive (Minority) Class Data Points	Percentage of Positive Class Data Points	Percentage Precision When TPR=1, FPR=0.01	Percentage Precision When TPR=1, FPR=0.05
PC2	5589	23	0.4	29.11	7.64
MC1	9466	68	0.7	41.98	12.64
PC5	17186	516	3.0	75.55	38.22
PC1	1107	76	6.9	88.37	59.38
MW1	403	31	7.7	88.57	62.00
KC3	458	43	9.4	91.49	67.19
CM1	505	48	9.5	90.57	67.61
PC3	1563	160	10.2	91.95	69.57
PC4	1458	178	12.2	93.19	73.55
KC1	2107	325	15.4	94.75	78.50
JM1	10878	2102	19.3	95.98	82.72
MC2	161	52	32.3	98.11	91.23
KC4	125	61	48.8	98.39	95.31

Table 6.2: Each of the NASA data sets ranked in ascending order of percentage data points in minority class.

Thus, relying solely on TPR and FPR or methods based around them, including: ROC analysis, AUC-ROC, and the *balance* metric, “can present an overly optimistic view of an algorithm’s performance if there is a large skew in the class distribution” [Davis 2006]. *Precision is required to give a more accurate representation of true performance in this context.*

6.2.3 Conclusions

Precision matters, especially when the class distribution of the data is highly skewed. When performing any kind of data mining experiment it is very important to document the characteristics of the data being used: where it came from, what preprocessing has been carried out, how many data points and features are present, what is the class distribution, etc. This makes it more accessible for other researchers to check the validity of the claimed results. For example, if such data characteristics are given, it is often possible to derive measures that are not explicitly reported, as was done here (and more recently in [Bowes 2012]). The inspiration for the work carried out here came from [Zhang 2007], where precision values omitted in [Menzies 2007b] were derived using the TPR, FPR, and class distribution data.

If classifier performance is to be reported with a single set of (hopefully optimised and thus suitable) parameters, I believe defect prediction researchers should be reporting a minimum of recall (TPR) and precision, in addition to the data characteristics just described. It is of no harm to also report the false positive rate. Note that it is necessary to take both recall and precision into account when assessing performance, a single one of these measures will not suffice. This is because an optimal recall can be achieved simply by predicting all data points as belonging to the positive class, and an optimal precision can be achieved by making only a single positive prediction, which turns out to be correct. The f-measure (see Table 3.2) is commonly used to combine both measures into a single value, and can simplify performance quantification effectively. When classifier performance is to be reported over a range of parameters, I believe precision and recall (PR) curves to be more suitable in this domain than ROC curves. This is because class distributions are often highly skewed [Menzies 2007a].

As described in Section 4.2, Lessmann et al. carried out a large-scale benchmarking defect prediction experiment with 22 classifiers [Lessmann 2008]. The top 17 of these classifiers were reported to have statistically indistinguishable performance using the AUC-ROC performance measure, and a statistical approach proposed by [Demšar 2006]. The data used in the study came from 10 of the NASA data sets. Davis and Goadrich point out that in highly imbalanced domains, PR curves are more powerful at distinguishing the performance of classification methods than ROC curves [Davis 2006]. Thus, I think it would be interesting for the experiment by Lessmann et al. to be replicated with PR curves and AUC-PR.

In a recent paper by Menzies et al. [Menzies 2010], it is stated that the "standard learning goal" of defect predictors is to maximize AUC-ROC. I argue that this should not be the standard learning goal of defect predictors. As shown here, by [Davis 2006], and by [Zhang 2007], precision needs to be taken into account in a typically highly imbalanced domain. Moreover, [Davis 2006] prove "that an algorithm that optimizes the area under the ROC curve is not guaranteed to optimize the area under the PR curve".

In addition to the comments made about defect predictor learning goals, the Menzies et al. paper [Menzies 2010] also states "that accuracy and precision are highly unstable performance indicators for data sets . . . where the target concept occurs with relative infrequency". In my view this assertion is only partly correct. While it is commonly reported within the data mining literature that accuracy is not a suitable measure when using imbalanced test sets, this is not the case for precision (when reported along with recall). To further clarify this, He and Garcia state that "when used properly, precision and recall can effectively evaluate classification performance in imbalanced learning scenarios" [He 2009]. It should be noted, however, that in the context of experiments to explore the *class imbalance problem* (see Section 3.4), the FPR is more suitable than precision [Batista 2004].

6.3 SVMs for Defect Prediction Revisited

Here, I present the results from a similar classification experiment to the one described in Section 5.1, but in this experiment all the major issues that have been discussed in both this chapter and Chapter 5 are addressed. An assumption made in this experiment was that the repeated and inconsistent instances are noise; therefore, the full data cleansing process defined in Section 6.1.2 was carried out. To avoid the undersampling issues described in the previous chapter, no such sampling was undertaken, and the original class distributions (post data cleansing) were used in all training and testing sets. This was realised by using stratification in the 10-fold cross-validation process. After pre-processing, the experimental design used in this study was very similar to that shown in Figure 3.13, with the following differences:

- The value of M was 10, meaning that results were averaged over 10 repetitions of stratified 10-fold cross-validation.

- All 13 of the pre-processed NASA data sets were used.

- The parameter range used during model optimisation was as follows (start, stop, step, all $\log_2$): *cost* -5, 15, 2 : *gamma* 3, -15, -2. More clarification on this is provided in sections 3.5 & 3.6.

Name	Features	Instances	No. Defective Instances	% Defective Instances
CM1	37	454	46	10
JM1	21	7722	1612	21
KC1	21	1163	294	25
KC3	39	324	42	13
KC4	13	113	57	50
MC1	38	1978	36	2
MC2	39	156	51	33
MW1	37	376	28	7
PC1	37	949	63	7
PC2	36	1389	21	2
PC3	37	1433	150	10
PC4	37	1286	176	14
PC5	38	1699	459	27

Table 6.3: Details of the NASA data sets post data cleansing.

The updated data set statistics post the full data cleansing process are given in Table 6.3. Recall that the data set statistics prior to data cleansing were given in Table 6.1. Interesting observations regarding the new statistics include that there are large class-distribution changes in some of the data sets. For data set PC5, 27% of data points comprise the minority class compared to 3% before cleansing. The level of imbalance has also been reduced for many other data sets. Another interesting point to note is that for data set KC4, there is now one more instance in the 'defective' class than there is in the 'non-defective' class. This means the 'defective' class is now the majority class in this data set, albeit by a very narrow margin.

The results from this experiment are given in Table 6.4. The last column in this table contains a measure that has not been previously described, namely: *Matthew's Correlation Coefficient (MCC)* [Matthews 1975]. This measure is defined as follows:

$$MCC = \frac{(TP * TN - FP * FN)}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{6.2}$$

The MCC, which is particularly popular in the field of bioinformatics, gives a value between -1 and 1, with 1 representing perfect classification. A notable property of MCC is that a value of 0 represents classification that is no better than random. Therefore, a simple baseline check regarding the worth of a classifier is to make sure its MCC is positive. The MCC equation takes all four components of the confusion matrix into account, and provides a high-level, overall view of classifier performance.

Name	Recall	Precision	F-Score	FP Rate	MCC
CM1	.28	.30	.29 (.18)	.07	.22
JM1	.19	.49	.28 **(.35)**	.05	.21
KC1	.38	.51	.43 (.40)	.12	.29
KC3	.29	.34	.31 (.23)	.09	.22
KC4	.82	.67	.74 (.67)	.41	.42
MC1	.09	.13	.11 (.04)	.01	.10
MC2	.45	.57	.50 (.49)	.17	.30
MW1	.31	.45	.37 (.14)	.03	.33
PC1	.31	.34	.32 (.12)	.04	.28
PC2	.07	.10	.08 (.03)	.01	.07
PC3	.28	.31	.29 (.19)	.07	.21
PC4	.50	.69	.58 (.24)	.04	.53
PC5	.43	.56	.49 (.43)	.13	.34

Table 6.4: The classification results from this experiment. The values in brackets after the f-score values show the f-score that would be achieved by making only defective-class predictions. Observe that the actual performance achieved for data set JM1 is worse than this simple baseline predictor.

The results presented in Table 6.4 show that there is a positive MCC value for each data set. This indicates that overall predictive performance is better than random; however, in the case of data sets PC2 and MC1 (with MCC values of .07 and .10, respectively), performance is only slightly better than random. The reason for this is almost certainly because these are the two data sets with the highest level of class imbalance: only around 2% of data points comprise the minority class in each case. Although overall predictive performance is better than random (as shown by the MCC values), of most interest in this typically imbalanced domain is performance on the 'defective' class. For this reason, all performance measures in Table 6.4 other than MCC are specific to this class. As previously discussed, the f-score (or f-measure) is often used to effectively combine precision and recall into a single value. It is most commonly defined as the harmonic mean of these two comprising metrics, as is the case here. A trivial baseline that can be used regarding f-score is to compare the actual f-score achieved with the f-score that would be obtained by making only positive-class predictions. Recall that in a binary-classification context, making only positive-class predictions would result in a recall of 1, and a precision that is equal to the proportion of positive data points in the test data. If we call this proportion d, then the resultant f-score would be equal to:

$$Baseline\_F\text{-}score = \frac{(2d)}{(1+d)} \tag{6.3}$$

This trivial baseline is a good starting point for assessing the worth of a predictor, particularly in imbalanced classification domains. The results in Table 6.4 show that with the exception of JM1, all data sets improved on this baseline, although in some cases the improvement was very small. The poor performance achieved for data set JM1 is surprising when considering that it is the largest of the data sets (in terms of instances), and that it is not highly imbalanced (around 21% of instances comprise the minority class). Best performance was achieved with data set PC4, which had both the greatest performance increase over the baseline f-score and the highest MCC value. The recall was .50, meaning that half of all faults were found in the modules predicted as defective, and the precision was .69, meaning that there was around a 70% chance that a module predicted as defective would contain one or more faults. Notice that the general level of performance in terms of recall and precision is low, even with the use of such highly sophisticated classifiers.

Confusion matrix-based performance measures can only go so far in describing the worth of a classifier. For example, such measures may not take domain-specific technicalities into account, such as the fact that, if a manual code review were to be undertaken for each module predicted as defective, then predicting a large module as defective would consume more resources than predicting a small module as defective. Moreover, it may be more cost effective to predict many small modules than few large modules. Work to address this problem has been undertaken (see [Arisholm 2007, Mende 2010]), but none of the proposed techniques are yet well-established. Furthermore, to truly assess the worth of such predictors, it would be necessary for the technical details regarding the severity of each and every known fault to be available. This highlights the suitability of open-source systems for use in defect prediction studies, as the bug history for such systems should be freely available for analysis.

Chapter 7

Obtaining Fault Data

Contents

Because of the data quality issues described in the previous chapter, I decided to undertake the manual construction of a new fault data set that would be suitable for defect prediction. This was considered a better option than using public-domain data sets, as these are often unverifiable and based on closed-source systems. Initially the new data was intended to come from the open-source Apache HTTP server; however, this system was deemed too large for a first attempt. Therefore, a smaller system was chosen instead, namely: *Barcode* (described in the following section). Although initial progress at constructing the new data set was satisfactory, it soon became apparent that determining whether or not a source file contains an intended fault-fix is a non-trivial task (see Section 7.1.3). In light of this, data set construction was halted for a more thorough investigation on the intricacies of constructing fault data. The findings from this investigation led to the casting of more doubts on the worth of many public-domain data sets, and in particular the NASA data sets. The next sections describe: the Barcode system, the database used in this study, the method used for categorising source file revisions, the findings and the conclusions.

7.1 Barcode

Barcode[1] is a C library for creating barcode labels which includes a command-line front-end. It is an open-source system released under the GNU General Public License, and has been utilised in prior research studies (such as [Meyers 2007]). Over the project's public life span of around 7 years, concurrent versions system (CVS) has been used as its revision control system. An interrogation of the CVS server[2] reveals that there have been 359 revisions (or *check-ins*) within 137 transactions (or *change-sets*). The Barcode CVS server stores the history of 58 regular files and 6 directories, the distribution of regular file types is shown in Table 7.1. Barcode was chosen for this study as it is:

- **Open source.** One of the major problems with public-domain fault data sets is that they are often based on commercial, closed-source systems. Thus it is not always possible for users of such data sets to access the original source code on which they are based. Access to the original source code is beneficial as it helps to confirm data integrity, which is especially useful with regard to *outliers,* data points which substantially differ from the rest of the population from which they are drawn. Because Barcode is open source, access to its original source code is free to the public via a CVS server allowing 'anonymous' checkouts. The main drawback of basing this study on an open-source system is that the conclusions derived from the gathered data may not be generalisable to proprietary projects [Wright 2010].

- **Well documented.** Barcode is well documented in that a change log is rigorously updated by the authors. A change log is a plain-text file containing concise, but typically more detailed information on what was changed, by whom, why, and where than is available in CVS commit messages alone. The change log is the most commonly edited file in the barcode project, with 58 separate revisions stored by the CVS server. Examining change log entries can be invaluable when trying to determine the reasoning behind file modifications.

- **Small.** Barcode is small in terms of the number of: source files (see Table 7.1), revisions (359), lines of code (approximately 4000), and CVS committing developers (two). This reduces the amount of time required to categorise revisions (see Section 7.1.2), which is desirable as this is both a labour intensive and cumbersome process. Additionally, having a small number of CVS committers reduces change history complexity.

[1] `http://ar.linux.it/software/#barcode`

[2] In June 2010, although no changes have been made between then and now.

File Type	Quantity	Percentage Total
C Source File	16	28%
C Header File	3	5%
Python Source File	1	2%
Compressed Archive	4	7%
Make File	5	9%
Readme File	12	21%
Miscellaneous	17	29%

Table 7.1: Details of the regular file types comprising Barcode.

7.1.1 Database Initialisation

Using the open-source tool: CVSanaly2[3], it is possible to extract most of the information held by a CVS server and store it in a database. A shortcoming of CVS is that commits are not atomic, thus it is not always clear as to which revisions are a part of which transactions. To alleviate this problem, I developed a Python script to group revisions into transactions. This involved modifying the CVSanaly2 generated database. The method used for grouping revisions to transactions was based on that described in [Zimmermann 2004]. This method involves sorting revisions by author, timestamp and log message. It uses a sliding time window approach between revisions, so can deal with transactions of arbitrary size and cases where network communication is slow.

The aforementioned Python script was also used to create a 'change category' table, which is a foreign key for each revision in each transaction. This table contains a set of possible reasons describing the intended purpose of the modifications made. The change category labels are shown in Table 7.2. Labelling each source code revision into one of these categories is the starting point for obtaining accurate fault data. Some of this task was carried out using automated methods, while the remaining revisions were classified manually.

[3]http://forge.morfeo-project.org/projects/libresoft-tools/

ID No.	Change Category
1	Definite Fault-Fix
2	Probable Fault-Fix
3	Unsure
4	Probable Non-Fault-Fix
5	Definite Non-Fault-Fix
6	Outlier
7	Not A Chosen File Type
8	Initial Revision
9	Yet To Be Assessed

Table 7.2: Each of the possible change categories.

7.1.2 Categorising Revisions

Initially each revision will have its change category assigned to one of the following three categories:

- Not A Chosen File Type
- Initial Revision
- Yet To Be Assessed

Table 7.1 shows that there are 16 C source files partially comprising Barcode. It is from the history of these files that function (or *module*) based fault data can be extracted. Module-level fault data is the most common granularity of fault data reportedly used within the literature. This is probably because module-level predictions, if accurate, would be far more useful in practice than package, file or class level ones (see Chapter 2). This is due to there being typically less code to search through and comprehend for each 'defective' prediction made, assuming a manual code review is the resulting action taken.

When initialising the Barcode fault database, all files not ending with a '.c' extension were labelled as "Not A Chosen File Type". This included C header files, as although they may contain faults, they do not contain module implementations. It also included the single Python file; it was decided not to include this file as it is not a main part of the system. Additionally, having to focus on another language as well as C would have increased complexity later on.

By definition, the first/initial revision of a file (revision 1.1 in CVS) cannot include a fault-fix of a previous version of the same file (that exists in the revision control system). Therefore, the initial revision of each Barcode file (as long as it is a 'chosen file type') is automatically labelled with the "Initial Revision" category. This category can conceptually be seen to imply the "Definite Non-Fault-Fix" category, as even if the new file contains code intended to rectify faults contained in other files, it cannot contain code intended to rectify faults contained in a previous version of the same file (that exists in the revision control system). Such within-file fault-fixes are of main interest in this study, as they are the most suitable when trying to determine the *fix-inducing changes* (see Chapter 2). It is worth mentioning that CVS has no native notion of file renaming; therefore, a problem with this method is that a renamed file containing a fault-fix will be missed.

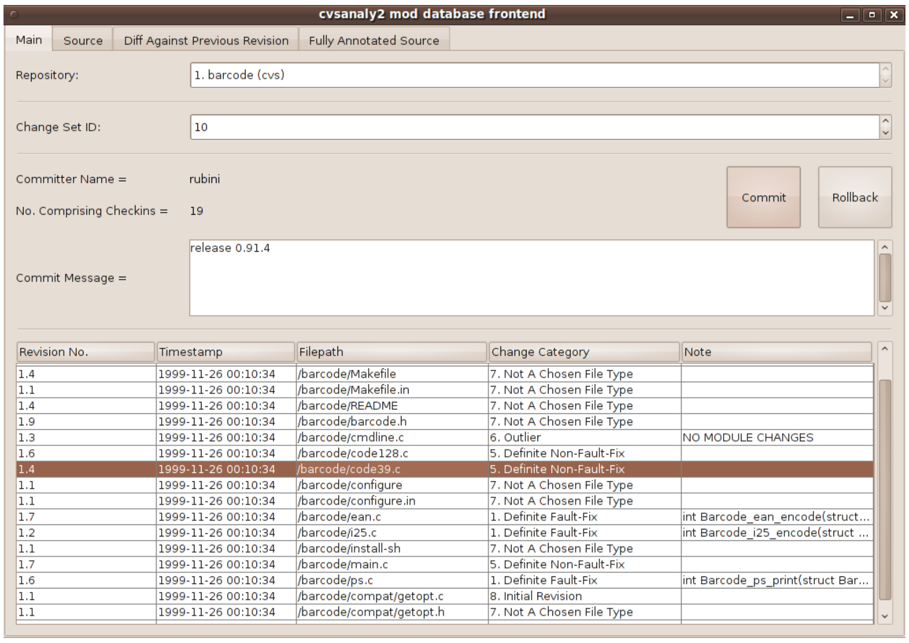
All revisions which had file names ending in '.c' and which were not initial revisions were thus labelled as "Yet To Be Assessed". It was these files that would be manually examined to determine the intended purpose of their modifications. There were 110 such revisions, 31% of all revisions in total.

The first 5 categories shown in Table 7.2 are on an ordinal scale. They describe the degree of membership a revision has with respect to being an intended fault-fix. The first category on this ordinal scale is "Definite Fault-Fix", the last is "Definite Non-Fault-Fix", and "Unsure" is the midpoint. The result of manually examining each revision is typically a classification into one of these five categories. If part of the changes to a revision are to a module, and those changes are deemed to be an intended fault-fix, then the revision is labelled as either a "Definite Fault-Fix" or a "Probable Fault-Fix", depending on the experts degree of certainty. Additionally, the module header where the fix is thought to have occurred is recorded in a textual database field (there can be multiple entries if required). All headers not recorded in this field are therefore implied to be in the "Definite Non-Fault-Fix" category for that particular revision.

The final category to be discussed: "Outlier", is used in circumstances where:

- **There are no module (function) changes.** Because the module level is the chosen fault granularity level in this study, any revision with no module changes should not be considered with respect to module-based faults. This can occur when, for example, a global variable change is the only modification made in a revision.

- **The only change is a syntax-error fix.** This occurs when developers commit their code to the server without first checking that it successfully compiles. Syntax errors in compiled languages (such as C) cannot affect end users as they prevent successful compilation. Software defect predictors are not intended to detect such errors as they can be found with far more efficiency by compilers.


cvsanaly2 mod database frontend
Main
Source
Diff Against Previous Revision
Fully Annotated Source
Commit
Rollback

Repository: 1. barcode (cvs)

Change Set ID: 10

Committer Name = rubini

No. Comprising Checkins = 19

Commit Message = release 0.91.4

Revision No.	Timestamp	Filepath	Change Category	Note
1.4	1999-11-26 00:10:34	/barcode/Makefile	7. Not A Chosen File Type	
1.1	1999-11-26 00:10:34	/barcode/Makefile.in	7. Not A Chosen File Type	
1.4	1999-11-26 00:10:34	/barcode/README	7. Not A Chosen File Type	
1.9	1999-11-26 00:10:34	/barcode/barcode.h	7. Not A Chosen File Type	
1.3	1999-11-26 00:10:34	/barcode/cmdline.c	6. Outlier	NO MODULE CHANGES
1.6	1999-11-26 00:10:34	/barcode/code128.c	5. Definite Non-Fault-Fix	
1.4	1999-11-26 00:10:34	/barcode/code39.c	5. Definite Non-Fault-Fix	
1.1	1999-11-26 00:10:34	/barcode/configure	7. Not A Chosen File Type	
1.1	1999-11-26 00:10:34	/barcode/configure.in	7. Not A Chosen File Type	
1.7	1999-11-26 00:10:34	/barcode/ean.c	1. Definite Fault-Fix	int Barcode_ean_encode(struct...
1.2	1999-11-26 00:10:34	/barcode/i25.c	1. Definite Fault-Fix	int Barcode_i25_encode(struct ...
1.1	1999-11-26 00:10:34	/barcode/install-sh	7. Not A Chosen File Type	
1.7	1999-11-26 00:10:34	/barcode/main.c	5. Definite Non-Fault-Fix	
1.6	1999-11-26 00:10:34	/barcode/ps.c	1. Definite Fault-Fix	int Barcode_ps_print(struct Bar...
1.1	1999-11-26 00:10:34	/barcode/compat/getopt.c	8. Initial Revision	
1.1	1999-11-26 00:10:34	/barcode/compat/getopt.h	7. Not A Chosen File Type	

Figure 7.1: The main screen of the revision-labelling front-end.

When labelling each of the revisions it was often possible to examine not only the source code and corresponding commit message, but also the previously mentioned change log. The change log is well maintained and typically contains concise details on what has been changed, by whom, why, and where. For example:

```
2000-01-26  Alessandro Rubini  <rubini@morgana.systemy.it>
* code128.c (Barcode_128_encode): new encoding: full-featured code128
```

Here it is shown that on the 26th of January 2000, Alessandro Rubini committed the 'code128.c' source file. The changes made were to the *'Barcode_ 128_ encode'* function, and the purpose of this modification was to add a new encoding (*'code128'*). Such textual notes can be invaluable when trying to determine the intention of modifications, especially for individuals who did not take part in the development process.

To simplify the process of manually labelling revisions, I developed a graphical Java front-end to the modified CVSanaly2 MySQL database. This front-end was used as the interface for the entire labelling process. A screenshot of the front-end main screen is shown in Figure 7.1. The figure shows that after selecting a repository name and a transaction (change-set) number, the following information is displayed:

- The transaction committer name.
- The number of transaction-comprising revisions (check-ins).
- The transaction commit message.
- And for each revision: its unique identification number, timestamp, relative filepath, change category, and corresponding note. Only the change category and note fields are editable by the user. The change category options are the same as those in Table 7.2, and are presented to the user via a combo box. The note field is for entering either the suspected fault-fix module header(s) ('+' separated list), or the justification for an outlier label (either 'NO MODULE CHANGES' or 'SYNTAX ERROR FIX ONLY').

In the table at the bottom of the window (in Figure 7.1) where each of the revision details are displayed, there will always be precisely one of the revisions highlighted. The highlighted revision indicates which revision the user is interested in when they select the tabs (at the top of the window) to display: the file source, the textual diff against revision $n-1$, or the fully annotated source, where each line is labelled with the revision number at which it was last modified.

The process for categorising revisions defined above was carried out twice. The first iteration was a trial run where all of the available categories were utilised. The second iteration was more rigorous than the first, and a more definitive classification was given to each revision. Thus, by the end of the second iteration, only three of the categories from Table 7.2 were used to classify the required source files: "Definite Fault-Fix", "Definite Non-Fault-Fix" and "Outlier".

7.1.3 Findings

The process of manually labelling each of the 110 Barcode source revisions typically involved examining the diff of both the source file and the change log (if present in the transaction), as well as the corresponding commit message. This was a lengthy and highly cumbersome process, and there were many failed attempts that resulted in the whole process having to be started afresh. The main problems were: difficulties in determining what was and was not a fault-fix, difficulties in being consistent with labellings across all revisions, and discovering part way through the labelling process that a database structure modification was required. The latter problem is understandable when considering that this was a first attempt at collecting fault data. This highlights the benefit of starting with a small system such as Barcode, as with a larger system having to restart the labelling process several times may have consumed too much time.

The failed labelling attempts, although highly demotivating and time consuming, did have the positive outcome of helping to more clearly define precisely what was considered a fault-fix. By the final labelling iteration a fault-fix was defined as: A module-based non-syntactic modification intended to rectify undesired program behaviour caused by one or more previous versions of the containing module. Note that 'undesired program behaviour' in this case is considered from the end-users perspective. Changes such as replacing the following line:

```
if (! strlen(characterArray))
```

with:

```
if (characterArray[0] == '\0')
```

to remove the unnecessary function call were not counted as a fault-fix because they should not affect an end user (in the context of the Barcode system). Note that in this context the tiny increase in execution speed and decrease in memory usage should not be a factor. Furthermore note that the *strlen* function is a part of the 1989 ANSI C (C89) standard, so compatibility issues should also not be a factor.

The distribution of file change categories for the 110 manually classified revisions is shown in Figure 7.2. This figure shows that the category for the fault-fixes had the highest number of revisions (50), the category for the non-fault-fixes had the least number of revisions (29), and there were a surprisingly large number of outliers (31). Most of these outliers (94%) were due to there being no module changes within the modified file. If it had been known beforehand that such a large number of revisions were to contain no module changes, such revisions would have been assigned their own exclusive category, and a tool for their automated classification would have been developed. This is potential future work.

There were many issues when trying to determine what was and was not considered a fault-fix. To revisit the example just described involving the *strlen* function, it could be reasonably argued that this was a fault-fix rather than a refactoring, as it resulted in more computationally efficient code. A second example is a revision involving the replacement of the *snprintf* function with the *sprintf* function, as the former does not conform to C89. I did not consider this to be a fault-fix as I was only interested in faults that could affect end users, and my definition of an end user is not someone who has to compile the software. It would be entirely reasonable to have a different definition where end users where expected to have to compile the software however, especially for open-source systems. A third example is that of revisions involving only minor output-formatting changes. Although I classified these as fault-fixes because they rectified undesired program behaviour, others may believe that such trivial output-formatting issues do not constitute a fault.

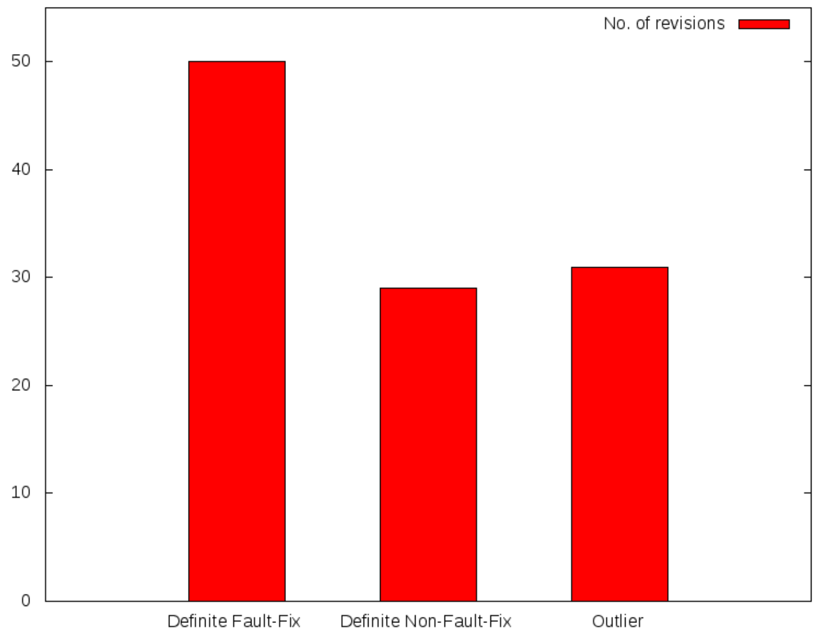


Figure 7.2: The distributions of the manually classified revisions.

The potential subjectivity of the fault-fix categorisation process is highlighted further by the findings in [Hall 2010], where three researchers independently labelled the Barcode source revisions and were found to have low inter-rater reliability. These findings suggest that detailed documentation of the labelling process is a prerequisite of reasonable quality fault data, especially if that data is publicly available. Without such documentation (and ideally the originating data sources) it is difficult to have a satisfactory level of confidence in the labels of the data points.

Although the issue of subjectivity was partly mitigated in this study because of the clear fault-fix definition (given previously) and the details of each categorisation being recorded, data set construction ceased because of a lack of domain expertise. I had no prior experience of barcode-label programming and was not especially familiar with such low-level C. This meant that for revisions where the change log was not particularly detailed, it was often very difficult to have confidence in a classification. This leads me to believe that perhaps categorising past revisions should be undertaken only by those who actively develop(ed) a system, or who are particularly knowledgeable in the application domain. Better still, more sophisticated support for documenting the intended purpose of modifications could be integrated into revision control software. This would help automate the process of data set construction, and could potentially increase the accuracy of fault data sets made in future.

7.2 Conclusions

The work described in this chapter illuminated that constructing accurate software fault data is far more difficult than may be initially perceived. This is especially true for those who did not take part in the development of the system being studied. A major difficulty when constructing fault data is the process of categorising whether or not a past revision contained a fault-fix. Along with the technical difficulties of comprehending the intended purpose of the modifications, there is also the subjectivity as to the definition of a fault. Therefore, to produce accurate and meaningful fault data, strict definitions must be made, documented, and consistently adhered to.

These findings bring back into light the lack of documentation available for the NASA and PROMISE data sets. It may be that this is a much more severe problem than previously thought. The sparse documentation available for the NASA data sets was described in Section 4.1.1. For the PROMISE data sets, there is typically even less documentation, if any is provided at all. For data sets such as the NASA ones where the original data sources (code and revision control archives) are not publicly available, the importance of data set documentation is magnified. This is because there is little opportunity for future data integrity checks. Thus, the worth of the NASA data sets is called into question once more.

Chapter 8

Finalising the Methodology

Contents

In this chapter I propose a new methodology for software defect prediction. Much of the content in this chapter has been previously discussed, thus there are many backward references. The methodology is proposed in four main parts: the first part is discussed in Section 8.1 and concerns the obtaining of fault data; the second part is discussed in Section 8.2 and concerns the analysis and potential cleansing of fault data; the third part is discussed in Section 8.3 and concerns the process of defect prediction; the fourth part is discussed in Section 8.4 and concerns quantifying predictive performance.

8.1 Obtaining Fault Data

This section is aimed at those wanting to gather primary data for use in defect prediction experiments. The software system from which to obtain this data may be either open or closed source. The following proposals are heuristics gathered from my experiences described in Chapter 7:

- Ensure the chosen system has documentation of known faults available in some form of bug repository.
- Ensure it is possible to obtain input from individuals who are experienced with the system's source code and/or the application domain.

- Make, document, and consistently adhere to a detailed definition of precisely what constitutes a fault. This definition should be used when determining whether or not each past revision contains one or more fault-fixes. If the final data set is made available to others as secondary data, the fault definition should be made available with it, even if the source code and/or bug repository cannot be distributed. In fact, as much documentation as possible regarding the chosen system and the data construction process should be made available. This is so that those either interpreting results based on the data (see [Hall 2012]) or making use of the physical data (if it is distributed) have sufficient contextual information.

- Record very many low-level static code metrics (such as the number of characters in total, number of assignment statements in total, and so on) all the way up to standard, high-level metrics (such as the number of lines in total). This will reduce the possibility of code units sharing identical metrics, which is beneficial as this leads to duplicate and inconsistent instances. Ideally, the size of the input space should be powerful enough to uniquely identify each data point (where this is possible). Recall that not all of these features will necessarily need using during classification, as using too many features may illuminate the *curse of dimensionality*, "the apparent intractability of accurately approximating a general high-dimensional function" [Donoho 2000].

8.2 Analysing & Cleansing Fault Data

Prior to undertaking a prediction experiment, it is imperative for a substantial data analysis to have been carried out. Although the former statement may appear obvious, it is worth noting that on many previous occasions this has not been the case (see chapters 4 & 6). Much in the same way that it is folly to begin the implementation of a software system without a sufficient requirements specification, it is folly to carry out a prediction experiment without an analysis of the data[1]. Witten and Frank [Witten 2005] correctly state: "Data cleaning is a time-consuming and labor-intensive procedure but one that is absolutely necessary for successful data mining . . . Time looking at your data is always well spent." Interestingly, they also state: "Preparing input for a data mining investigation usually consumes the bulk of the effort invested in the entire data mining process."

[1]The only exception to this could be in cases where some of the data is removed early on to serve as the test data, and in an effort to ensure that this data is entirely unseen and that there is no possibility of overfitting it, it is not included in data analysis and only the training data is analysed. In which case a retrospective analysis should be performed on the test data post experimentation.

A good starting point when analysing data is to discover where it came from and what was its intended purpose. Using data for a different purpose from that what it was originally intended may be problematic, and in some cases could introduce legal issues. Another important factor is whether the data is first-hand or second-hand. Typically, being further removed from the data means there must be more blind faith; however, utilising a well-used, publicly available data set does have the benefit of there being a greater chance that it has already been thoroughly analysed (although this was not previously the case for the well-used NASA data sets). If details of a prior analysis are available, then the analysis should be repeated to ensure consistency. This can be a good method of checking whether the version of the data set being used is the same as the version used by those who carried out the prior analysis. Possible avenues for extending the analysis should then be examined, some of these will be detailed later in this section.

After obtaining the background and context information regarding the data, a logical next step is to gather some basic statistics for each attribute (or feature). The type of statistics that can be recorded will vary depending on the attribute's data type. For numeric attributes there will be many commonly used statistics that can be recorded; however, there are some statistics that should be recorded regardless of attribute type. These include the number and proportion of both missing and distinct values. Missing values were discussed in Section 6.1.2.4, along with a data cleansing procedure to address them in the NASA data sets. For distinct values it should be ensured that each feature contains more than one, as otherwise this indicates a constant attribute. Constant attributes are of no use to a classifier for discriminating against instances during training, and should typically be removed as described in Section 6.1.2.2. Domain expertise is invaluable when analysing the general properties of attributes, as it is important to clarify that each attribute describes what it is thought to describe. This is something that has previously been, but should not be taken for granted. A good example was mentioned in Section 6.1.1, where some former PROMISE data sets had a value of 1.1 in an attribute supposedly describing a LOC count. Such a clearly erroneous attribute value indicates that its corresponding data point needs to be validated (where possible) and/or removed.

In addition to analysing features individually, correlations between features should also be analysed. As discussed in sections 6.1.2.3 & 6.1.2.7, repeated attributes, cases where two or more features have the same value for each instance, are typically either problematic or simply a waste of computational resources. A simple way to address such attributes is to remove all duplicates, thus ensuring the feature is only being represented once. Other, less trivial correlations between attributes may exist in the data. Section 6.1.2.5 gave examples of integrity checks that can be carried out for each data point to ensure that attribute relationships hold. For example, if attribute A should be equal to the sum of attributes B and C, then checking this is a simple method of trying to discover erroneous data. Domain expertise may illuminate other potential checks that can be made whereby certain combinations of features should be impossible. The more of these integrity checks there are the better. Lastly, as described in Section 6.1.2.7, highly correlated attributes can be harmful to classifier performance. Therefore, it may be that computed attributes such as those just mentioned, although initially useful for validating data integrity, are not always required when training a classifier.

As shown in Section 6.1, repeated data points can lead to serious problems in classification experiments; therefore, it is important to check if they are present in the data. If they are, then the first task is to try and determine their provenance, that is, whether or not there really are multiple quantities of modules with the same metrics in the software system. This is important as the repeated data points could have originated from a problem in the data construction process. Validating the data should be a trivial task if the originating data sources are accessible; however, if this is not the case, then such validation is typically impossible. If primary data is being used and the repeated data points are found to be genuine, then this may indicate that more metrics should be recorded for each module, so that they are potentially better differentiated. If secondary data is being used and it is not possible to validate the source code, then a decision must be made as to whether or not the repeats are genuine. This decision should be made in light of the repeated data point's features; for example, many repeats of a module with 2 LOC may be more plausible than many repeats of a module with 200 LOC. If the repeated data points are assumed to be noise (incorrect data) then the method of removing them (along with inconsistent instances) described in Section 6.1.2.6 is a viable approach. However, if the repeated data points are believed to be genuine, then a different approach is required. Details of such an approach will be given in Section 8.3.1.

During data analysis, is it important to be aware of the class distribution of the data. The background to this was given in sections 3.4 & 6.1.2.7. Perhaps the biggest problem presented by heavily imbalanced data is that if there is only a scarce amount of data available for a class, then a classifier may struggle to learn from it. Although the class distribution is something to be aware of, note that at this stage it is typically unsuitable to carry out any form of sampling to address the class imbalance problem. This is because such sampling, if performed at this time, could potentially lead to the assumption of unseen data being broken later on (see Chapter 5).

Data visualisation is a valuable tool when trying to get a 'feel for' your data. In Section 5.2, principal components analysis was used to show the two main principal components of a pre-processed NASA data set (Figure 5.2). PCA was extremely valuable in this case as it showed there to be little correlation between the features and the class label (other than for a few outliers). This finding was then confirmed by analysing an SVM classifier trained on the data, as it built a very complex model (one with many support vectors). Data visualisation techniques also have the potential to enable the discovery of natural clusters in the data. For these reasons, they should be used as a matter of course during data analysis.

Data analysis may indicate that data cleansing is required, in which case the data cleansing process defined in Section 6.1 may be a good starting point. Data cleansing may be necessary both to remove noise (erroneous data) and to prepare the data sets for experimental use. It is worth noting that it may not be sufficient to carry out a single data cleansing process prior to experimentation. It may be that, during a 10-fold cross-validation experiment for example, each of the 10 training sets will need to be individually pre-processed. This may be because the size of each training subset is smaller than the size of the full data set, thus there may be issues in a training subset that did not exist in the full data set. For example, a constant attribute may appear in a training subset if there were only a few distinct values for that attribute in the original data set to begin with. Also, as described in the next section, repeated data points may require pre-processing in each training subset.

8.2.1 Summary

To summarise this section, when analysing and cleansing fault data:

- Ensure a substantial data analysis is undertaken prior to experimentation.
- Discover where the data came from and its intended purpose. If details of a prior analysis are available then this should be repeated and validated.
- Analyse each attribute individually; include in the recorded statistics the number and proportion of both missing and distinct values. It is important to try and verify that each attribute describes what it is thought to describe.
- Analyse correlations between attributes; start by looking for redundant attributes, then validate relationships between attributes to try and discover erroneous data. Look for any other highly correlated features.
- Search for repeated and inconsistent data points; if they are present in the data then they need to be addressed. If such data points are believed to be noise (incorrect data) then they should be removed (as described in Section 6.1.2.6), whereas if they are believed to be genuine then the new approach proposed in Section 8.3.1 may be suitable.
- Be aware of the class distribution of the data; having a heavily imbalanced data set could lead to problems when trying to build a suitable classifier.
- Utilise a data visualisation technique (such as PCA) to get a ‘feel for’ the data.
- Carry out any data cleansing that has been identified as necessary during the prior stages of analysis. The data cleansing process defined in Section 6.1 may be a good starting point.

8.3 The Process of Defect Prediction

This section is concerned with the process of defect prediction: the construction and use of classifiers that are intended to predict the presence of defects in future code units. As discussed in Chapter 3, when trying to obtain an estimate of potential real-world predictive performance, it is crucial for classifiers to be tested on unseen data, data that was not used during model construction. Therefore, the original data sample should typically be re-sampled into one or more training and testing set pairs. Some of the methods of doing this were described in Section 3.3.1, a good method would usually be stratified 10-fold cross-validation. As discussed in the previous section, each training set may require its own pre-processing. It is often a necessity that this pre-processing is carried out on only the training data, as otherwise the assumption of unseen data could be broken (see chapters 4 & 5). For example, if a high-level of class imbalance was identified during data analysis, it may be that an undersampling and/or oversampling technique should be utilised on only the training data (see Section 3.4). Also, as will be shown in the next section, repeated data points may require addressing in only the training data.

When trying to produce as good a classification model as possible, it is important to carry out a model optimisation phase (see Section 3.5). Note that this is not always possible for some simple classifiers, and that there are other, more sophisticated classification methods (random forests for example) that can perform competitively without such tuning; these methods typically do not require an additional, explicit model optimisation phase, although one can often be carried out if desired. Nevertheless, for many classifiers (such as SVMs) an explicit model optimisation phase is often a prerequisite for obtaining a satisfactory classification model. The pseudo-code for a typical, full classification experiment which includes model optimisation was given in Figure 3.6. Note that during this process it is imperative for the test data to remain entirely unseen. Also note that it is good practice for a coarse parameter search to be undertaken to begin with, and then a finer search on the area identified as best by the coarse search [Hsu 2003].

8.3.1 How to address the issues caused by repeated data points

As already mentioned, the simplest way to address the issues caused by repeated data points is to discard them as part of the contextual data cleansing process, making each consistent data point unique (for details see Section 6.1.2.6). However, carrying out such a pre-processing step may not be the best approach, as repeated data points may occur in the real world. For this reason, I propose the following method for addressing the issues caused by repeated data points in artificial (as opposed to real-world) classification experiments. This method was first proposed in an EASE journal paper [Gray 2012], which was an extended version of the conference paper described in Section 6.1. The full journal paper can be found in Appendix E.

1. After the initial divide into training and testing set, discard all training data points with feature vectors common to the testing set. This ensures that performance is measured on unseen data, while the test set remains unmodified.

2. If the class distribution of the training set is adversely altered as a consequence of step 1, consider sampling techniques (see [He 2009, Chawla 2002]) to help maintain the original (or a more balanced) distribution. If oversampling is required, I recommend the synthetic minority oversampling technique (SMOTE [Chawla 2002]) rather than oversampling by duplication, as it reduces the likelihood of overfitting [Chawla 2002, Chawla 2003, Cieslak 2006].

3. During model optimisation/tuning (if performed), remove all duplicates from the validation set, as recommended by [Kołcz 2003]. Next, discard all validation set data points with feature vectors common to the corresponding training set (the training subset). If the class distribution of the validation set is adversely altered, consider the use of sampling techniques, as described in step 2. The purpose of this step is to help avoid overfitting.

This proposed approach is most suitable when researchers believe the repeated data points to be genuine, not noise. This is because test sets remain unmodified. With the simple approach of removing all repeated data points during data cleansing, test sets are indirectly modified before the data separation process has occurred. Note that a possible addition to the proposed approach is to remove all duplicates from the training set, to further reduce the possibility of overfitting. The best place for this to occur would be in-between steps 1 and 2.

8.3.2 Summary

To summarise this section, during the process of defect prediction:

- Help ensure that classifiers will be tested against unseen data by partitioning the data into one or more training and testing set pairs. Stratified 10-fold cross-validation is typically a good method of doing this (see Section 3.3.1).

- Carry out a model optimisation phase if suitable to do so based on the classification method being used (see Section 3.5). Good practice is for a coarse parameter search to be undertaken to begin with, and then a finer search on the area identified as best by the coarse search [Hsu 2003].

- As discussed in the previous section, if there are genuine repeated data points present in the data that cannot be addressed by gathering more features (to better discriminate them), then a suitable approach is described in Section 8.3.1. This approach keeps test sets unmodified, while preventing any contamination of training and testing data.

8.4 Quantifying Predictive Performance

To assess the predictive performance of classifiers, that is, how accurately they can predict the labels of previously unseen data points, several factors should be taken into account. The required background to this topic was given in Section 3.3. Firstly, when describing classifier performance results of any kind, it is imperative to present the characteristics of the data being used: where it came from, what pre-processing has been carried out, how many data points and features are present, what is the class distribution, etc. This will give those interpreting the results a basic overview of the data, and allow them to independently validate the consistency of the results (as done in [Zhang 2007, Gray 2011a, Bowes 2012]).

As demonstrated in Section 6.2, if the class distribution of the data (in a binary-classification context) means that less than around 5 to 10 percent of data points comprise the minority class, then the recall and precision achieved on the minority class should be reported. These measures should either be reported directly or via the area under the PR curve (AUC-PR). As class distributions become more balanced, it becomes more permissible to use other measures, such as to report the false positive rate instead of precision. Personally, I find precision to be the more immediately informative of these two measures, and would typically still choose to report it (along with recall) in my headline figures. Note that in cases where it is reasonably practical, it is worthwhile to report other performance measures in addition to recall and precision, such as the false positive rate and the correct classification rate (accuracy). The main reason for this is that it is helpful to have access to such measures when validating the consistency of results. In fact, the best way to report results that can be easily externally validated is to report the full confusion matrix (or an averaged approximation of it).

Regardless of the class distribution of the data, one measure that should always be examined is Matthew's Correlation Coefficient (MCC, see Equation 6.2 and Section 6.3). MCC takes all four components of the confusion matrix into account, and yields a value between -1 and $+1$. Perhaps the most attractive feature of this metric is that a value of 0 represents classification that is no better than random. Therefore, an initial benchmark regarding the worth of a classifier is to ensure its performance yields an MCC greater than 0.

Although estimating the real-world benefit of a classification system (perhaps by using some sort of cost-benefit analysis) is a very challenging task, and one that is beyond the scope of this dissertation, simple performance baselines (such as the one involving MCC just described) are useful in aiding to quantify predictive performance. A second simple baseline, this time involving recall and precision, is to determine the f-score (often defined as the harmonic mean of recall and precision) that would be achieved by a classifier that always makes positive-class predictions. Such a classifier would therefore achieve a recall of 1, and a precision equal to the proportion of positive class data points in the test data. This baseline f-score can be calculated using Equation 6.3. A predictor that is of any potential worth should achieve a real f-score that is greater than the baseline f-score.

There are other simple performance baselines that can be used to give indication toward the worth of a classifier, such as comparing the performance achieved with the performance that would be achieved by a very simple predictor. An example of this was given in Section 6.2.2.5, where, firstly, the average number of positive-class predictions made by a classifier during a 10-fold cross-validation experiment was determined. This number is denoted n. A predictor was then built which ranked the test data on a single attribute (in this case *LOC total* in descending order) and predicted the first n data points as being in the positive (defective) class. It turned out that, for 3 out of the 8 data sets used in [Menzies 2007b], this simple predictor achieved very similar performance to the actual (naïve Bayes) predictors used. This is known as *LOC module-order modelling* (see [Khoshgoftaar 2003] & [Mende 2009]), and in this case highlights both the poor actual predictive performance of the classifiers, and that research into making defect predictors 'effort aware' is worthwhile (see [Arisholm 2007] & [Mende 2010]). An 'effort aware' defect predictor is one that takes into consideration that predicting a module with 1000 lines of code will typically result in more required effort to manually examine than predicting a module with 10 lines of code. Furthermore, it may be more cost-effective to predict many small modules than few large modules. The worth of comparing performance with that of simple classifiers was highlighted many years ago in the machine learning community [Holte 1993].

8.4.1 Summary

To summarise this section, when quantifying predictive performance:

- Ensure the characteristics of the data are available; these include: origin, size, class distribution, and details of any pre-processing that has been carried out.

- For binary-classification tasks, report the recall and precision of the positive class. Where reasonably practical it is also worthwhile reporting other performance measures (such as the false positive rate), or the full confusion matrix.

- Compare the performance achieved against simple baselines. Firstly, it should be ensured the MCC (Matthew's Correlation Coefficient) is greater than 0. Secondly, the f-score achieved should be greater than the f-score that would be achieved by a predictor that makes only positive-class predictions. Lastly, it is worthwhile to compare performance with that of very simple classifiers. An example of such a classifier is one that makes judgements based solely on a linear ranking of a single attribute (such as *LOC total*).

Chapter 9

Conclusions

Contents

This is the concluding chapter of the dissertation; it contains a summary of each of the preceding chapters, and my claimed contributions to knowledge. After this, areas of future work that follow on from the dissertation are examined. A closing discussion is given in Section 9.4, followed by a list of publications in Section 9.5. Finally, there is a short note on personal reflection in Section 9.6.

9.1 Summary of Chapters

- **Chapter 1** gave an introduction to software defect prediction and an outline of this dissertation.

- **Chapter 2** gave the required background to software metrics and defects, with particular focus on static code metrics. An introduction to collecting fault data was also given, a subject revisited in Chapter 7.

- **Chapter 3** gave the required background to machine learning, with particular focus on supervised learning, or learning by example. Support vector machines were introduced in this chapter, as they are the most well-utilised classifier in my experiments.

- **Chapter 4** gave summaries of the most relevant publications to this dissertation; major methodological shortcomings in some of these studies were highlighted. These shortcomings included the breaking of the assumption of unseen data, and the reporting of grossly misleading performance figures.

- **Chapter 5** gave descriptions of the first two major experiments carried out during my PhD study. Both of these experiments utilised support vector machine classifiers. The first experiment was to obtain an estimate of current, state-of-the-art predictive performance; the second experiment was to explore the inner workings of these sophisticated classifiers. The experience gained during these early experiments was substantial, and a key influence on the methodology proposed in Chapter 8.

- **Chapter 6** gave details of the issues with the NASA MDP data sets that were discovered during the experiments described in Chapter 5. A data cleansing protocol to address these issues was presented, and the main problems caused by repeated data points in classification experiments discussed. Following on from this, the topic of classifier performance measures was revisited. Practical examples were given demonstrating why *precision* is often required to quantify predictive performance effectively in this domain, and how omitting this measure can lead to misleading results. Lastly, the results from a similar classification experiment to the one described in Chapter 5 were presented. This experiment addressed all of the main issues discussed in both chapters 5 & 6.

- **Chapter 7** gave details of a study where the aim was to construct a new fault data set from an open-source system. The motivation for this study came from there being so many data quality issues with the well-used NASA MDP data sets. The findings indicated that constructing fault data is a very difficult and intricate task. This was mainly due to the problems encountered while categorising whether or not a past revision contains a fault-fix. It appears that for this task to be completed successfully, a strict definition of precisely what constitutes a fault must be made, documented, and consistently adhered to. If this is not the case, it will be difficult (often impossible) to know what the dependent variable in the new data actually describes. As no such documentation is available for the NASA MDP data sets (and most other public-domain data sets), it follows that they are of limited value, and that the findings from results based on them should be treated with caution.

- **Chapter 8** gave details of the new proposed methodology for software defect prediction. This methodology was mainly a synthesis of points raised earlier in the dissertation, although it also included a novel approach to dealing with genuine repeated data points in classification experiments.

9.2 Contributions to Knowledge

The contributions to knowledge I have made in this dissertation include:

- **Highlighting the issues with much prior work.** In chapters 4, 5 & 6, I have identified many methodological problems in much previous work (including my own). This includes the incorrect application of data mining techniques, and the misleading reporting of classifier performance. Such methodological mistakes threaten the validity of the findings presented in studies, often rendering them void. In this domain, the quantity and severity of previous shortcomings is damning.

- **Highlighting the issues with the NASA MDP data sets.** As shown in Section 6.1, these well-utilised data sets contain many quality problems. The most severe of these problems is that of the repeated data points. This is an issue that is not specific to the NASA data sets, and one that requires further research, particularly in the context of this research area. A data cleansing process for the NASA data sets was presented in Section 6.1.2; this may also be a good starting point when using other software fault data sets. In addition to these low-level data quality issues, the lack of documentation available for the NASA data sets is another major concern. Chapter 7 demonstrated how difficult the construction of fault data can be, and how important it is for the process to be properly documented. Without such documentation, it can be difficult (perhaps impossible) to determine what the dependent variable actually describes. The current lack of readily available, reasonable quality data is another considerable issue in this domain.

- **Proposing a new methodology for software defect prediction.** The methodology was proposed in Chapter 8, and includes details regarding: obtaining fault data, data analysis and cleansing, the process of constructing defect prediction models, and quantifying predictive performance. The motivation for the methodology, the aim of which is to be able to obtain a realistic estimate of potential real-world predictive performance, came from there being so many methodological mistakes made in prior research (see the first contribution). The methodology is centred around data analysis, as this seems to have been largely ignored in the past, with serious repercussions (see the second contribution).

9.3 Future Work

A clear direction of future study opened-up by the work described in this dissertation is to further analyse the effect that repeated data points can have during classification experiments. As discussed in Chapter 6, a good starting point would be to analyse other software fault data sets, to see whether the proportions of repeated data points in the NASA data sets are typical of fault data sets in general. This will help to determine the extent of the problem, and provide some (albeit limited) indication as to the likelihood of the rates of duplication in the NASA data sets being legitimate. A different direction to explore would be to test the new approach of addressing the issues caused by repeated data points in classification experiments (proposed in Section 8.3.1). This approach is yet to be subject of independent scrutiny, and will almost certainly require improving and refining. In fact, the same can be said for the entire methodology proposed in Chapter 8.

A challenging but highly fruitful task that leads on mainly from the work described in Chapter 7 is to construct some accurate software fault data from an open-source system. The main benefit of using such a system is that the source code for each module will be readily available, leading to transparency and enabling the external validation of the data. Also, the availability of the source code would make it possible to extract additional metrics at a later date if desired. As detailed in chapters 7 & 8, it would be extremely beneficial to have access to individuals who know the code well. This is because it is often extremely difficult to comprehend the reasons behind code modifications if the corresponding commit message (in the revision control archive) is sparse and/or complex. Therefore, a direct working relationship with the system's developers would be highly beneficial. Lastly, recall that the importance of documenting a detailed definition of precisely what constitutes a fault cannot be overstressed (see chapters 7 & 8).

9.4 Discussion

The findings presented in this dissertation demonstrate current software defect prediction research to be plagued by technical shortcomings and data quality issues. Such methodological problems are extremely serious, as they threaten the validity of studies. Moreover, I believe the number of prior studies that have contained technical failings, when coupled with the typically appalling quality of the data sets used in this area, mean that serious questions must now be asked. These questions are much bigger than simply whether or not individual studies are at fault; they relate to the current body of knowledge as it stands, and whether or not we really know what we think we know.

Interestingly, some of the findings in this dissertation are echoes of the past. Over 13 years ago in 1999, Fenton & Neil carried out a critique of defect prediction research [Fenton 1999]. In their conclusions they state that "many methodological and theoretical mistakes have been made. Many past studies have suffered from a variety of flaws from model misspecification to use of inappropriate data." It appears then that little has changed in the last 13 years. Regarding the many methodological mistakes, there have been many concrete examples presented in chapters 4, 5 & 6. The most persistent offence is the violation of the assumption of unseen data, whether that be by test set contamination or otherwise. Regarding the use of inappropriate data, recall the study of Liebchen & Shepperd (described in Section 4.4) where an SLR of empirical software engineering studies spanning from 1995 to early 2008 was carried out [Liebchen 2008]. The focus of the SLR was data quality, and the findings indicated this as a topic of little interest in the software engineering community. An interesting point raised in the study is that the community "may be wasting research effort on data sets that contain such levels of noise as to prevent meaningful conclusions." In this dissertation the case study was the well-used 13 NASA data sets, some of which, especially in unprocessed form, I do believe to contain such levels of noise. Note that it could be reasonably argued that these noise levels mean that some of the data sets should be rejected outright, with no cleansing attempt made. For example, the original data set MC1 (according to the metrics) contains 4841 modules (51% of modules in total) with no lines of code.

I believe repeated data points to be highly problematic in classification experiments, especially in this domain. The experiments described in Section 6.1 showed the effect that test set contamination can have on classifier performance, which provided empirical evidence to confirm the well known, intuitive and obvious fact that training and testing set pairs must be completely disjunct to prevent potentially unrealistic performance results. In cases where there are genuine repeated occurrences of the artefact being modelled in the data, which cannot be easily distinguished by adding extra, directly relevant features, then there are two suitable approaches. The first approach is better suited to artificial classification experiments (as opposed to real-world use), and is to use the method described in Section 8.3.1. This method keeps test sets unmodified and prevents test set contamination, thus helping ensure test sets are representative while at the same time guaranteeing that classifiers are tested against unseen data. The second approach is better suited to real-world classifier use, and is to have a distinction in the reported performance figures between performance achieved on seen data (data used during model construction) and performance achieved on unseen (novel) data. This ensures that the performance figures appropriately describe the extent to which the classifier has learnt and is successfully generalising. Clearly, in the real world the test set labels would not be readily available following classification (and in fact may never be discovered); however, the proportion of seen feature vectors would be immediately determinable, and would describe the extent to which the classifier was being used to predict novel input patterns.

Recall that repeated feature vectors may be the artefact of too few features (attributes) being recorded for each occurrence in the data. Therefore, a straightforward way to avoid these complications may be, where possible, to simply record more features. This should hopefully lead to a reduction in the quantity of repeated feature vectors, as each occurrence may become more distinguishable. When constructing a new data set it is important to keep in mind the proportion of unique feature vectors, as ideally the feature set chosen should be able to uniquely identify each point. Note that although this is the ideal case it may not always be possible. Also note that in some domains (this one included) it is possible for the same feature vector to occur in more than one class (i.e. for there to be inconsistent data points).

An observation regarding repeated data points that has not been previously mentioned and is specific to this domain is related to fault data that is collected at different versions during the lifetime of a software system. Imagine an example project P where fault data has been recorded for versions 3.0 and 3.1. In such an example, it may be tempting to train a classifier on version 3.0 and test it on version 3.1. However, there may be many code units that have remained unchanged between these two versions, and as such will share identical metrics. These identical metrics are likely to yield repeated feature vectors in the data set for each version, potentially leading to the problems described in Section 6.1. In such a situation, it could be that the predictive performance appears impressive from the raw performance figures, when in fact the high performance is merely an artefact of the similarity between the training data (version 3.0) and the testing data (version 3.1).

9.5 Publications

During my PhD study I have been the lead author of 4 conference papers [Gray 2009, Gray 2010, Gray 2011a, Gray 2011b] and a journal paper [Gray 2012]. Each of these are presented in full in the appendix. The first two conference papers were discussed in Chapter 5, the two remaining were discussed in Chapter 6. The journal paper was an extended version of one of the conference papers [Gray 2011b], and was discussed in chapters 6 & 8. I have also been the co-author of 2 conference papers [Sun 2010, Bowes 2012] and 2 journal papers [Hall 2012, Hall 2011]. The latter of the conference papers was based on recomputing classifier performance measures that have been omitted from studies; it won the best paper award at PROMISE 2012, and more formally proved the main shortcomings in [Elish 2008] (described in Section 4.3). The journal papers were based on a systematic literature review (SLR) of fault prediction performance, as described in Section 4.5.

9.6 Personal Reflection

I was first introduced to defect prediction by David Bowes while studying the highly enjoyable Software Engineering Practice & Experience module on my MSc at the University of Hertfordshire. Having been handed what then seemed a dauntingly complex paper, I set about trying defect prediction for myself, knowing almost nothing about it, and far less about machine learning. With the help of my excellent supervision team (and notable others), the distance I have come on since then is profound. For me, the biggest highlight (other than successful completion) is that after many paper rejections and negative reviews, much perseverance (and the stubbornness of a mule) resulted in the research community finally beginning to pay attention to the data quality issues I had first discovered back in 2008. Although others had discovered several of these issues before, I believe I was the first to realise their potential severity, and to want to publicise this in the pursuit of better science. Data quality should never be underestimated in an empirical study; as the classic saying goes: garbage in, garbage out. My main hope now is that this dissertation will lead to a better standard of work in this area.

Bibliography

[Arisholm 2007] Erik Arisholm, Lionel C. Briand and Magnus Fuglerud. *Data Mining Techniques for Building Fault-proneness Models in Telecom Java Software.* In ISSRE '07: Proceedings of the The 18th IEEE International Symposium on Software Reliability, pages 215–224, Washington, DC, USA, 2007. IEEE Computer Society. (Cited on pages 88, 94 and 113.)

[Batista 2004] Gustavo E. A. P. A. Batista, Ronaldo C. Prati and Maria Carolina Monard. *A study of the behavior of several methods for balancing machine learning training data.* SIGKDD Explor. Newsl., vol. 6, pages 20–29, June 2004. (Cited on pages 74 and 91.)

[Bezerra 2007] M.E.R. Bezerra, A.L.I. Oliveira and S.R.L. Meira. *A Constructive RBF Neural Network for Estimating the Probability of Defects in Software Modules.* In Neural Networks, 2007. IJCNN 2007. International Joint Conference on, pages 2869–2874, Aug. 2007. (Cited on pages 65, 67 and 70.)

[Boetticher 2006] G.D. Boetticher. *Improving Credibility of Machine Learner Models in Software Engineering.* In Advanced Machine Learner Applications in Software Engineering, Software Engineering and Knowledge Engineering, pages 52–72, 2006. (Cited on pages 65, 67, 76, 77 and 82.)

[Boser 1992] Bernhard E. Boser, Isabelle M. Guyon and Vladimir N. Vapnik. *A Training Algorithm for Optimal Margin Classifiers.* In Proceedings of the 5th Annual ACM Workshop on Computational Learning Theory, pages 144–152. ACM Press, 1992. (Cited on page 38.)

[Bottou 2007] Léon Bottou and Chih-Jen Lin. *Support Vector Machine Solvers.* In Léon Bottou, Olivier Chapelle, Dennis decoste and Jason Weston, editeurs, Large Scale Kernel Machines, pages 301–320, Cambridge, MA, USA, 2007. MIT Press. (Cited on page 54.)

[Bowes 2012] David Bowes, Tracy Hall and David Gray. *Comparing the performance of fault prediction models which report multiple performance measures: re-computing the confusion matrix.* In PROMISE '12: Proceedings of the 8th International Conference on Predictive Models in Software Engineering, pages 109–118, New York, NY, USA, 2012. ACM. (Cited on pages 46, 90, 112 and 120.)

[Bradford 2005] James R. Bradford and David R. Westhead. *Improved prediction of protein-protein binding sites using a support vector machines approach.* Bioinformatics, vol. 21, no. 8, pages 1487–1494, April 2005. (Cited on page 38.)

[Breiman 1996] Leo Breiman. *Bagging predictors.* Mach. Learn., vol. 24, pages 123–140, August 1996. (Cited on page 22.)

[Breiman 2001] Leo Breiman. *Random Forests.* Mach. Learn., vol. 45, pages 5–32, October 2001. (Cited on page 22.)

[Catal 2007] Cagatay Catal, Banu Diri and Bulent Ozumut. *An Artificial Immune System Approach for Fault Prediction in Object-Oriented Software.* In DEPCOS-RELCOMEX '07: Proceedings of the 2nd International Conference on Dependability of Computer Systems, pages 238–245, Washington, DC, USA, 2007. IEEE Computer Society. (Cited on page 8.)

[Ceriani 2011] Lidia Ceriani and Paolo Verme. *The origins of the Gini index: extracts from Variabilità e Mutabilità (1912) by Corrado Gini.* Journal of Economic Inequality, pages 1–23, 2011. 10.1007/s10888-011-9188-x. (Cited on page 20.)

[Challagulla 2005] Venkata U. B. Challagulla, Farokh B. Bastani, I-Ling Yen and Raymond A. Paul. *Empirical Assessment of Machine Learning based Software Defect Prediction Techniques.* In WORDS '05: Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, pages 263–270, Washington, DC, USA, 2005. IEEE Computer Society. (Cited on page 65.)

[Challagulla 2006] Venkata U. B. Challagulla, Farokh B. Bastani and I-Ling Yen. *A Unified Framework for Defect Data Analysis Using the MBR Technique.* In ICTAI '06: Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence, pages 39–46, Washington, DC, USA, 2006. IEEE Computer Society. (Cited on page 65.)

[Chang 2001] Chih-Chung Chang and Chih-Jen Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`. (Cited on page 51.)

[Chawla 2002] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall and W. Philip Kegelmeyer. *SMOTE: Synthetic Minority Over-sampling Technique.* Journal of Artificial Intelligence Research, vol. 16, pages 321–357, 2002. (Cited on page 111.)

[Chawla 2003] Nitesh V. Chawla, Aleksandar Lazarevic, Lawrence O. Hall and Kevin W. Bowyer. *SMOTEBoost: Improving Prediction of the Minority Class in Boosting.* In In Proceedings of the Principles of Knowledge Discovery in Databases, PKDD-2003, pages 107–119. Springer, 2003. (Cited on page 111.)

[Chawla 2004] Nitesh V. Chawla, Nathalie Japkowicz and Aleksander Kolcz. *Special issue on learning from imbalanced datasets.* SIGKDD Explor. Newsl., vol. 6, no. 1, pages 1–6, 2004. (Cited on pages 32, 67 and 74.)

[Chen 2004] Chao Chen, Andy Liaw and Leo Breiman. *Using Random Forest to Learn Imbalanced Data.* Rapport technique, Department of Statistics, University of California, Berkeley. Technical Report 666, 2004. (Cited on page 80.)

[Chidamber 1994] Shyam R. Chidamber and Chris F. Kemerer. *A metrics suite for object-oriented design.* IEEE Computer Society Trans. Software Engineering, vol. 20, no. 6, pages 476–493, June 1994. (Cited on pages 8 and 49.)

[Cieslak 2006] D.A. Cieslak, N.V. Chawla and A. Striegel. *Combating imbalance in network intrusion datasets.* In Granular Computing, 2006 IEEE International Conference on, pages 732–737, may 2006. (Cited on page 111.)

[Cong 2010] Jin Cong, Dong En-Mei and Qin Li-Na. *Software Fault Prediction Model Based on Adaptive Dynamical and Median Particle Swarm Optimization.* In Multimedia and Information Technology (MMIT), 2010 Second International Conference on, volume 1, pages 44–47, 2010. (Cited on page 65.)

[Cortes 1995] Corinna Cortes and Vladimir Vapnik. *Support-Vector Networks.* Machine Learning, vol. 20, pages 273–297, 1995. (Cited on page 38.)

[Davis 2006] Jesse Davis and Mark Goadrich. *The relationship between Precision-Recall and ROC curves.* In Proceedings of the 23rd international conference on Machine learning, ICML '06, pages 233–240, New York, NY, USA, 2006. ACM. (Cited on pages 74, 85, 90 and 91.)

[de Carvalho 2010] AndrÃĺ B. de Carvalho, Aurora Pozo and Silvia Regina Vergilio. *A symbolic fault-prediction model based on multiobjective particle swarm optimization.* Journal of Systems and Software, vol. 83, no. 5, pages 868–882, 2010. (Cited on page 65.)

[DeMarco 1986] T. DeMarco. Controlling software projects: Management, measurement, and estimates. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1986. (Cited on page 5.)

[Demšar 2006] Janez Demšar. *Statistical Comparisons of Classifiers over Multiple Data Sets.* J. Mach. Learn. Res., vol. 7, pages 1–30, December 2006. (Cited on pages 44 and 90.)

[Dijkstra 1968] Edsger W. Dijkstra. *Go To statement considered harmful.* Comm. ACM, vol. 11, no. 3, pages 147–148, 1968. letter to the Editor. (Cited on page 58.)

[Donoho 2000] D. L. Donoho. *High-dimensional data analysis: the curses and blessings of dimensionality.* In American Mathematical Society Conf. Math Challenges of the 21st Century. 2000. (Cited on page 106.)

[Dudoit 2003] Sandrine Dudoit and Jane Fridlyand. *Classification in microarray experiments*. In Statistical Analysis of Gene Expression Microarray Data. Chapman and Hall/CRC, 2003. (Cited on page 80.)

[Elish 2008] Karim O. Elish and Mahmoud O. Elish. *Predicting defect-prone software modules using support vector machines*. J. Syst. Softw., vol. 81, no. 5, pages 649–660, 2008. (Cited on pages 10, 41, 45, 49, 65, 68 and 120.)

[Fenton 1998] Norman E. Fenton and Shari Lawrence Pfleeger. Software metrics: A rigorous and practical approach. PWS Publishing Co., Boston, MA, USA, 2nd édition, 1998. (Cited on pages 6, 7 and 8.)

[Fenton 1999] Norman E. Fenton and Martin Neil. *A Critique of Software Defect Prediction Models*. IEEE Trans. Softw. Eng., vol. 25, pages 675–689, September 1999. (Cited on pages 9 and 119.)

[Flach 2003] P.A. Flach and S. Wu. *Repairing concavities in ROC curves*. In Proc. 2003 UK Workshop on Computational Intelligence, pages 38–44. University of Bristol, August 2003. (Cited on page 29.)

[Forman 2010] George Forman and Martin Scholz. *Apples-to-apples in cross-validation studies: pitfalls in classifier performance measurement*. SIGKDD Explorations, vol. 12, no. 1, pages 49–57, 2010. (Cited on pages 25 and 35.)

[Fowler 1999] Martin Fowler. Refactoring: Improving the design of existing code. Addison Wesley, Boston, MA, USA, 1999. (Cited on page 60.)

[Gray 2009] David Gray, David Bowes, Neil Davey, Yi Sun and Bruce Christianson. *Using the Support Vector Machine as a Classification Method for Software Defect Prediction with Static Code Metrics*. In Dominic Palmer-Brown, Chrisina Draganova, Elias Pimenidis and Haris Mouratidis, editeurs, Engineering Applications of Neural Networks, volume 43 of *Communications in Computer and Information Science*, pages 223–234. Springer Berlin Heidelberg, 2009. (Cited on pages 51 and 120.)

[Gray 2010] David Gray, David Bowes, Neil Davey, Yi Sun and Bruce Christianson. *Software defect prediction using static code metrics underestimates defect-proneness*. In International Joint Conference on Neural Networks (IJCNN), Barcelona, Spain, pages 1–7. IEEE, 2010. (Cited on pages 51 and 120.)

[Gray 2011a] David Gray, David Bowes, Neil Davey, Yi Sun and Bruce Christianson. *Further Thoughts on Precision*. In Evaluation and Assessment in Software Engineering (EASE), pages 129–133, 2011. (Cited on pages 64, 74, 112 and 120.)

[Gray 2011b] David Gray, David Bowes, Neil Davey, Yi Sun and Bruce Christianson. *The Misuse of the Nasa Metrics Data Program Data Sets for Automated Software Defect Prediction*. In Evaluation and Assessment in Software Engineering (EASE), pages 96–103, 2011. (Cited on pages 64, 68 and 120.)

[Gray 2012] David Gray, David Bowes, Neil Davey, Yi Sun and Bruce Christianson. *Reflections on the NASA MDP data sets.* IET Software, vol. 6, no. 6, pages 549–558, 2012. (Cited on pages 110 and 120.)

[Guo 2003] L. Guo, B. Cukic and H. Singh. *Predicting fault prone modules by the Dempster-Shafer belief networks.* In Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on, pages 249–252, Oct. 2003. (Cited on page 65.)

[Guo 2004] L. Guo, Y. Ma, B. Cukic and Harshinder Singh. *Robust prediction of fault-proneness by random forests.* In Software Reliability Engineering. ISSRE 2004. 15th International Symposium on, pages 417–428, Nov. 2004. (Cited on pages 65, 79 and 80.)

[Guyon 2003] Isabelle Guyon and André Elisseeff. *An introduction to variable and feature selection.* J. Mach. Learn. Res., vol. 3, pages 1157–1182, March 2003. (Cited on page 79.)

[Hall 1999] Mark A. Hall. *Correlation-based Feature Subset Selection for Machine Learning.* PhD thesis, Department of Computer Science, University of Waikato, Hamilton, New Zealand, apr 1999. (Cited on page 74.)

[Hall 2010] Tracy Hall, David Bowes, Gernot Armin Liebchen and Paul Wernick. *Evaluating Three Approaches to Extracting Fault Data from Software Change Repositories.* In PROFES, pages 107–115, 2010. (Cited on pages 11 and 103.)

[Hall 2011] Tracy Hall, Sarah Beecham, David Bowes, David Gray and Steve Counsell. *Developing Fault-Prediction Models: What the Research Can Show Industry.* IEEE Software, vol. 28, no. 6, pages 96–99, 2011. (Cited on page 120.)

[Hall 2012] Tracy Hall, Sarah Beecham, David Bowes, David Gray and Steve Counsell. *A Systematic Literature Review on Fault Prediction Performance in Software Engineering.* Software Engineering, IEEE Transactions on, vol. 38, no. 6, pages 1276–1304, nov.-dec. 2012. (Cited on pages 8, 47, 50, 106 and 120.)

[Halstead 1977] Maurice H. Halstead. Elements of software science (operating and programming systems series). Elsevier Science Inc., New York, NY, USA, 1977. (Cited on pages 5, 6 and 71.)

[Hamer 1982] Peter G. Hamer and Gillian D. Frewin. *M.H. Halstead's Software Science - a critical examination.* In ICSE '82: Proceedings of the 6th international conference on Software engineering, pages 197–206, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press. (Cited on page 7.)

[He 2009] Haibo He and Edwardo A. Garcia. *Learning from Imbalanced Data.* IEEE Transactions on Knowledge and Data Engineering, vol. 21, pages 1263–1284, 2009. (Cited on pages 67, 74, 75, 91 and 111.)

[Holte 1993] Robert C. Holte. *Very simple classification rules perform well on most commonly used datasets.* In Machine Learning, pages 63–91, 1993. (Cited on page 113.)

[Howley 2006] Tom Howley, Michael G. Madden, Marie-Louise O'Connell and Alan G. Ryder. *The effect of principal component analysis on machine learning accuracy with high-dimensional spectral data.* Knowl.-Based Syst., vol. 19, no. 5, pages 363–370, 2006. (Cited on page 74.)

[Hsu 2003] C. W. Hsu, C. C. Chang and C. J. Lin. *A practical guide to support vector classification.* Rapport technique, Taipei, 2003. (Cited on pages 38, 110 and 111.)

[Huang 2003] J. Huang, J. Lu and C.X. Ling. *Comparing naive Bayes, decision trees, and SVM with AUC and accuracy.* In Data Mining, 2003. ICDM 2003. Third IEEE International Conference on, pages 553–556, nov. 2003. (Cited on page 23.)

[iee 1990] *IEEE Standard Glossary of Software Engineering Terminology.* IEEE Std 610.12-1990, pages 1–84, 1990. (Cited on page 9.)

[Jiang 2007] Yue Jiang, Bojan Cukic and Tim Menzies. *Fault Prediction using Early Lifecycle Data.* In Software Reliability, 2007. ISSRE '07. The 18th IEEE International Symposium on, pages 237–246, nov. 2007. (Cited on pages 65 and 80.)

[Jiang 2008a] Yue Jiang, Bojan Cukic and Yan Ma. *Techniques for evaluating fault prediction models.* Empirical Softw. Engg., vol. 13, no. 5, pages 561–595, 2008. (Cited on pages 65 and 85.)

[Jiang 2008b] Yue Jiang, Bojan Cukic and Tim Menzies. *Can data transformation help in the detection of fault-prone modules?* In DEFECTS '08: Proceedings of the 2008 workshop on Defects in large software systems, pages 16–20, New York, NY, USA, 2008. ACM. (Cited on pages 65, 79 and 80.)

[Jiang 2008c] Yue Jiang, Bojan Cukic, Tim Menzies and Nick Bartlow. *Comparing design and code metrics for software quality prediction.* In Proceedings of the 4th international workshop on Predictor models in software engineering, PROMISE '08, pages 11–18, New York, NY, USA, 2008. ACM. (Cited on pages 65, 79 and 80.)

[Jiang 2009] Yue Jiang and Bojan Cukic. *Misclassification cost-sensitive fault prediction models.* In Proceedings of the 5th International Conference on Predictor Models in Software Engineering, PROMISE '09, pages 20:1–20:10, New York, NY, USA, 2009. ACM. (Cited on pages 65 and 87.)

[Joachims 1998] Thorsten Joachims. *Text categorization with support vector machines: learning with many relevant features.* In Claire Nédellec and Céline Rouveirol, editeurs, Proceedings of ECML-98, 10th European Conference on Machine Learning, pages 137–142, Heidelberg et al., 1998. Springer. (Cited on page 38.)

[Jones 2008] Capers Jones. Applied Software Measurement: Global Analysis of Productivity and Quality. McGraw-Hill Osborne Media, 3 édition, April 2008. (Cited on page 6.)

[Kaminsky 2004] K. Kaminsky and G. Boetticher. *Building a genetically engineerable evolvable program (GEEP) using breadth-based explicit knowledge for predicting software defects.* In Fuzzy Information, 2004. Processing NAFIPS '04. IEEE Annual Meeting of the, pages 10–15 Vol.1, June 2004. (Cited on page 67.)

[Khoshgoftaar 2003] Taghi M. Khoshgoftaar and Edward B. Allen. *Ordering Fault-Prone Software Modules.* Software Quality Control, vol. 11, pages 19–37, May 2003. (Cited on pages 88 and 113.)

[Khoshgoftaar 2004] Taghi M. Khoshgoftaar and Naeem Seliya. *The Necessity of Assuring Quality in Software Measurement Data.* In METRICS '04: Proceedings of the Software Metrics, 10th International Symposium, pages 119–130, Washington, DC, USA, 2004. IEEE Computer Society. (Cited on pages 65 and 67.)

[Kim 2006] Sunghun Kim, Thomas Zimmermann, Kai Pan and E. James Jr. Whitehead. *Automatic Identification of Bug-Introducing Changes.* In ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering, pages 81–90, Washington, DC, USA, 2006. IEEE Computer Society. (Cited on pages 11 and 64.)

[Kim 2008] Sunghun Kim, Jr. E. James Whitehead and Yi Zhang. *Classifying Software Changes: Clean or Buggy?* IEEE Transactions on Software Engineering, vol. 34, no. 2, pages 181–196, March/April 2008. (Cited on page 64.)

[Kohavi 1995] Ron Kohavi. *A study of cross-validation and bootstrap for accuracy estimation and model selection.* In Proceedings of the 14th international joint conference on Artificial intelligence - Volume 2, pages 1137–1143, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc. (Cited on page 25.)

[Kołcz 2003] Aleksander Kołcz, Abdur Chowdhury and Joshua Alspector. *Data duplication: an imbalance problem?* In ICML 2003 Workshop on Learning from Imbalanced Datasets,, 2003. (Cited on pages 75, 81 and 111.)

[Koru 2005a] A.G. Koru and H. Liu. *Building effective defect-prediction models in practice.* Software, IEEE, vol. 22, no. 6, pages 23–29, Nov.-Dec. 2005. (Cited on page 65.)

[Koru 2005b] Akif Günes Koru and Hongfang Liu. *An investigation of the effect of module size on defect prediction using static measures.* ACM SIGSOFT Software Engineering Notes, vol. 30, no. 4, pages 1–5, 2005. (Cited on pages 65 and 77.)

[Kutlubay 2007] O. Kutlubay, B. Turhan and A.B. Bener. *A Two-Step Model for Defect Density Estimation.* In Software Engineering and Advanced Applications, 2007. 33rd EUROMICRO Conference on, pages 322–332, Aug. 2007. (Cited on page 65.)

[Lessmann 2008] Stefan Lessmann, Bart Baesens, Christophe Mues and Swantje Pietsch. *Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings.* Software Engineering, IEEE Transactions on, vol. 34, no. 4, pages 485–496, 2008. (Cited on pages 10, 41, 44, 45, 49, 65, 68, 87 and 90.)

[Levinson 2001] M. Levinson. *Let's Stop Wasting $78 Billion per Year.* CIO Magazine, 2001. (Cited on page 1.)

[Li 2007] Zhan Li and M. Reformat. *A practical method for the software fault-prediction.* Information Reuse and Integration. IRI 2007. IEEE International Conference on, pages 659–666, Aug. 2007. (Cited on page 65.)

[Liebchen 2008] Gernot A. Liebchen and Martin Shepperd. *Data sets and data quality in software engineering.* In PROMISE '08: Proceedings of the 4th international workshop on Predictor models in software engineering, pages 39–44, New York, NY, USA, 2008. ACM. (Cited on pages 41, 46, 49, 66 and 119.)

[Liu 2010] Yi Liu, T.M. Khoshgoftaar and N. Seliya. *Evolutionary Optimization of Software Quality Modeling with Multiple Repositories.* Software Engineering, IEEE Transactions on, vol. 36, no. 6, pages 852–864, nov.-dec. 2010. (Cited on page 65.)

[Lu 2012] Huihua Lu and Bojan Cukic. *An adaptive approach with active learning in software fault prediction.* In Proceedings of the 8th International Conference on Predictive Models in Software Engineering, PROMISE '12, pages 79–88, New York, NY, USA, 2012. ACM. (Cited on page 65.)

[Ma 2006] Yan Ma, Lan Guo and Bojan Cukic. *A Statistical framework for the prediction of fault-proneness.* In Advances in Machine Learning Application in Software Engineering, Idea Group Inc., pages 237–265 Vol.1, June 2006. (Cited on pages 45 and 65.)

[Matthews 1975] B.W. Matthews. *Comparison of the predicted and observed secondary structure of T4 phage lysozyme.* Biochimica et Biophysica Acta (BBA) - Protein Structure, vol. 405, no. 2, pages 442 – 451, 1975. (Cited on page 92.)

[McCabe 1976] Thomas J. McCabe. *A complexity measure.* In ICSE '76: Proceedings of the 2nd international conference on Software engineering, page 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press. (Cited on pages 8, 59 and 71.)

[Mende 2009] Thilo Mende and Rainer Koschke. *Revisiting the evaluation of defect prediction models.* In Proceedings of the 5th International Conference on Predictor Models in Software Engineering, PROMISE '09, pages 7:1–7:10, New York, NY, USA, 2009. ACM. (Cited on pages 65, 88 and 113.)

[Mende 2010] Thilo Mende and Rainer Koschke. *Effort-Aware Defect Prediction Models.* Software Maintenance and Reengineering, European Conference on, vol. 0, pages 107–116, 2010. (Cited on pages 65, 88, 94 and 113.)

[Menzies 2004a] Tim Menzies and Justin S. Di Stefano. *How good is your blind spot sampling policy?* In High Assurance Systems Engineering, 2004. Proceedings. Eighth IEEE International Symposium on, pages 129–138, march 2004. (Cited on page 65.)

[Menzies 2004b] Tim Menzies, Justin S. Di Stefano, A. Orrego and R. Chapman. *Assessing Predictors of Software Defects.* Proceedings of Workshop on Predictive Software Models, 2004. (Cited on pages 65 and 66.)

[Menzies 2007a] Tim Menzies, Alex Dekhtyar, Justin Distefano and Jeremy Greenwald. *Problems with Precision: A Response to "Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors"'.* IEEE Transactions on Software Engineering, vol. 33, pages 637–640, 2007. (Cited on pages 84, 87 and 90.)

[Menzies 2007b] Tim Menzies, Jeremy Greenwald and Art Frank. *Data Mining Static Code Attributes to Learn Defect Predictors.* Software Engineering, IEEE Transactions on, vol. 33, no. 1, pages 2–13, Jan. 2007. (Cited on pages 10, 41, 42, 43, 44, 45, 49, 65, 68, 77, 82, 84, 85, 87, 88, 90 and 113.)

[Menzies 2008] Tim Menzies, Burak Turhan, Ayse Bener, Gregory Gay, Bojan Cukic and Yue Jiang. *Implications of ceiling effects in defect predictors.* In Proceedings of the 4th international workshop on Predictor models in software engineering, PROMISE '08, pages 47–54, New York, NY, USA, 2008. ACM. (Cited on page 65.)

[Menzies 2010] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang and Ayşe Bener. *Defect prediction from static code features: current results, limitations, new approaches.* Automated Software Engg., vol. 17, no. 4, pages 375–407, 2010. (Cited on pages 65 and 91.)

[Mertik 2006] M. Mertik, M. Lenic, G. Stiglic and P. Kokol. *Estimating Software Quality with Advanced Data Mining Techniques.* In Software Engineering Advances, International Conference on, page 19, oct. 2006. (Cited on page 65.)

[Meyers 2007] Timothy M. Meyers and David Binkley. *An empirical study of slice-based cohesion and coupling metrics.* ACM Trans. Softw. Eng. Methodol., vol. 17, no. 1, pages 2:1–2:27, December 2007. (Cited on page 96.)

[Mockus 2000] Audris Mockus and Lawrence G. Votta. *Identifying Reasons for Software Changes Using Historic Databases.* In ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00), page 120, Washington, DC, USA, 2000. IEEE Computer Society. (Cited on page 10.)

[Nickerson 2001] Adam S. Nickerson, Nathalie Japkowicz and Evangelos Milios. *Using unsupervised learning to guide resampling in imbalanced data sets.* In In Proceedings of the Eighth International Workshop on AI and Statitsics, pages 261–265, 2001. (Cited on page 67.)

[Noyes 1992] James L. Noyes. Artificial intelligence with common lisp: fundamentals of symbolic and numeric processing. D. C. Heath and Company, Lexington, MA, USA, 1992. (Cited on page 16.)

[Núñez 2002] Haydemar Núñez, Cecilio Angulo and Andreu Català. *Rule extraction from support vector machines.* In Michel Verleysen, editeur, ESANN, pages 107–112, 2002. (Cited on page 53.)

[Oral 2007] A.D. Oral and A.B. Bener. *Defect prediction for embedded software.* In Computer and information sciences, 2007. ISCIS 2007. 22nd international symposium on, pages 1–6, Nov. 2007. (Cited on page 65.)

[Oram 2010] Andy Oram and Greg Wilson. Making software: What really works, and why we believe it. O'Reilly Media, 2010. (Cited on page 49.)

[Ostrand 2005] Thomas J. Ostrand, Elaine J. Weyuker and Robert M. Bell. *Predicting the Location and Number of Faults in Large Software Systems.* IEEE Trans. Software Eng., vol. 31, no. 4, pages 340–355, 2005. (Cited on page 11.)

[Osuna 1997] E. Osuna, R. Freund and F. Girosit. *Training support vector machines: an application to face detection.* In Computer Vision and Pattern Recognition, 1997. Proceedings., 1997 IEEE Computer Society Conference on, pages 130–136, jun 1997. (Cited on page 38.)

[Pearson 1901] K. Pearson. *On lines and planes of closest fit to systems of points in space.* Philosophical Magazine, vol. 2, no. 6, pages 559–572, 1901. (Cited on page 56.)

[Pelayo 2007] L. Pelayo and S. Dick. *Applying Novel Resampling Strategies To Software Defect Prediction.* In Fuzzy Information Processing Society, 2007. NAFIPS '07. Annual Meeting of the North American, pages 69–72, June 2007. (Cited on page 65.)

[Quinlan 1993] J. Ross Quinlan. C4.5: programs for machine learning. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. (Cited on page 20.)

[Rezwan 2011] Faisal Rezwan, Yi Sun, Neil Davey, Rod Adams, Alistair G. Rust and Mark Robinson. *Effect of using varying negative examples in transcription factor binding site predictions.* In Proceedings of the 9th European conference on Evolutionary computation, machine learning and data mining in bioinformatics, EvoBIO'11, pages 1–12, Berlin, Heidelberg, 2011. Springer-Verlag. (Cited on page 38.)

[Rodriguez 2007] D. Rodriguez, R. Ruiz, J. Cuadrado-Gallego and J. Aguilar-Ruiz. *Detecting Fault Modules Applying Feature Selection to Classifiers.* In Information Reuse and Integration. IRI 2007. IEEE International Conference on, pages 667–672, Aug. 2007. (Cited on page 65.)

[Rosenberg 1997] J. Rosenberg. *Some Misconceptions About Lines of Code.* In Proceedings of the 4th International Symposium on Software Metrics, METRICS '97, pages 137–, Washington, DC, USA, 1997. IEEE Computer Society. (Cited on page 6.)

[Rosenblatt 1958] F. Rosenblatt. *The perceptron: a probabilistic model for information storage and organization in the brain.* Psychological review, vol. 65, no. 6, page 386, 1958. (Cited on page 19.)

[Schölkopf 2001] Bernhard Schölkopf and Alexander J. Smola. Learning with kernels: Support vector machines, regularization, optimization, and beyond (adaptive computation and machine learning). The MIT Press, 2001. (Cited on page 37.)

[Schröter 2006] Adrian Schröter, Thomas Zimmermann and Andreas Zeller. *Predicting Component Failures at Design Time.* In Proceedings of the 5th International Symposium on Empirical Software Engineering, pages 18–27, September 2006. (Cited on page 64.)

[Segal 2004] Mark R Segal. *Machine Learning Benchmarks and Random Forest Regression.* Rapport technique, UC San Francisco: Centre for Bioinformatics and Molecular Biostatistics., 2004. (Cited on page 80.)

[Segaran 2007] Toby Segaran. Programming collective intelligence. O'Reilly, first édition, 2007. (Cited on pages 14 and 51.)

[Segata 2009] Nicola Segata, Enrico Blanzieri and Pádraig Cunningham. *A Scalable Noise Reduction Technique for Large Case-Based Systems.* In Proceedings of the 8th International Conference on Case-Based Reasoning: Case-Based Reasoning Research and Development, ICCBR '09, pages 328–342, Berlin, Heidelberg, 2009. Springer-Verlag. (Cited on page 16.)

[Segata 2010] Nicola Segata, Enrico Blanzieri, Sarah Delany and Pádraig Cunningham. *Noise reduction for instance-based learning with a local maximal margin approach.* Journal of Intelligent Information Systems, vol. 35, pages 301–331, 2010. (Cited on page 16.)

[Seliya 2005] N. Seliya, T.M. Khoshgoftaar and S. Zhong. *Analyzing software quality with limited fault-proneness defect data.* In High-Assurance Systems Engineering, 2005. HASE 2005. Ninth IEEE International Symposium on, pages 89–98, Oct. 2005. (Cited on page 65.)

[Shen 1983] V. Y. Shen, S. D. Conte and H. E. Dunsmore. *Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support.* IEEE Trans. Softw. Eng., vol. 9, no. 2, pages 155–165, 1983. (Cited on page 7.)

[Shenoy 2011] Aruna Shenoy, Sue H. Anthony, Ray J. Frank and Neil Davey. *Categorizing facial expressions: a comparison of computational models.* Neural Computing and Applications, vol. 20, no. 6, pages 815–823, 2011. (Cited on page 38.)

[Shepperd 1988] Martin Shepperd. *A critique of cyclomatic complexity as a software metric.* Softw. Eng. J., vol. 3, no. 2, pages 30–36, 1988. (Cited on page 8.)

[Shivaji 2009] S. Shivaji, E.J. Whitehead, R. Akella and Sunghun Kim. *Reducing Features to Improve Bug Prediction.* In Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on, pages 600–604, nov. 2009. (Cited on page 64.)

[Singh 2008] Yogesh Singh, Arvinder Kaur and Ruchika Malhotra. *Predicting Software Fault Proneness Model Using Neural Network.* In Proceedings of the 9th international conference on Product-Focused Software Process Improvement, PROFES '08, pages 204–214, Berlin, Heidelberg, 2008. Springer-Verlag. (Cited on page 65.)

[Śliwerski 2005] Jacek Śliwerski, Thomas Zimmermann and Andreas Zeller. *When do changes induce fixes?* SIGSOFT Softw. Eng. Notes, vol. 30, no. 4, pages 1–5, 2005. (Cited on pages 10, 11 and 64.)

[Soares 2004] Carlos Soares, Pavel B. Brazdil and Petr Kuba. *A Meta-Learning Method to Select the Kernel Width in Support Vector Regression.* Mach. Learn., vol. 54, no. 3, pages 195–209, March 2004. (Cited on pages 38 and 45.)

[Sommerville 2006] Ian Sommerville. Software engineering: (8th edition) (international computer science series). Addison Wesley, 2006. (Cited on pages 1 and 8.)

[Song 2011] Qinbao Song, Zihan Jia, M. Shepperd, Shi Ying and Jin Liu. *A General Software Defect-Proneness Prediction Framework.* Software Engineering, IEEE Transactions on, vol. 37, no. 3, pages 356–370, may-june 2011. (Cited on pages 43, 45, 49 and 65.)

[Spackman 1989] Kent A. Spackman. *Signal Detection Theory: Valuable Tools for Evaluating Inductive Learning.* In ML, pages 160–163, 1989. (Cited on page 29.)

[Sun 2006] Yi Sun, Mark Robinson, Rod Adams, Rene Te Boekhorst, Alistair G. Rust and Neil Davey. *Using sampling methods to improve binding site predictions.* In Proceedings of ESANN, 2006. (Cited on page 45.)

[Sun 2010] Yi Sun, Lun Tak Lam, Gary P. Moss, Maria Prapopoulou, Rod Adams, Neil Davey, David Gray and Marc B. Brown. *Predicting drug absorption rates through human skin.* In International Joint Conference on Neural Networks (IJCNN), Barcelona, Spain, pages 1–5. IEEE, 2010. (Cited on page 120.)

[Tao 2010] Wang Tao and Li Wei-hua. *Naive Bayes Software Defect Prediction Model.* In Computational Intelligence and Software Engineering (CiSE), 2010 International Conference on, pages 1–4, dec. 2010. (Cited on page 65.)

[Tosun 2009] Ayse Tosun and Ayse Bener. *Reducing false alarms in software defect prediction by decision threshold optimization.* In Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement, ESEM '09, pages 477–480, Washington, DC, USA, 2009. IEEE Computer Society. (Cited on page 65.)

[Turhan 2007] Burak Turhan and Ayse Bener. *A Multivariate Analysis of Static Code Attributes for Defect Prediction.* In QSIC '07: Proceedings of the Seventh International Conference on Quality Software, pages 231–237, Washington, DC, USA, 2007. IEEE Computer Society. (Cited on page 65.)

[Turhan 2009a] Burak Turhan, Gozde Kocak and Ayse Bener. *Data mining source code for locating software bugs: A case study in telecommunication industry.* Expert Systems with Applications, vol. 36, no. 6, pages 9986–9990, 2009. (Cited on page 65.)

[Turhan 2009b] Burak Turhan, Tim Menzies, Ayşe B. Bener and Justin Di Stefano. *On the relative value of cross-company and within-company data for defect prediction.* Empirical Softw. Engg., vol. 14, pages 540–578, October 2009. (Cited on page 65.)

[Vandecruys 2008] O. Vandecruys, D. Martens, B. Baesens, C. Mues, M. De Backer and R. Haesen. *Mining software repositories for comprehensible software fault prediction models.* Journal of Systems and Software, vol. 81, no. 5, pages 823–839, 2008. (Cited on page 65.)

[Vangelis Metsis 2006] Vangelis Metsis, Ion Androutsopoulos and Georgios Paliouras. *Spam Filtering with Naive Bayes – Which Naive Bayes?* In Third Conference on Email and Anti-Spam, 2006. (Cited on page 23.)

[Vapnik 1963] V Vapnik and A Lerner. *Pattern Recognition using Generalized Portrait Method.* Automation and Remote Control, vol. 24, no. 6, pages 774–780, 1963. (Cited on page 38.)

[Vivanco 2010] Rodrigo A. Vivanco, Yasutaka Kamei, Akito Monden, Ken ichi Matsumoto and Dean Jin. *Using search-based metric selection and oversampling to predict fault prone modules.* In CCECE, pages 1–6. IEEE, 2010. (Cited on page 65.)

[Williams 2008] Chadd Williams and Jaime Spacco. *SZZ revisited: verifying when changes induce fixes.* In Proceedings of the 2008 workshop on Defects in large software systems, DEFECTS '08, pages 32–36, New York, NY, USA, 2008. ACM. (Cited on pages 11 and 64.)

[Witten 2005] Ian H. Witten and Eibe Frank. Data mining: Practical machine learning tools and techniques. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, second édition, June 2005. (Cited on pages 14, 15, 20, 22, 24, 65, 66, 72, 74, 83, 85 and 106.)

[Wright 2010] Hyrum K. Wright, Miryung Kim and Dewayne E. Perry. *Validity concerns in software engineering research.* In Proceedings of the FSE/SDP workshop on Future of software engineering research, FoSER '10, pages 411–414, New York, NY, USA, 2010. ACM. (Cited on page 96.)

[Zhang 2007] Hongyu Zhang and Xiuzhen Zhang. *Comments on "Data Mining Static Code Attributes to Learn Defect Predictors".* IEEE Trans. Softw. Eng., vol. 33, pages 635–637, September 2007. (Cited on pages 43, 49, 74, 84, 88, 90, 91 and 112.)

[Zhang 2009] Hongyu Zhang. *An investigation of the relationships between lines of code and defects.* In Software Maintenance, 2009. ICSM 2009. IEEE International Conference on, pages 274–283, sept. 2009. (Cited on page 65.)

[Zhang 2010] Hongyu Zhang, Adam Nelson and Tim Menzies. *On the value of learning from defect dense components for software defect prediction.* In Proceedings of the 6th International Conference on Predictive Models in Software Engineering, PROMISE '10, pages 14:1–14:9, New York, NY, USA, 2010. ACM. (Cited on page 65.)

[Zhong 2004] Shi Zhong, T.M. Khoshgoftaar and N. Seliya. *Unsupervised learning for expert-based software quality estimation.* In High Assurance Systems Engineering, 2004. Proceedings. Eighth IEEE International Symposium on, pages 149–155, march 2004. (Cited on page 65.)

[Zimmermann 2004] Thomas Zimmermann and Peter Weißgerber. *Preprocessing CVS Data for Fine-grained Analysis.* In Proceedings of the First International Workshop on Mining Software Repositories, pages 2–6, May 2004. (Cited on page 97.)

Appendices

Appendix A

International Conference on Engineering Applications of Neural Networks 2009: Using the Support Vector Machine as a Classification Method for Software Defect Prediction with Static Code Metrics

Using the Support Vector Machine as a Classification Method for Software Defect Prediction with Static Code Metrics

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

Science and Technology Research Institute,
University of Hertfordshire, UK
(d.gray, d.h.bowes, n.davey, y.2.sun, b.christianson)@herts.ac.uk

Abstract. The automated detection of defective modules within software systems could lead to reduced development costs and more reliable software. In this work the static code metrics for a collection of modules contained within eleven NASA data sets are used with a Support Vector Machine classifier. A rigorous sequence of pre-processing steps were applied to the data prior to classification, including the balancing of both classes (defective or otherwise) and the removal of a large number of repeating instances. The Support Vector Machine in this experiment yields an average accuracy of 70% on previously unseen data.

1 Introduction

Software defect prediction is the process of locating defective modules in software and is currently a very active area of research within the software engineering community. This is understandable as "Faulty software costs businesses $78 billion per year" ([1], published in 2001), therefore any attempt to reduce the number of latent defects that remain inside a deployed system is a worthwhile endeavour.

Thus the aim of this study is to observe the classification performance of the Support Vector Machine (SVM) for defect prediction in the context of eleven data sets from the NASA Metrics Data Program (MDP) repository; a collection of data sets generated from NASA software systems and intended for defect prediction research. Although defect prediction studies have been carried out with these data sets and various classifiers (including an SVM) in the past, this study is novel in that thorough data cleansing methods are used explicitly.

The main purpose of static code metrics (examples of which include the number of: lines of code, operators (as proposed in [2]) and linearly independent paths (as proposed in [3]) in a module) is to give software project managers an indication toward the quality of a software system. Although the individual worth of such metrics has been questioned by many authors within the software engineering community (see [4], [5], [6]), they still continue to be used.

Data mining techniques from the field of artificial intelligence now make it possible to predict software defects; undesired outputs or effects produced by software, from static code metrics. Views toward the worth of using such metrics for defect prediction are as varied within the software engineering community as those toward the worth of static code metrics. However, the findings within this study suggest that such predictors are useful, as on the data used here they correctly classify modules with an average accuracy of 70%.

2 Background

2.1 Static Code Metrics

Static code metrics are measurements of software features that may potentially relate to quality. Examples of such features and how they are often measured include: size, via lines of code (LOC) counts; readability, via operand and operator counts (as proposed by [2]) and complexity, via linearly independent path counts (as proposed by [3]).

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 (also known as the cyclomatic complexity [3]) is 3. These are just a few examples of the many metrics that can be statically computed from source code.

Fig. 1. An example C program.

```
#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;
}
```

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. [7] points out that such collections of metrics can be used in the following contexts:

– **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 highlighted as such can then be used as pointers to where developers should be focusing their efforts. [8] points out that this is common practice amongst several large US government contractors.

2.2 The Support Vector Machine

A Support Vector Machine (SVM) is a supervised machine learning algorithm that can be used for both classification and regression [9]. SVMs are known as maximum margin classifiers as they find the best separating hyperplane between two classes. This process can also be applied recursively to allow the separation of any number of classes. Only those data points that are located nearest to this dividing hyperplane, known as the *support vectors,* are used by the classifier. This enables SVMs to be used successfully with both large and small data sets. Moreover, the process of finding the decision boundary is a convex optimisation problem, so there are no problems with local minima.

Although maximum margin classifiers are strictly intended for linear classification, they can also be used successfully for non-linear classification (such as the case here) via the use of 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 [10]. The benefit of using a kernel function is that the data is more likely to be linearly separable in the higher feature space. Additionally, the actual 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 have varying characteristics and are suitable for different problem domains. The one used here is the *Gaussian radial basis function* (RBF), as it can handle non-linear problems, requires fewer parameters than other non-linear kernels and is computationally less demanding than the polynomial kernel [11]. In fact, this kernel implicitly maps the data into an infinite dimensional feature space.

When an SVM is used with a Gaussian RBF kernel, there are two user-specified parameters, C and γ. C is the error cost parameter; a variable that determines the trade-off between minimising the training error and maximizing the margin (see Fig. 2). γ controls the width / radius of the Gaussian RBF. The performance of an SVM is largely dependant on these parameters, and the optimal values need to be determined *for each training set* via a systematic search.

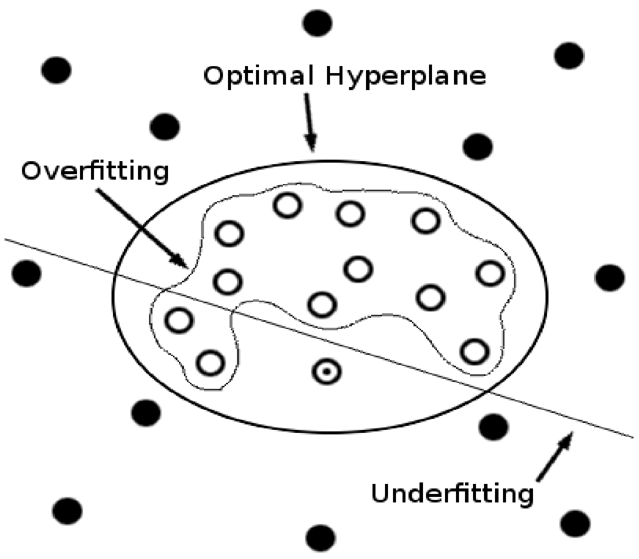


Fig. 2. The importance of optimal parameter selection. The solid and hollow dots represent the training data for two classes. The hollow dot with a dot inside is the test data. Observe that the test dot will be misclassified if too simple (underfitting, the straight line) or too complex (overfitting, the jagged line) a hyperplane is chosen. The optimum hyperplane is shown by the oval line.

2.3 Data

The data used within this study was obtained from the NASA Metrics Data Program (MDP) repository[1]. This repository currently contains thirteen data sets, each of which represent a NASA software system / subsystem and contain the static code metrics and corresponding fault data for each comprising module. Note that a module in this domain can refer to a function, procedure or method. Eleven of these thirteen data sets were used in this study: brief details of each are shown in Table 1. A total of 42 metrics and a unique module identifier comprise each data set (see Table 5, located in the appendix), with the exception of MC1 and PC5 which do not contain the decision density metric.

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

Table 1. The eleven NASA MDP data sets that were used in this study. Note that KLOC refers to thousand lines of code.

Name	Language	Total KLOC	No. of Modules	% Defective Modules
CM1	C	20	505	10
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	C	26	5589	0.4
PC3	C	40	1563	10
PC4	C	36	1458	12
PC5	C++	164	17186	3

All the metrics shown within Table 5 with the exception of *error count* and *error density*, 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 that were issued for each module via a bug tracking system. Each error report increments the value by one. *Error density* is derived from *error count* and *LOC total,* and describes the number of errors per thousand lines of code (KLOC).

3 Method

3.1 Data Pre-processing

The process for cleansing each of the data sets used in this study is as follows:

Initial Data Set Modifications Each of the data sets initially had their module identifier and *error density* attribute removed, as these are not required for classification. 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.

Removing Repeated and Inconsistent Instances Repeated feature vectors, whether with the same (repeated instances) or different (inconsistent instances) class labels, are a known problem within the data mining community [12].

Ensuring that training and testing sets do not share instances guarantees that all classifiers are being tested against *previously unseen data.* This is very important as testing a predictor upon the data used to train it can greatly overestimate performance [12]. The removal of inconsistent items from training data is also important, as it is clearly illogical in the context of binary classification for a classifier to associate the same data point with both classes.

Carrying out this pre-processing stage showed that some data sets (namely MC1, PC2 and PC5) had an overwhelmingly high number of repeating instances (79%, 75% and 90% respectively, see Table 2). Although no explanation has yet been found for these high number of repeated instances, it appears highly unlikely that this is a true representation of the data, i.e. that 90% of modules within a system / subsystem could possibly have the same number of: lines, comments, operands, operators, unique operands, unique operators, conditional statements, etc.

Table 2. The result of removing all repeated and inconsistent instances from the data.

Name	Original Instances	Instances Removed	% Removed
CM1	505	51	10
KC3	458	134	29
KC4	125	12	10
MC1	9466	7470	79
MC2	161	5	3
MW1	403	27	7
PC1	1107	158	14
PC2	5589	4187	75
PC3	1563	130	8
PC4	1458	116	8
PC5	17186	15382	90
Total	**38021**	**27672**	**73**

Removing Constant Attributes If an attribute has a fixed value throughout all instances then it is obviously of no use to a classifier and should be removed.

Each data set had between 1 and 4 attributes removed during this phase with the exception of KC4 that had a total of 26. Details are not shown here due to space limitations.

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.

Manual inspection of these missing values indicated that they were almost certainly supposed to be representing zero, and were replaced accordingly.

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. When such *imbalanced* data is used with a supervised classification algorithm such as an SVM, the classifier will be expected to over predict the majority class [10], as this will produce lower error rates in the test set.

There are various techniques that can be used to balance data (see [13]). The approach taken here is the simplest 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, along with the final number of instances contained within each data set, are shown in Table 3.

Table 3. The result of balancing each data set.

Name	**Instances Removed**	**% Removed**	**Final no. of Instances**
CM1	362	80	**92**
KC3	240	74	**84**
KC4	1	1	**112**
MC1	1924	96	**72**
MC2	54	35	**102**
MW1	320	85	**56**
PC1	823	87	**126**
PC2	1360	97	**42**
PC3	1133	79	**300**
PC4	990	74	**352**
PC5	862	48	**942**

Normalisation All values within the data sets used in this study are numeric, so to prevent attributes with a large range dominating the classification model all values were normalised between -1 and +1. Note that this pre-processing stage was performed just prior to training for each training and testing set, and that each training / testing set pair were scaled in the same manner [11].

Randomising Instance Order The order of the instances within each data set were randomised to defend against *order effects*, where the performance of a predictor fluctuates due to certain orderings within the data [14].

3.2 Experimental Design

When splitting each of the data sets into training and testing sets it is important to ameliorate possible anomalous results. To this end we use five-fold cross-validation. Note that to reduce the effects of sampling bias introduced when randomly splitting each data set into five bins, the cross-validation process was repeated 10 times for each data set in each iteration of the experiment (described below).

As mentioned in Section 2.2, an SVM with an RBF kernel requires the selection of optimal values for parameters C and γ for maximal performance. Both values were chosen for each training set using a five-fold *grid search* (see [11]), a process that uses cross-validation and a wide range of possible parameter values in a systematic fashion. The pair of values that yield the highest average accuracy are then taken as the optimal parameters and used when generating the final model for classification.

Due to the high percentage of information lost when balancing each data set (with the exception of KC4), the experiment is repeated fifty times. This is in order to further minimise the effects of sampling bias introduced by the random undersampling that takes place during balancing.

Pseudocode for the full experiment carried out in this study is shown in Fig 3. Our chosen SVM environment is LIBSVM [15], an open source library for SVM experimentation.

Fig. 3. Pseudocode for the experiment carried out in this study.

```
M = 50     # No. of times to repeat full experiment
N = 10     # No. of cross-validation repetitions
V = 5      # No. of cross-validation folds

DATASETS = ( CM1, KC3, KC4, MC1, MC2, MW1, PC1, PC2, PC3, PC4, PC5 )

results = ( )     # An empty list

repeat M times:
  for dataSet in DATASETS:
    dataSet = pre_process(dataSet)     # As described in Section 3.1
    repeat N times:
      for i in 1 to V:
        testerSet = dataSet[i]
        trainingSet = dataSet - testerSet
        params = gridSearch(trainingSet)
        model = svm_train(params, trainingSet)
        results += svm_predict(model, testerSet)

FinalResults = avg(results)
```

4 Assessing Performance

The measure used to assess predictor performance in this study is *accuracy*. Accuracy is defined as the ratio of instances correctly classified out of the total number of instances. Although simple, accuracy is a suitable performance measure for this study as each test set is balanced. For imbalanced test sets more complicated measures are required.

5 Results

The average results for each data set are shown in Table 4. The results show an average accuracy of 70% across all 11 data sets, with a range of 64% to 82%. Notice that there is a fairly high deviation shown within the results. This is to be expected due to the large amount of data lost during balancing and supports the decision for the experiment being repeated fifty times (see Fig. 3). It is notable that the accuracy for some data sets is extremely high, for example with data set PC4, four out of every five modules were being correctly classified.

The results show that all data sets with the exception of PC2 have a mean accuracy greater than two standard deviations away from 50%. This shows the statistical significance of the classification results when compared to a dumb classifier that predicts all one class (and therefore scores an accuracy of 50%).

Table 4. The results obtained from this study.

Name	**% Mean Accuracy**	**Std.**
CM1	68	5.57
KC3	66	6.56
KC4	71	4.93
MC1	65	6.74
MC2	64	5.68
MW1	71	7.3
PC1	71	5.15
PC2	64	9.17
PC3	76	2.15
PC4	82	2.11
PC5	69	1.41
Total	**70**	**5.16**

6 Analysis

Previous studies ([16], [17], [18]) have also used data from the NASA MDP repository and an SVM classifier. Some of these studies briefly mention data pre-processing, however we believe that it is important to explicitly carry out all of the data cleansing stages described here. This is especially true with regard to the removal of repeating instances, ensuring that all classifiers are being tested against previously unseen data.

The high number of repeating instances found within the MDP data sets was surprising. Brief analysis of other defect prediction data sets showed a repeating average of just 1.4%. We are therefore suspicious of the suitability of the data held within the MDP repository for defect prediction and believe that previous studies which have used this data and not carried out appropriate data cleansing methods may be reporting inflated performance values.

An example of such a study is [18], where the authors use an SVM and four of the NASA data sets, three of which were used in this study (namely CM1, PC1 and KC3). The authors make no mention of data pre-processing other than the use of an attribute selection algorithm. They then go on to report a minimum average *precision*, the ratio of correctly predicted defective modules to the total number of modules predicted as defective, of 84.95% and a minimum average *recall*, the ratio of defective modules detected as such, of 99.4%. We believe that such high classification rates are highly unlikely in this problem domain due to the limitations of static code metrics and that not carrying out appropriate data cleansing methods may have been a factor in these high results.

7 Conclusion

This study has shown that on the data studied here the Support Vector Machine can be used successfully as a classification method for defect prediction. We hope to improve upon these results in the near future however via the use of a one-class SVM; an extension to the original SVM algorithm that trains upon only defective examples, or a more sophisticated balancing technique such as SMOTE (Synthetic Minority Over-sampling Technique).

Our results also show that previous studies which have used the NASA data may have exaggerated the predictive power of static code metrics. If this is not the case then we would recommend the explicit documentation of what data pre-processing methods have been applied. Static code metrics can only be used as probabilistic statements toward the quality of a module and further research may need to be undertaken to define a new set of metrics specifically designed for defect prediction.

The importance of data analysis and data quality has been highlighted in this study, especially with regard to the high quantity of repeated instances found within a number of the data sets. The issue of data quality is very important within any data mining experiment as poor quality data can threaten the validity of both the results and the conclusions drawn from them [19].

8 Appendix

Table 5. The 42 metrics originally found within each data set.

Metric Type	**Metric Name**
McCabe	01. Cyclomatic Complexity
	02. Cyclomatic Density
	03. Decision Density
	04. Design Density
	05. Essential Complexity
	06. Essential Density
	07. Global Data Density
	08. Global Data Complexity
	09. Maintenance Severity
	10. Module Design Complexity
	11. Pathological Complexity
	12. Normalised Cyclomatic Complexity
Raw Halstead	13. Number of Operators
	14. Number of Operands
	15. Number of Unique Operators
	16. Number of Unique Operands
Derived Halstead	17. Length (N)
	18. Volume (V)
	19. Level (L)
	20. Difficulty (D)
	21. Intellegent Content (I)
	22. Programming Effort (E)
	23. Error Estimate (B)
	24. Programming Time (T)
LOC Counts	25. LOC Total
	26. LOC Executable
	27. LOC Comments
	28. LOC Code and Comments
	29. LOC Blank
	30. Number of Lines (opening to closing bracket)
Misc.	31. Node Count
	32. Edge Count
	33. Branch Count
	34. Condition Count
	35. Decision Count
	36. Formal Parameter Count
	37. Modified Condition Count
	38. Multiple Condition Count
	39. Call Pairs
	40. Percent Comments
Error	41. Error Count
	42. Error Density

References

1. Levinson, M.: Lets stop wasting $78 billion per year. CIO Magazine (2001)
2. Halstead, M.H.: Elements of Software Science (Operating and programming systems series). Elsevier Science Inc., New York, NY, USA (1977)
3. McCabe, T.J.: A complexity measure. In: ICSE '76: Proceedings of the 2nd international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1976) 407
4. Hamer, P.G., Frewin, G.D.: M.H. Halstead's Software Science - a critical examination. In: ICSE '82: Proceedings of the 6th international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1982) 197–206
5. Shen, V.Y., Conte, S.D., Dunsmore, H.E.: Software Science Revisited: A critical analysis of the theory and its empirical support. IEEE Trans. Softw. Eng. **9**(2) (1983) 155–165
6. Shepperd, M.: A critique of cyclomatic complexity as a software metric. Softw. Eng. J. **3**(2) (1988) 30–36
7. Sommerville, I.: Software Engineering: (8th Edition) (International Computer Science Series). Addison Wesley (2006)
8. Menzies, T., Greenwald, J., Frank, A.: Data mining static code attributes to learn defect predictors. Software Engineering, IEEE Transactions on **33**(1) (Jan. 2007) 2–13
9. Schölkopf, B., Smola, A.J.: Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond (Adaptive Computation and Machine Learning). The MIT Press (2001)
10. Sun, Y., Robinson, M., Adams, R., Boekhorst, R.T., Rust, A.G., Davey, N.: Using sampling methods to improve binding site predictions. In: Proceedings of ESANN. (2006)
11. Hsu, C.W., Chang, C.C., Lin, C.J.: A practical guide to support vector classification. Technical report, Taipei (2003)
12. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques. Second edn. Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann (June 2005)
13. Wu, G., Chang, E.Y.: Class-boundary alignment for imbalanced dataset learning. In: ICML 2003 Workshop on Learning from Imbalanced Data Sets. (2003) 49–56
14. Fisher, D.: Ordering effects in incremental learning. In: Proc. of the 1993 AAAI Spring Symposium on Training Issues in Incremental Learning, Stanford, California (1993) 34–41
15. Chang, C.C., Lin, C.J.: LIBSVM: a library for support vector machines. (2001) Software available at http://www.csie.ntu.edu.tw/∼cjlin/libsvm.
16. Li, Z., Reformat, M.: A practical method for the software fault-prediction. Information Reuse and Integration, 2007. IRI 2007. IEEE International Conference on (Aug. 2007) 659–666
17. Lessmann, S., Baesens, B., Mues, C., Pietsch, S.: Benchmarking classification models for software defect prediction: A proposed framework and novel findings. Software Engineering, IEEE Transactions on **34**(4) (2008) 485–496
18. Elish, K.O., Elish, M.O.: Predicting defect-prone software modules using support vector machines. J. Syst. Softw. **81**(5) (2008) 649–660
19. Liebchen, G.A., Shepperd, M.: Data sets and data quality in software engineering. In: PROMISE '08: Proceedings of the 4th international workshop on Predictor models in software engineering, New York, NY, USA, ACM (2008) 39–44

Appendix B

International Joint Conference on Neural Networks 2010: Software Defect Prediction Using Static Code Metrics Underestimates Defect-Proneness

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 pre-processing) 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 pre-processing 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 $\div$ 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

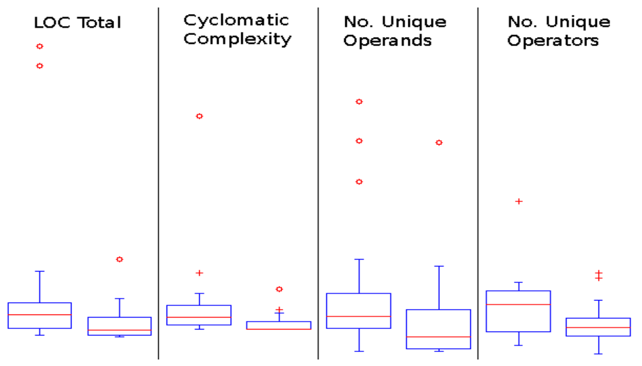


Fig. 2. A box plot showing basic statistics for data set PC2. Boxes on the left in each column are the defective instances while boxes on the right are the non-defective instances.

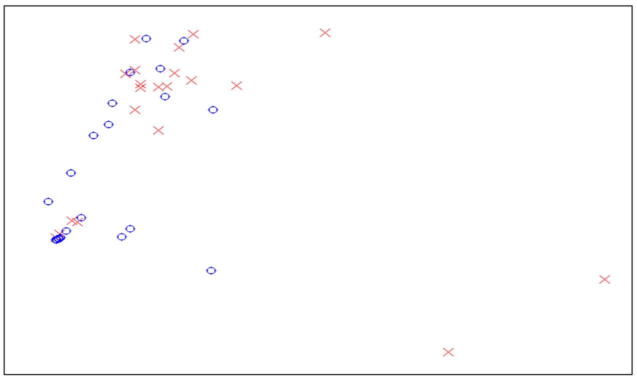


Fig. 3. Principal Components Analysis on data set PC2. Crosses represent modules labelled as defective while circles represent modules labelled as non-defective. Observe the extreme outliers belonging to the defective class in the bottom right corner.

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.

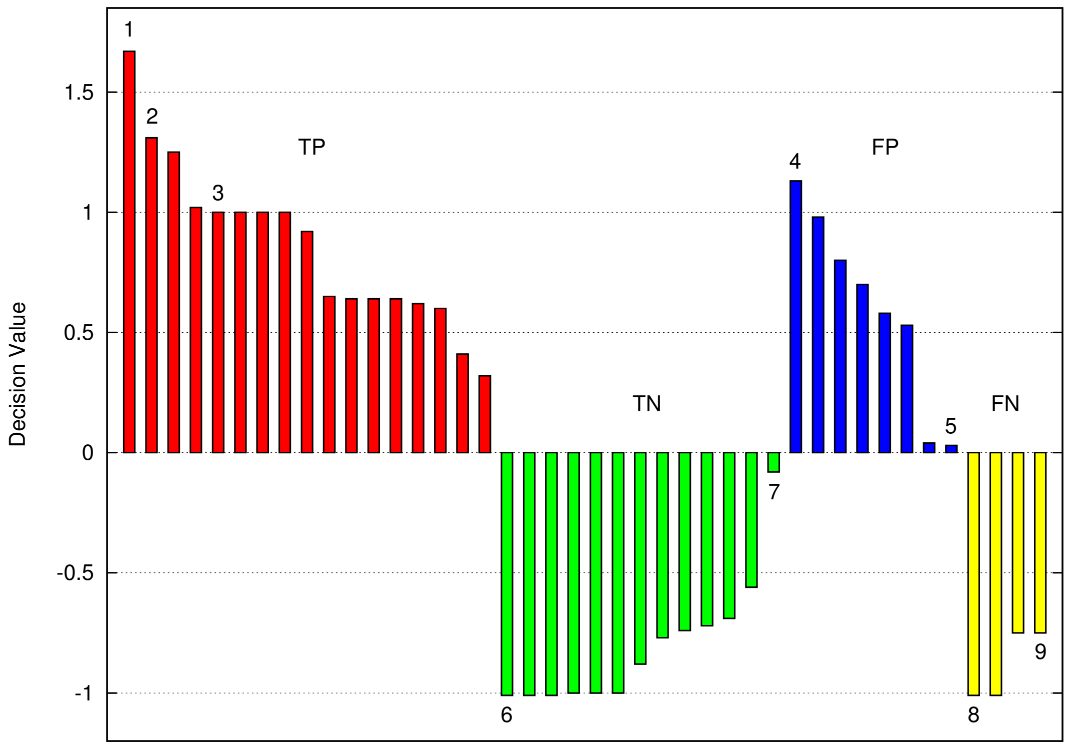


Fig. 4. The decision value and classification result for each of the 42 modules in data set PC2.

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.

Appendix C

International Conference on Evaluation and Assessment in Software Engineering 2011: The Misuse of the NASA Metrics Data Program Data Sets for Automated Software Defect Prediction

The Misuse of the NASA Metrics Data Program Data Sets for Automated Software Defect Prediction

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

***Abstract*—**
Background: The NASA Metrics Data Program data sets have been heavily used in software defect prediction experiments.
Aim: To demonstrate and explain why these data sets require significant pre-processing in order to be suitable for defect prediction.
Method: A meticulously documented data cleansing process involving all 13 of the original NASA data sets.
Results: Post our novel data cleansing process; each of the data sets had between 6 to 90 percent less of their original number of recorded values.
Conclusions:
One: Researchers need to analyse the data that forms the basis of their findings in the context of how it will be used.
Two: Defect prediction data sets could benefit from lower level code metrics in addition to those more commonly used, as these will help to distinguish modules, reducing the likelihood of repeated data points.
Three: The bulk of defect prediction experiments based on the NASA Metrics Data Program data sets may have led to erroneous findings. This is mainly due to repeated data points potentially causing substantial amounts of training and testing data to be identical.

I. Introduction

Automated software defect prediction is a process where classification and/or regression algorithms are used to predict the presence of non-syntactic implementational errors (henceforth; *defects*) in software source code. To make these predictions such algorithms attempt to generalise upon *software fault data;* observations of software product and/or process metrics coupled with a *level of defectiveness* value. This value typically takes the form of a *number of faults reported* metric, for a given software unit after a given amount of time (post either code development or system deployment).

The predictions made by defect predictors may be continuous (level of defectiveness) or categorical (set membership of either: {'defective', 'non-defective'}). The current trend by researchers is typically to report the latter (categorical predictions) only. However, this may not be the best approach, as continuous predictions allow software units to be ranked according to their predicted quality factor (this is known as *module-order modelling*, see (Khoshgoftaar & Allen 2003)). A software quality ranking system has the real world benefit of producing an ordered list of the seemingly most error-prone code units. These code units can then be subjected to some form of further inspection, in descending order of predicted level of defectiveness, for as long as resources allow.

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 order to carry out a software defect prediction experiment there is naturally a requirement for reasonable quality data. However, software fault data is very difficult to obtain. Commercial software development companies often do not have a fault measurement program in place. And, even if such a process is in place, it is typically undesirable from a business perspective to publicise fault data. This is particularity true for systems where quality has been a serious problem, i.e. where it would be most useful to publicise such data and give researchers an opportunity to discover why this was the case.

Open-source systems are a good place for researchers to construct their own fault data sets. This is simplest when the system has been developed whilst using a bug tracking system to record the faults encountered by developers. If bug information has been correctly and consistently entered into version control commit messages, it is then possible to autonomously locate the *fault-fixing revisions*. From here it is possible to (fairly accurately) map fault-fixing revisions back to where the fault was first introduced (the *bug-introducing change*, see (Kim, Zimmermann, Pan & Whitehead 2006)). The major problem with constructing fault data from open-source systems is that it can be a very time consuming task to do accurately. This is because human intervention is often required to check the validity of the automated mappings.

Thus difficulty in obtaining software fault data is the major factor why public domain fault data repositories, such as those hosted by NASA[1] and PROMISE[2], have become so popular among researchers. These repositories host numerous data sets, which require no data analysis and little or no pre-preprocessing, before machine learning tools such as Weka[3] will classify them. The ease of this process can be dangerous to the inexperienced researcher. Results can be obtained without any scrutiny of the data. Furthermore, researchers may naively assume the NASA Metrics Data Program (MDP) data sets are of reasonable quality for data mining. This issue is worsened by the hosting sites not indicating the main problems, and by so many previous researchers using these data sets without appropriate pre-processing. The aim of this study is to illuminate why the NASA MDP data sets require significant pre-processing; or contextual *data cleansing,* before they become suitable for data mining. It is hoped that this paper will encourage researchers to take data quality seriously, and to question the results of some studies based on these data sets.

[1]http://mdp.ivv.nasa.gov/
[2]http://promisedata.org/
[3]http://www.cs.waikato.ac.nz/~ml/weka/

The NASA MDP repository currently consists of 13 data sets explicitly intended for software metrics research. Each data set represents a NASA software system/subsystem and contains the static code metrics and corresponding fault data for each comprising module. Note that 'module' in this case can refer to a function, procedure or method. A substantial amount of research based wholly or partially on these data sets has been published over the last decade, including: (Menzies, Stefano, Orrego & Chapman 2004), (Menzies & Stefano 2004), (Menzies, Greenwald & Frank 2007), (Menzies, Milton, Turhan, Cukic, Jiang & Bener 2010), (Jiang, Cukic & Menzies 2008), (Jiang & Cukic 2009), (Khoshgoftaar & Seliya 2004), (Seliya, Khoshgoftaar & Zhong 2005), (Lessmann, Baesens, Mues & Pietsch 2008), (Boetticher 2006), (Mende & Koschke 2009), (Koru & Liu 2005), (Tosun & Bener 2009), (Bezerra, Oliveira & Meira 2007), (Turhan, Menzies, Bener & Di Stefano 2009), (Pelayo & Dick 2007), (Li & Reformat 2007) and (Elish & Elish 2008).

It is widely accepted by the data mining community that in order to accurately assess the potential real world performance of a classification model, the model must be tested against different data from that upon which it was trained (Witten & Frank 2005). This is why there is a distinction between a *training set* and a *testing set.* A testing set is also referred to as an *independent test set;* as it is intended to be independent from the training set (i.e. models should be tested against *unseen data,* see (Witten & Frank 2005)). This is very basic data mining knowledge, and is no surprise to the defect prediction community. In 2004 Menzies et al. state: "if the goal of learning is to generate models that have some useful future validity, then the learned theory should be tested on data not used to build it. Failing to do so can result in a excessive over-estimate of the learned model..." (Menzies et al. 2004). Despite this fact being well known, numerous studies based on the NASA MDP data sets (henceforth, NASA data sets) have potentially had high proportions of identical data points in their training and testing sets. This is because the NASA data sets contain varied quantities of *repeated data points;* i.e. observations of module metrics and their corresponding fault data occurring more than once. Thus, when this data is used in a machine learning context, training and testing sets potentially have large proportions of identical data points. This will result in the aforementioned excessive estimate of performance, as classifiers can memorise rather than learn.

In this study we develop and carry out a meticulously documented data cleansing process involving all 13 of the original NASA data sets. The purpose of this data cleansing is both to make the data sets suitable for machine learning, and to remove *noise* (i.e. inaccurate/incorrect data points, see (Liebchen & Shepperd 2008)). We show that after this process each of the data sets had between 6 to 90 percent less of their original recorded values. We then discuss at length the problems caused by repeated data points when data mining, and why using lower level metrics in fault data sets (such as character counts) may alleviate this problem, by helping to distinguish non-identical modules.

The rest of this paper is presented as follows: in the next section we discuss related work; papers where issues with the NASA data sets have been documented or discussed. In Section III we document our novel data cleansing process in incremental stages. Section IV contains our findings, which include a demonstration of the effect of repeated data points during an artificial classification experiment. Our conclusions are presented in Section V.

II. Related Studies

The major issue with the original NASA data sets is that when they are used in a machine learning context, repeated data points may result in training data inadvertently being included in testing sets, potentially invalidating the experiment. This is not a new finding. However, we believe it needs spelling out to researchers, as previous studies mentioning this issue seem to have been ignored. In this section the most relevant studies surrounding this issue are discussed.

The earliest mention of repeated data in NASA data sets that we can find was made in (Kaminsky & Boetticher 2004). In this study the authors state that they eliminated "redundant data", but give no further explanation as to why. The data set used in this study was NASA data set KC2, which is no longer available from the NASA MDP repository. Although this data set is currently available from the PROMISE repository, we did not use it in our study in an effort to use only the original, unmodified data.

In (Boetticher 2006) five NASA data sets were used in various classification experiments. The author states that "data pre-processing removes all duplicate tuples from each data set along with those tuples that have questionable values (e.g. LOC equal to 1.1)." Interestingly, it is only the PROMISE versions of the NASA data sets that contain these clearly erroneous non-integer LOC values. The author goes into detail on repeated data points, stating that "to avoid building artificial models, perhaps the best approach would be not to allow duplicates within datasets." One of the experiments carried out in this study was intended to show the effect of the repeated data in the five NASA data sets used. This was in the context of a 10-fold cross-validation classification experiment with a C4.5 decision tree. The claimed result was that the data sets with the repeats included achieved significantly higher performance than those without. Although this result is to be expected, there was an unfortunate technical shortcoming in the experimental design. When reporting the performance of classifiers on data sets with imbalanced class distributions, 'accuracy' (or its inverse: 'error rate') should not be used (Nickerson, Japkowicz & Milios 2001). In addition to this, care is required when performing such an experiment, as the proportion of repeated data in each class is not related to the class distribution. Therefore, post the removal of repeated data points, the data sets could have substantially different class distributions. This may boost or reduce classifier performance, because of the *class imbalance problem* (see (Chawla, Japkowicz & Kolcz 2004)).

Classification experiments utilising probabilistic outputs were carried out in (Bezerra et al. 2007). Here the authors used all 13 of the original NASA data sets and state that they removed both “redundant and inconsistent patterns”. Inconsistent data points are another of the problems when data mining with the NASA data sets. They occur when repeated feature vectors (module metrics) describe data points with differing class labels. Thus in this domain they occur where the same set of metrics is used to describe both a module labeled as ‘defective’ and a module labeled as ‘non-defective’. We believe the removal of such instances was first carried out in (Khoshgoftaar & Seliya 2004).

The work described here differs from that previously described, as it is not based on classification experiments. It is instead based on the analysis and cleansing of data. This study demonstrates: the poor quality of the NASA data sets; the extent to which repeated data points disseminate into training and testing sets; and the effect of testing sets containing *seen data* during classification experiments.

III. Method

The NASA data sets are available from the aforementioned NASA MDP and PROMISE repositories. For this study we used the original versions of the data sets from the NASA MDP repository. Note however that the same issues also apply to the PROMISE versions of these data sets, which are for the most part simply the same data in a different format.

A. Initial Pre-processing: Binarisation of Class Variable & Removal of Module Identifier and Extra Error Data Attributes

In order to be suitable for binary classification, the *error count* attribute is commonly reported in the literature (see (Menzies, Greenwald & Frank 2007), (Lessmann et al. 2008) and (Elish & Elish 2008) for example) as being binarised as follows:

$$defective = (error\_count \geq 1)$$

It is also necessary to remove the ‘unique module identifier’ attribute as this gives no information toward the defectiveness of a module. Lastly, it is necessary to remove all other error based attributes to make the classification task worthwhile. This initial pre-processing is summarised as follows:

```
attributes = [    MODULE, ERROR_DENSITY,
        ERROR_REPORT_IN_6_MON,
        ERROR_REPORT_IN_1_YR,
        ERROR_REPORT_IN_2_YRS    ]

for dataSet in dataSets:
    for attribute in attributes:
        if attribute in dataSet:
            dataSet = dataSet - attribute
    dataSet.binarise(error_count)
    dataSet.rename(error_count, defective)
```

The NASA data is often reportedly used in defect prediction experiments post this initial pre-processing. We therefore present an overview of each data set in Table I. In this table the number of original recorded values is defined as the number of attributes (features) multiplied by the number of instances (data points). For simplicity we do not take missing values into account. We use the number of recorded values as a method of quantifying how much data is available in each data set. We shall come back to these values post data cleansing to judge how much data has been removed.

B. Stage 1: Removal of Constant Attributes

An attribute which has a constant/fixed value throughout all instances is easily identifiable as it will have a *variance* of zero. Such attributes contain no information with which to discern modules apart, and are at best a waste of classifier resources. Each data set had from 0 to 10 percent of their total attributes removed during this stage, with the exception of data set KC4. This data set has 26 constant attributes out of a total of 40, thus 65 percent of available recorded values contain no information upon which to data mine.

C. Stage 2: Removal of Repeated Attributes

In addition to constant attributes, repeated attributes occur where two attributes have identical values for each instance. This effectively results in a single attribute being over-represented. Amongst the NASA data sets there is only one pair of repeated attributes (post stage 1), namely the *‘number of lines’* and *‘loc total’* attributes in data set KC4. The difference between these two metrics is poorly defined at the NASA MDP repository. However, they may be identical for this data set as (according to the metrics) there are no modules with any lines either containing comments or which are empty. For this data cleansing stage we removed one of the attributes so that the values were only being represented once. We chose to keep the *‘loc total’* attribute label as this is common to all 13 NASA data sets.

TABLE I
DETAILS OF THE NASA MDP DATA SETS POST INITIAL PRE-PROCESSING.

Name	Language	Features	Instances	Recorded Values	% Defective Instances
CM1	C	40	505	20200	10
JM1	C	21	10878	228438	19
KC1	C++	21	2107	44247	15
KC3	Java	40	458	18320	9
KC4	Perl	40	125	5000	49
MC1	C & C++	39	9466	369174	0.7
MC2	C	40	161	6440	32
MW1	C	40	403	16120	8
PC1	C	40	1107	44280	7
PC2		40	5589	223560	0.4
PC3		40	1563	62520	10
PC4		40	1458	58320	12
PC5	C++	39	17186	670254	3

D. Stage 3: Replacement of Missing Values

Missing values may or may not be problematic for machine learners depending on the classification method used. However, dealing with missing values within the NASA data sets is very simple. Seven of the data sets contain missing values, but all in the same single attribute: *'decision density'*. This attribute is defined as *'condition count'* divided by *'decision count'*, and for each missing value both these base attributes have a value of zero. In the remaining NASA data set which contains all three of the aforementioned attributes but does not contain missing values, all instances with *'condition count'* and *'decision count'* values of zero also have a *'decision density'* of zero. This appears logical, and it is clear that missing values have occurred because of a division by zero error. Because of this we replace all missing values with zero. Note that in (Bezerra et al. 2007) all instances which contained missing values within the NASA data sets were discarded. It is more desirable to cleanse data than to remove it, as the quantity of possible information to learn from will thus be maximised.

E. Stage 4: Enforce Integrity with Domain Specific Expertise

The NASA data sets contain varied quantities of correlated attributes, which are useful for checking data integrity. Additionally, it is possible to use domain specific expertise to validate data integrity, by searching for theoretically impossible occurrences. The following is a non-exhaustive list of possible checks that can be carried out for each data point:

- Halstead's length metric (see (Halstead 1977)) is defined as: *'number of operators'* + *'number of operands'*.
- Each token that can increment a module's cyclomatic complexity (see (McCabe 1976)) is counted as an operator according to the NASA MDP repository. Therefore, the cyclomatic complexity of a module should not be greater than the number of operators + 1. Note that 1 is the minimum cyclomatic complexity value.
- The number of function calls within a module is recorded by the *'call pairs'* metric. A function call operator should be counted as an operator, therefore the number of function calls should not exceed the number of operators.

These three simple rules are a good starting point for removing noise in the NASA data sets. Any data point which does not pass all of the checks contains noise. Because the original NASA software systems/subsystems from where the metrics are derived are not publicly available, it is impossible for us to investigate this issue of noise further. The most viable option is therefore to discard each offending instance. Note that a prerequisite of each check is that the data set must contain all of the relevant attributes. Six of the data sets had data removed during this stage, between 1 to 12 percent of their data points in total.

During this stage it may be tempting to not only remove noise (i.e. inaccurate/incorrect data points), but also *outliers*. A module which (reportedly) contains no lines of code and no operands and operators should be an empty module containing no code. So at this stage of data cleansing, should such a module be discarded? As it is impossible for us to check the validity of the metrics against the original code, this is a grey area. An empty module may still be a valid part of a system, it may just be a question of time before it is implemented. Furthermore, a module missing an implementation may still have been called by an unaware programmer. As the module is unlikely to have carried out the task its name implies, it may also have been reported to be faulty.

F. Stage 5: Removal of Repeated and Inconsistent Instances

As previously mentioned, repeated/redundant instances occur when the same feature vectors (module metrics) describe multiple modules with the same class label. While this situation is clearly possible in the real world, such data points are problematic in the context of machine learning, where is it imperative that classifiers are tested upon data points independent from those used during training (see (Witten & Frank 2005)). The issue is that when data sets containing repeated data points are split into training and testing sets (for example by a *x%* training, *1-x%* testing split, or *n-fold cross-validation*), it is possible for identical instances to appear in both sets. This either simplifies the learning task or reduces it entirely to a task of recollection. Ultimately however, if the experiment was intended to show how well a classifier could generalise upon future, unseen data points, the results will be erroneous as the experiment is invalid.

Inconsistent instances are similar to repeated instances, as they also occur when the same feature vectors describe multiple modules. The difference between repeated and inconsistent instances is that with the latter, the class labels differ, thus (in this domain) the same metrics would describe both a 'defective' and a 'non-defective' module. This is again possible in the real world, and while not as serious an issue as the repeated instances, inconsistent data points are problematic during binary classification tasks. When building a classifier which outputs a predicted class set membership of either *'defective'* or *'non-defective'*, it is clearly illogical to train such a classifier with data instructing that the same set of features is resultant in both classes.

Adding all data points into a mathematical set is the simplest way of ensuring that each one is unique. This ensures classifiers will be tested on unseen data. From here it is possible to remove all inconsistent pairs of modules, to ensure that all feature vectors (data points irrespective of class label) are unique. The proportion of instances removed from each data set during this stage is shown in Figure 1. All data sets had instances removed during this stage, and in some cases the proportion removed was very large (90, 78, and 74 percent for data sets PC5, MC1, and PC2, respectively). Note that for most data sets the proportion of inconsistent instances removed is negligible.

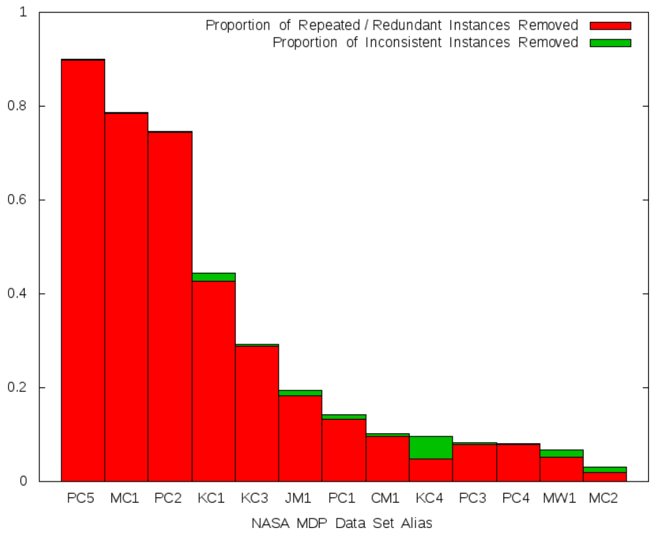


Fig. 1. The proportion of instances removed during stage 5.

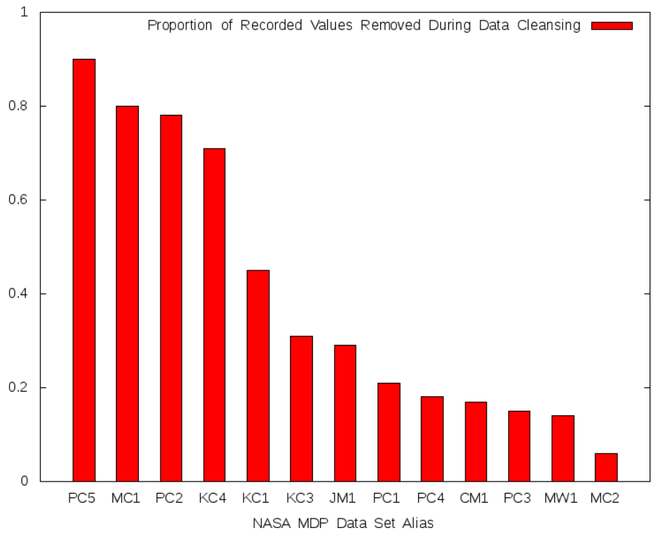


Fig. 2. The proportion of recorded values removed during data cleansing.

IV. Findings

Figure 2 shows the proportion of recorded values removed from the 13 NASA data sets (after basic pre-processing, see Table I) post our 5 stage data cleansing process. Stages 1 and 2 of this process can remove attributes, stage 3 can replace values, and stages 4 and 5 can remove instances (data points). This was the motivation to use the number of recorded values ($attributes * instances$) metric, as it takes both attributes and instances into account. Figure 2 shows that between 6 to 90 percent of recorded values in total were removed from each data set during our data cleansing process.

The purpose of our data cleansing process is to ensure that all data sets are suitable for machine learning (stages 1, 2, 5, and to some extent stage 3), and to remove or repair what can be confidently assumed to be noise (stage 4, and to some extent stage 3). Note however that there are almost certainly noisy data points in the NASA data sets that can not be confidently assumed to be erroneous using such simple methods. But, as the data sets are based on closed source, commercial software, it is impossible for us to investigate the potential issues further. For example, the original data set MC1 (according to the metrics) contains 4841 modules (51% of modules in total) with no lines of code.

Of the data cleansing processes with the potential to reduce the quantity of recorded values, it is the removal of repeated and inconsistent instances (stage 5) that is responsible for the largest average proportions of data removed (see Figure 1). This raises the following questions: *Is the complete removal of such instances really necessary? Why are there so many repeated data points and what can be done in future to avoid them? What proportion of seen data points could end up in testing sets if this data was used in classification experiments? What effect could having such quantities of seen data points in testing sets have on classifier performance?* Each of these questions are addressed in the sections that follow.

A. Is the complete removal of repeated and inconsistent instances prior to classification really necessary?

Our data cleansing method is a work in progress. Post data cleansing, researchers will be able to use off the shelf data mining tools (such as Weka) to carry out experiments that yield meaningful results (or at least, far more meaningful results than had the issues described not been addressed). However, not every part of the cleansing process will be required in all contexts. In this section we describe the issues that researchers should be aware of with regard to addressing the repeated and inconsistent instances.

Figure 1 shows that the proportion of inconsistent instances removed during stage 5 is negligible. This is partly a consequence of repeated instances being removed before inconsistent ones however, as it is possible for a data point to be both repeated and inconsistent. The complete removal of inconsistent instances prior to classification may not always be necessary or desirable. Because defect prediction data sets typically have an imbalanced class distribution (Menzies, Dekhtyar, Distefano & Greenwald 2007), some researchers may wish to retain each inconsistent minority class data point, in order to keep as much data as possible regarding the minority class (typically modules labeled as ‘defective’). During training, some learning methods (such as various probabilistic learners) may be able to robustly handle conflicting information. Therefore, the inclusion of inconsistent instances in training sets would not be problematic in this context. During testing, some researchers may feel that it is more appropriate to include inconsistent data points, as they may occur in the real world. Note that the inclusion of inconsistent data points in testing sets would typically introduce an upper bound on the level of performance achievable.

The training of learning methods on data containing repeated data points is typically not overly problematic. For example, a very simple oversampling technique is to duplicate minority class data points during training. Using training sets which contain repeated data points is reasonable, *as long as no training data points are included in the testing set.* The same also applies if researchers feel that testing sets should include repeated data points. There are potentially serious issues to be aware of when training sets contain repeated data points however, as pointed out in (Kołcz, Chowdhury & Alspector 2003). When optimising model parameters using a validation set (a withheld subset of the training set) that contains duplicate data points, *over-fitting* may occur, potentially harming performance during assessment. It is for this reason that (Kołcz et al. 2003) recommend "tuning a trained classifier with a duplicate-free sample". Note that this also applies when using multiple validation sets, for example via n-fold cross-validation. Kołcz et al. also point out that feature selection techniques can be affected by duplicate data points. An approach to repeated data points proposed by (Boetticher 2006) is to add an extra attribute: *number of duplicates*. This will ensure data points are unique, and help reduce information being lost.

B. Why so much repeated data and how can it be avoided?

As previously stated, the NASA data sets are based on closed source, commercial software, so it is impossible for us to validate whether the repeated data points are truly a representation of each software system/subsystem, or whether they are noise. Despite this, a probable factor in why the repeated (and inconsistent) data is a part of these data sets is because of the poor differential capability of the metrics used. Intuitively, 40 metrics describing each software module seems like a large set. However, many of the metrics are simple equations of other metrics. Because of this, it may be highly beneficial in future to also record lower level metrics, such as character counts. These will help to distinguish modules apart; particularly small modules, which statistically result in more repeated data points than large modules. Additionally, machine learners may be able to utilise such low level data for helping to detect potentially troublesome modules (in terms of error-proneness).

C. What proportion of seen data points could end up in testing sets if this data was used in classification experiments?

In order to find the answer to this question, a small Java program was developed utilising the Weka machine learning tool's libraries (version 3.7.1). The Weka libraries were chosen because they have been heavily used in defect prediction experiments (see (Menzies, Greenwald & Frank 2007), (Boetticher 2006) and (Koru & Liu 2005), for example). In this experiment a standard stratified 10-fold cross-validation was carried out. During each of the 10 folds, the number of instances in the testing set which were also in the training set were counted. After all 10 folds, the average number of shared instances in each testing set was calculated. This process was repeated 1000 times with different pseudo-random number seeds to defend against order effects. For this experiment we used the NASA data sets post basic pre-processing (see Table I). We did this because it was as representative as possible of what will have happened in some previous studies. It is for the same reason that we chose 10-fold cross-validation, the Weka *Explorer* default. The results from this experiment are shown in Figures 3 and 4. From these figures can be seen that for each data set, the proportion of seen data points in the testing sets is larger than the proportion of repeated data points in total. Additionally, this relationship can be seen in Figure 4 to have a strong positive correlation. It is worth emphasising that in some cases the average proportion of seen data points in the testing sets was very large (91, 84, and 82 percent for data sets PC5, MC1, and PC2, respectively).

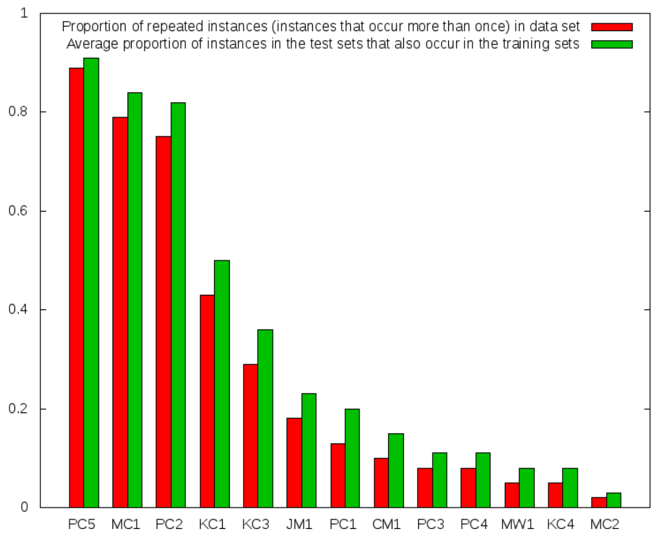


Fig. 3. Proportions of repeated data and seen data in testing sets.

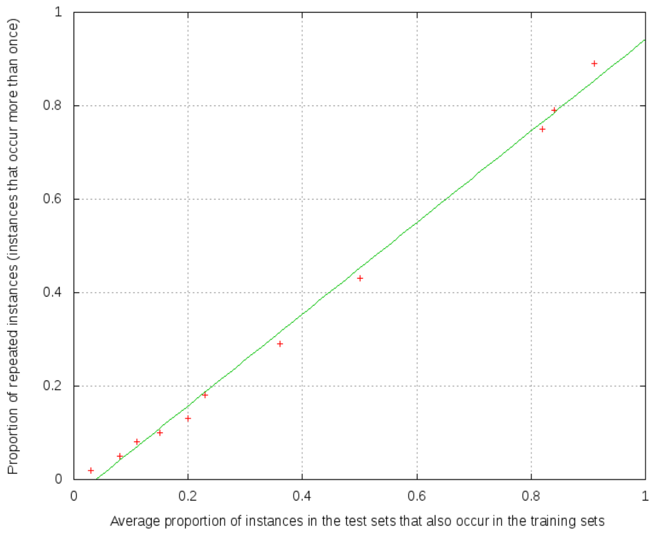


Fig. 4. Proportions of repeated data verses seen data in testing sets.

D. What effect could having such quantities of seen data points in testing sets have on classifier performance?

To answer this question we do not use the NASA data sets because of the point regarding class distributions mentioned in Section II. Instead, we construct an artificial data set. This data set has 10 numeric features and 1000 data points. The numeric features were all generated by a pseudo-random number generator and have a range between 0 and 1 inclusive. The data set has a balanced class distribution, with 500 data points representing each class.

Using the Weka *Experimenter*, we ran 10 repetitions of a 10-fold cross-validation experiment with the data set just described, and the following variations of it:

- 25% repeats from each class, an extra copy of 125 instances in each class, 1250 instances in total.
- 50% repeats from each class, an extra copy of 250 instances in each class, 1500 instances in total.
- 75% repeats from each class, an extra copy of 375 instances in each class, 1750 instances in total.
- 100% repeats, two copies of every instance, 2000 instances in total.

We used a random forest meta decision tree learner for this experiment, with 100 trees and all other parameters set to the Weka defaults. The results from this experiment showed accuracy levels for each data set: original, 25% repeats, 50% repeats, 75% repeats and 100% repeats of 48.30, 65.20, 80.47, 87.49 and 93.50. This clearly shows that repeated data points can have a huge influence on the performance of classifiers, even with pseudo-random data. As the proportion of repeated information increases, so does the performance of the classifier. It is worth pointing out that the severity of repeated data points is algorithm specific. Naive Bayes classifiers have been reported to be fairly resilient to duplicates (Kołcz et al. 2003).

V. Conclusions

Regardless of whether repeated data points are, or are not noise, it is unsuitable to have seen data in testing sets during defect prediction experiments intended to show how well a classifier could potentially perform on future, unseen data points. This is because having identical data points in training and testing sets can result in an excessive estimate of performance, occurring because classifiers can, to varying degrees, memorise rather than learn. There is an important distinction between learning from and simply memorising data: only if you learn the structure underlying the data can you be expected to correctly predict unseen data.

Some researchers may argue that as it is possible for modules with identical metrics to be contained within a software system, such data points should be tolerated rather than removed. While the initial part of this argument is true, if, in the real world, you happen to have a data point in your testing set which is also contained in your training set, chance is on your side. However, it is not scientific for chance to be on the side of every researcher experimenting with the NASA data sets, as when unseen data is presented to the classifiers, performance may plummet from the expected.

If researchers believe that repeated data points are a correct representation of the software system (i.e. not noise), there are two options available. Firstly, it is possible to use data containing repeated data points, *so long as there are no common instances shared between training and testing sets*. This may lead researchers to designate the task of ensuring no seen data is contained within testing sets to machine learning software (i.e. during the data separation process; for example during cross-validation). Note however that this complicates the task of stratification. The second option, proposed in (Boetticher 2006), is to use an extra attribute: *number of duplicates*. This will help to ensure that information is not lost, and is most useful when data sets are believed to be of high quality.

A possible reason why there are so many repeated (and inconsistent) data points within the NASA data sets is because of the poor differential power of the metrics used. It may be highly beneficial in future to also record lower level metrics (such as character counts), as these will help to distinguish non-identical modules, reducing the likelihood of modules sharing identical metrics.

Data quality is very important during any data mining experiment, time spent analysing data is time well spent. We believe the data cleansing process defined in this paper will ensure that the NASA data sets become suitable for machine learning. This process may also be a good starting point when using other software fault data sets. Experiments based on the NASA data sets which included the repeated data points may have led to erroneous findings. Future work may be required to (where possible) repeat these studies with appropriately processed data. Other areas of future work include extending the list of integrity rules described in stage 4 of the cleansing process, and analysing other fault data sets to see whether the proportions of repeated data points in the NASA data sets are typical of fault data sets in general.

References

Bezerra, M., Oliveira, A. & Meira, S. (2007), A constructive rbf neural network for estimating the probability of defects in software modules, pp. 2869–2874.

Boetticher, G. (2006), Improving credibility of machine learner models in software engineering, *in* 'Advanced Machine Learner Applications in Software Engineering', Software Engineering and Knowledge Engineering.

Chawla, N. V., Japkowicz, N. & Kolcz, A. (2004), 'Special issue on learning from imbalanced datasets', *SIGKDD Explor. Newsl.* **6**(1).

Elish, K. O. & Elish, M. O. (2008), 'Predicting defect-prone software modules using support vector machines', *J. Syst. Softw.* **81**(5), 649–660.

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

Jiang, Y. & Cukic, B. (2009), Misclassification cost-sensitive fault prediction models, *in* 'Proceedings of the 5th International Conference on Predictor Models in Software Engineering', PROMISE '09, ACM, New York, NY, USA, pp. 20:1–20:10.

Jiang, Y., Cukic, B. & Menzies, T. (2008), Can data transformation help in the detection of fault-prone modules?, *in* 'DEFECTS '08: Proceedings of the 2008 workshop on Defects in large software systems', ACM, New York, NY, USA, pp. 16–20.

Kaminsky, K. & Boetticher, G. (2004), Building a genetically engineerable evolvable program (GEEP) using breadth-based explicit knowledge for predicting software defects, pp. 10–15 Vol.1.

Khoshgoftaar, T. M. & Allen, E. B. (2003), 'Ordering fault-prone software modules', *Software Quality Control* **11**, 19–37.

Khoshgoftaar, T. M. & Seliya, N. (2004), The necessity of assuring quality in software measurement data, *in* 'METRICS '04: Proceedings of the Software Metrics, 10th International Symposium', IEEE Computer Society, Washington, DC, USA, pp. 119–130.

Kim, S., Zimmermann, T., Pan, K. & Whitehead, E. J. J. (2006), Automatic identification of bug-introducing changes, *in* 'ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering', IEEE Computer Society, Washington, DC, USA, pp. 81–90.

Kołcz, A., Chowdhury, A. & Alspector, J. (2003), Data duplication: an imbalance problem?, *in* 'ICML 2003 Workshop on Learning from Imbalanced Datasets,'.

Koru, A. G. & Liu, H. (2005), 'An investigation of the effect of module size on defect prediction using static measures', *ACM SIGSOFT Software Engineering Notes* **30**(4), 1–5.

Lessmann, S., Baesens, B., Mues, C. & Pietsch, S. (2008), 'Benchmarking classification models for software defect prediction: A proposed framework and novel findings', *Software Engineering, IEEE Transactions on* **34**(4), 485–496.

Li, Z. & Reformat, M. (2007), 'A practical method for the software fault-prediction', *Information Reuse and Integration, 2007. IRI 2007. IEEE International Conference on* pp. 659–666.

Liebchen, G. A. & Shepperd, M. (2008), Data sets and data quality in software engineering, *in* 'PROMISE '08: Proceedings of the 4th international workshop on Predictor models in software engineering', ACM, New York, NY, USA, pp. 39–44.

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

Mende, T. & Koschke, R. (2009), Revisiting the evaluation of defect prediction models, *in* 'Proceedings of the 5th International Conference on Predictor Models in Software Engineering', PROMISE '09, ACM, New York, NY, USA, pp. 7:1–7:10.

Menzies, T., Dekhtyar, A., Distefano, J. & Greenwald, J. (2007), 'Problems with precision: A response to "comments on 'data mining static code attributes to learn defect predictors'"', *IEEE Transactions on Software Engineering* **33**, 637–640.

Menzies, T., Greenwald, J. & Frank, A. (2007), 'Data mining static code attributes to learn defect predictors', *Software Engineering, IEEE Transactions on* **33**(1), 2–13.

Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y. & Bener, A. (2010), 'Defect prediction from static code features: current results, limitations, new approaches', *Automated Software Engg.* **17**(4), 375–407.

Menzies, T. & Stefano, J. S. D. (2004), 'How good is your blind spot sampling policy?', *High-Assurance Systems Engineering, IEEE International Symposium on* **0**, 129–138.

Menzies, T., Stefano, J. S. D., Orrego, A. & Chapman, R. (2004), 'Assessing predictors of software defects', *Proceedings of Workshop on Predictive Software Models* .

Nickerson, A. S., Japkowicz, N. & Milios, E. (2001), Using unsupervised learning to guide resampling in imbalanced data sets, *in* 'In Proceedings of the Eighth International Workshop on AI and Statitsics', pp. 261–265.

Pelayo, L. & Dick, S. (2007), Applying novel resampling strategies to software defect prediction, pp. 69–72.

Seliya, N., Khoshgoftaar, T. & Zhong, S. (2005), Analyzing software quality with limited fault-proneness defect data, pp. 89–98.

Tosun, A. & Bener, A. (2009), Reducing false alarms in software defect prediction by decision threshold optimization, *in* 'Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement', ESEM '09, IEEE Computer Society, Washington, DC, USA, pp. 477–480.

Turhan, B., Menzies, T., Bener, A. B. & Di Stefano, J. (2009), 'On the relative value of cross-company and within-company data for defect prediction', *Empirical Softw. Engg.* **14**, 540–578.

Witten, I. H. & Frank, E. (2005), *Data Mining: Practical Machine Learning Tools and Techniques*, Morgan Kaufmann Series in Data Management Systems, second edn, Morgan Kaufmann.

Appendix D

International Conference on Evaluation and Assessment in Software Engineering 2011: Further Thoughts on Precision

Further Thoughts on Precision

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

***Abstract*—**
Background: There has been much discussion amongst automated software defect prediction researchers regarding use of the *precision* and *false positive rate* classifier performance metrics.
Aim: To demonstrate and explain why failing to report *precision* when using data with highly imbalanced class distributions may provide an overly optimistic view of classifier performance.
Method: Well documented examples of how dependent class distribution affects the suitability of performance measures.
Conclusions: When using data where the minority class represents less than around 5 to 10 percent of data points in total, failing to report *precision* may be a critical mistake. Furthermore, deriving the *precision* values omitted from studies can reveal valuable insight into true classifier performance.

I. Introduction

It is surprisingly difficult to characterise appropriately the performance of data mining classification algorithms from the field of machine learning. Deciding which performance measures to use involves taking several factors into account, including the costs associated with misclassification, and the class distribution. We believe the inappropriate use of classifier performance measures to be a current problem in the reporting of defect prediction research, and this short paper explains why.

In January 2007 the Menzies et al. paper *'Data Mining Static Code Attributes to Learn Defect Predictors'* was published in the IEEE Transactions on Software Engineering (Menzies, Greenwald & Frank 2007). In this study many defect prediction experiments were carried out. Classifier performance was reported using the two metrics used in *receiver operating characteristic (ROC)* analysis, the *true positive rate* and the *false positive rate*. In the journal these metrics were referred to as the *probability of detection (pd)* and the *probability of false alarm (pf)*, respectively. The use of these metrics motivated a comments paper by Zhang and Zhang (Zhang & Zhang 2007). They argued that the prediction "models built in (Menzies, Greenwald & Frank 2007) are not satisfactory for practical use". This was because the *precision;* the proportion of modules predicted as being defective which were also originally labeled as being defective, was low for 7 out of the 8 data sets (between 2.02 and 31.55 percent). The authors conclude by suggesting that, for reporting on the performance of software defect prediction models, the true positive rate be used with precision rather than with the false positive rate.

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

The Zhang and Zhang comments paper motivated a response by two of the original journal authors and two others (Menzies, Dekhtyar, Distefano & Greenwald 2007). Here the main arguments were that "detectors learned in the domain of software engineering rarely yield high precision", and that low precision predictors can be useful in practice. While it is true that low precision predictors can be useful in certain contexts, and that lowering precision in order to increase the true positive rate may be desirable depending on your objectives, it is clearly inappropriate in a classification domain to disregard precision completely.

In this paper we demonstrate that when using data with a highly imbalanced class distribution, relying on true positive rates and false positive rates alone (this includes ROC analysis) may provide an overly optimistic view of classifier performance. We demonstrate this by showing that even when pairs of values for these measures appear to be near optimal, there is still considerable room for improvement in practical terms. This is not a novel finding. However, many defect prediction researchers have continued to report their classification results inappropriately since the publication of (Zhang & Zhang 2007). The contribution made here is the intuitive and easily comprehensible presentation of the examples given in Section III.

The rest of this paper is laid out as follows: Section II provides a background to machine learning classifier performance metrics. Section III describes the problem at hand, and why precision is required to appropriately describe classifier performance in highly imbalanced domains. Our conclusions and advice for researchers is presented in Section IV.

II. Background

This section presents an overview of machine learning classifier performance metrics. In this study we limit our scope to that of binary classification problems. For each data point predicted during binary classification, there can be only one of four possible outcomes:

- A *true positive (TP)* occurs when a data point labeled as positive (typically *'defective'* in this domain) is correctly predicted as positive.
- A *true negative (TN)* occurs when a data point labeled as negative (typically *'non-defective'* in this domain) is correctly predicted as negative.
- A *false positive (FP)* occurs when a negative labeled data point is incorrectly predicted as positive.
- A *false negative (FN)* occurs when a positive labeled data point is incorrectly predicted as negative.

These values can be put into a *confusion matrix* (Figure 1).

	labeled positive	labeled negative
predicted positive	TP	FP
predicted negative	FN	TN

Fig. 1. A confusion matrix.

It is worth pointing out that there is a symmetry between the positive and negative classes. However, the positive class typically refers to the class of most interest (*'defective'* modules), which is commonly (in this domain and many others) the minority class.

Useful data statistics and commonly used classifier performance metrics can be derived from a confusion matrix. A subset of these are defined in Table I. Note that in this table, the last two measures defined (*f-measure* and *balance*) are in their most commonly used form. It is however possible to weight them in order to favour either of their comprising measures (Jiang, Cukic & Ma 2008). Additionally note that the *balance* measure was defined in (Menzies, Greenwald & Frank 2007), it is a measurement of distance from a point on a ROC curve to the ideal point, which is typically defined as where the true positive rate is 1 and the false positive rate is 0.

The three measures of most interest in this paper are: the true positives rate, the false positive rate, and precision. The true positive rate describes the proportion of data points labeled as positive which were correctly predicted as such; the optimal value is 1. The false positive rate describes the proportion of data points labeled as negative which were incorrectly predicted as positive; the optimal value is 0. Precision describes the proportion of modules predicted as defective which were correct; the optimal value is 1.

In addition to observing classifier performance with a fixed set of parameters, it may also be desirable to observe how performance varies across a range of parameters. Doing so can be especially beneficial when performing classifier comparisons. ROC curve analysis is commonly used for this task, and involves exploring the relationship between the true positive rate and the false positive rate of a classifier while (typically) varying its *decision threshold*. A trade-off commonly exists between the true positive rate and the false positive rate, and this is demonstrated by ROC analysis via a two dimensional plot of the false positive rate on the x-axis and the true positive rate on the y-axis. The area under the ROC curve (AUC-ROC) is commonly used to summarise a ROC curve in a single measure. The optimal AUC-ROC value is 1.

Precision and recall (PR) curves can be used in the same manner as ROC curves. On a *PR curve*, *recall* is on the x-axis and precision on the y-axis. A trade-off commonly exists between these two measures, and is thus shown on a PR curve. Note that recall is another alias for the true positive rate used in ROC analysis. PR curves are "an alternative to ROC curves for tasks with a large skew in the class distribution" (Davis & Goadrich 2006). The area under the PR curve (AUC-PR) can be computed and used similarly to AUC-ROC.

Alias / Aliases	**Defined As**
Testing Set No. Instances	$TP + TN + FP + FN$
No. Instances in Class 1 (Positive Class)	$TP + FN$
No. Instances in Class 2 (Negative Class)	$TN + FP$
Accuracy **+** Correct Classification Rate 1 - Error Rate	$\frac{TP + TN}{TP + TN + FP + FN}$
Error Rate **-** Incorrect Classification Rate 1 - Accuracy	$\frac{FP + FN}{TP + TN + FP + FN}$
True Positive Rate **+** Recall Sensitivity Probability of Detection (pd) 1 - False Negative Rate	$\frac{TP}{TP + FN}$
True Negative Rate **+** Specificity 1 - False Positive Rate	$\frac{TN}{TN + FP}$
False Positive Rate **-** Type 1 Error Rate Probability of False Alarm (pf) 1 - True Negative Rate	$\frac{FP}{FP + TN}$
False Negative Rate **-** Type 2 Error Rate 1 - True Positive Rate	$\frac{FN}{FN + TP}$
Precision **+**	$\frac{TP}{TP + FP}$
F-Measure **+** F-Score	$\frac{2 * Recall * Precision}{Recall + Precision}$
Balance **+** Distance from ROC optimal point	$1 - \frac{\sqrt{(0-pf)^2+(1-pd)^2}}{\sqrt{2}}$

TABLE I
A SUBSET OF STATISTICS DERIVED FROM A CONFUSION MATRIX. MEASURES MARKED WITH '+' HAVE AN OPTIMAL VALUE OF 1. MEASURES MARKED WITH '-' HAVE AN OPTIMAL VALUE OF 0.

III. WHY CLASS DISTRIBUTION AFFECTS THE SUITABILITY OF MEASURES

Consider a perfectly balanced data set with 1000 data points, 500 in each class. If we achieve a classification performance of *true positive rate (TPR)* = 1 and *false positive rate (FPR)* = 0.01, it appears as though our classifier has performed very well. All of the data points in the positive class have been correctly classified (TPR = 1), and only 1 percent of data points in the negative class (5 data points) have been incorrectly classified (FPR = 0.01). If we calculate the precision of such a classifier, it works out to 0.99 (to two significant figures). Thus 99 percent of data points predicted to be in the positive class turned out to be correct. In this first example, where we have a balanced class distribution, using the TPR and FPR provided an honest and accurate representation of classifier performance, as a near optimal pair of values likewise resulted in a near optimal precision. A confusion matrix for this example is presented in Figure 2.

	labeled positive	labeled negative
predicted positive	TP = 500	FP = 5
predicted negative	FN = 0	TN = 495

Fig. 2. TPR = 1, FPR = 0.01, Precision = 0.99

Now consider a data set with a highly imbalanced class distribution, 1000 data points in total but with only 10 data points in the positive class (1 percent). If we again achieve TPR = 1 and FPR = 0.01, classifier performance appears to be equal to that of the previous classifier. This is inappropriate, misleading, and exaggerates the classifiers real performance, as the precision of this classifier is 0.50 as opposed to 0.99. Thus, *because of the imbalanced class distribution (i.e. much more data in one class than the other),* the same supposedly near optimal TPR and FPR (or point on a ROC curve) represented vastly different performance. In the first example 99 of every 100 positive predictions were correct, whereas in the second example, the same TPR and FPR represented a classifier where only half of all positive predictions were correct. A confusion matrix for this example is presented in Figure 3.

	labeled positive	labeled negative
predicted positive	TP = 10	FP = 10
predicted negative	FN = 0	TN = 980

Fig. 3. TPR = 1, FPR = 0.01, Precision = 0.50

Turning attention toward the domain of software defect prediction, researchers often use data sets where the minority class represents less than 1 percent of data points in total (see (Menzies, Greenwald & Frank 2007), (Lessmann, Baesens, Mues & Pietsch 2008) and (Jiang & Cukic 2009), for example). We now present an example using the most imbalanced of the NASA Metrics Data Program data sets[1], *PC2*. This data set contains 5589 data points, each consisting of module product metrics and the associated fault data. Just 23 of the data points are labeled as *'defective',* 0.4 percent of data points in total. Thus, with a TPR of 1 all 23 data points labeled as *'defective'* would be correctly classified. An FPR of 0.01 means that 1 percent of the 5566 data points labeled as *'non-defective'* were incorrectly classified. With approximately 56 false positives, a total of 79 modules would be predicted to require further attention. This works out to a precision of 0.29 (to two significant figures), despite the other metrics implying near optimal performance. A confusion matrix for this example is presented in Figure 4.

	labeled positive	labeled negative
predicted positive	TP = 23	FP = 56
predicted negative	FN = 0	TN = 5510

Fig. 4. TPR = 1, FPR = 0.01, Precision = 0.29

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

In this domain, where the cost of false positives is typically not prohibitively large (see (Menzies, Dekhtyar, Distefano & Greenwald 2007)), such a classifier would, in most environments, be very attractive. Observe however that because of the highly imbalanced class distribution, small changes in the FPR have a large effect on the actual number of false positives. Thus if classifier performance changes to TPR = 1 and FPR = 0.05, the number of false positives increases to 280. The precision in turn drops to 0.08 (to two significant figures). In some environments examining 100 modules to find 8 of them to be defective would not be feasible. A confusion matrix for this example is presented in Figure 5.

	labeled positive	labeled negative
predicted positive	TP = 23	FP = 280
predicted negative	FN = 0	TN = 5286

Fig. 5. TPR = 1, FPR = 0.05, Precision = 0.08

As defect predictors do not achieve performance even close to TPR = 1 and FPR = 0.05, are these theoretical experiments valid? Zhang and Zhang (Zhang & Zhang 2007) indirectly answer this question by highlighting the precision achieved in (Menzies, Greenwald & Frank 2007) on data set *PC2*. Here, despite results appearing to be acceptable (TPR = 0.72, FPR = 0.14), and claims of the naive Bayes classifiers being "demonstrably useful", the precision was just 2.02 percent. The classifier predicted that approximately 796 modules were in the 'defective' class, of which only approximately 17 actually were. This highlights the poor predictive performance achieved, and raises the question of whether such a classifier could be of any practical worth. An optimistic approximation of the confusion matrix for the entire data set (the results in the study were generated after 10 repeated runs of 10-fold cross-validation) is presented in Figure 6.

	labeled positive	labeled negative
predicted positive	TP = 17	FP = 779
predicted negative	FN = 6	TN = 4787

Fig. 6. TPR = 0.74, FPR = 0.14, Precision = 0.02

Note that an identical confusion matrix to the one in Figure 6 can be obtained by simply ranking all data points in data set *PC2* by their *lines of code total* attribute (descending order), and predicting the first $n = 796$ data points only as defective. It is a similar case for 2 more of the 8 data sets used in (Menzies, Greenwald & Frank 2007), where n is the approximate average number of *'defective'* predictions made. This is known as *LOC module-order modelling* (see (Khoshgoftaar & Allen 2003) and (Mende & Koschke 2009)), and highlights both the poor predictive performance of the classifiers, and that research into making defect predictors 'effort aware' is worthwhile (see (Arisholm, Briand & Fuglerud 2007) and (Mende & Koschke 2010)).

Table II presents statistics for each of the 13 NASA Metrics Data Program data sets. These data sets were chosen as they have been heavily used in software defect prediction research.

NASA Data Set Alias	No. Data Points	No. Positive (Minority) Class Data Points	Percentage of Positive Class Data Points	Percentage Precision @ TPR=1, FPR=0.01	Percentage Precision @ TPR=1, FPR=0.05
PC2	5589	23	0.4	29.11	7.64
MC1	9466	68	0.7	41.98	12.64
PC5	17186	516	3.0	75.55	38.22
PC1	1107	76	6.9	88.37	59.38
MW1	403	31	7.7	88.57	62.00
KC3	458	43	9.4	91.49	67.19
CM1	505	48	9.5	90.57	67.61
PC3	1563	160	10.2	91.95	69.57
PC4	1458	178	12.2	93.19	73.55
KC1	2107	325	15.4	94.75	78.50
JM1	10878	2102	19.3	95.98	82.72
MC2	161	52	32.3	98.11	91.23
KC4	125	61	48.8	98.39	95.31

TABLE II
EACH OF THE NASA METRICS DATA PROGRAM DATA SETS RANKED IN ASCENDING ORDER OF PERCENTAGE DATA POINTS IN MINORITY CLASS.

The table shows class distribution details for each data set as well as the precision when TPR = 1, FPR = 0.01 and when TPR = 1, FPR = 0.05. Note that the false positives in these calculations were rounded to the nearest integer. The data sets are ranked in ascending order of the percentage of modules in the positive (minority) class. From the table it can be seen that for the three data sets with the highest class imbalance especially, near optimal values of TPR and FPR results in classifiers which are far from optimal in practical terms, or in terms of precision.

Thus, relying on TPR and FPR or methods based around them, including: ROC analysis, AUC-ROC, and the *balance* metric, "can present an overly optimistic view of an algorithm's performance if there is a large skew in the class distribution" (Davis & Goadrich 2006). *Precision is required to give a more accurate representation of true performance in this context.*

IV. CONCLUSIONS

Precision matters, especially when class distributions are highly skewed. When performing any kind of data mining experiment we believe it is very important to document the characteristics of the data being used: where it came from, what pre-processing has been carried out, how many data points and features are present, what is the class distribution, etc. This makes it more accessible for other researchers to check the validity of the claimed results. For example; if such data characteristics are given, it is often possible to derive measures that are not explicitly reported, as done here. The inspiration for the work carried out here came from (Zhang & Zhang 2007), where precision values omitted in (Menzies, Greenwald & Frank 2007) were derived using the TPR, FPR, and class distribution data.

If classifier performance is to be reported with a single set of (hopefully validated and thus suitable) parameters, we believe defect prediction researchers should be reporting a minimum of recall (TPR) and precision, in addition to the data characteristics just described. It is of no harm to also report the false positive rate. Note that it is necessary to take both recall and precision into account when accessing performance, a single one of these measures will not suffice. This is because an optimal recall can be achieved by simply predicting all data points as belonging to the positive class, and an optimal precision can be achieved by only making a single positive prediction, which turns out to be correct. The f-measure (see Table I) is commonly used to combine both measures into a single value, and can simplify performance quantification. When classifier performance is to be reported over a range of parameters, we believe precision and recall (PR) curves to be more suitable in this domain than ROC curves. This is because class distributions are often highly skewed (Menzies, Dekhtyar, Distefano & Greenwald 2007).

Lessmann et al. carried out a large scale benchmarking defect prediction experiment with 22 classifiers (Lessmann et al. 2008). The top 17 of these classifiers were reported to have statistically indistinguishable performance using the AUC-ROC performance measure, and a statistical approach proposed by (Demšar 2006). The data used in the study came from 10 of the NASA Metrics Data Program data sets. Davis and Goadrich in (Davis & Goadrich 2006) point out that with highly imbalanced data sets, PR-curves are more powerful at distinguishing the performance of classification methods than ROC curves. Thus, we think it would be interesting for the experiment by Lessmann et al. to be replicated with PR-curves and AUC-PR.

In a recent paper by Menzies et al. (Menzies, Milton, Turhan, Cukic, Jiang & Bener 2010), it is stated that the "standard learning goal" of defect predictors it to maximize AUC-ROC. We would argue that this should not be the standard learning goal of defect predictors. As shown here, by (Davis & Goadrich 2006) and by (Zhang & Zhang 2007), precision is required in a typically imbalanced domain. Moreover, (Davis & Goadrich 2006) prove "that an algorithm that optimizes the area under the ROC curve is not guaranteed to optimize the area under the PR curve".

In addition to the comments made about defect predictor learning goals, the Menzies et al. paper also states "that accuracy and precision are highly unstable performance indicators for data sets ... where the target concept occurs with relative infrequency". While it is commonly reported within the data mining literature that accuracy is not suitable for imbalanced data sets, the same can not be said (at all) for precision. This is true other than in the context of experiments to explore the *class imbalance problem* (see (Batista, Prati & Monard 2004)), where the FPR is better suited than precision.

REFERENCES

Arisholm, E., Briand, L. C. & Fuglerud, M. (2007), Data mining techniques for building fault-proneness models in telecom java software, *in* 'ISSRE '07: Proceedings of the The 18th IEEE International Symposium on Software Reliability', IEEE Computer Society, Washington, DC, USA, pp. 215–224.

Batista, G. E. A. P. A., Prati, R. C. & Monard, M. C. (2004), 'A study of the behavior of several methods for balancing machine learning training data', *SIGKDD Explor. Newsl.* **6**, 20–29.

Davis, J. & Goadrich, M. (2006), The relationship between precision-recall and roc curves, *in* 'Proceedings of the 23rd international conference on Machine learning', ICML '06, ACM, New York, NY, USA, pp. 233–240.

Demšar, J. (2006), 'Statistical comparisons of classifiers over multiple data sets', *J. Mach. Learn. Res.* **7**, 1–30.

Jiang, Y. & Cukic, B. (2009), Misclassification cost-sensitive fault prediction models, *in* 'Proceedings of the 5th International Conference on Predictor Models in Software Engineering', PROMISE '09, ACM, New York, NY, USA, pp. 20:1–20:10.

Jiang, Y., Cukic, B. & Ma, Y. (2008), 'Techniques for evaluating fault prediction models', *Empirical Softw. Engg.* **13**(5), 561–595.

Khoshgoftaar, T. M. & Allen, E. B. (2003), 'Ordering fault-prone software modules', *Software Quality Control* **11**, 19–37.

Lessmann, S., Baesens, B., Mues, C. & Pietsch, S. (2008), 'Benchmarking classification models for software defect prediction: A proposed framework and novel findings', *Software Engineering, IEEE Transactions on* **34**(4), 485–496.

Mende, T. & Koschke, R. (2009), Revisiting the evaluation of defect prediction models, *in* 'Proceedings of the 5th International Conference on Predictor Models in Software Engineering', PROMISE '09, ACM, New York, NY, USA, pp. 7:1–7:10.

Mende, T. & Koschke, R. (2010), 'Effort-aware defect prediction models', *Software Maintenance and Reengineering, European Conference on* **0**, 107–116.

Menzies, T., Dekhtyar, A., Distefano, J. & Greenwald, J. (2007), 'Problems with precision: A response to "comments on 'data mining static code attributes to learn defect predictors'"', *IEEE Transactions on Software Engineering* **33**, 637–640.

Menzies, T., Greenwald, J. & Frank, A. (2007), 'Data mining static code attributes to learn defect predictors', *Software Engineering, IEEE Transactions on* **33**(1), 2–13.

Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y. & Bener, A. (2010), 'Defect prediction from static code features: current results, limitations, new approaches', *Automated Software Engg.* **17**(4), 375–407.

Zhang, H. & Zhang, X. (2007), 'Comments on "data mining static code attributes to learn defect predictors"', *IEEE Trans. Softw. Eng.* **33**, 635–637.

Appendix E

Selected Papers Special Issue on Evaluation and Assessment in Software Engineering 2011: Reflections on the NASA MDP Data Sets

Reflections on the NASA MDP data sets

D. Gray D. Bowes N. Davey Y. Sun B. Christianson

Computer Science Department, University of Hertfordshire, UK
E-mail: d.gray@herts.ac.uk

Abstract: Background: The NASA metrics data program (MDP) data sets have been heavily used in software defect prediction research. Aim: To highlight the data quality issues present in these data sets, and the problems that can arise when they are used in a binary classification context. Method: A thorough exploration of all 13 original NASA data sets, followed by various experiments demonstrating the potential impact of duplicate data points when data mining. Conclusions: Firstly researchers need to analyse the data that forms the basis of their findings in the context of how it will be used. Secondly, the bulk of defect prediction experiments based on the NASA MDP data sets may have led to erroneous findings. This is mainly because of repeated/duplicate data points potentially causing substantial amounts of training and testing data to be identical.

1 Introduction

Modern software defect prediction research typically involves classification and/or regression algorithms being used to predict the presence of non-syntactic implementational errors in software source code. To make these predictions such algorithms attempt to generalise upon software fault data, observations of software product and/or process metrics coupled with a ‘level of defectiveness’ value. This value typically takes the form of a ‘number of faults reported’ metric, for a given software unit after a given amount of time (either post code development or system deployment). Note that in this paper we use the following terms interchangeably: defect, error, fault and bug.

The predictions made by defect predictors may be continuous (level of defectiveness) or categorical [often set membership of either: (‘defective’, ‘non-defective’)]. The current trend by researchers is typically to report the latter (categorical predictions) only. However, this may not be the best approach, as continuous predictions allow software units to be ranked according to their predicted quality factor (this is known as module-order modelling, see [1]). A software quality ranking system has the real-world benefit of producing an ordered list of the seemingly most error-prone code units. These code units can then be subjected to some form of further inspection, in descending order of predicted level of defectiveness, for as long as resources allow.

In order to carry out a software defect prediction experiment there is a requirement for reasonable quality data. However, software fault data is very difficult to obtain. Commercial software development companies often do not have a fault measurement program in place. Moreover, even if such a program is in place, it is typically undesirable from a business perspective to publicise fault data. This is particularity true for systems where quality has been a serious problem, that is, where it would be most useful to publicise such data and give researchers an opportunity to discover why this was the case.

Open-source systems are frequently used by researchers to construct their own fault data sets [2–6]. Such systems are often developed while using a bug-tracking system to record the faults encountered by developers. If bug information has been correctly and consistently entered into version control commit messages, it is then possible to autonomously locate the fault-fixing revisions. From here it is possible to (fairly accurately) map fault-fixing revisions back to where the fault was first introduced (the bug-introducing change, see [5–7]). The major problem with constructing fault data from open-source systems is that it can be a very time consuming task to do accurately. This is because human intervention is often required to check the validity of the automated mappings. For systems of even moderate size, the quantity of these mappings can make this task infeasible.

Thus difficulty in obtaining software fault data is the major factor why public domain fault data repositories, such as those hosted by NASA and PROMISE, have become so popular among researchers. These repositories host numerous data sets, which require no data analysis and little or no pre-preprocessing, before machine learning tools such as Weka will classify them. The ease of this process can be dangerous to the inexperienced researcher. Results can be obtained without any scrutiny of the data. Furthermore, researchers may naively assume the NASA metrics data program (MDP) data sets are of reasonable quality for data mining. This issue is worsened by the hosting sites not indicating the main problems, and by so many researchers using these data sets inappropriately. The aim of this study is therefore to illuminate: the data quality issues present in these data sets, and the problems that can arise when they are used (as they often are) in a binary classification context. It is hoped that this study will encourage researchers to take data quality seriously, and to question the results of some studies based on these data sets.

The NASA MDP Repository was previously available at http://mdp.ivv.nasa.gov/ and is currently available in a more basic, less documented form at http://filesanywhere.com/fs/v.aspx?v=896a648c5e5e6f799b. The repository currently contains 13 module-level data sets explicitly intended for software metrics research. Each data set represents a NASA software system/subsystem and contains static code metrics and fault data for each comprising module. Note that 'module' in this case refers to either a function, procedure or method. The static code metrics recorded include lines-of-code (LOC)-count, Halstead [8] and McCabe [9]-based measures. The primary fault data takes the form of an error-count metric, which was reportedly calculated from the number of error reports issued for each module via a bug-tracking system. From the details given at the original NASA MDP Repository, it is unclear precisely how these error reports were mapped back to the individual modules; however, it was stated that 'If a module is changed due to an error report (as opposed to a change request), then it receives a one up count. It cannot receive more than a one up for a given error report.' It was also stated that the error-count metric describes 'the number of changes due to error'. The originating source code for these data sets is entirely closed-source, making the validation of data integrity more difficult. A substantial amount of research based wholly or partially on these data sets has been published over the last decade, including [10–55]. Note that in this study we focus solely on the data sets just described, because they are widely used; we do not make use of the available object-oriented metrics data set, or any other data set affiliated with NASA.

The most common usage for the NASA data sets (as reported in the literature) is in binary classification experiments. Typically, a classifier is trained on binary-labelled data, and then each new set of module metrics is predicted as belonging to either a 'faulty' module, or a 'non-faulty' module. This is clearly a huge simplification of the real world, for two main reasons. Firstly, fault quantity is disregarded: there is typically no distinction between a module with one reported fault and a module with 31 reported faults, they are both simply labelled as 'faulty'. Secondly, fault severity is disregarded: there is typically no distinction between a trivial fault and a life-threatening fault. Despite these crude simplifications, binary classification defect prediction studies continue to be very prolific.

It is widely accepted by the data mining community that in order to accurately assess the potential real-world performance of a classification model, the model must be tested against entirely different data from that upon which it was trained [56]. This is why there is a distinction between a training set and a testing set. A testing set is also referred to as an independent test set, as it is intended to be independent from the training set (i.e. models should be tested against 'unseen data', see [56]). This is very basic data mining knowledge, and is no surprise to the defect prediction community. In 2004 Menzies *et al.* [11] state: 'if the goal of learning is to generate models that have some useful future validity, then the learned theory should be tested on data not used to build it. Failing to do so can result in a excessive over-estimate of the learned model...'. Despite this fact being well-known, numerous studies based on the NASA MDP data sets (henceforth, NASA data sets) have potentially had high proportions of data points common to both their training and testing sets. This is because the NASA data sets contain varied quantities of repeated data points, observations of module metrics and their corresponding fault data occurring more than once. Thus, when this data is used in a classification context, the separation into training and testing sets may result in both sets containing large proportions of common data points. This can yield the aforementioned excessive estimate of performance, as classifiers can memorise rather than generalise. This is very serious, as when data mining 'it is important that the test data was not used in any way to create the classifier' [56].

In this study we thoroughly analyse all 13 of the original NASA data sets. We are interested in data quality in terms of noise, inaccurate/incorrect data (see [57]). Additionally, because these data sets are typically used in binary classification experiments, we are also interested in the issues specific to this context. Firstly, we highlight the basic data quality problems via our novel data cleansing process. This process is for removing noise, and for preparing the data sets for binary classification. Next, we present the more complex issues still remaining after the cleansing process, including the issue of repeated data points. We discuss at length the potential problems caused by repeated data points when data mining, and why using lower level metrics (such as character counts) in fault data sets may alleviate these problems, by helping to distinguish non-identical modules.

The rest of this paper is presented as follows: in the next section we discuss related work, papers where problems with the NASA data sets have been documented or discussed. In Section 3 we document our novel data cleansing process in incremental stages. Section 4 contains a discussion of the issues still remaining after our cleansing process, including a demonstration of the potential effects of repeated data points during classification experiments. A new method to address these issues is proposed at the end of Section 4. Our conclusions are presented in Section 5.

2 Related studies

The major problem when using the NASA data sets in a classification context is that repeated data points may result in training data inadvertently being included in testing sets, potentially invalidating the experiment. This is not a new finding; however, we believe it needs spelling out to researchers, as previous studies mentioning this issue seem to have been ignored. In this section the most relevant studies surrounding this issue are discussed.

The earliest mention of repeated data points in NASA data sets that we can find was made in [58]. The authors state that they eliminated 'redundant data', but give no further explanation as to why. The data set used was NASA data set KC2, which is not available from the NASA MDP Repository. Although this data set is currently available from the PROMISE Repository, we did not use it in our study in an effort to use only the original, unmodified data.

In [33] five NASA data sets were used in various classification experiments. The author states that 'data pre-processing removes all duplicate tuples from each data set along with those tuples that have questionable values (e.g. LOC equal to 1.1).' Interestingly, it is only the PROMISE versions of the NASA data sets that contain these clearly erroneous non-integer LOC-count values. The author goes into detail on repeated data points, stating that 'to avoid building artificial models, perhaps the best approach would be not to allow duplicates within datasets'. One of the experiments carried out was intended to show the effect of the repeated data points in the five NASA data sets used. This was in a 10-fold cross-validation classification experiment with a C4.5 decision tree. The claimed result was that the data sets with the repeats included achieved significantly better performance than those without. Although this result is to be expected, there was an unfortunate technical shortcoming in the experimental design. When reporting the performance of classifiers on test sets with imbalanced class distributions, 'accuracy' (or its inverse: 'error rate') should not be used [59]. In addition to this, care is required when performing such an experiment, as the proportion of repeated data points in each class is not related to the class distribution. Therefore, post the removal of repeated data points, the data sets could have substantially different class distributions. This may boost or reduce classifier performance, because of the class imbalance problem (see [60, 61]).

Classification experiments utilising probabilistic outputs were carried out in [39]. Here the authors used five of the original NASA data sets and state that they removed both 'redundant and inconsistent patterns'. Inconsistent data points are another of the problems when data mining with the NASA data sets. They occur when repeated feature vectors (module metrics) describe data points with differing class labels. Thus, in this domain they occur where the same set of metrics is used to describe both a module labelled as 'defective' and a module labelled as 'non-defective'. We believe the removal of such instances was first carried out in [25].

The work described here differs from that previously described, as it is not based on classification experiments. It is instead based on the analysis and cleansing of data. This study demonstrates: the poor quality of the NASA data sets; the extent to which repeated data points disseminate into training and testing sets; and the effect of testing sets containing seen data during classification experiments.

3 Method: data cleansing

The NASA data sets are available from the aforementioned NASA MDP and PROMISE repositories. For this study, we used the original versions of the data sets from the NASA MDP Repository. Note, however, that the main issues also apply to the PROMISE versions of these data sets (available at http://promisedata.org/), which are for the most part simply the same data in a different format. Also note that updated versions of the data sets have recently been uploaded at PROMISE. These new versions address many of the issues pointed out in our earlier work (see [62]).

3.1 Initial pre-processing: binarisation of class variable and removal of module identifier and extra error data attributes

In order to be suitable for binary classification, the error-count attribute is commonly reported in the literature (see [13, 29, 55] for example) as being binarised as follows

$$\text{defective?} = (\text{error\_count} \geq 1)$$

It is then necessary to remove the 'unique module identifier' attribute, as this gives no information towards the defectiveness of a module. Lastly, it is necessary to remove all other error-based attributes, to make the classification task worthwhile. This initial pre-processing is summarised in Fig. 1. As the NASA data are often reportedly used post this initial pre-processing, we present an overview of each data set in Table 1.

3.2 Stage 1: removal of constant attributes

A numeric attribute which has a constant/fixed value throughout all instances is easily identifiable as it will have a variance of zero. Such attributes contain no information with which to discern modules apart, and are at best a waste of classifier resources. Each data set had from 0 to 10% of their total attributes removed during this stage, with the exception of data set KC4. This data set has 26 constant attributes out of a total of 40, thus 65% of available data contains no information with which to train a classifier.

This stage removes data that may be genuine, but in the context of machine learning it is of no use and is therefore discarded. Regarding data set KC4, it appears as though many of the metrics have not been collected; instead of leaving them out of the data set originally however, they were instead included with all values equal to zero.

An additional note regarding data set KC4 is that two of its attributes: 'essential complexity' and 'essential density' have two unique values each, but in each case, one of the values occurs just once. This data may be valid, but after the data divide into training and testing set, it may be that the training data contains a constant attribute. This can be problematic for some learning techniques, and is therefore something that researchers should be aware of.

```
rmAttributes = [ MODULE, ERROR_DENSITY, ERROR_REPORT_IN_6_MON,
                 ERROR_REPORT_IN_1_YR, ERROR_REPORT_IN_2_YRS ]

for dataSet in dataSets:
    for rmAttribute in rmAttributes:
        if rmAttribute in dataSet:
            dataSet = dataSet - rmAttribute
    dataSet.binarise(ERROR_COUNT)
    dataSet.rename(ERROR_COUNT, DEFECTIVE)
```

Fig. 1 *Initial pre-processing pseudo-code*

Table 1 Details of the NASA data sets post initial pre-processing

Name	Language	Features	Instances	Defective instances, %
CM1	C	40	505	10
JM1	C	21	10 878	19
KC1	C++	21	2107	15
KC3	Java	40	458	9
KC4	Perl	40	125	49
MC1	C and C++	39	9466	0.7
MC2	C	40	161	32
MW1	C	40	403	8
PC1	C	40	1107	7
PC2	C	40	5589	0.4
PC3	C	40	1563	10
PC4	C	40	1458	12
PC5	C++	39	17 186	3

3.3 Stage 2: removal of repeated attributes

In addition to constant attributes, repeated attributes occur where two or more attributes have identical values for each instance. Such attributes are therefore fully correlated, which may effectively result in a single attribute being over-represented. Among the NASA data sets there are two repeated attributes (post stage 1), namely the 'number of lines' and 'loc total' attributes in data set KC4. The difference between these two metrics is poorly defined at the NASA MDP Repository. However, they may be identical for this data set as (according to the metrics) there are no modules with any lines either containing comments or which are empty. For this data-cleansing stage we removed one of the attributes so that the values were only being represented once. We chose to keep the 'loc total' attribute label as this is common to all 13 NASA data sets.

This stage again removes data that may be genuine, because it can be problematic when data mining. It is interesting that data set KC4 has had so much data removed in these first two stages. Table 1 shows that KC4 is unique in that it is the only data set based on Perl code. Therefore it may be that the metrics collection tool (McCabeIQ 7.1) was more limited in the metrics it could collect for this language.

3.4 Stage 3: replacement of missing values

Missing values may or may not be problematic for learners depending on the classification method used. However, dealing with missing values within the NASA data sets is very simple. Seven of the data sets contain missing values, but all in the same single attribute: 'decision density'. This attribute is defined as 'condition count' divided by 'decision count', and for each missing value both these base attributes have a value of zero. It therefore appears as though missing values have occurred because of a division by zero error. In the remaining data set which contains all three of the aforementioned attributes but does not contain missing values, all instances with 'condition count' and 'decision count' values of zero also have a 'decision density' of zero. Because of this we replace all missing values with zero, ensuring consistency between data sets. Note that in [39] all instances which contained missing values within the NASA data sets were discarded. It is more desirable to cleanse data than to remove it, as the quantity of possible information to learn from will thus be maximised.

This stage adds data via the replacement of missing values, because they are problematic for many learning techniques. Note, however, that some researchers may not wish to carry out this stage, if they are using a learning method that is resilient to missing values (such as naive Bayes). Additionally, some researchers may wish to exclude derived features (such as 'decision density') altogether. There is more discussion on this in Section 4.

3.5 Stage 4: enforce integrity with domain-specific expertise

The NASA data sets contain varied quantities of attributes derived from simple equations of other attributes, which are useful for checking data integrity. In addition, it is possible to use domain-specific expertise to validate data integrity, by searching for theoretically impossible occurrences. The following is a non-exhaustive list of checks that can be carried out for each data point:

- Halstead's length metric (see [8]) is defined as 'number of operators' + 'number of operands'.
- Each token that can increment a module's cyclomatic complexity (see [9]) is counted as an operator according to the original NASA MDP Repository. Therefore the cyclomatic complexity of a module should not be greater than the number of operators + 1. Note that the minimum cyclomatic complexity is 1.
- The number of function calls within a module is recorded by the 'call pairs' metric. A function call operator is counted as an operator according to the original NASA MDP Repository, therefore the number of function calls should not exceed the number of operators.

These three simple rules are a good starting point for removing noise in the NASA data sets. Any data point which does not pass all of the checks contains noise. As the original NASA software systems/subsystems from where the metrics are derived are not publicly available, it is impossible for us to investigate this issue of noise further. The most viable option is therefore to discard each offending instance. Note that a prerequisite of each check is that the data set must contain all of the relevant attributes (post stage 1). Six of the data sets had data removed during this stage, between 1 and 12% of their data points in total.

During this stage, it is possible to not only remove noise (inaccurate/incorrect data), but also problematic data. A module which (reportedly) contains no lines of code and no operands and operators should be an empty module containing no code. Should such a data point be discarded? As it is impossible for us to check the validity of the metrics against the original code, this is a grey area. An empty module may still be a valid part of a system, it may just be a question of time before it is implemented. Furthermore, a module missing an implementation may still have been called by an unaware programmer. As the module is unlikely to have carried out the task its name implies, it may also have been reported to be faulty. Despite this, researchers need to decide for themselves what to do with data that cannot be proved to be noisy, but is nonetheless strange. For example, the original data set MC1 (according to the metrics) contains 4841 modules (51% of modules in total) with no lines of code. We feel that it would therefore not be unreasonable to remove such data points, or even reject the entire data set altogether.

4 Further issues

Our data-cleansing process demonstrates issues with the NASA data sets in terms of noise (stage 4) and classification-specific problems (stages 1–3). Although there are almost certainly more noisy data points that could not be identified using such simple methods, it is difficult for us to explore this further because of the closed-source, proprietary nature of the software. However, there are additional, more serious classification-specific problems, which we now discuss.

The most well-known issue regarding use of the NASA data sets in classification experiments is that of the varied levels of class imbalance (see Table 1). The table shows that data set KC4 has an almost balanced class distribution, whereas data set PC2 has only 0.4% of data points belonging to the minority class. This is an issue that researchers should be aware of. Learning from imbalanced data is an active area of research within the data mining community, we therefore refer readers to standard texts [56, 60, 61, 63]. Note, however, that defect prediction researchers need to be very careful in the way they assess the performance of their classifiers when using highly imbalanced data (see [64–66]).

Another issue is that, as mentioned previously, there are attributes within the NASA data sets that are simple equations of other attributes. Although useful for checking data integrity, they can be problematic (or simply a waste of computational resources) depending on the learning technique used. For example, support vector machines utilising a Gaussian radial basis kernel will typically not benefit from the inclusion of such attributes, as they will be implicitly calculated. Additionally, other highly correlated attributes can be found within the data sets, which are known to harm classification performance with many learning techniques [67, 68]. Therefore in some contexts, researchers may wish to address these issues. This usually involves removing attributes during pre-processing and/or utilising a feature selection technique on the training data.

The most severe issue when using the NASA data sets for classification experiments is that of repeated data points. Unfortunately, such data points are often ignored in the defect prediction literature. Repeated, redundant or duplicate data points are data points (or instances) that appear more than once within a data set. They occur either because of a data quality problem (a faulty data collection process), or (in this domain) when many modules have the same values for all measured metrics; for example, when they have the same number of: lines of code, lines of comments, blank lines, operands, operators, unique operands, unique operators, function calls and so on. Additionally, these modules have also been assigned the same class label referring to whether they are or are not ‘defective’. This situation is clearly possible in the real world; for example, in an object-oriented system, there may be many simple accessor and mutator methods that have not been reported as faulty and share identical metrics. However, such data points may be problematic in the context of machine learning, where is it imperative that classifiers are tested upon data points independent from those used during training [56]. The issue is that when data sets containing repeated data points are split into training and testing sets (e.g. by a x% training, $100 - x$% testing split, or n-fold cross-validation), it is possible for there to be instances common to both sets. With test data included in the training data, the learning task is either simplified or reduced entirely to a task of recollection. Ultimately however, if the experiment is intended to show how well a classifier could generalise upon future, unseen data points, the results will be erroneous as the experiment is invalid. This is because the assumption of unseen data has been violated, as the test data has been contaminated with training data.

Inconsistent (or conflicting) instances are another issue, and are very similar to repeated instances in that both occur when the same feature vectors describe multiple modules. The difference between repeated and inconsistent instances is that with the latter, the class labels differ, thus (in this domain) the same metrics would describe both a ‘defective’ and a ‘non-defective’ module. This is again possible in the real world, and while not as serious an issue as the repeated instances (in the case of the NASA data sets), inconsistent data points can be problematic during binary classification tasks. When building a classifier which outputs a predicted class set membership of either ‘defective’ or ‘non-defective’, it is illogical to train such a classifier with data instructing that the same set of features is resultant in both classes. We focus more on repeated data points than inconsistent ones in this study, as for most data sets the proportion of repeated instances is considerably larger. Note, however, that it is possible for a data point to be both repeated and inconsistent.

The proportion of repeated data points in each data set (post initial pre-processing, as this is how they are most frequently used) is shown in Fig. 2. Note that in some cases the proportion is very large (89, 79 and 75% for data sets PC5, MC1 and PC2, respectively). In an earlier study [62], we recommended the removal of such instances as part of our data cleansing process, ensuring that each consistent data point is unique. This is a simple and acceptable way to address the issues caused by repeated data points, which has been carried out in prior studies (see [39, 58]). However, in this study we propose a novel and robust approach, where test sets remain unmodified. We come back to this at the end of this section, after addressing the following, more immediate questions: Why are there so many repeated data points and what can be done in future to avoid them? What proportion of seen data points could end up in testing sets if this data were to be used in classification experiments? What effect could having such quantities of seen data points in testing sets have on classifier performance? Each of these questions are addressed in the sections that follow.

4.1 *Why are there so many repeated data points and how can they be avoided in future?*

As previously stated, the NASA data sets are based on closed-source, proprietary software, so it is impossible for us to validate whether the repeated data points are truly a representation of each software system/subsystem, or whether they are noise. Despite this, a probable factor in why the repeated (and inconsistent) data points are a part of these data sets is because of the poor differential capability of the metrics used. Intuitively, 40 metrics describing each software module seem like a large set. However, many of the metrics are simple equations of other metrics. Because of this, it may be highly beneficial in future to also record lower level metrics, such as character counts. These will help to distinguish modules apart, particularly small

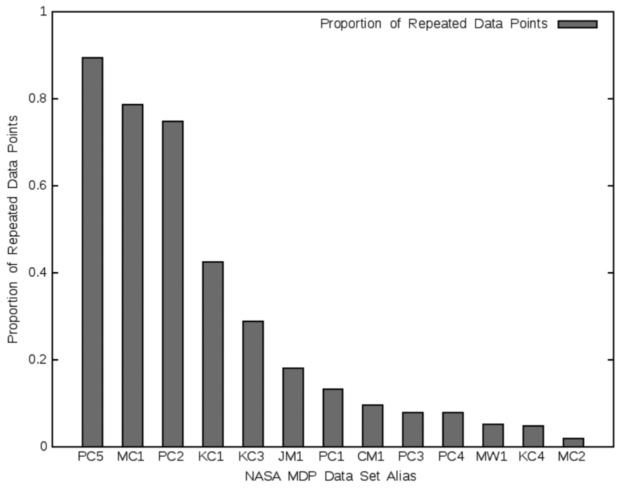


Fig. 2 *Proportion of repeated data points in each NASA data set*

modules, which statistically result in more repeated data points than large modules. Additionally, learners should be able to utilise such low-level data for helping to detect potentially troublesome modules (in terms of error-proneness).

The size of the input space in these data sets and fault data sets in general has not been previously discussed in detail to the best of our knowledge. All base attributes (attributes not derived from simple equations of other attributes) in these data sets contain only discrete values ≥ 0. Therefore the probability of a repeated data point in these data sets is much greater than, for example, a data set with real-valued measurements for the same number of attributes. This is why we believe that in future, researchers should additionally record lower level metrics (such as character counts), to alleviate the issues with repeated data points by helping to distinguish non-identical modules.

4.2 What proportion of seen data points could end up in testing sets if this data were to be used in classification experiments?

In order to find the answer to this question, a small Java program was developed utilising the Weka machine learning tool's libraries (version 3.7.5). These libraries were chosen as they have been heavily used in defect prediction research (see [13, 33, 36], for example). In this experiment a standard stratified 10-fold cross-validation was carried out. During each of the 10-folds, the number of instances in the testing set which were also in the training set were counted. After all 10-folds, the average number of shared instances in each testing set was calculated. This process was repeated 10 000 times with different pseudo-random number seeds to defend against order effects. For this experiment, we used the NASA data sets post basic pre-processing (see Table 1). We did this because it was as representative as possible of what will have happened in many previous studies. It is for the same reason that we chose stratified 10-fold cross-validation, the Weka default. The results from this experiment are shown in Figs. 3*a* and 3*b*. These figures show that for each data set, the average proportion of seen data points in the testing sets was greater than the proportion of repeated data points in total. Additionally, this relationship can be seen in Fig. 3*b* to have a strong positive correlation. It is worth emphasising that in some cases the average proportion of seen data points in the testing sets was very large (91, 84 and 82% for data sets PC5, MC1 and PC2, respectively).

4.3 What effect could having such quantities of seen data points in testing sets have on classifier performance?

To answer this question we do not use the NASA data sets because of the point regarding class distributions mentioned in Section 2. Instead, for clarity, we construct an artificial data set, with 10 numeric features and 1000 data points. All features were generated by a uniform pseudo-random number generator and have a possible range between 0 and 1 inclusive. The data set has a balanced class distribution, with 500 data points representing each class.

Using the Weka Experimenter, we ran 10 repetitions of a stratified 10-fold cross-validation experiment with the data set just described, and the following variations of it:

- 25% repeats from each class, an extra copy of 125 unique instances in each class, 1250 instances in total.
- 50% repeats from each class, an extra copy of 250 unique instances in each class, 1500 instances in total.
- 75% repeats from each class, an extra copy of 375 unique instances in each class, 1750 instances in total.
- 100% repeats, two copies of every instance, 2000 instances in total.

We used random forest learners for this experiment, with 500 trees and all other parameters set to the Weka defaults. We chose 500 trees as this was the number used in [19, 21, 22]. The results of this experiment are shown in Fig. 4. The mean accuracy levels for each data set: original, 25, 50, 75 and 100% repeats were 47.84, 67.38, 80.22, 88.37 and 95.00. This clearly shows that seen data points can have a huge influence on the performance of classifiers, even with pseudo-random data. As the percentage of duplicates increases, so does the performance of the classifiers. There are several major factors why this is the case, including that each node (or tree) comprising a random forest is

- *Unpruned*: Meaning that the training data is essentially memorised. Therefore in cases where the test data contains seen data points, these points are highly likely to be classified correctly, provided they are consistent in the training data.
- *Trained on a bootstrap sample of the original training data*: Such samples are made with replacement, meaning that many original training instances will be repeated and some not chosen at all. Clearly, if the original training data contains duplicates to begin with, the proportion of repeated instances trained on by each node will likely increase dramatically.
- *Trained on a subset of available features*: Meaning that, with the input space of each data point reduced because of this subset, there is a greater likelihood of repeated (and inconsistent) data.

The results from this experiment are very interesting, as random forests have been reported to work well with the NASA data sets [19–22]. Note that random forests have also been reported to struggle with imbalanced data [69–71]. This makes them not the obvious choice for use with the NASA data sets, as most of them are imbalanced (see Table 1). Despite this, favourable performance has been observed [19–22], which we believe is partly because of the reasons just described. Note that we have confirmed these findings using version 5.1 of Breiman and Cutler's original Fortran code (available at http://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm).

It is worth pointing out that the effect of learning on repeated data points is algorithm specific. For example, naive Bayes classifiers have been reported to be fairly resilient to duplicates [72]. We confirm this by repeating the experiment described previously with naive Bayes classifiers. The results are shown in Fig. 5. Although these results are not as striking as those for the random forests, we believe they are still noteworthy, and that in practice the effect may be significant.

4.4 How to address the issues caused by repeated data points

An additional issue with repeated data points, separate to the problem of test set contamination, may occur as a result of

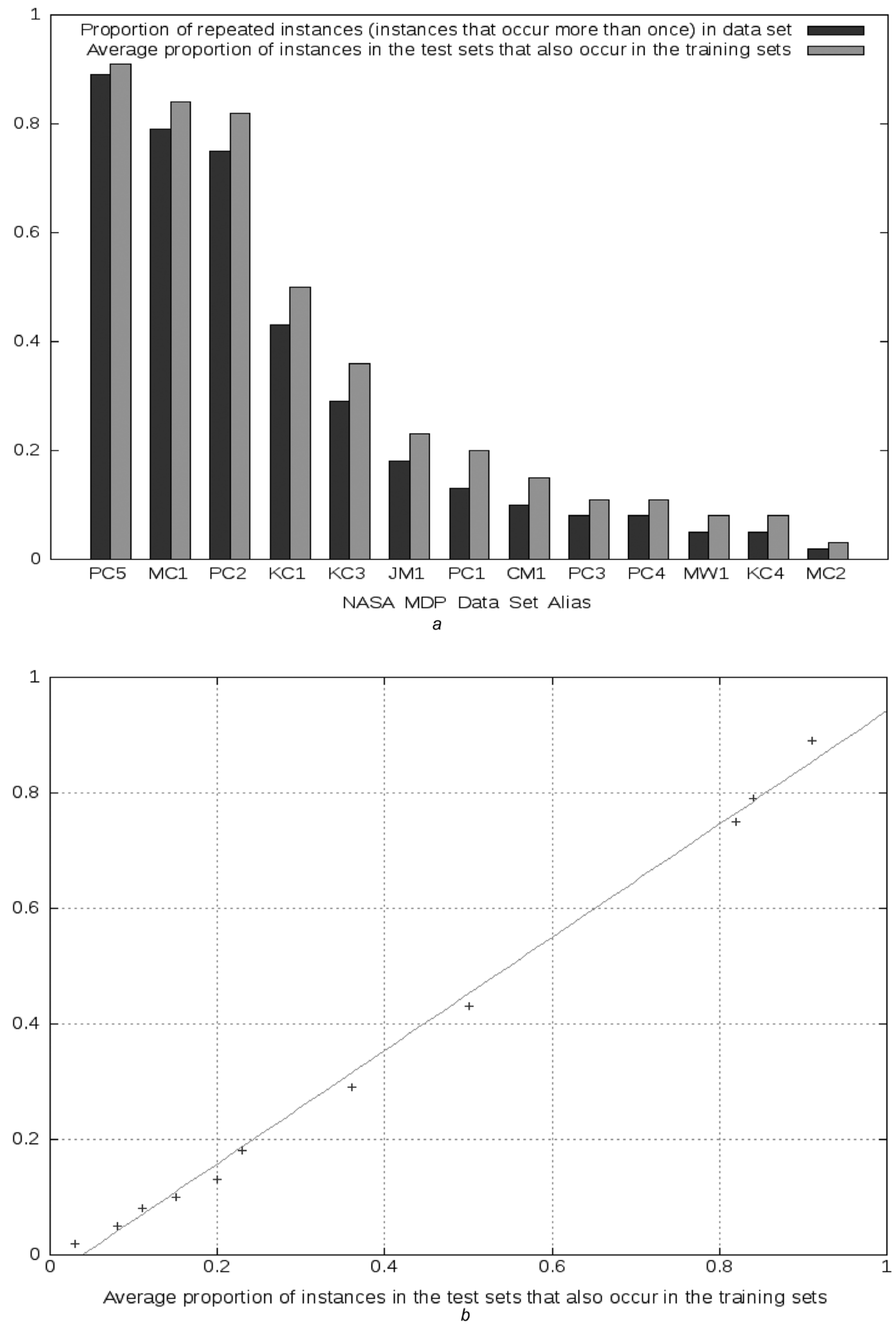


Fig. 3 *Dispersion of repeated data points in the NASA data sets*

a Proportions of repeated data and seen data in testing sets
b Proportions of repeated data against seen data in testing sets

model construction (training and optimisation). Training a model on data containing small proportions of repeated data points is typically non-problematic. For example, a simple oversampling technique is to duplicate minority class data points. Using training sets which contain repeated data points is reasonable, as long as training and testing sets share no common instances. Note however that the issue with excessive oversampling: overfitting, may also occur here. Overfitting can be identified when a model obtains good training performance, but poor performance on unseen test data [61]. It is also possible to cause overfitting by optimising model parameters using a validation set (a withheld subset of the training set) containing duplicate data points. It is for this reason that [72] recommend 'tuning a trained classifier with a duplicate-free sample'. Note that this also applies when optimising using multiple validation sets, for example via *n*-fold cross-validation. It is also worth noting that feature selection techniques can be negatively affected by duplicates [72].

As already mentioned, the simplest way to address the issues caused by repeated data points is to discard them as part of the contextual data-cleansing process, making each

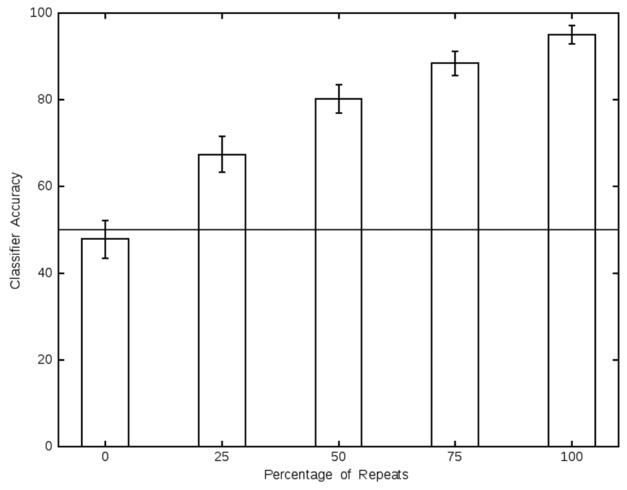


Fig. 4 *Random Forest classifiers with repeated data points*

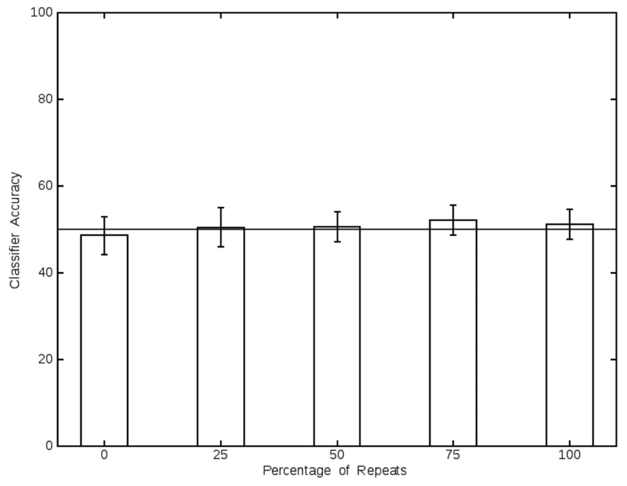


Fig. 5 *Naive Bayes classifiers with repeated data points*

consistent data point unique (for details see our earlier paper [62]). However, carrying out such a pre-processing step may not be the best approach, as repeated data points may occur in the real world. For this reason, we propose the following method for addressing the issues caused by repeated data points in classification experiments:

1. After the initial stratified-divide into training and testing set, discard all training data points with feature vectors common to the testing set. This ensures that performance is measured on unseen data, while the test set remains unmodified.
2. If the class distribution of the training set is adversely altered as a consequence of step 1, consider sampling techniques (see [61, 73]) to help maintain the original (or a more balanced) distribution. If oversampling is required, we recommend the synthetic minority oversampling technique (SMOTE [73]) rather than oversampling by duplication, as it reduces the likelihood of overfitting [73–75].
3. During model optimisation/tuning (if performed), remove all duplicates from the validation set, as recommended by [72]. Next, discard all validation set data points with feature vectors common to the corresponding training set (the training subset). If the class distribution of the validation set is adversely altered, consider the use of sampling techniques, as described in step 2. The purpose of this step is to help avoid overfitting.

This proposed approach is most suitable when researchers believe the repeated data points to be genuine, not noise. This is because test sets remain unmodified. With the simple approach of removing all repeated data points during data cleansing, test sets are indirectly modified before the data separation process has occurred. Therefore researchers who believe the rates of duplication in the NASA data sets to be feasible may wish to use this new approach, whereas researchers believing otherwise may wish to use the more simple approach (described in our earlier paper [62]). Note that a possible addition to the proposed approach is to remove all duplicates from the training set, to further reduce the possibility of overfitting. The best place for this to occur would be between steps 1 and 2.

5 Conclusions

Regardless of whether repeated data points are, or are not noise, it is unsuitable to have seen data in testing sets during experiments intended to show how well a classifier could potentially perform on future, unseen data. This is because seen data points can result in an excessive estimate of performance, occurring because classifiers can, to varying degrees, memorise rather than generalise. There is an important distinction between learning from and simply memorising data: only if you learn the structure underlying the data can you be expected to correctly predict unseen data.

Some researchers may argue that repeated data points should be tolerated and subject of no special treatment, as it is possible for modules with identical metrics to be contained within a software system. However, we have demonstrated that machine learning experiments based on data containing repeated data points can lead to invalid results. This is because it is possible for the test data to be (perhaps inadvertently) contaminated with training data, violating the assumption of unseen data. Even if researchers choose to ignore this fact, it is folly to report the performance of classification experiments with no distinction between performance on seen and unseen data. This is because the generalisation ability of classifiers may be far worse than the performance obtained implies.

If, in the real world, you happen to have a feature vector in your test set which is also contained in your training set, a simple lookup may be all that is required for best performance. A system could therefore be implemented where training data is stored in a hash table, and each test vector checked to see if it is contained within the hash table prior to classification. If so, the class label could be looked up directly with the classification model being unneeded (or some form of ensemble method used). A confidence interval could be derived using the lookup method, with a weight assigned to each training data point based on the number of times it occurs (because of replication) in the training data. This could also be extended to deal with inconsistent instances, perhaps by predicting the most frequently occurring of the classes. If the number of copies of the inconsistent instance in each class is equal, it may be best to report this, and/or make independent use of the classification model(s).

A simple approach to address the issues caused by repeated data points is to discard them prior to classification (for details see our earlier paper [62]). A more sophisticated approach was proposed in this paper (see Section 4.4), which keeps test sets unmodified, and may be a more realistic approach. Another possible option was proposed in [33], and is to use an extra attribute: number of duplicates. This will help to ensure that information is not lost, and like the approach proposed in this paper, is most viable when repeats are believed to be genuine.

A possible reason why there are so many repeated (and inconsistent) data points within the NASA data sets is because of the poor differential power (and the small input space) of the metrics used. It may be highly beneficial in future to also record lower level metrics (such as character counts), as these will help to distinguish non-identical modules, reducing the likelihood of modules sharing identical metrics. In addition to having lower level metrics to begin with, researchers should be careful when discarding attributes, as this typically reduces the size of the input space, increasing the probability of repeated feature vectors. For example, [13] used eight of the NASA data sets with just two or three of the original 38 features. Removing over 92% of the available features in each of these data sets typically yields dramatic increases in the proportions of repeated and inconsistent instances.

'Data cleaning is a time-consuming and labor-intensive procedure but one that is absolutely necessary for successful data mining … Time looking at your data is always well spent.' [56] We believe the data cleansing process defined in this paper will increase the accuracy of the NASA data sets, and make them more suitable for machine learning. This process may also be a good starting point when using other software fault data sets. Experiments based on the NASA data sets which blindly included the repeated data points may have led to erroneous findings. This is because results are often reported as if they were based on unseen data, when in fact (to varying degrees) they were not. The impression given from the literature is that many defect prediction researchers using this data have not been aware of this issue. Future work may be required to (where possible) repeat these studies with appropriately processed data. Other areas of future work include

- Extending the list of integrity rules described in stage 4 of the cleansing process. This will help to catch as many infeasible feature vectors, sets of metric values that can be proved to be noisy, as possible.
- Analysing other fault data sets to see whether the proportions of repeated data points in the NASA data sets are typical of fault data sets in general. This will help to determine the extent of this problem.
- Experimenting with, improving and refining the proposed new method of addressing the issues caused by repeated data points.

6 References

1 Khoshgoftaar, T.M., Allen, E.B.: 'Ordering fault-prone software modules', *Softw. Qual. Control*, 2003, **11**, pp. 19–37

2 Kim, S., James, E., Whitehead, J., Zhang, Y.: 'Classifying software changes: clean or buggy?', *IEEE Trans. Softw. Eng.*, 2008, **34**, (2), pp. 181–196

3 Schröter, A., Zimmermann, T., Zeller, A.: 'Predicting component failures at design time'. Proc. Fifth Int. Symp. on Empirical Software Engineering, 2006, pp. 18–27

4 Shivaji, S., Whitehead, E.J., Akella, R., Kim, S.: 'Reducing features to improve bug prediction'. 24th IEEE/ACM Int. Conf. Automated Software Engineering, 2009. ASE'09, 2009, pp. 600–604

5 Śliwerski, J., Zimmermann, T., Zeller, A.: 'When do changes induce fixes?', *SIGSOFT Softw. Eng. Notes*, 2005, **30**, (4), pp. 1–5

6 Kim, S., Zimmermann, T., Pan, K., Whitehead, E.J.J.: 'Automatic identification of bug-introducing changes'. ASE'06: Proc. 21st IEEE/ACM Int. Conf. on Automated Software Engineering, Washington, DC, USA, 2006, pp. 81–90

7 Williams, C., Spacco, J.: 'SZZ revisited: verifying when changes induce fixes'. Proc. 2008 Workshop on Defects in Large Software Systems. DEFECTS'08, New York, USA, 2008, pp. 32–36

8 Halstead, M.H.: 'Elements of software science (operating and programming systems series)' (Elsevier Science Inc., New York, USA, 1977)

9 McCabe, T.J.: 'A complexity measure'. ICSE'76: Proc. Second Int. Conf. on Software Engineering, Los Alamitos, CA, USA, 1976, p. 407

10 Song, Q., Jia, Z., Shepperd, M., Ying, S., Liu, J.: 'A general software defect-proneness prediction framework', *IEEE Trans. Softw. Eng.*, 2011, **37**, (3), pp. 356–370

11 Menzies, T., Stefano, J.S.D., Orrego, A., Chapman, R.: 'Assessing predictors of software defects'. Proc. Workshop on Predictive Software Models, 2004

12 Menzies, T., Stefano, J.S.D.: 'How good is your blind spot sampling policy?'. Proc. Eighth IEEE Int. Symp. on High Assurance Systems Engineering, 2004, pp. 129–138

13 Menzies, T., Greenwald, J., Frank, A.: 'Data mining static code attributes to learn defect predictors', *IEEE Trans. Softw. Eng.*, 2007, **33**, (1), pp. 2–13

14 Menzies, T., Turhan, B., Bener, A., Gay, G., Cukic, B., Jiang, Y.: 'Implications of ceiling effects in defect predictors'. Proc. Fourth Int. Workshop on Predictor Models in Software Engineering. PROMISE'08, New York, USA, 2008, pp. 47–54

15 Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., Bener, A.: 'Defect prediction from static code features: current results, limitations, new approaches', *Autom. Softw. Eng.*, 2010, **17**, (4), pp. 375–407

16 Zhang, H.: 'An investigation of the relationships between lines of code and defects'. IEEE Int. Conf. on Software Maintenance, 2009. ICSM 2009, 2009, pp. 274–283

17 Zhang, H., Nelson, A., Menzies, T.: 'On the value of learning from defect dense components for software defect prediction'. Proc. Sixth Int. Conf. on Predictive Models in Software Engineering. PROMISE'10, New York, USA, 2010, p. 14:1–14:9

18 Guo, L., Cukic, B., Singh, H.: 'Predicting fault prone modules by the Dempster–Shafer belief networks'. Proc. 18th IEEE Int. Conf. on Automated Software Engineering, 2003, pp. 249–252

19 Guo, L., Ma, Y., Cukic, B., Singh, H.: 'Robust prediction of fault-proneness by random forests'. 15th Int. Symp. on Software Reliability Engineering. ISSRE 2004, 2004, pp. 417–428

20 Jiang, Y., Cukic, B., Menzies, T.: 'Fault prediction using early lifecycle data'. 18th IEEE Int. Symp. on Software Reliability, 2007. ISSRE'07, 2007, pp. 237–246

21 Jiang, Y., Cukic, B., Menzies, T., Bartlow, N.: 'Comparing design and code metrics for software quality prediction'. Proc. Fourth Int. Workshop on Predictor Models in Software Engineering. PROMISE'08, New York, USA, 2008, pp. 11–18

22 Jiang, Y., Cukic, B., Menzies, T.: 'Can data transformation help in the detection of fault-prone modules?'. DEFECTS'08: Proc. 2008 Workshop on Defects in Large Software Systems, New York, USA, 2008, pp. 16–20

23 Jiang, Y., Cukic, B., Ma, Y.: 'Techniques for evaluating fault prediction models', *Empir. Softw. Eng.*, 2008, **13**, (5), pp. 561–595

24 Jiang, Y., Cukic, B.: 'Misclassification cost-sensitive fault prediction models'. Proc. Fifth Int. Conf. on Predictor Models in Software Engineering. PROMISE'09, New York, USA, 2009, p. 20:1–20:10

25 Khoshgoftaar, T.M., Seliya, N.: 'The necessity of assuring quality in software measurement data'. METRICS'04: Proc. Software Metrics, 10th Int. Symp., Washington, DC, USA, 2004, pp. 119–130

26 Zhong, S., Khoshgoftaar, T.M., Seliya, N.: 'Unsupervised learning for expert-based software quality estimation'. Proc. Eighth IEEE Int. Symp. on High Assurance Systems Engineering, 2004, pp. 149–155

27 Seliya, N., Khoshgoftaar, T.M., Zhong, S.: 'Analyzing software quality with limited fault-proneness defect data'. Ninth IEEE Int. Symp. on High-Assurance Systems Engineering, 2005. HASE 2005, 2005, pp. 89–98

28 Liu, Y., Khoshgoftaar, T.M., Seliya, N.: 'Evolutionary optimization of software quality modeling with multiple repositories', *IEEE Trans. Softw. Eng.*, 2010, **36**, (6), pp. 852–864

29 Lessmann, S., Baesens, B., Mues, C., Pietsch, S.: 'Benchmarking classification models for software defect prediction: a proposed framework and novel findings', *IEEE Trans. Softw. Eng.*, 2008, **34**, (4), pp. 485–496

30 Turhan, B., Bener, A.: 'A multivariate analysis of static code attributes for defect prediction'. QSIC'07: Proc. Seventh Int. Conf. on Quality Software, Washington, DC, USA, 2007, pp. 231–237

31 Turhan, B., Menzies, T., Bener, A.B., Di Stefano, J.: 'On the relative value of cross-company and within-company data for defect prediction', *Empir. Softw. Eng.*, 2009, **14**, pp. 540–578

32 Turhan, B., Kocak, G., Bener, A.: 'Data mining source code for locating software bugs: a case study in telecommunication industry', *Expert Syst. Appl.*, 2009, **36**, (6), pp. 9986–9990

33 Boetticher, G.D.: 'Improving credibility of machine learner models in software engineering'. Advanced Machine Learner Applications in Software Engineering (Series on Software Engineering and Knowledge Engineering), 2006, pp. 52–72

34 Mende, T., Koschke, R.: 'Revisiting the evaluation of defect prediction models'. Proc. Fifth Int. Conf. on Predictor Models in Software Engineering. PROMISE'09, New York, USA, 2009, pp. 7:1–7:10

35 Mende, T., Koschke, R.: 'Effort-aware defect prediction models'. European Conf. on Software Maintenance and Reengineering, 2010, pp. 107–116

36 Koru, A.G., Liu, H.: 'An investigation of the effect of module size on defect prediction using static measures', *ACM SIGSOFT Softw. Eng. Notes*, 2005, **30**, (4), pp. 1–5

37 Koru, A.G., Liu, H.: 'Building effective defect-prediction models in practice', *IEEE Softw.*, 2005, **22**, (6), pp. 23–29

38 Tosun, A., Bener, A.: 'Reducing false alarms in software defect prediction by decision threshold optimization'. Proc. 2009 Third Int. Symp. on Empirical Software Engineering and Measurement. ESEM'09, Washington, DC, USA, 2009, pp. 477–480

39 Bezerra, M.E.R., Oliveira, A.L.I., Meira, S.R.L.: 'A constructive RBF neural network for estimating the probability of defects in software modules'. Int. Joint Conf. on Neural Networks, 2007. IJCNN 2007, 2007, pp. 2869–2874

40 Singh, Y., Kaur, A., Malhotra, R.: 'Predicting software fault proneness model using neural network'. Proc. Ninth Int. Conf. on Product-Focused Software Process Improvement. PROFES'08, Berlin, Heidelberg, 2008, pp. 204–214

41 Challagulla, V.U.B., Bastani, F.B., Yen, I.L., Paul, R.A.: 'Empirical assessment of machine learning based software defect prediction techniques'. WORDS'05: Proc. 10th IEEE Int. Workshop on Object-Oriented Real-Time Dependable Systems, Washington, DC, USA, 2005, pp. 263–270

42 Challagulla, V.U.B., Bastani, F.B., Yen, I.L.: 'A unified framework for defect data analysis using the mbr technique'. ICTAI'06: Proc. 18th IEEE Int. Conf. on Tools with Artificial Intelligence, Washington, DC, USA, 2006, pp. 39–46

43 Pelayo, L., Dick, S.: 'Applying novel resampling strategies to software defect prediction'. Annual Meeting of the North American Fuzzy Information Processing Society, 2007. NAFIPS'07, 2007, pp. 69–72

44 Kutlubay, O., Turhan, B., Bener, A.B.: 'A two-step model for defect density estimation'. 33rd EUROMICRO Conf. Software Engineering and Advanced Applications, 2007, pp. 322–332

45 Ma, Y., Guo, L., Cukic, B.: 'A statistical framework for the prediction of fault-proneness', in 'Advances in machine learning application in software engineering' (Idea Group Inc., 2006, vol. 1), pp. 237–265

46 Oral, A.D., Bener, A.B.: 'Defect prediction for embedded software'. 22nd Int. Symp. on Computer and Information Sciences, 2007. ISCIS 2007, 2007, pp. 1–6

47 Rodriguez, D., Ruiz, R., Cuadrado-Gallego, J., Aguilar-Ruiz, J.: 'Detecting fault modules applying feature selection to classifiers'. IEEE Int. Conf. on Information Reuse and Integration. IRI 2007, 2007, pp. 667–672

48 Vandecruys, O., Martens, D., Baesens, B., Mues, C., De Backer, M., Haesen, R.: 'Mining software repositories for comprehensible software fault prediction models', *J. Syst. Softw.*, 2008, **81**, (5), pp. 823–839

49 Mertik, M., Lenic, M., Stiglic, G., Kokol, P.: 'Estimating software quality with advanced data mining techniques'. Int. Conf. on Software Engineering Advances, 2006, p. 19

50 Cong, J., En-Mei, D., Li-Na, Q.: 'Software fault prediction model based on adaptive dynamical and median particle swarm optimization'. 2010 Second Int. Conf. on Multimedia and Information Technology (MMIT), 2010, vol. 1, pp. 44–47

51 de Carvalho, A.B., Pozo, A., Vergilio, S.R.: 'A symbolic fault-prediction model based on multiobjective particle swarm optimization', *J. Syst. Softw.*, 2010, **83**, (5), pp. 868–882

52 Tao, W., Wei-hua, L.: 'Naive Bayes software defect prediction model'. 2010 Int. Conf. on Computational Intelligence and Software Engineering (CiSE), 2010, pp. 1–4

53 Li, Z., Reformat, M.: 'A practical method for the software fault-prediction'. IEEE Int. Conf. on Information Reuse and Integration. IRI 2007, August 2007, pp. 659–666

54 Vivanco, R.A., Kamei, Y., Monden, A., Matsumoto, K.-i., Jin, D.: 'Using search-based metric selection and oversampling to predict fault prone modules'. IEEE CCECE, 2010, pp. 1–6

55 Elish, K.O., Elish, M.O.: 'Predicting defect-prone software modules using support vector machines', *J. Syst. Softw.*, 2008, **81**, (5), pp. 649–660

56 Witten, I.H., Frank, E.: 'Data mining: practical machine learning tools and techniques', in 'Morgan Kaufmann series in data management systems' (Morgan Kaufmann, 2005, 2nd edn.)

57 Liebchen, G.A., Shepperd, M.: 'Data sets and data quality in software engineering'. PROMISE'08: Proc. Fourth Int. Workshop on Predictor Models in Software Engineering, New York, USA, 2008, pp. 39–44

58 Kaminsky, K., Boetticher, G.: 'Building a genetically engineerable evolvable program (GEEP) using breadth-based explicit knowledge for predicting software defects'. IEEE Annual Meeting of the Fuzzy Information, 2004. Processing NAFIPS'04, 2004, vol. 1, pp. 10–15

59 Nickerson, A.S., Japkowicz, N., Milios, E.: 'Using unsupervised learning to guide resampling in imbalanced data sets'. Proc. Eighth Int. Workshop on AI and Statistics, 2001, pp. 261–265

60 Chawla, N.V., Japkowicz, N., Kolcz, A.: 'Special issue on learning from imbalanced datasets', *SIGKDD Explor. Newsl.*, 2004, **6**, (1), pp. 1–6

61 He, H., Garcia, E.A.: 'Learning from Imbalanced Data', *IEEE Trans. Know. Data Eng.*, 2009, **21**, pp. 1263–1284

62 Gray, D., Bowes, D., Davey, N., Sun, Y., Christianson, B.: 'The misuse of the NASA metrics data program data sets for automated software defect prediction'. Evaluation and Assessment in Software Engineering (EASE), 2011, pp. 96–103

63 Batista, G.E.A.P.A., Prati, R.C., Monard, M.C.: 'A study of the behavior of several methods for balancing machine learning training data', *SIGKDD Explor. Newsl.*, 2004, **6**, pp. 20–29

64 Davis, J., Goadrich, M.: 'The relationship between Precision-Recall and ROC curves'. Proc. 23rd Int. Conf. on Machine Learning. ICML'06, New York, USA, 2006, pp. 233–240

65 Zhang, H., Zhang, X.: 'Comments on "data mining static code attributes to learn defect predictors"', *IEEE Trans. Softw. Eng.*, 2007, **33**, pp. 635–637

66 Gray, D., Bowes, D., Davey, N., Sun, Y., Christianson, B.: 'Further thoughts on precision'. Evaluation and Assessment in Software Engineering (EASE), 2011, pp. 129–133

67 Hall, M.A.: 'Correlation-based feature subset selection for machine learning'. Department of Computer Science, University of Waikato, Hamilton, New Zealand, 1999

68 Howley, T., Madden, M.G., O'Connell, M.L., Ryder, A.G.: 'The effect of principal component analysis on machine learning accuracy with high-dimensional spectral data', *Knowl.-Based Syst.*, 2006, **19**, (5), pp. 363–370

69 Chen, C., Liaw, A., Breiman, L.: 'Using random forest to learn imbalanced data', Technical report 666, Department of Statistics, University of California, Berkeley, 2004

70 Segal, M.R.: 'Machine learning benchmarks and random forest regression' (Centre for Bioinformatics and Molecular Biostatistics, UC, San Francisco, 2004)

71 Dudoit, S., Fridlyand, J.: 'Classification in microarray experiments', in 'Statistical analysis of gene expression microarray data' (Chapman and Hall/CRC, 2003)

72 Kołcz, A., Chowdhury, A., Alspector, J.: 'Data duplication: an imbalance problem?'. ICML 2003 Workshop on Learning from Imbalanced Datasets, 2003

73 Chawla, N.V., Bowyer, K.W., Hall, L.O., Kegelmeyer, W.P.: 'SMOTE: synthetic minority over-sampling technique', *J. Artif. Intell. Res.*, 2002, **16**, pp. 321–357

74 Chawla, N.V., Lazarevic, A., Hall, L.O., Bowyer, K.W.: 'SMOTEBoost: improving prediction of the minority class in boosting'. Proc. Principles of Knowledge Discovery in Databases (PKDD-2003), 2003, pp. 107–119

75 Cieslak, D.A., Chawla, N.V., Striegel, A.: 'Combating imbalance in network intrusion datasets'. 2006 IEEE Int. Conf. Granular Computing, 2006, pp. 732–737