Monash University
Faculty of Information Technology

Metric Learning for Software Defect Prediction

This thesis is presented in partial fulfillment of the requirements for the degree of Master of Information Technology (Honours) at Monash University

By:
Linh Nhat Chu
24734756

Supervisors:
Dr. Yuan-Fang Li
Dr. Reza Haffari

Semester 1 – 2015

DECLARATION

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the work of others has been acknowledged.

Signed by ……

Name

Date

ACKNOWLEDGEMENTS

At the outset, I would like to express my deep gratitude to my supervisors, Dr. Yuan-Fang Li and Dr. Reza Haffari for all their great guidance and support throughout this research. As my supervisors, their observations, guidance and comments help me find the right direction of the project to move forward and to complete the thesis. I am very much indebted to them.

I am really thankful to Assoc Prof Tim Menzies and Dr. Jelber Sayyad Shirabad for allowing us to use data from PROMISE software engineering repository. The data sets were very helpful for our experiments and evaluation.

To my family, thank you for your support through my study. I would like to dedicate the thesis to my mother for her unconditional love, sacrifice and encouragement. Her support means so much to me.

ABSTRACT

Software is playing an increasingly vital role in many industries. However, defects are not only inconvenient and annoying, but can also have serious consequences for software systems, especially for mission-critical systems. Therefore, software defect prediction models are useful for understanding, evaluating and improving the quality of a software system. Machine learning techniques have been employed to make predictions about the defectiveness of software components by exploiting historical data of software components and their defects. In order to predict software defects, many studies using distance-based classification algorithms with Mahalanobis distance function have been proposed. However, the common implementations of the Mahalanobis distance function do not take into account the label information (defective or defect-free) from the training data.

In this thesis, we propose a novel approach utilizing metric learning for software defect prediction. The goal of the approach is to make use of label information for improving the performance of classification algorithms such as *k*-nearest neighbor. Specifically, we apply the Maximally Collapsing Metric Learning (MCML) technique to learn a Mahalanobis distance metric for use in classifiers. The effectiveness of our proposed method is evaluated using historical data from the PROMISE software engineering repository, by comparing it with a *k*-nearest neighbor baseline. Our evaluation on a widely used data set shows that our method significantly improves the performance of the *k*-nearest neighbor classifier.

Keywords: *Metric learning, software defect prediction, software metrics, Mahalanobis distance, predictors.*

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 - INTRODUCTION

1.1 Research Background and Significance

Recent decades have witnessed a significant development in the field of software engineering. In the age of information technology, enterprises are increasingly using software to support their works. This leads to a considerable rise in the demand for software quality. Although there are several different ways to define software quality, it is a fact that a complex software application with many bugs would be perceived as a poor-quality product. Unfortunately, large software systems tend to contain a lot of defects (Zhang, 2010). Therefore, software quality assurance is still a challenging problem for software development practice.

According to Zhang (2010), whereas a small number of defects are caused by compilers that produce incorrect code, many stem from errors and mistakes made by programmers in the design and coding process. These defects not only reduce the quality of software but also push up the cost of testing (Whittaker, 2000). Indeed, many software development companies including Microsoft have spent a vast amount of money and effort on testing their software products before releasing them to customers (Kaner, Hendrickson & Smith-Brock, 2001).

In order to better manage software defects, additional human resources need to be hired as software testers. In many companies, the tester to developer ratio is even 1:1 (Kaner et al., 2001). However, since complete testing is very expensive and infeasible given the finite budget and personnel resources, many final products still contain defects. Myers, Sandler and Badgett (2011) construct a simple program that covers 11 boundary conditions and 15 categories of test cases. The exhaustive path testing on this program requires tests for 10^{14} paths. In real programs, it would be infeasible to test every path in a software system because of the exponential growth of paths. As a result, it is essential to identify source code instances that are *likely* to contain defects. By focusing on fault-prone instances, software development firms can reduce the cost and enhance the overall effectiveness of the testing process through more informed resource allocation. The

possibility of identifying software components containing defects early also helps a software firm on planning and accurately estimating the effort needed for completing a software project (Dejaeger, Verbeke, Martens & Baesens, 2012). Hence, the ability to accurately predict the defectiveness of software components is highly desirable for the improvement of software quality.

1.2 Objective and Contributions

The process of identifying defect-prone software components in advance is called software defect prediction. Many studies have been published in the literature. Most of them aim to build defect prediction models by making use of software metrics (e.g., number of lines of code), historical data and classification algorithms for data mining. In fact, the performance of distance-based classification algorithms such as *k*-nearest neighbor significantly depends on the performance of used distance functions. A common method for estimating the distance between instances is to use Mahalanobis distance function with the covariance matrix. This method, however, does not use the label information (defective or defect-free) from the training data. In this study, we propose to use Mahalanobis distance with metric learning to make use of the label information. More specifically, we utilize the Maximally Collapsing Metric Learning (MCML) algorithm to learn a Mahalanobis distance metric for improving classification capability of predictors. Through a comprehensive evaluation using a widely used benchmark data set, we have shown that the use of Mahalanobis distance with MCML significantly improves classification over a *k*-nearest neighbor baseline.

1.3 Outline of The Thesis

The rest of the thesis is structured as follows:

Chapter 2 reviews the literature of software defect prediction studies. Firstly, this chaper presents the typical process of defect prediction. Secondly, software metrics and classification algorithms used in the literature are analyzed in depth. Then, traditional methods for data preprocessing including normalization, noise reduction and attribute

selection are also given. In the next section, we introduce methods for evaluating the performance of prediction models. The last section discusses some challenges and some proposed solutions of software defect prediction.

Chapter 3 discusses the specific research approach used in the thesis. It includes an introduction of metric learning, Mahalanobis distance function used for estimating the similarity between pairs of instances and MCML to learn a Mahalanobis distance metric for improving the classification performance.

Chapter 4 thoroughly describes experiments followed by the analysis and evaluation. Specifically, the chapter contains descriptions of data sets used, design of the evaluation framework, experimental settings, and result analysis of the experiments.

Finally, chapter 5 discusses research contributions and practical issues, as well as provides possible direction for future research.

CHAPTER 2 – BACKGROUND AND RELATED WORK

2.1 Software Defect Prediction Process

The very common process of predicting software defects is to make use of machine learning techniques that provide computer systems the ability to learn from data without being explicitly programmed (Smola & Vishwanathan, 2008). Firstly, data sets are generated from software repositories including defect tracking systems, source code changes, mail archives, data extraction and version control systems. Those data sets consist of instances, which can be software components, files, classes, functions and modules. Based on particular metrics like static code attributes (Menzies, Greenwald & Frank, 2007) extracted from the software repositories, an instance is labeled as defective or defect-free. The collected data sets are then cleaned using preprocessing methods such as noise detection and reduction (Kim, Zhang, Wu & Gong, 2011), data normalization (Nam, Pan & Kim, 2013), and attribute selection (Menzies et al., 2007). After that, the preprocessed data sets are used for building a defect prediction model that is to predict whether new instances contain defects or not. Apart from the binary classification, this model can estimate the number of defects in each instance. In terms of machine learning, this estimation is also called regression (Smola & Vishwanathan, 2008).

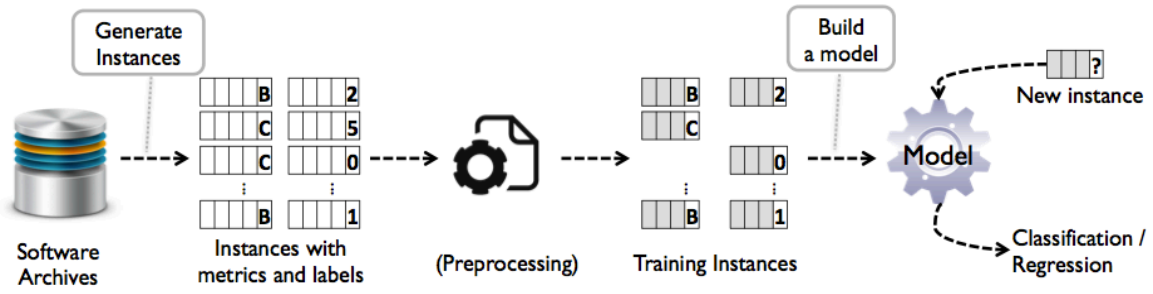


Figure 2.1: A common software defect prediction process (Nam, 2009)

2.2 Software Metrics

Software metrics can be considered as a quantitative measurement that assigns symbols or numbers to features of predicted instances (Bieman, 1997). In fact, they are features, or

attributes, that describe many properties such as reliability, effort, complexity and quality of software products. These metrics play a key role in building an effective software defect predictor. They can be divided into two main categories: code metrics and process metrics (Nam, 2009).

2.2.1 Code metrics

Code metrics, also called product metrics, are directly collected from existing source code. These metrics measure complexity of source code based on the assumption that complex software components are more likely to contain bugs. Throughout the history of software engineering, various code metrics have been used for software defect prediction.

Size: The first metric is size metric introduced by Akiyama (1971). In order to predict the number of bugs, the author uses the number of lines of code as the only metric. Afterwards, numerous software defect prediction studies have applied this metric for building predictors (D'Ambros, Lanza & Robbes, 2012; Hata, Mizuno & Kikuno, 2012; Lee et al., 2011; Menzies et al., 2007; Shihab et al., 2011; Song et al., 2011). However, using only this metric is too simple to measure the complexity of software products.

Halstead and McCabe: For this reason, other useful, widely used and easy to use metrics have been applied for creating defect predictiors (Menzies et al., 2007; Turhan, Menzies, Bener & Di Stefano, 2009; Song, Jia, Shepperd, Ying & Liu, 2011). Those metrics are called static code attributes introduced by McCabe (1976) and Halstead (1977). Halstead attributes are selected based on the reading complexity of source code. They are defined using several basic metrics collected from a software instance including:

Symbol	Description
μ_1	Number of distinct operators
μ_2	Number of distinct operands
N_1	Total number of operators
N_2	Total number of operands
μ_1^*	Minimum possible number of operators

μ_2^*	Minimum possible number of operands

Table 2.1: Halstead basic measurements (Menzies, Di Stefano, Chapman, McGill; 2003)

The first four metrics are self-explanatory whereas μ_1^* and μ_2^* are potential operator and operant counts in a software instance. For example, $\mu_1^* = 2$ is the minimum number of operators for a default function with name of function and a grouping symbol while μ_2^* is the number of parameters passed to the function, with no repetition.

The Halstead metrics defined using the above metrics include:

Name	**Description**
$Length: N = N_1 + N_2$	The program length
$Vocabulary: \mu = \mu_1 + \mu_2$	The vocabulary size
$Volume: V = N * \log_2 \mu$	The information content of a program
$Potential\ volume: V^* = (2 + \mu_2^*) \log_2 (2 + \mu_2^*)$	The volume of the minimal size implementation of a program
$Level: L = V^*/V$	The program level
$Difficulty: D = 1/L$	The difficulty level of a program
$Error\ estimate: \hat{L} = \frac{2}{\mu_1} * \frac{\mu_2}{N_2}$	Error estimate for a program
$Content: I = \hat{L} * V$	The intelligence content of a program
$Effort: E = \frac{V}{L} = \frac{\mu_1 N_2 N \log_2 \mu}{2\mu_2}$	The effort required to generate a program
$Programming\ time: T = E/18\ (seconds)$	The programming time required for a program

Table 2.2: Halstead metrics (Menzies et al.; 2003)

McCabe attributes are cyclomatic metrics representing the complexity of a software product. The attributes proposed based on the assumption that "the complexity of pathways between module symbols is more insightful that just a count of the symbols" (Menzies et al., 2007, p.5). Differing from Halstead attributes, McCabe attributes measure the complexity of source code structure. They are obtained by computing the number of connected components, arcs and nodes in control flow charts of source code. Each node of the flow chart represents a program statement while an arc is the flow of

control from a statement to another. The following are three complexity attributes introduced by McCabe (1976).

- Cyclomatic complexity, denoted by $v(G)$, represents the number of linearly independent paths through the flow chart. $v(G) = e - n + 2$ in which G is the flow chart, e represents the number of arcs, and n is the number of nodes (Menzies et al., 2003). As shown in Figure 2.2, the numbers of arcs and nodes are 6 and 6 respectively, so the cyclomatic complexity is 2, $v(G) = 6 - 6 + 2 = 2$.
- Essential complexity, denoted by $ev(G)$, measures the degree to which a flow chart is able to reduce by decomposing all the sub flow charts that are proper one-entry one-exit, or D-structured primes as defined by Bieman (1997). $ev(G) = v(G) - m$ in which m is the number of sub flow charts of G.
- Design complexity, denoted by $iv(G)$, represents the cyclomatic complexity of a reduced flow chart of a class or module. The reason of reducing the flow chart is to remove complexities that do not affect the interrelationship between design classes or modules (McCabe & Butler, 1989).

```
void foo(void)
{
    if (x == y)
        one();
    else
        two();
    three();
}
```

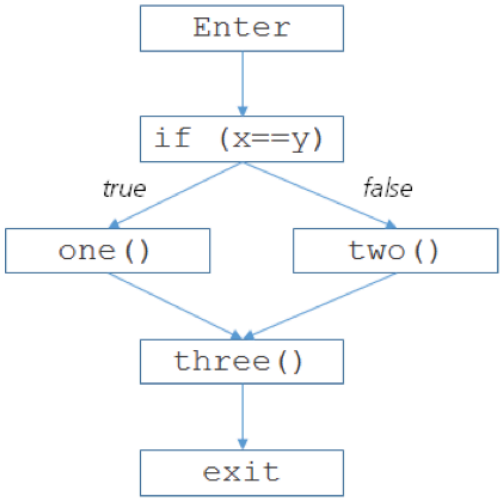


Figure 2.2: A simple function and flow chart

There are many approaches utilizing McCabe and Halstead attributes in the combination with machine learning techniques such as decision trees (J48), *k*-nearest neighbor, support vector machine and Naïve Bayes (Witten & Frank, 2005) to build software defect

prediction models (Lessmann Baesens, Mues & Pietsch, 2008; Menzies et al., 2007; Nam et al., 2013; Ohlsson & Alberg, 1996; Song et al., 2011; Turhan et al., 2009).

Even though these models deliver outstanding performances that are fairly similar to the best results (Turhan & Bener, 2007), Menzies et al. (2010) argue that the approaches using static code attributes have methodological and technical issues. Firstly, the indiscriminate use of machine learning techniques wastes software quality assurance budgets. Machine learners should be selected and customized for particular goals. For example, the machine learners used to maximize the probability of defect detection should differ from those used to get low probability of false alarm with an acceptable detection probability. Secondly, the McCabe and Halstead attributes are intra-class metrics. Since unsafe operations usually arise from unstudied interactions between software components (Menzies et al., 2007), it is necessary to apply inter-class metrics for providing better defect predictions.

Object-oriented: In fact, such inter-class metrics have been produced by Henry and Kafura (1981) with fan-in metrics that represent the number of software components invoking a given component, and fan-out metrics that represent the number of software components invoked by a given component. Besides fan-in and fan-out, other metrics measuring quantity and volume of source code have also been introduced (Abreu & Carapua, 1994; D'Ambros, Lanza & Robbes, 2012). As listed in Table 2.3, several of these metrics are quite simple to compute by counting the number of public and private attributes and methods. In practice, these metrics are designed based on characteristics of object-oriented models including inheritance, reusability, cohesion, encapsulation and coupling. Therefore, the collection of these metrics, also known as object-oriented (OO) metrics, is suitable for evaluating object-oriented systems. With the popularity of object-oriented programming, OO metrics are becoming increasingly widely used for building software defect prediction models. Many of such models have been proposed by D'Ambros et al. (2012); Kim, Zhang, Wu and Gong (2011); Lee, Nam, Han, Kim and Hoh (2011); Pai and Dugan (2007); Wu, Zhang, Kim and Cheung (2011); Zimmermann and Nagappan (2008); Pan and Yang (2010).

Name	Description
Fan-in	The number of other classes that reference the measured class
Fan-out	The number of classes referenced by the measured class
NOA	The number of attributes
NOPA	The number of public attributes
NOPRA	The number of private attributes
NOAI	The number of attributes inherited
LOC	The number of lines of code in a class
NOM	The number of methods
NOPM	The number of public methods
NOPRM	The number of private methods
NOMI	The number of methods inherited

Table 2.3: Examples of class-level object-oriented metrics (D'Ambros et al., 2012)

Apart from the above object-oriented metrics, several other metrics have been empirically proven to be effective for predicting defects in object-oriented programs. Particularly, in 1994, Chidamber and Kemerer used the ontology of Bunge as the theoretical basis to introduce a set of so-called CK metrics. Afterwards, the metrics have been used in numerous studies to create software defect predictors (Bacchelli, D'Ambros & Lanza, 2010; D'Ambros et al., 2012; Kamei et al., 2010; Kim et al., 2011; Lee et al., 2011; Pai & Dugan, 2007; Wu et al., 2011; Basili, Briand and Melo, 1996; Arisholm et al., 2007).

Name	Description
WMC	Weighted methods per class
DIT	Depth of inheritance tree
NOC	Number of children
CBO	Coupling between object classes
RFC	Response for a class
LCOM	Lack of cohesion of methods

Table 2.4: CK metrics (Chidamber & Kemerer, 1994)

The metrics listed in Table 2.4 can be described as follows:

- Weighted methods per class (WMC): This metric measures the complexity of an individual class. It is a weighted sum of all methods in a class.
- Depth of inheritance tree (DIT): This metric measures the length of the longest path of inheritance ending at a class. If the inheritance tree for the measured class is deeper then it is more difficult to estimate the behaviour of the class.
- Number of children (NOC): This metric counts the number of immediate child classes that inherit from the current class.
- Coupling between object classes (CBO): This metric measures the dependency of a class on others by counting the number of other classes coupled to the measured class. A class is coupled to others if it invokes variables or functions of the other classes (Tang, Kao & Chen; 1999).
- Response for a class (RFC): This metric counts the number of methods potentially executed in response to a message received by an object of a class (Subramanyam & Krishnan; 2003).
- Lack of cohesion of methods (LCOM): This metric is the subtraction of the number of method pairs sharing no member variable from the number of method pairs sharing at least one member variable.

2.2.2 Process metrics

In addition to the above code metrics, the history of software defect prediction has also witnessed the appearance of process metrics. Like code metrics, process metrics are also widely used for building defect prediction models (D'Ambros, Lanza & Robbes, 2012). However, rather than directly computed from the existing source code, the process metrics are generated from software repositories such as defect tracking systems and version control systems. Those metrics focus on attributes related to the process of software development; for example, changes of source code, cost or effectiveness of methods used. Table 2.5 shows seven representatives of process metrics.

Process metrics	Number of metrics	Source systems
Relative code change churn	8	Version control
Change	17	Version control
Change entropy	1	Version control
Code metric churn and code entropy	2	Version control
Popularity	5	Email archive
Ownership	4	Version control
Micro interaction metrics	56	Mylyn

Table 2.5: Representatives of process metrics (Nam, 2009)

Relative code change churn: In order to predict the amount of code change, Nagappan and Ball (2005) present relative code change churn metrics that have been proven as useful metrics to predict defect density of a system. They are a set of eight metrics from M1 to M8. The set is identified based on various normalized changes in a software component.

- M1 is determined from churned lines of code, the accumulative number of added and deleted lines between the old and new version of a class, divided by total number of lines of code.
- M2 is the division of deleted lines of code by total lines of codes between two versions.
- M3 is the division of the number of changed files by the number of files complied between two versions.
- M4 is the division of the number of changes made to the files by the number of changed files between two versions.
- M5 is the division of the accumulative times that a file is opened for editing by the number of files complied between two versions.
- M6 is the division of the summary of churned lines of code and deleted lines of code by the accumulative times of opening a file for editing between two versions.

- M7 is the division of churned lines of code by deleted lines of code between two versions.
- M8 is the division of the summary of churned lines of code and deleted lines of code by the number of changes made to the files between two versions.

Change: Another representative of process metrics, also taking into account of added and deleted lines of code, is change metrics. The metrics proposed by Moser, Pedrycz and Succi (2008) consider the extent of code changes in historical data extracted from version control systems. Examples of change metrics are average lines of code added per revision and number of revisions of a file. Table 2.6 lists seventeen change metrics generated from the Eclipse repositories to carry out a comparative analysis between change and code metrics. After conducting the analysis, Moser et al. (2008) conclude that change metrics outperform code metrics in terms of predicting software defects.

Name	Description
Revisons	Number of revisions of a file
Refactorings	Number of times a file has been refactored
Bugfixes	Number of times a file is involved in bug-fixing
Authors	Number of distinct authors checked a file into the repository
Loc_Added	Sum over all revisions of lines of code added to a file
Max_Loc_Added	Maximum number of lines of code added for all revisions
Ave_Loc_Added	Average lines of code added per revision
Loc_Deleted	Sum over all revisions of lines of code deleted from a file
Max_Loc_Deleted	Maximum number of lines of code deleted for all revisions
Ave_Loc_Deleted	Average lines of code deleted per revision
Codechurn	Sum of added and deleted lines of code over all revisions
Max_Codechurn	Maximum codechurn for all revisions
Ave_Codechurn	Average codechurn per revision
Max_Changeset	Maximum number of files committed together to the repository
Ave_Changeset	Average number of files committed together to the repository

Age	Age of file in weeks – counting backwards from a specific release
Weighted_Age	$\frac{\sum_{i=1}^{N} Age(i) * Loc\_Added(i)}{\sum_{i=1}^{N} Loc\_Added(i)}$

Table 2.6: List of change metrics (Moser et al., 2008)

Change entropy: Change entropy, also called history complexity metric, is introduced by Hassan (2009) to capture the complexity of changes. This metric is calculated based on the number of file changes. In order to validate the change entropy, Hassan (2009) builds a statistical linear regression models to compare the effectiveness of change entropy and two change metrics. The comparison, implemented on six open-source projects, indicates that the predictor built using change entropy is better than that using two change metrics. However, the drawback of using change entropy is that the measurement is only taken at the subsystem level instead of the file level (Nam, 2009).

Code metric churn and code entropy: Code metric churn and code entropy are file-level metrics proposed by D'Ambros (2010). While code metric churn captures the change of code metrics every two weeks, code entropy considers the number of involved files when a change occurs to a certain code metric. The limitation of using those metrics is heavy computation due to the track of changes made every two weeks.

Popularity: Other typical process metrics are popularity metrics obtained from email archives (Bacchelli et al., 2010). The ground assumption of the metrics is that the classes discussed more in email archives are more defect-prone. List of the popularity metrics is shown in Table 2.7. Although the idea of using popularity metrics is creative, the evaluation conducted by Bacchelli et al. (2010) indicates that these metrics are not better than code metrics or other process metrics in terms of predicting software defects.

Name	**Description**
Pop_Nom	Number of emails discussing a class
Pop_Nocm	Number of character in emails discussing a class
Pop_Not	Number of threads discussing different topics for a class
Pop_Nomt	Number of emails in a thread dicussing a class

Pop_Noa	Number of authors talking about the same class

Table 2.7: List of popularity metrics (Bacchelli et al., 2010)

Ownership: Ownership metrics are introduced by Bird, Nagappan, Murphy, Gall and Devanbu (2011) based on the idea of measuring the influence of ownership on software quality. To understand the relationship between ownership and software quality, the researchers propose four metrics as follows:

- Ownership: Proportion of ownership of the contributor having the highest proportion of ownership. The proportion of ownership of a contributor is the ratio of the number of changes or commits, which a developer has made to a software instance, to the total number of changes or commits for that instance.
- Minor: Number of minor contributors. A minor contributor is a developer who has made changes or commits to an instance but his/her ownership is less than 5%.
- Major: Number of major contributors. A major contributor is a developer who has made changes or commits to an instance but his/her ownership is equal to or greater than 5%.
- Total: Total number of contributors who have made changes or commits to a software instance.

The study of Bird et al. (2011) has shown that the higher levels of ownership lead to less defect-prone and that an instance touched many developers is more likely to be defective.

Micro interaction metrics: Another study on the relationship between developers and software instances is conducted by Lee et al. (2011). The authors propose micro interaction metrics generated from Mylyn, an Eclipse plug-in recording the context of tasks of developers like inserting or updating a file. The assumption of using micro interaction metrics is that software defects could stem from developers' mistakes. In the study, Lee et al. (2011) have selected top 56 ranked metrics among 113 metrics based on gain ratio (Kullback, 1968; Kullback & Leibler, 1951) to build defect prediction models (Appendix A). Their experiments show that micro interaction metrics are better than code metrics and other process metrics in both regression and classification. Nevertheless,

because of the heavy dependence of micro interaction metrics upon Mylyn, it is difficult to use the metrics in the software systems that do not support Mylyn (Lee et al., 2011).

2.2.3 Discussion on software metrics

Figure 2.3 shows the frequency of use of software metrics in the literature. As their earlier appearance in the history of software defect prediction, it is unsurprising that code metrics are used more frequently than process metrics (Nam, 2009). Furthermore, when a new kind of metrics is produced, it is usually compared to code metrics for eveluating the performance. Meanwhile, process metrics have been introduced later when software repositories including defect-tracking systems, source code changes, mail archives, data extraction and version control systems become widely used.

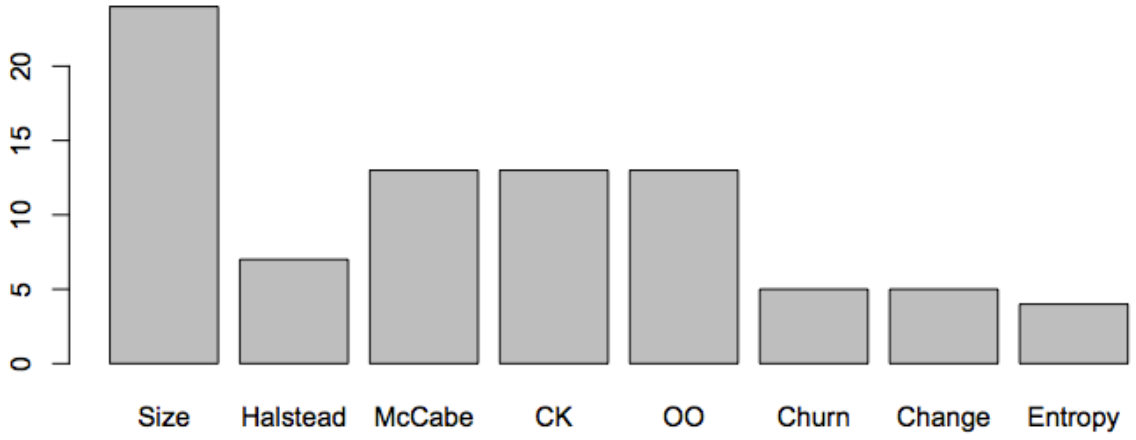


Figure 2.3: The frequency of use of software metrics in representative studies (Nam, 2009)

In the area of software defect prediction, there are also a lot of debates on which kind of metrics performs better. While Menzies et al. (2007) state that static code metrics are still efficient for creating defect predictors; Rahman and Devanbu (2011) believe that recently, process metrics are more useful due to the stagnation of source code metrics. In fact, the effectiveness of process metrics for predicting software defect has been confirmed in a number of studies (Moser, Pedrycz & Succi, 2008; Hassan, 2009; Lee et al., 2011; Rahman & Devanbu, 2013). Even so, only using software metrics is not enough to build effective predictors. In the literature, many researchers have proven that software

defect predictors perform better when making use of machine learning techniques to learn from historical data (Nam, 2009).

2.3 Machine Learning

Machine learning is a scientific discipline of exploring the construction and study of techniques that allow computer programs to learn from data without explicitly programmed (Smola & Vishwanathan, 2008). Basically, machine learning provides computer programs with capabilities to imitate the human learning process. This process is to observe a phenomenon and to generalize from the observations (Japkowicz & Shah, 2011). Machine learning is typically divided into three main categories: supervised and unsupervised learning. In unsupervised learning, algorithms are used to learn predictors from unlabeled data. Meanwhile, supervised learning learns prediction models based on a set of input data with label information.

In supervised learning, the outputs can be real numbers in regression or class labels in classification. As classifying an input into two or more classes, supervised learning is sometimes known as classification. There are a variety of classification techniques that have been widely exploited in the literature for labeling software instances as defective or defect-free.

2.3.1 Decision Tree classification

Decision tree is one of the popular prediction algorithms applied to a broad range of tasks in statistics, data mining and machine learning. This algorithm aims to build a decision tree for classifying a target instance based on input features. It could also be represented as if-then statements for enhancing human readability (Sahana, 2013).

Table 2.8 describes a small training data set for deciding whether or not to play golf. An example of a decision tree, which is to support the decision-making process, is presented in Figure 2.4. The decision tree firstly is constructed by sorting the features down from the root to several leaves. Each leaf node represents a test on a feature while each branch

is a possible outcome of the test. In order to label an instance, the tests are taken place at each node from the root to leaf nodes through appropriate branches (Quinlan, 1986).

Day	Outlook	Temperature	Humidity	Wind	Play golf
1	Sunny	Hot	High	Weak	Yes
2	Sunny	Hot	High	Strong	Yes
3	Overcast	Hot	High	Weak	No
4	Rain	Cool	Normal	Weak	Yes
5	Rain	Mild	High	Weak	Yes
6	Rain	Cool	Normal	Strong	No
7	Overcast	Cool	Normal	Weak	No
8	Sunny	Mild	High	Weak	Yes
9	Sunny	Cool	Normal	Weak	Yes
10	Rain	Mild	Normal	Strong	No
11	Sunny	Mild	Normal	Strong	Yes
12	Overcast	Mild	High	Strong	Yes
13	Overcast	Hot	Normal	Weak	Yes
14	Rain	Mild	High	Strong	No

Table 2.8: A small training data set (Sahana, 2013)

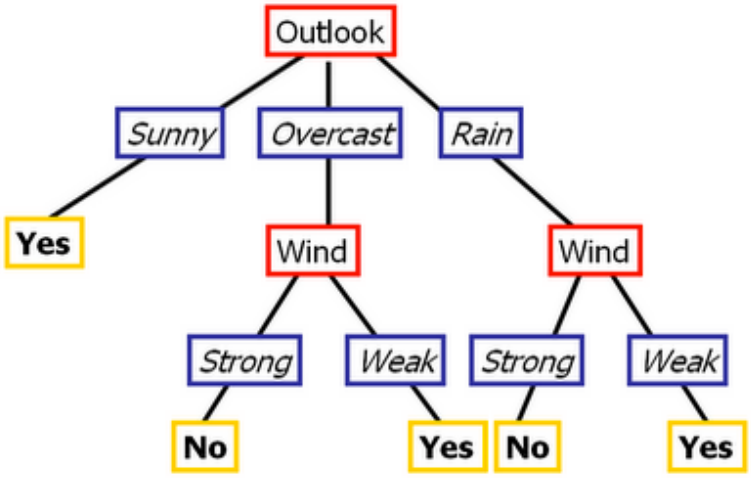


Figure 2.4: A simple decision tree (Sahana, 2013)

There are many models developed using decision tree for predicting software defects (Menzies, 2007; Liu, 2005; Knab, Pinzger & Bernstein, 2006; Peng, Wang & Wang, 2012; Giger, Pinzger & Gall, 2010). J48 learner, which is a JAVA implementation of the C4.5 algorithm (Quinlan, 2014), can be seen as the most widely used one. As a normal decision tree algorithm, J48 recursively splits a data set based on tests on feature values to separate possible outcomes. However, according to Liu (2005), the learner is innovative because of producing a set of rules with measures and their cutoff values for easily interpreting and using in software defect prediction. The cutoff values are chosen using information theory (Shannon, 1993). Assuming that the percentages of defect-free and defective instances in a data set are 75 and 25 respectively. Then, the data set has an outcome distribution X in which two classes $x_1 = defect - free$ and $x_2 = defect$ have frequencies $n_1 = 0.75$ and $n_2 = 0.25$ respectively, and $p(x)$ is the probability of $x \in X$. The number of bits, also known as information entropy, needed to encode an outcome distribution X is $H(X)$ defined as follows (Menzies, 2007):

$$N = \sum_{x \in X} n(x)$$

$$p(x) = \frac{n(x)}{N}$$

$$H(X) = -\sum_{x \in X} p(x) \log_2 p(x)$$

In our case:

$$\begin{aligned} H(X) &= H(0.75, 0.25) \\ &= -(0.75 \log_2 0.75) - (0.25 \log_2 0.25) \\ &= 0.81 \end{aligned}$$

Menzies (2007) presents a simple example of executing J48 on the KC4 data set generated from PROMISE repository (Shirabab & Menzies, 2005).

$CALL\_PAIRS \leq 0: defect - free$
$CALL\_PAIRS > 0$
$| \; NUMBER\_OF\_LINES \leq 3.12: defect - free$
$| \; NUMBER\_OF\_LINES > 3.12$
$| \; | \; NORMALIZED\_CYCLOMATIC\_COMPLEXITY \leq 0.02$

$|\,|\,|\ NODE\_COUNT \leq 3.47: defective$
$|\,|\,|\ NODE\_COUNT > 3.47: defect - free$
$|\,|\ NORMALIZED\_CYCLOMATIC\_COMPLEXITY \leq 0.02: defective$

The following is an explanation of the result: "A software instance is classified as defective if it has non-zero call-pairs, more than 3.12 lines and a high value of normalized cyclomatic complexity (greater than 0.02), or a low value of normalized cyclomatic complexity and less than or equal to 3.47 node count."

2.3.2 Naïve Bayes classification

Another way to construct software defect prediction models is to use a very useful machine learning technique, Naïve Bayes. The technique is one of probability classifiers based on Bayes' theorem with independence assumptions between attributes (Bishop, 2006; Murphy, 2006). With no complicated iterative parameter estimation, a Naïve Bayes classifier is easy to construct and suitable for high-dimensional input data. Despite its simplicity, comparative studies of Langley and Sage (1994) have shown that the classifier is effective for large data sets, and often outperforms other more sophisticated classifiers such as decision tree in supervised learning domains.

According to Bayes theorem, a Naïve Bayes algorithm assumes that the value of a particular attribute (x) on given class (c) is independent of the values of others (Bishop, 2006; Murphy, 2006). Bayes theorem describes the relationship between $P(c|x), P(x|c), P(c)\ and\ P(x)$ as follows:

$$P(c|x) = \frac{P(x|c)P(c)}{P(x)}$$

$$P(c|X) = P(x_1|c) * P(x_2|c) * \ldots * P(x_n|c) * P(c)$$

Where:

$P(c|x)$ is the posterior probability of class (c) given attribute (x).

$P(c)$ is the prior probability of class (c).

$P(x|c)$ is the likelihood which is the probability of attribute (x) given class (c).

$P(x)$ is the prior probability of attribute (x).

In plain English, the equation of calculating the posterior probability can be written as:

$$P(Outcome/Evidence) = \frac{P(Likelihood\ of\ Evidence) * Prior\ Probability\ of\ Outcome}{P(Evidence)}$$

$$P(Outcome/Evidence) = P(Evidence1/Outcome) * P(Evidence2/Outcome) * \dots * P(EvidenceN/Outcome) * P(Outcome)$$

A simple example of calculating the posterior probability is shown in Figure 2.5 (Sayad, 2015). Firstly, a frequency table is constructed for each attribute against each class. The frequency table then is transformed into a likelihood table. Finally, the posterior probability for each class is calculated using the Naïve Bayes equation. The outcome of prediction is the class with the highest value of the posterior probability.

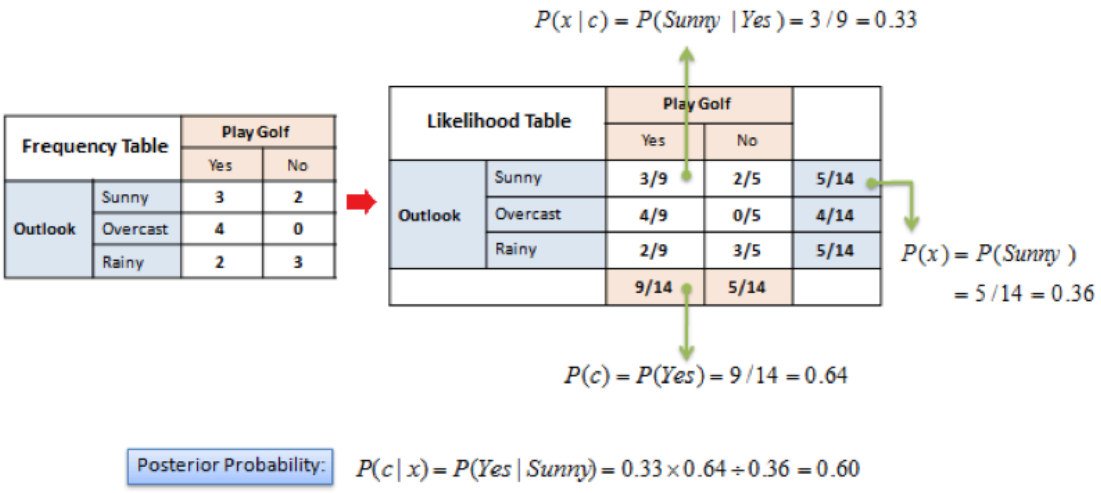


Figure 2.5: An example of the Naïve Bayes classifier (Sayad, 2015)

In the field of software defect prediction, the Naïve Bayes technique can be used to construct prediction models by analyzing historical data generated from software repositories. (Turhan & Bener, 2009; Menzies, Greenwald & Frank, 2007; Turhan et al., 2009; Ma, Luo, Zeng & Chen, 2012; Wang & Li, 2010). When creating the prediction models, the posterior probability of each class, defective or defect-free, is computed

using attributes of software instances extracted from the historical data sets such as static code attributes (McCabe, 1976; Halstead, 1977). Since the attributes are numerical, a common practice is to make use of the probability density function for a normal distribution (Witten & Frank, 2005).

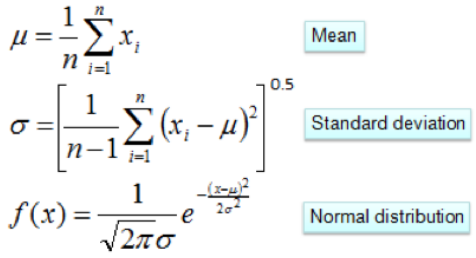


Figure 2.6: The probability density function for a normal distribution (Sayad, 2015)

The following is an example of predicting a target instance that has the number lines of code is 68 based on data from 14 other instances. By only using one attribute, the result indicates that the target instance is likely to be defective.

Defective	**The lines of code (LOC)**									**Mean**	**Standard Deviation**
True	81	91	75	60	65	75	65	85	70	74.1	10.2
False	80	85	65	90	86					81.2	9.7

Table 2.9: An example of predicting a target instance

$$P(LOC = 68|Defective = true) = \frac{1}{\sqrt{2\pi}(10.2)} e^{\frac{-(68-74.1)^2}{2(10.2)^2}} = 0.032$$

$$P(LOC = 68|Defective = false) = \frac{1}{\sqrt{2\pi}(9.7)} e^{\frac{-(68-81.2)^2}{2(9.7)^2}} = 0.016$$

2.3.3 *K*-Nearest Neighbor classification

Apart from J48 decision tree and Naïve Bayes, *k*-nearest neighbor (Cover & Hart, 1967), one of the simple distance-based algorithms, is also commonly applied for pattern classification. In the field of software defect prediction, many studies have also used *k*-nearest neighbor for classifying test data sets (Turhan, Menzies, Bener & Di Stefano, 2009; El-Emam, Benlarbi, Goel & Rai, 2001; Khoshgoftaar et al., 1997; Ganesan, Khoshgoftaar & Allen, 1999). Despite its simplicity, Weinberger and Saul (2009) state

that *k*-nearest neighbor algorithm often works well and produces competitive results in practice. Notably, the algorithm can be improved significantly when combined with prior knowledge obtained from metric learning phase (Belongie, Malik & Puzicha, 2002; Simard, LeCon & Decker, 1993).

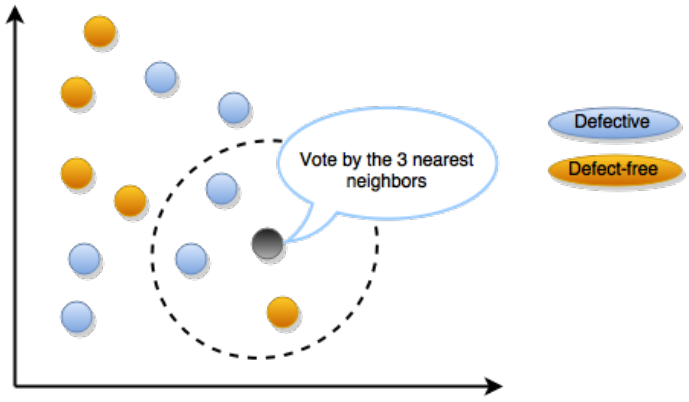


Figure 2.7: An example of classifying a software instance

K-nearest-neighbor algorithm classifies a new instance based on measuring the similarity between it and every other existing instance. The similarity is measured by distance functions such as Manhattan distance (Black, 2006), Euclidean distance (Deza & Deza, 2009) and Mahalanobis distance (Mahalanobis, 1936). When it comes to *k*-nearest neighbor classification, the number of nearest neighbors used for prediction could influence the prediction performance. Typically, this number is odd if classing test instances into two categories. The test instances will be labeled based on the majority of votes. Khoshgoftaar et al. (1997) and Ganesan et al. (1999) have built case-based reasoning (CBR) systems for classifying software components by only selecting $k = 1$ as the single nearest neighbor. El-Emam et al., in 2001, improved the CBR classifier by using a majority vote in cases of three and five nearest neighbors. While selecting a small number of nearest neighbors may lead to a higher impact of noise on classifiers, a large number of those will increase computational cost. Therefore, a simple approach is to set the number of nearest neighbors to square root of the number of training instances ($k = \sqrt{n}$).

Another straightforward extension, which can improve the accuracy of prediction models, is to apply weights. The rationale for using weights is varying degrees of similarity

between the instance to be classified and its neighbors. Thus, rather than giving an equal weight to all nearest neighbors (Ganesan et al., 1999), weights of training instances are set differently depending on their distances to the test instance (Khoshgoftaar et al., 1997). There are a variety of ways to achieve weights. As proposed by Cunningham and Delany (2007), a fairly general technique is to use the inverse of the distance as weight of each instance. This means that closer neighbors have higher weights than father ones. Another way to set weight is based on the number of instances of each class in training data. By simple dividing the number of nearest neighbors of a class by the number of instances of this class in the training data set, this method can become a good solution for mitigating class imbalance problem.

2.3.4 Support Vector Machine classification

Support Vector Machine (SVM) is another classifier commonly used in many applications. SVM is a kernel based learning technique proposed by Boser, Guyon and Vapnik in 1992, which basically deals with two-class pattern recognition problems (Vapnik, 1995; Cortes & Vapnik, 1995; Elish & Elish, 2008). In order to perform classification, the SVM algorithm finds the optimal hyperplane that splits all instances of one class from those of the other. The optimal hyperplane, defined by a number of support vectors (Cristianini & Shawe-Taylor, 2000), is obtained when maximizing the width of the margin between the two classes. The support vectors are data points that lie on the boundary of the margin.

Separable data: There is no interior data point within the margin. Figure 2.8 shows an SVM that linearly separates two classes in a two-dimensional space. In the figure, green circles represent class C_1 and red circles represent class C_2. The SVM aims to set a linear boundary between the two classes in such a way to maximize the margin between solid lines.

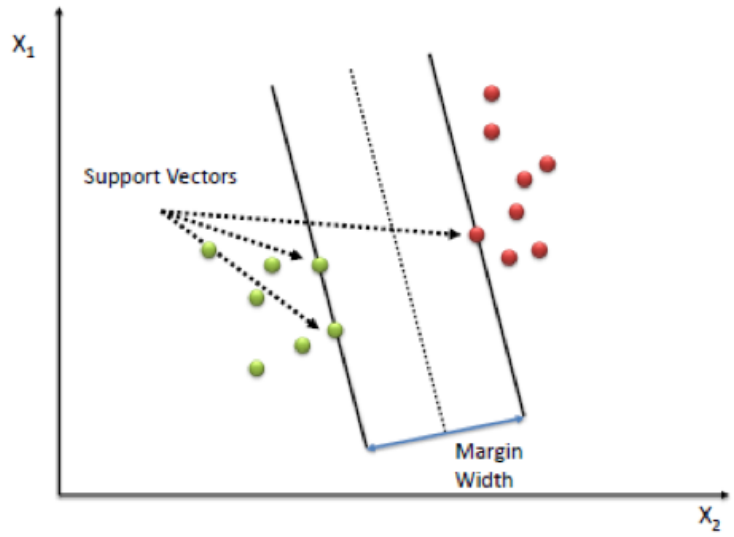


Figure 2.8: The maximum-margin hyperplane, margin and support vectors for an SVM trained with instances from two classes (Sayad, 2015)

The goal of a binary classification problem is to estimate a function $f: R^d \rightarrow \{\pm 1\}$ based on a training data set. The class C_1 is represented with instances $x \in C_1$ and label $y = 1$ while the class C_2 is represented with instances $x \in C_2$ and label $y = -1$ in which $(x_i, y_i) \in R^d \times \{\pm 1\}$. If there exists a hyperplane that linearly separable the training data set in a d-dimensional space then there also exists a pair $(\vec{w}, b)$ that is a solution for the problem:

$$Maximize \left\{ \frac{2}{\|\vec{w}\|} \right\}$$

$$Subject\ to$$

$$\vec{w}.x_i + b \geq 1 \; \forall \; x_i \in C_1$$

$$\vec{w}.x_i + b \leq -1 \; \forall \; x_i \in C_2$$

in which

$\frac{2}{\|\vec{w}\|}$ is the width of the margin.

$x_i = x_1, x_2, x_3, \ldots, x_n$ and n is the number of the training data set.

$\vec{w}$ is a d-dimensional normal vector orthogonal to the hyperplane.

b is the bias of the estimator (Brown, 1947).

"." denotes the inner product, or also dot product (Hazewinkel, 2001).

Those inequality constraints above can be written as:

$$y_i(\vec{w}.x_i + b) \geq 1 \; \forall \; x_i \; \in \; C_1 \cup \; C_2$$

The support vector are x_i on the boundary such that $y_i(\vec{w}.x_i + b) = 1$. Figure 2.9 shows an optimal hyperplane defined by maximizing the width of the margin.

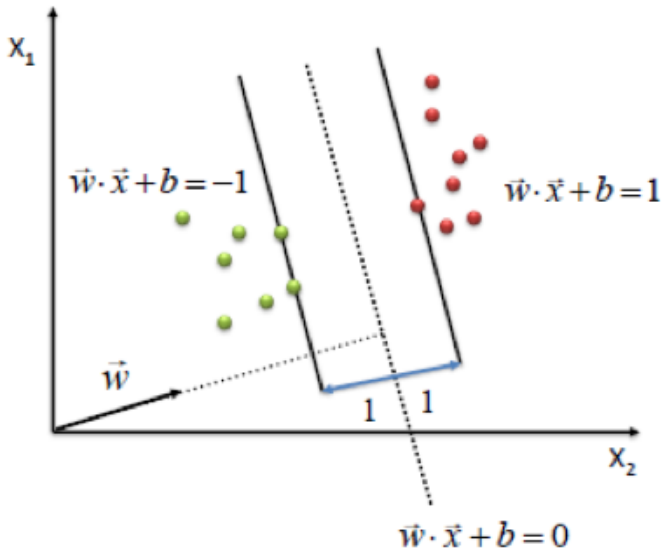


Figure 2.9: Optimal hyperplane separating the two classes (Sayad, 2015)

Since the learning problem depends on the norm of $\vec{w}$ ($\|\vec{w}\|$) that involves a square root, it is difficult to tackle. For mathematic convenience, this problem can be given as the equivalent problem of minimizing $\frac{1}{2}\|\vec{w}\|^2$ without changing the solution. The learning problem now is a quadratic programming problem.

$$Minimize \left\{\frac{1}{2}\|\vec{w}\|^2\right\}$$

$Subject\ to$

$$y_i(\vec{w}.x_i + b) \geq 1 \; \forall \; x_i = x_1, \; x_2, \; x_3, \ldots, x_n$$

In addition to the primal form, the dual form of the learning problem can be solved using standard Lagrangian duality algorithms (Vapnik, 1995; Burges, 1998):

$$F(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\|\vec{w}\|^2$$

$$F(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j x_i . x_j$$

in which $\alpha = \alpha_1, \alpha_2, \ldots, \alpha_n$ are the positive Lagrange multipliers. According to Abe (2005), this model is called hard-margin SVM and the optimization problem is to maximize the function $F(\alpha)$ on the subject of $\alpha_i \geq 0$.

$$Maximize \left\{ \sum_{i=1}^{n} \alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j x_i . x_j \right\}$$

$$Subject\ to \sum_{i=1}^{n} \alpha_i y_i = 0\ and\ \ \alpha_i \geq 0\ \forall\ i = 1,2, \ldots, n$$

The non-zero α_i is the solution for the dual problem while the data points x_i corresponding to the non-zero α_i are support vectors defining the hyperplane. The optimal solution enables SVM to classify an instance t by the decision function:

$$class(x_t) = sign\left(\sum_{i=1}^{n} \alpha_i y_i x_i . x_j \ + b \right)$$

Non-separable data: There are one or more interior data points within the margin. In this case, the SVM aims to find the hyperplane that separates many but not all instances, and minimizes the number of errors. Thus, the optimization problem must be modified to allow misclassification errors. As suggested by Cortes and Vapnik (1995), non-negative slack variables $\xi_i \ \geq 0$, which measure the degree of misclassification of the instances x_i, are given. Figure 2.10 shows the linear hyperplane for non-separable data in two-dimensional space.

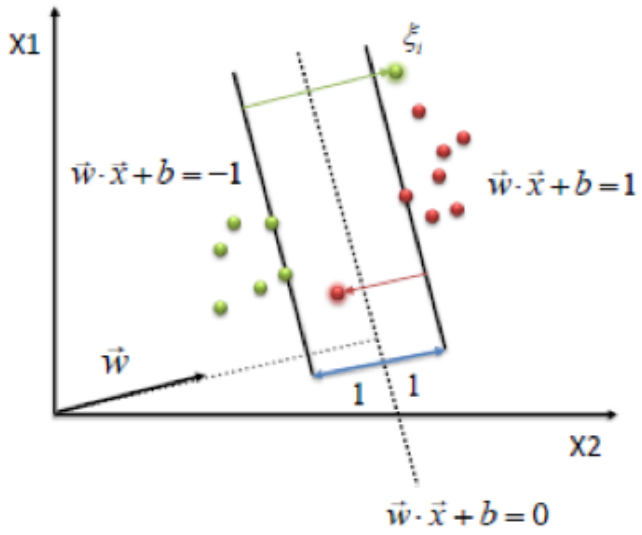


Figure 2.10: Optimal hyperplane for non-separable data in two-dimensional space (Sayad, 2015)

The optimization problem is now written as follows:

$$Minimize \left\{\frac{1}{2}\|\vec{w}\|^2 + C\sum_{i=1}^{n}\xi_i\right\}$$

$$Subject\ to\ y_i(\vec{w}.x_i + b) \geq 1 - \xi_i\ \forall\ i = 1, 2, \dots, n$$

in which C is a penalty parameter for determining the trade-off between maximizing the width of the margin and minimizing the number of the misclassified instances (Gun, 1998; Cristianini & Shawe-Taylor, 2000). Similar to the case of separable data, the solution to the optimization problem is to:

$$Maximize \left\{\sum_{i=1}^{n}\alpha_i - \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j x_i.x_j\right\}$$

$$Subject\ to \sum_{i=1}^{n}\alpha_i y_i = 0\ and\ \ 0 \leq \alpha_i \leq C\ \forall\ i = 1,2, \dots, n$$

The bounds of the Lagrange multipliers are modified by adding an upper bound of C for α_i. As proposed by Abe (2005), the used SVM for non-separable data is a soft-margin SVM.

Nonlinear classification: In reality, there are situations where a simple hyperplane does not exist as a useful separating criterion. Boser, Guyon and Vapnil (1992), therefore, propose a kernel function to handle this problem. The kernel function maps input data into a higher dimensional feature space by replacing the inner product. Then, a hyperplane allowing the linear separation can be found in the higher dimensional space (Burges, 1998). Figure 2.11 shows the transformation from the low-dimensional space to the high-dimensional space using a kernel function.

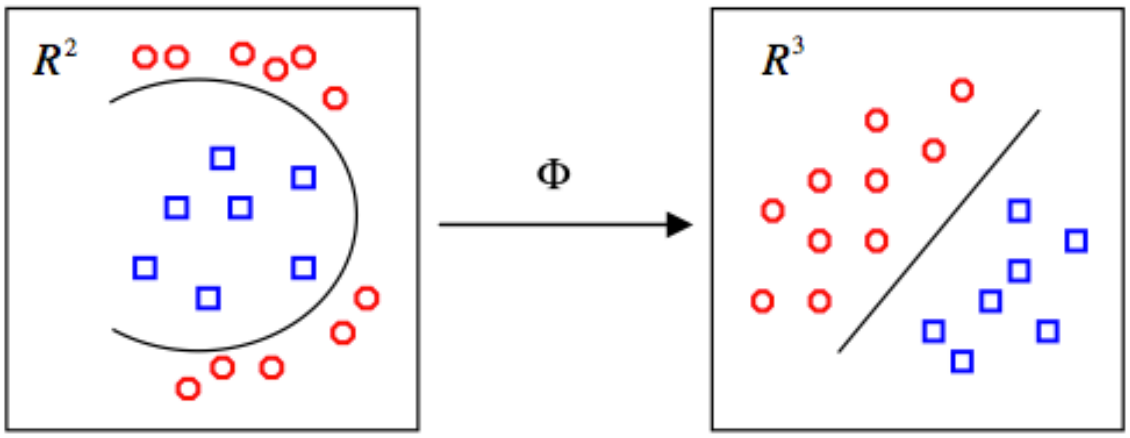


Figure 2.11: Mapping nonlinearly separable data from two-dimensional space into three-dimensional space (Elish & Elish, 2007)

The kernel function $K(x_i, x_j) = \phi(x_i).\phi(x_j)$ transforms the data between spaces using a nonlinear mapping ϕ, such that $x \rightarrow \phi(x)$ $in\ which\ \phi: R^n \rightarrow R^m$. After applying the kernel function, the optimization problem and decision function for classifying an instance t become:

$$Maximize \left\{ \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j K(x_i, x_j) \right\}$$

$$Subject\ to \sum_{i=1}^{n} \alpha_i y_i = 0\ and\ \alpha_i \geq 0\ \forall\ i = 1,2,\ldots,n$$

$$class(x_t) = sign\left(\sum_{i=1}^{n} \alpha_i y_i K(x_i, x_t) + b\right)$$

According to Gun (1998), Burges (1998), Abe (2005) and Ivanciuc (2007), there are several common kernel functions in machine learning:

$$Linear\ (homogeneous)\colon\ K(x_i, x_j) = x_i . x_j$$

$$Polynomial\ (inhomogeneous)\colon\ K(x_i, x_j) = \left(x_i . x_j + 1\right)^d$$

$$Radial\ basis\ function\ (gaussian)\colon\ K(x_i, x_j) = \exp\left(-\gamma \left\| x_i - x_j \right\|^2\right), for\ \gamma > 0$$

$$Neural\ (sigmoid, tanh)\colon\ K(x_i, x_j) = \tanh\left(\gamma\left(x_i . x_j\right) - r\right), for\ some\ \gamma, r > 0$$

Smola (1998) argues that the radial basis function is the most popular kernel used in SVM because of giving better performance than others. In terms of prediction, using SVM brings several advantages (Elish & Elish, 2007) that make SVM a useful classifier for predicting software defects.

- Since it is formulated as a quadratic programming problem, SVM provides a global optimum solution.
- The learning capability of SVM is independent of the attribute space dimensionality because it can be created effectively even in high-dimensional spaces under small training sample conditions.
- By using a kernel function introduced by Boser, Guyon & Vapnik (1992), the model is able to deal with nonlinear functional relationships that are problematic for other algorithms.
- Since it uses the margin parameter C for controlling misclassification errors, SVM is robust to outliers.

2.4 Data Preprocessing

Preprocessing techniques are important and widely used in machine learning, the foundation of most software defect prediction studies (Nam, 2009). There are many factors negatively affecting the performance of machine learning algorithms such as unreliable and irrelevant information or noisy data (Kotsiantis, Kanellopoulos & Pintelas, 2006). These problems can be tackled by using data preprocessing that offers techniques including data cleaning, normalization, attribute selection and extraction. Since differences in selecting models, subjects and metrics among studies, data preprocessing techniques may or may not be used and they are applied in various ways depending on each study.

2.4.1 Normalization

Data normalization is a basic preprocessing task in machine learning and data mining (Graf & Borer, 2001), which aims to improve performance of classification models by giving an equal weight to all attributes of a data set (Nam et al., 2013). There are a lot of usable normalization methods.

One of them is to use a log-filtering preprocessor presented by Menzies et al. (2007) to normalize values of static code attributes having exponential distributions. In this method, the logarithmic filter is used to replace all numeric values n with their logarithms $\ln n$. Experiments conducted by Menzies et al. (2007) show that after log-filtered, these values become more even and thus it is easier for prediction models to work on them. The log-filtering method has also been applied in several other studies having the same experimental subjects (Turhan, Menzies, Bener & Di Stefano, 2009; Turhan, Misirli & Bener, 2013).

Apart from log-filtering technique, other common normalization methods are *difference* (Boetticher, 2005) and feature scaling (Juszczak, Tax & Duin, 2002). These methods transform values of an original data set into a range from 0 to 1, and ensure that each attribute receives an equal weight. While the former is simple dividing every numeric value of each attribute by the *difference* $= x_{max} - x_{min}$, the latter subtracts the minimum

value from each attribute before implementing the division.). In the literature, the method of normalizing data by feature scaling is also known as min-max (Han, Kamber & Pei, 2012), and its formula is shown as below:

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Another normalization method, which is useful for normally distributed populations, is standard score, also called normal score or z-score (Kotsiantis et al., 2006).

$$x' = \frac{x - mean}{standard\ deviation}$$

The method of calculating standard score is based on the distribution mean and standard deviation for each attribute. After determining the mean and standard deviation, the mean is subtracted from each attribute. Then, values obtained from the subtractions of each attribute are divided by its standard deviation. According to Nam et al. (2013), this normalization method is very popular in many machine learning techniques, and it leads to a higher accuracy in predicting software defects.

2.4.2 Noise reduction

In order to build and evaluate prediction models, defect data is often extracted from log files, version controls and bug reports in bug tracking databases automatically by using tools (Kim et al., 2011). However, recent studies have shown that collected data from version controls, bug reports and change logs may be noisy. For example, studies of Aranda and Venolia (2009) have proven that a lot of information is missing in defect reports. In 2009, Bird et al. also found systematic bias in code version histories and bug tracking systems, which is a cause of the incorrectness of predicted results.

For measuring noise resistance of defect predictors, Kim et al. (2011) firstly propose to add false negative and positive information in training data sets while leaving test data unchanged. This aims to measure the accuracy of prediction algorithms. The authors then produce a noise detection method called closest list noise identification. This method uses Euclidean distance to compute the ratio of similarity between a training component and a

list of noisy components. The component will be considered as noisy if the ratio reaches a given threshold. According to Kim et al. (2011), the method is beneficial because collected data will be more suitable for defect prediction when noise can be detected and eliminated beforehand.

2.4.3 Attribute selection

In addition to noisy data, poor performance of defect predictors is also caused by the redundancy of learning attributes (Shivaji, Whitehead, Akella & Kim, 2013). In order to achieve precise predictions, all operators, comments, class names, variables and programming language keywords could be taken into consideration as attributes for training the predictors. Furthermore, static code metrics, object-oriented metrics and other metrics can also be used together for training data sets. These lead to a large attribute set. Nevertheless, it is typically infeasible for prediction models to handle such a large attribute set along with the presence of noise and complex interactions (Shivaji et al., 2013).

A possible solution for this problem is to select a subset of attributes providing the best performance of predictors. This method is known as feature selection, or attribute selection. There are a large number of approaches that have been proposed in the area of machine learning for attribute selection (Rodriguez, Ruiz, Cuadrado-Gallego & Aguilar-Ruiz, 2007; Jong, Marchiori, Sebag & Van Der Vaart, 2004; Ilczuk, Mlynarski, Kargul & Wakulicz-Deja; 2007; Forman, 2003; Chen, Menzies, Port & Boehm; 2005). Most of them are categories into two kinds: filter and wrapper approaches. The filter approaches, such as Chi-Squared, Gain Ratio and significance attribute evaluation (SAE), discard attributes having lowest significance until reaching optimal prediction performance. They use heuristics based on the characteristics of data for evaluating attributes (Hall, 1999). Meanwhile, the wrapper approaches evaluate attributes based on scores given by learning algorithms such as Naïve Bayes and support vector machine (Liljeson & Mohlin, 2014).

As argued by Hall (2000), filter approaches are faster than wrapper ones; thus, they are more suitable for data sets with a vast number of attributes. In addition, the filter approaches robust to over-fitting whereas the wrapper approaches may lead to over-

fitting if the number of training instances is not sufficient (Hammon, 2013). Another advantage of using filter approaches is the ability to work in conjunction with any machine learning algorithm while wrapper approaches use the same learning algorithms for both attribute selection and classification (Liljeson & Mohlin, 2014).

Nevertheless, the limitation of filter approaches is that they may cause redundancy of selected attributes (Khan et al., 2014). For example, a traditional filter approach is Relief (Arauzo-Azofra, Benitez & Castro; 2004) that focuses on retaining relevant attributes. After assigning a relevance value to each attribute, the technique assesses each attribute individually by using an evaluation function. Attributes are selected if their relevance values are greater than a given threshold. This approach does not take into account the dependencies between attributes. Therefore, they tend to select redundant attributes (Reena & Rajan, 2014).

2.5 Evaluation Measures

2.5.1 Measures for classification

In order to evaluate a software defect prediction model, the evaluation measures that include false positive rate, accuracy, precision, recall, balance and F-measure (Lee, Nam, Han, Kim & In, 2011) are applied.

Beforehand, the symbols *A*, *B*, *C*, *D* are used in which:

A is the number of defective modules predicted as defective.

B is the number of defective modules classified as defect-free.

C is the number of defect-free modules predicted as defective.

D is the number of defect-free modules predicted as defect-free.

Accuracy: The ratio is the number of modules correctly predicted to the number of total modules.

$$accuracy = (A + D) / (A + B + C + D)$$

From the equation, it is apparent that the accuracy is heavily affected by class balance. However, software data sets typically have a lot more defect-free modules than defective modules (Rahman, Posnett & Devanbu, 2012). If a model predicts all modules to be defect-free, the accuracy will be very high although no defective modules are correctly predicted. For instance, average defective rate of PROMISE data sets (Peters & Menzies, 2012) is 18%. If a prediction model declares all modules as defect-free, its accuracy will be 82% without correctly predicting any defective module. Due to the imbalance between classes in the data sets, accuracy cannot be considered as a proper measure for comparing prediction models (Nam, 2009).

False positive rate: False positive rate is also known as probability of false alarm (*pf*) (Menzies et al., 2007). The ratio is the number of defect-free modules wrongly predicted as defective to the number of defect-free modules.

$$pf = C / (C + D)$$

Recall: Recall is also known as true positive rate or probability of detection (*pd*) (Menzies et al., 2007). The ratio is the number of defective modules correctly predicted as defective to the number of defective modules.

$$recall = pd = A / (A + B)$$

Balance: In order to select an optimal pair of probability of false alarm and probability of detection (*pf, pd*), Menzies et al. (2007) propose a measure called balance (*bal*). The measure is a normalized Euclidean distance from the desired point (*pd = 1, pf = 0*) to a pair of (*pf, pd*).

$$balance = bal = 1 - \frac{\sqrt{(1 - pd)^2 + (0 - pf)^2}}{\sqrt{2}}$$

Although the balance is a fairly useful measure used in evaluation of defect predictors (Menzies et al., 2007; Jiang, Cukic & Ma, 2008), it is uncommon and still has drawbacks. As the balance is a distance, prediction models having different values of (*pf, pd*) could have the same value of the balance. However, this does not mean that those models have equal performance in practice (Song et al., 2011). Also, Menzies et al. (2007) state that

using a pair of (*pf, pd*) is impractical in classifying imbalanced data sets because of the low ratio of precision.

Precision: The ratio is the number of defective modules correctly predicted as defective to the number of modules predicted as defective.

$$precision = A / (A + C)$$

A prediction model presents a good performance if achieving higher values of recall, precision and lower values of false positive rate. Nevertheless, it is well-known that recall may be improved by diminishing precision and vice versa (Rahman et al., 2012). Because of the trade-off between precision and recall, it is not easy to compare performances of defect predictors based on only either recall or precision (Kim, Whitehead & Zhang, 2008; Alpaydin, 2010; Zheng, Wu & Srihari, 2004). As a result, F-measure, a composite measure of recall and precision, has been used to compare prediction outcomes (Lee et al., 2011; Nam et al., 2013).

F-measure: The harmonic function of recall and precision (Lee et al., 2011).

$$F - measure = 2 * recall * precision / (recall + precision)$$

F-measure has been utilized in numerous defect prediction papers (Kim et al., 2011; Lee et al., 2011; Nam et al., 2013; Rahman & Devanbu, 2013; Wu, Zhang, Kim & Cheung; 2011). This function presents a unified score to evaluate prediction model after balancing out the trade-off between recall and precision. The value of F-measure is directly proportional to the performance of a model.

It is obvious that all of the above measures are computed based on the values of A, B, C, D, which are results of binary decisions from the predictor. Nevertheless, many defect prediction models classify a model by giving a defective probability instead of a binary decision (Lessman, Baesens, Mues & Pietsch, 2008; Mende, 2010). This poses a question on how to define the discretization of the continuous probability. To address this question, Rahman et al. (2012) propose to use a threshold of the probability in which a module is considered as defective if its defect proneness probability reaches the threshold. Otherwise, the module is classified as defect-free.

AUC: AUC signifies the area under the receiver operating characteristic (ROC) curve. AUC is a non-parametric measure that is independent of the threshold and is not influenced by class imbalance (Rahman et al., 2012). ROC is a two-dimensional curve that is plotted by probability of false alarm (x-axis) and probability of detection (y-axis). According to Rahman and Devanbu (2013), a model is better when its ROC curve is close to the point of *pd = 1* and *pf = 0*. A perfect predictor, hence, has AUC of 1. In contrast, a negative curve illustrates a poor model with high probability of false alarm and low probability of detection. Elsewhere, Menzies, DiStefano, Orrego and Chapman (2004) have found that if the model negates its prediction results, the negative curve will turn out to be a preferred curve representing for a good predictor. A random model offering no information always has AUC of 0.5 because it tends to be close to the diagonal *pd = pf* (Rahman et al., 2012; Menzies et al., 2007). Probability of false alarm and probability of detection are various depending on the threshold for prediction probability of each predicted module (Nam, 2009). Nonetheless, AUC takes all possible threshold values into consideration of prediction performance. Therefore, AUC is independent of the threshold, and it is a stable measure.

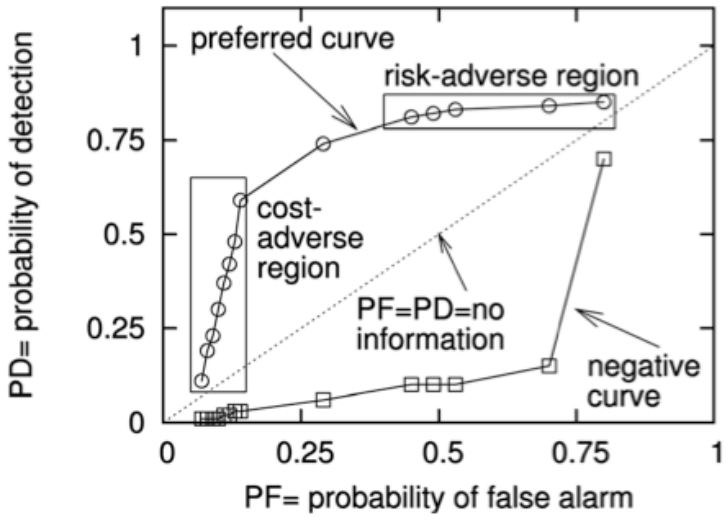


Figure 2.12: A typical ROC curve (Menzies et al., 2007)

AUCEC: Although AUC is a useful measure for evaluating the performance of software defect predictors, it does not take cost-effectiveness into account. Therefore, Arisholm, Briand and Fuglerud (2007) present a measure known as area under cost-effectiveness

curve (AUCEC). The cost-effectiveness in defect prediction is identified based on the percentage of bugs that can be found among a certain percentage of lines of code inspected (Nam, 2009). More specifically, a model will be more cost-effective if it can find more defects with less inspecting effort.

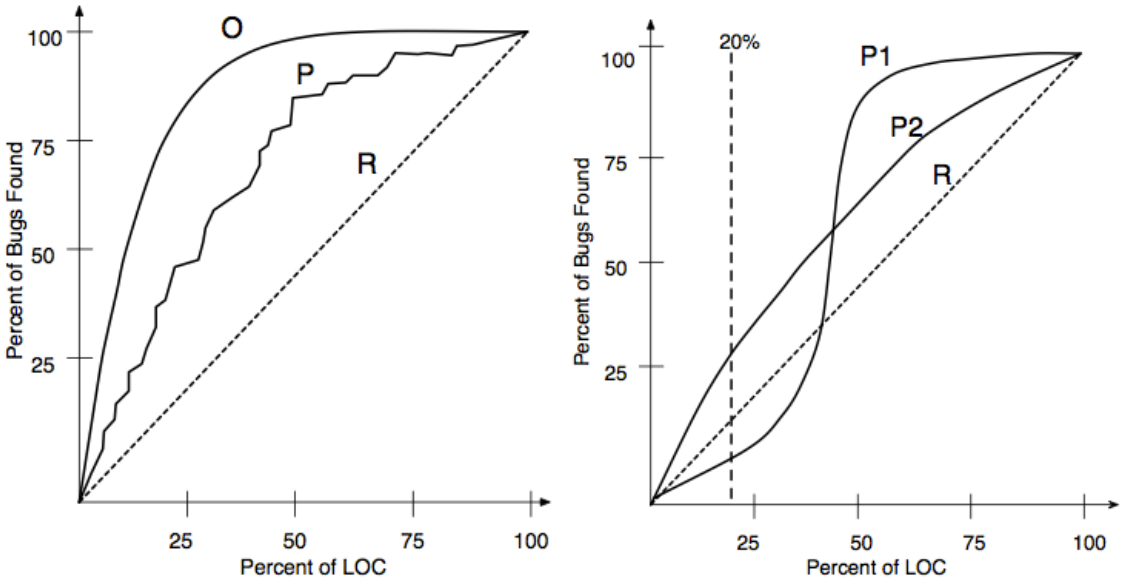


Figure 2.13: Cost-effective curve (Rahman, Posnett, Hindle, Barr & Devanbu, 2011)

An example of cost-effectiveness curves is shown in Figure 2.13 where x-axis represents the percentage of lines of code and y-axis is the percentage of bugs found. The left graph shows random, practical and optimal predictors that are represented by R, P and O curves respectively. Since a random predictor would pick random modules as defective, the percentages of defects found and lines of code inspected are equal. In this case, it has AUCEC of 0.5. Meanwhile, according to Rahman et al. (2012), AUCEC of the optimal predictor is highest compared to others. A higher AUCEC means that the predictor can find more bugs by testing fewer lines of code than others.

The right graph illustrates cost-effectiveness curves of two different predictors, P1 and P2. The two models have identical overall performance because their AUCEC are equal when testing the whole lines of code. However, with a threshold of 20% of lines, P2 has higher AUCEC than P1. This means that if inspecting only 20% of lines of code, P2 outperforms P1. As a result, in order to apply AUCEC as a defect prediction measure, it

is necessary to consider a particular threshold for the percentage of lines of code (Nam, 2009). As argued by Arisholm et al. (2007), this concept of AUECE is suitable for predicting defects in telecommunications software.

2.5.2 Discussion on measures

The statistics implemented by Nam (2009) show the number of each measure employed in typical software defect prediction studies for evaluating classification models. As illustrated in Figure 2.14, F-measure is the most often tool utilized for prediction model evaluation. In fact, this is reasonable because the trade-off between recall and precision causes difficulties to evaluate performance of prediction models with high recall but low precision and vice versa. Thus, F-measure, a harmonic of the two measures, has been chosen in numerous defect prediction studies (Peters & Menzies, 2012; Rahman et al., 2012; Shihab, Mockus, Kamei, Adams & Hassan, 2011). Nonetheless, different thresholds for prediction probability of a software component make F-measure varying. When a predictor classifies a component as defective, it gives prediction probability that is greater than a threshold. Lessmann et al. (2008) have indicated that, in software defect prediction literature, thresholds are often overlooked. This leads to inconsistent prediction outcomes across studies. In order to tackle this problem, AUC and AUCEC measures, which are independent from thresholds, have been used (Giger, D'Ambros, Pinzger & Gall, 2012; Lessmann et al., 2008; Rahman et al., 2012; Song et al., 2011).

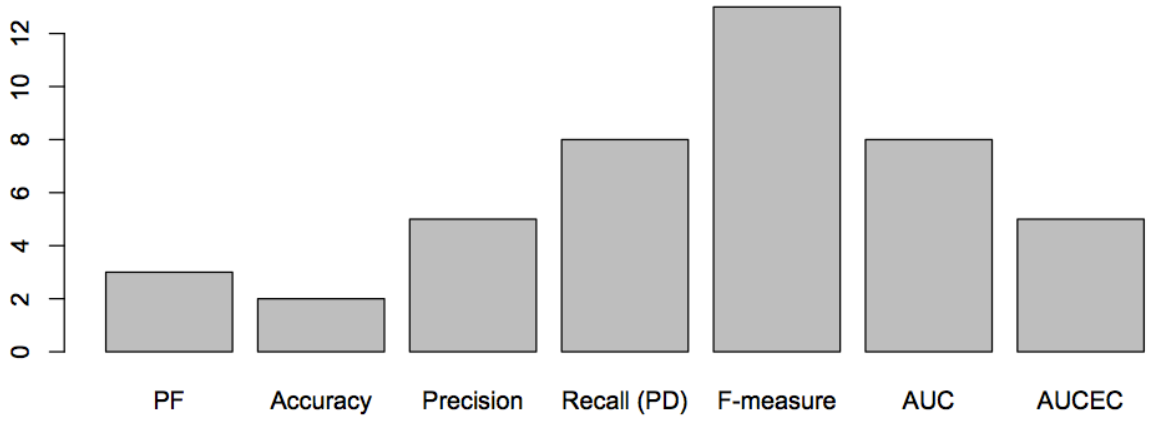


Figure 2.14: Total of evaluation measures employed in representative software defect prediction studies for evaluating classification models (Nam, 2009)

2.6 Challenges and Proposed Solutions of Software Defect Prediction

There are several challenges of software defect prediction in the literature. Firstly, although software defect predictors can be applied before releasing software products, it would be more useful if defects can be predicted whenever source code is changed. From this idea, Mockus and Votta (2000) produce a model for software changes based on historical databases. Recently, this type of predictors, which is known as just-in-time model, has also appeared in several studies (Kim, Whitehead & Zhang, 2008; Fukushima, Kamei, McIntosh, Yamashita & Ubayashi, 2014).

Secondly, the key of software defect prediction is how to effectively analyze and use existing historical data for creating more precise classifiers (Jing, Ying, Zhang, Wu & Liu, 2014). Nevertheless, the classification approaches often encounter certain difficulties including the issue of misclassification cost (Zheng, 2010) and the class imbalance problem (Zhou & Liu, 2006; Menzie et al., 2007; He & Garcia, 2009). Misclassification implies that defect-free modules are predicted as defective or vice versa while class imbalance happens when a data set contains significantly fewer defective modules than defect-free ones, which negatively impacts decision of classifiers. In order to deal with those problems, Jing et al. (2014) propose to use a Cost-sensitive Discriminative Dictionary Learning (CDDL) method. For enhancing the classification capability, CDDL takes misclassification cost into consideration by increasing punishment on the second type of misclassification (defective modules are predicted as defect-free). Moreover, in order to mitigate the class imbalance problem, CDDL uses the principal component analysis technique for initializing sub-dictionary for each class. Another solution for this problem is to apply data-resampling strategies before learning prediction models. In fact, the strategies such as random under- and over-sampling have been proposed by Pelayo and Dick (2012). In the literature, a number of studies have used these strategies (Wang & Yao, 2013; Liu, Miao & Zhang, 2014; Laradji, Alshayeb & Ghouti; 2015; Roh, Stoeckel & Lee; 2015).

Although the above approaches can tackle complex problems in predicting software defects, they are only useful in within-project settings (Nam et al., 2013). For new

software projects and projects having limited historical data, it is important to build a defect prediction model based on sufficient data of existing software projects and then employ this model for new projects. Since process metrics are increasingly widely used, Zimmermann, Nagappan, Gall, Giger and Murphy (2009) consider this drawback as one of the most considerable difficulties in the study of software defect prediction. In order to overcome this difficulty, Zimmermann and Nagappan (2008) produce a cross-project defect prediction model. Experiments have been conducted based on training data from 12 projects with 622 cross-project predictions, but the results indicate a low success rate of only 3.4%. Pan and Yang (2010) argue that performances of cross-project predictors are normally poor because source and target projects are significantly different in feature distribution. Most machine learning techniques are designed with a common assumption that training and test data are in the same feature space and have the same data distribution. If the distribution changes, a prediction model needs to be rebuilt based on newly collected data. Unfortunately, it is difficult and expensive to recollect training data as well as rebuild prediction models in real-world applications. Hence, it is desirable to apply transfer learning techniques to cross-project defect prediction (Pan & Yang, 2010).

The transfer learning methods mostly aim to generate knowledge from a source domain and transfer it to a target domain. The transfer knowledge is utilized for training a defect prediction model. In practice, several methods have been produced (Fan, Davidson, Zadrozny & Yu, 2005; Dai, Xue, Yang & Yu, 2007; Watanabe, Kaiya, & Kaijiri, 2008; Turhan, Menzies, Bener & Di Stefano, 2009; Ma, Luo, Zheng & Chen, 2012). One of popular methods for domain adaption is Transfer Component Analysis (TCA) proposed by Pan et al. (2012). The basic idea of this method is that there may have common components underlying source and target domains if they are related to each other. Several of these components may be the reason for the differences between two domains while several may capture discriminative information and intrinsic structure of the original data. Therefore, the method aims to find the components in which the feature distributions of the two domains are similar by utilizing projection. The projection is a feature generation technique in machine learning for reducing differences between the feature distributions and still preserving the original data characteristics. Once those components are learned, the data of two projects can be mapped onto them. By using

TCA, a classifier is built on source data and then is applied to transformed target data for defect prediction.

Although using feature distributions in transforming data between source and target projects instead of defect information, Nam et al. (2013) observe that outcomes of cross-project defect prediction using TCA vary depending on normalization options applied for data preprocessing. Therefore, Nam et al. (2013) produce Transfer Component Analysis + (TCA+) by extending TCA to improve cross-project defect prediction performance. Specifically, TCA+ adds a set of decision rules for selecting appropriate normalization options into TCA. In fact, with the benefits in transferring knowledge across domain, TCA is worth to be applied to not only cross-project defect prediction but also other prediction and recommendation systems (Rigby & Hassan, 2007; Jeong, Kim & Zimmermann, 2009; Shin, Mennely, Williams & Osborne, 2011).

Another limitation in the literature is that many studies do not take into account label information from training data. Indeed, most of the implementations are to use software metrics and machine learning techniques for mining data sets from software archives. This method is helpful to produce effective models for predicting software defects. Nevertheless, many studies learn prediction models without using label information, which is important to distance-based classification algorithms like *k*-nearest neighbor (Cover & Hart, 1967). Motivated by this issue, we propose a novel approach to improve the performance of software defect prediction models, *"metric learning for software defect prediction"*. The common objective of metric learning is to learn an appropriate metric for distance functions such as Mahalanobis distance (Mahalanobis, 1936), which computes similarity between instances. With the ability of fully exploiting label information, metric learning used in conjunction with *k*-nearest neighbor classifier would be useful to create software defect predictors.

CHAPTER 3 – METRIC LEARNING FOR SOFTWARE DEFECT PREDICTION

3.1 Introduction to Metric Learning

One of the fundamental concepts in machine learning is distance metric learning, which is to measure distances between different objects. Specifically, metric learning attempts to adapt pairwise metric functions to measure similarity or dissimilarity between pairs of objects. The distance between a similar pair should be relatively smaller than that between a dissimilar pair. A simple example for this statement is that, in classification settings, instances in the same class can be considered as similar while those in the distinct classes are dissimilar. In fact, the performance of distance-based algorithms such as *k*-nearest neighbor (Cover & Hart, 1967) classification significantly relies on the performance of used metric learning method (Weinberger & Saul, 2009; Ying & Li, 2012; Bellet, Habrard & Sebban, 2013). Good metric learning methods can provide insight into latent structure of data, and they are useful for creating better data visualization via embedding. If the method could correctly identify whether objects were similar or not, as well as accurately estimate the degree of similarity or dissimilarity, subsequent tasks of prediction models would be trivial (Weinberger & Saul, 2009). For this reason, it is desirable to apply an appropriate distance metric learning for each specific problem.

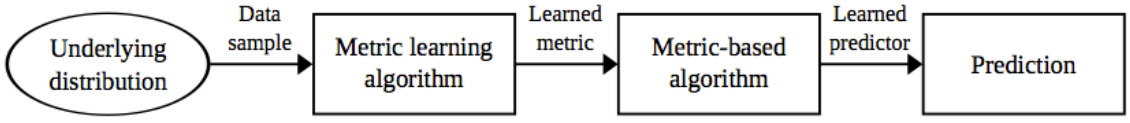


Figure 15: A common metric learning process (Bellet et al., 2013)

3.2 Mahalanobis Distance

A common function of metric learning is Mahalanobis distance (Mahalanobis, 1936) that is sufficiently effective to work on real-world problems (Davis, Kulis, Jain, Sra &

Dhillon, 2007). In the field of software defect prediction, this method is useful for measuring how similar a test software component is to a training component. Pairs of components with smaller distances should be more similar.

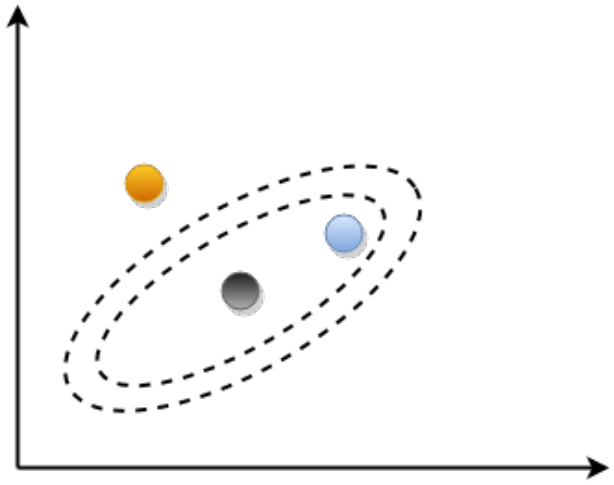


Figure 16: The distances between objects

The Mahalanobis distance between two vectors $\vec{x}, \vec{z}$ is defined as follows:

$$d(\vec{x}, \vec{z}) = \sqrt{(\vec{x} - \vec{z})^T C^{-1} (\vec{x} - \vec{z})}$$

where $\vec{x}$ is vector of a test data instance, $\vec{z}$ is vector of each training data instance and C^{-1} is the inverse of covariance matrix of data.

$$C = \frac{1}{n} \sum_{i=1}^{n} (\vec{z}_i - \bar{z})(\vec{z}_i - \bar{z})^T$$

where n is the number of training data points, $\bar{z}$ is the mean of the training data set. Using the inverse of covariance matrix is very common in estimating the Mahalanobis distance. However, this method has a few drawbacks. Firstly, it does not use the label information. This could lead to poor performances of software prediction models. Secondly, Maesschalck, Jouan-Rimbaud and Massart (2000) argue that in many cases, the covariance matrix is singular or nearly singular, and may not be inverted due to multi-collinearity in data sets. Specifically, the data sets may contain much highly correlated or redundant information when they are measured based on a large number of attributes.

Furthermore, a fundamental requirement of calculating the covariance matrix is that the number of attributes needs to be smaller than the number of instances in data. Therefore, as proposed in section 2.4.3, it is necessary to apply attribute selection in advance.

In fact, Globerson and Roweis (2005) claim that metric learning and attribute selection have a close connection in terms of measuring distances between instances in an attribute space. For example, distance between $x_1 \; and \; x_2 \in X$ can be measured using Euclidean distance $d[f(x_1),\ f(x_2)]$ in a r-dimensional attribute space. By fixing function $d[f(x_1),\ f(x_2)]$, an attribute selection method can be seen as a metric learning method. Assuming that $f(x) = Wx$ is a linear projection of $x \in R^r$ in which r is the number of attributes then the Euclidean distance $d[f(x_1),\ f(x_2)]$ becomes the Mahalanobis distance $\|f(x_1) - f(x_2)\|^2 = (x_1 - x_2)^T M (x_1 - x_2)$ in which $M = W^T W$ is a positive semi-definite matrix. In terms of classification, there exists an equivalence relation in which instances in the same class should be close, and instances in different classes should be far. Learning matrix M for minimizing various separation criteria between classes is the key objective of metric learning (Goldberger, Roweis, Hinton & Salakhutdinov, 2004; Xing, Andrew Ng, Jordan & Russell, 2004).

3.3 Maximally Collapsing Metric Learning

Maximally Collapsing Metric Learning (MCML) proposed by Goldberger and Roweis (2005) is an innovative approach to learning metric. The main idea of the approach is to assume that all instances in the same class are close, and those in different classes are far. The approach uses the label information for mapping all instances in the same class into a single place in attribute space and those in other classes into other places. This gives an optimal approximation of the equivalence relation. Also, the approach can estimate fairly accurately local covariance structure of data.

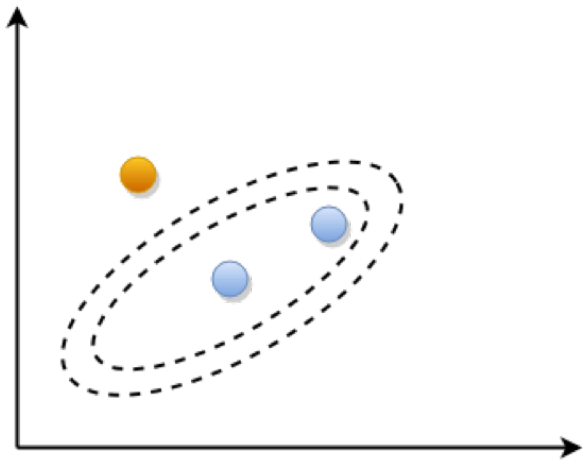


Figure 17: Instances in the same class are close, and those in different classes are far

3.1.1 Approach of collapsing classes

Given a set of n labeled training instances (x_i, y_i) in which

$$x_i \in R^r, r \; is \; the \; number \; of \; attributes$$

$$y_i \in \{1, 2, \dots, k\}, k \; is \; the \; number \; of \; classes$$

The distance of two instances in X input space is learned using Mahalanobis function under a positive semi-definite matrix M.

$$d_{ij}^M = (x_i - x_j)^T M (x_i - x_j)$$

The metric after learned would make instances in the same class close and those in different classes far. To begin with, the approach assumes that distance between instances in the same class is zero while distance between instances in different classes is infinite. In other words, the approach tries to map same class instances into a single point using a linear projection. The ideal distribution is presented as follows:

$$p_0(j|i) \propto \begin{Bmatrix} 1 & if \; y_i = y_j \\ 0 & if \; y_i \neq y_j \end{Bmatrix}$$

Then, a conditional distribution for each training instance over others is defined as follows:

$$p^M(j|i) = \frac{e^{-d_{ij}^M}}{Z_i} = \frac{e^{-d_{ij}^M}}{\sum_{k \neq i} e^{-d_{ik}^M}} \qquad where\ i \neq j$$

The goal of the approach is to find matrix M such that $p^M(j|i)$ is as close as possible to $p_0(j|i)$. In order to match those distributions, KL divergence (Kullback & Leibler, 1951), which measures the difference between two probability distributions, is minimized.

$$\min_M f(M) = \min_M \sum_i KL[p_0(j|i)|p^M(j|i)]$$

The objective function can be rewritten as follows:

$$f(M) = -\sum_{i,j:y_i=y_j} \log p^M(j|i)$$

3.1.2 Optimization of the minimal problem

As proven by Goldberger and Roweis (2005), a critical characteristic of this minimization problem is the convexities of matrix M and also $f(M)$. Thus, there would be only a single minimum value for the globally optimal solution. There are various convex optimization algorithms that can be used for optimizing the problem. One of them is the projected gradient algorithm introduced by Xing, Jordan and Russell (2004). The following is a summary of the algorithm:

Input: A set of n labeled training instances (x_i, y_i)

Output: A positive semi-definite metric M that optimally collapses classes

Initialization: Using an identity matrix for the initialization of matrix M_0

Iteration:

$Set\ M_{t+1} = M_t - \varepsilon * g(M_t)\ in\ which$

$$g(M_t) = \sum_{ij}\left(p_0(j|i) - p^M(j|i)\right)\left(x_j - x_i\right)\left(x_j - x_i\right)^T$$

$Compute\ the\ eigen - decomposition\ of\ M_{t+1}$

$$M_{t+1} = \sum_k \lambda_k v_k v_k^T\ , then\ set\ M_{t+1} = \sum_k \max(\lambda_k, 0) v_k v_k^T$$

Given a set of n labeled training instances with vector feature x_i and label y_i, firstly, matrix M_0 is initialized as an identity matrix. Then, an Armijo step-size rule (Armijo, 1966; Bertsekas, 1976) in the direction of the negative gradient is taken at each iteration. Finally, every negative eigenvalue of the matrix M_{t+1} will be replaced by zero to obtain a positive semi-definite matrix.

CHAPTER 4 – EXPERIMENTS AND EVALUATION

4.1 Framework Design

In general, the software defect prediction framework includes two stages: scheme evaluation and software defect prediction as shown in Figure 4.1 (Song et al., 2011). Before constructing software defect predictors and using them for future predictions, researchers firstly need to design learning schemes and evaluate them based on historical data. Then, according to the evaluation, the learning scheme with better performance would be taken to build a defect predictor for new data. The historical data used at the first stage is divided into two data sets: a training data set for constructing learners and a test data set for evaluating them. Meanwhile, at the second stage, the software defect prediction model is built using all of the historical data, which could improve the general performance of the model (Song et al., 2011).

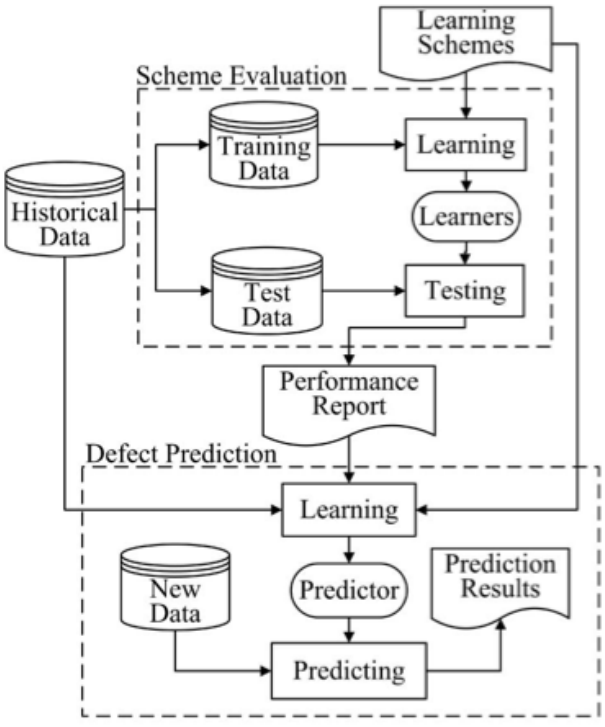


Figure 18: The general software defect prediction framework (Song, 2011)

In the thesis, we design different learning schemes to build different predictors, and evaluate them based on various measures. Firstly, in order to obtain training and test data sets, we apply 10-fold cross validation to historical data extracted from PROMISE repository (Sayyad Shirabab & Menzies, 2005). At each round of the cross validation, the original data set is partitioned into 10 subsets in which 9 subsets are used for training learners and the remaining subset is treated as test data. By doing so, the test data set will not be used in building the predictors. The independence of the test set from constructing predictors is crucial for correctly assessing performances of the predictors.

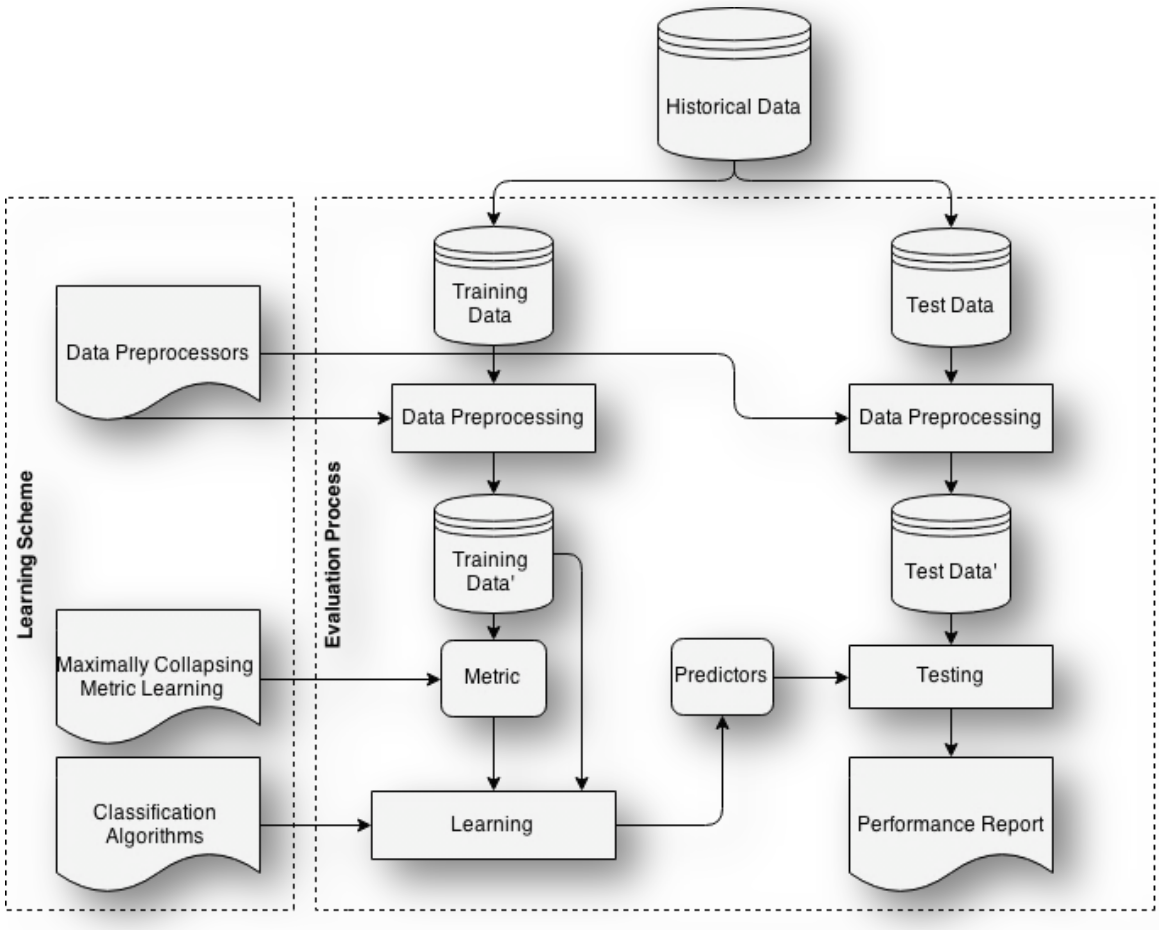


Figure 19: Our framework design for software defect prediction

Then, data normalization methods are applied to both training and test sets for data preprocessing. In the experiments, we use three different methods of data normalization including log-filter (Menzies et al., 2007), feature scaling (Juszczak et al., 2002) and

standard score (Kotsiantis et al., 2006). As stated in section 2.4.1, normalizing data is essential to handle missing values, to remove outliers as well as to discretize and transform numeric features.

After preprocessing data, MCML technique is applied to learn a metric from the preprocessed training data. This metric will later be used in conjunction with Mahalanobis distance to estimate the similarity between test and training data.

Once the metric has been learned, the predictors are constructed based on a very common classification algorithm: *k*-nearest neighbor. The numbers of nearest neighbors (*k*) are chosen as odd numbers from 1 to 11. In order to improve the performance of *k*-nearest neighbor classifier, two weighting methods are also applied. Those methods are based on the inverse of the distance and the number of instances of each class in the training data set (section 2.3.3).

Finally, the predictors are evaluated by comparing predicted values and actual values of the test set via label information: defective or defect-free. There are various measures used for performance evaluation introduced in section 2.5. Performances of the predictors are evaluated based on two widely used performance measures in the field of software defect prediction: *balance* and *f-measure*. While *balance* represents an optimal pair of *probability of false alarm* and *recall*, *f-measure* is a unified score to evaluate predictors after balancing out the trade-off between *recall* and *precision*. A good predictor would present high values of *balance* and *f-measure*.

Apart from examining the effectiveness of software defect prediction models, the main objective of the experiments and evaluation is to compare the performance between *k*-nearest neighbor predictors using MCML and the baseline *k*-nearest neighbor predictors using the inverse of covariance matrix.

The pseudo code from Figure 4.3 describes detailed learning scheme and evaluation process. The function involves 10 cross validation iterations, 3 different ways for data normalization, 6 different values for the number of nearest neighbors. When constructing the predictors, 2 different weighting methods are also applied. In summary, we have

designed 72 different learning schemes including cases of not using any data normalization and weighting method.

```
Procedure Evaluation(historicalData)
    Input: historicalData /* the data extracted from PROMISE repository */
    Output: avgResult /* the mean performance of a learner over 10-fold cross validation */

    R = Randomize(historicalData);
    Generate 10 subsets from R;
    for i = 1 to 10 do
        test = subset[1];
        train = R - test;
        for j = 1 to 4 do
            /*
                * 1: no normalization applied
                * 2: standard score
                * 3: feature scaling
                * 4: log-fiter
            */
            [test, train] = Normalization(test, train, j);

            /* learning M using Maximally Collapsing Metric Learning */
            M = mcml(train);

            for k = 1 to 11 step 2 do
                /* k used for kNN should be odd */
                predictor = Construction(train, M, k);
                result = Evaluation(test, predictor);
            end
        end
    end
    avgResult = mean(result);
end
```

Figure 20: The pseudo code for detailed learning scheme

4.2 Data Sets

For the experiments, 5 NASA data sets generated from PROMISE software engineering repository have been used (Shirabab & Menzies, 2005). Those projects have been developed in C and C++ programming languages. The data is collected from various projects including spacecraft instrument, storage management, flight software, scientific data processing and real time projects (Zhang, Zhang, & Gu 2007). The collection of those publicly available data sets, shown in Table 4.1, is a useful tool for creating predictive software models.

Data set	Project	Language	Instances	Class distribution (%)	
				Defective	Defect-free
KC2	Data processing	C++	522	20.55	79.55
KC1	Store management	C++	2109	15.45	84.55

JM1	Real time	C	10885	19.35	80.65
PC1	Flight control	C	1109	6.94	93.06
CM1	Instrument	C	498	9.84	90.16

Table 10: The description of 5 NASA data sets

Each data set consists of a number of software instances. An instance is labeled as defective if it contains one or more bugs. Otherwise, it is labeled as defect-free. Apart from label information, NASA also adds numeric values of 21 different features to each data set. The set of features includes 3 McCabe metrics, 4 base Halstead metrics, 8 derived Halstead metrics, 5 different line-of-code metrics and a branch count of the flow graph. Table 4.2 depicts details of the features and their description.

Feature	Description
loc	McCabe: line count of code
v(g)	McCabe: cyclomatic complexity
ev(g)	McCabe: essential complexity
iv(g)	McCabe: design complexity
n	Halstead: total operands and operators
v	Halstead: volume
l	Halstead: program length
d	Halstead: difficulty
i	Halstead: intelligence
e	Halstead: effort
b	Halstead: delivered defects
t	Halstead: time estimator
lOCode	Halstead: line count
lOComment	Halstead: line count of comments
lOBlank	Halstead: count of blank lines
lOCodeAndComment	Line count of code and comment
uniq_Op	Unique operators
uniq_OPnd	Unique operands
total_Op	Total operators
total_Opnd	Total operands
branchCount	Branch count of the flow graph

Table 11: The collection of used features

4.3 Results and Analysis

Table 4.3 describes the performance of software defect predictors built using *k*-nearest neighbor with maximally collapsing metric learning (*k*-NN with MCML) and the baseline *k*-nearest neighbor with the inverse of covariance matrix (*k*-NN with ICM). The two methods have been evaluated in terms of *balance* and *f-measure*. As expected, our method presents satisfactory results, and it is superior to the baseline for all five data sets. The overall average *balance* of the proposed method is 50.97 percent, which is better than the baseline (44.12 percent). Similarly, the mean prediction *f-measure* of *k*-NN with ICM over the five data sets is 25.65 percent whereas the mean prediction *f-measure* of *k*-NN with MCML is 35.77 percent, with an improvement of 10.12 percent.

Data set	***K*-NN with ICM (%)**		***K*-NN with MCML (%)**	
	Balance	*F-measure*	*Balance*	*F-measure*
KC2	53.86	36.01	60.57	46.80
CM1	32.34	18.59	39.65	29.89
JM1	50.68	35.49	60.12	45.79
PC1	31.89	12.01	37.88	20.55
KC1	51.81	26.15	56.62	35.80
Overall average	**44.12**	**25.65**	**50.97**	**35.77**

Table 12: Comparison of our method and the baseline for all five data sets

It is noticeable that the models created using either method produce the best results when performing on the KC2 data set. By contrast, they perform worst on the PC1 data set. For the KC2 data set, the models using *k*-NN with MCML achieve a 60.57 percent *balance* and 46.80 percent *f-measure* while the percentages for PC1 are only 37.88 percent *balance* and 20.55 percent *f-measure*. The difference in performance can also be seen at prediction models using *k*-NN with ICM. This difference can be explained by the class imbalance problem that has an adverse effect on the performance of software defect prediction models. Indeed, it is clear from Table 4.1 that the ratio of defective instances to defect-free instances in the KC2 data set is 20.55 percent to 79.55 percent. Meanwhile, the ratio in the PC1 data set is much lower, with only 6.94 percent to 93.06 percent. Since the KC2 data set has less influence of class imbalance than others, the results of predictors built on this data set are taken for the more detailed comparison.

Normalization	*K*-NN with ICM (%)		*K*-NN with MCML (%)	
	Balance	*F-measure*	*Balance*	*F-measure*
None	61.21	40.45	62.56	47.00
Standard score	51.75	32.21	65.64	45.84
Feature scaling	50.64	36.88	56.31	47.10
Log-filter	50.49	34.49	59.11	47.24
Overall average	**53.52**	**36.01**	**60.91**	**46.80**

Table 13: Comparison of our method and the baseline for different data normalization techniques and KC2

Table 4.4 describes the performance of prediction models on KC2 with different data normalization techniques. From the table, the models using *k*-NN with ICM perform significantly worse when applying data normalization, but there is no corresponding decrease in the performance of models using *k*-NN with MCML. Although applying data normalization does not improve the performance of predictors, our method outperforms the baseline for every normalization technique. On average, our method presents an improvement of 7.39 percent *balance* and 10.79 percent *f-measure* compared to the method of using *k*-NN with ICM. While the results for feature scaling and log-filter are very similar, the performance of models using standard score is better in *balance* but worse in *f-measure* compared to those using other data normalization techniques.

K	***K*-NN with ICM (%)**		***K*-NN with MCML (%)**	
	Balance	*F-measure*	*Balance*	*F-measure*
1	54.40	35.47	59.12	45.05
3	53.16	37.01	60.16	48.01
5	54.25	37.44	60.89	45.57
7	53.92	36.45	61.09	47.24
9	53.55	35.49	61.04	46.54
11	53.87	34.17	61.11	46.36
Overall average	**53.86**	**36.01**	**60.57**	**46.46**

Table 14: Comparison of our method and the baseline for different values of chosen nearest neighbors and KC2

Besides evaluated by different data normalization techniques, the prediction performances of the two methods are also compared based on different choices of the number of nearest neighbors (*k*). As can be seen from Table 4.5, our method produces

better results than the baseline for all six ways of choosing *k*, with a gap of 6.71 percent and 10.45 percent in *balance* and *f-measure* respectively. It is interesting that the performance of prediction models using *k*-NN with MCML is slightly better in terms of *balance* when increasing value of *k*. However, the overall performance for either method tends to remain stable for all choices of *k*. This means that the prediction models are robust to the number of nearest neighbors.

Weight	***K*-NN with ICM (%)**		***K*-NN with MCML (%)**	
	Balance	*F-measure*	*Balance*	*F-measure*
None	55.67	34.08	63.12	46.17
ID	56.18	34.35	62.66	46.08
NIEC	49.73	39.58	55.92	48.14
Overall average	**53.86**	**36.00**	**60.57**	**46.80**

Table 15: Comparison of our method and the baseline for different weighting techniques and KC2

Another criterion for the evaluation is based on two different weighting techniques (section 2.3.3) using the inverse of the distance (ID) and the number of instances of each class (NIEC). The information from Table 4.6 indicates that again, our method is substantially superior to the baseline. On average, the differences between the two methods in terms of *balance* and *f-measure* are 6.71 percent and 10.80 percent, respectively. There is an interesting point that the models using NIEC technique perform worse in terms of *balance* but better in terms of *f-measure* than others. As stated in section 2.5, due to the low ratio of *precision*, *balance* is not very practical for evaluating predictors built on imbalanced data sets. Therefore, the improvement in terms of *f-measure* demonstrates that NIEC technique can be a useful tool to mitigate the class imbalance problem. For this reason, the performance of predictors using NIEC technique will be discussed in more detail.

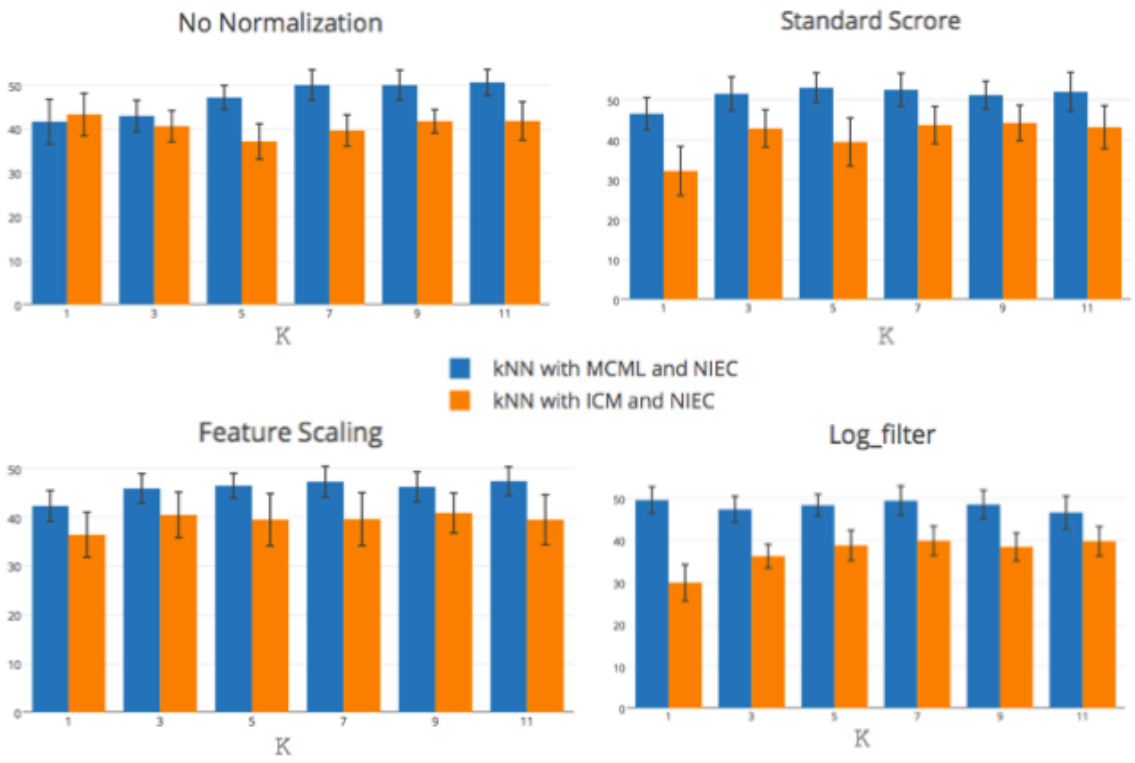


Figure 21: Comparison of f-measure of our method and the baseline for NIEC and KC2

The bar charts from Figure 4.4 show a visual comparison in terms of *f-measure* between defect prediction results of models using NIEC weighting technique. Overall, our method outperforms the baseline for all data normalization techniques and all choices of *k* except for *k* equal to one and without normalization. The biggest gap can be seen in the case of combining log-filter with the choice of *k* equal to one. In this case, the difference is significant, with around 20 percent *f-measure*. Specifically, the performance of predictors using *k*-NN with MCML and NIEC achieves roughly 50 percent *f-measure* while the *f-measure* of those using *k*-NN with ICM and NIEC is just over 30 percent.

CHAPTER 5 – CONCLUSION

5.1 Contributions

The main objective of this thesis is to propose a novel approach for improving the performance of software defect prediction models. The approach is *"metric learning for sofware defect prediction"*. Specifically, we apply Maximally Collapsing Metric Learning (MCML) to learn a Mahalanobis distance metric for improving classification capability of classifiers such as *k*-nearest neighbor.

At the outset, a systematic review has been conducted in order to gain an insight into the literature, and to investigate whether our approach is feasible. The review provides details of various kinds of software metrics, widely used machine learning algorithms, data preprocessing techniques and evaluation measures in the field of software defect prediction. Also, the review introduces a number of outstanding studies for building software defect prediction models that can efficiently assist in software effort estimation.

Many of those studies employ software metrics and distance-based classification algorithms such as *k*-nearest neighbor with Mahalanobis distance for mining data sets from software repositories. However, the common implementations of the Mahalanobis distance function do not use the label information from the training data. Motivated by this issue, we have decided to propose the approach of using metric learning, which makes use of the label information, to improve the performance of software defect predictors.

After introducing general ideas of metric learning and providing details of our approach, we have conducted experiments followed by the analysis and comparisons to verify the effectiveness of the approach. Firstly, an experimental framework with different learning schemes was designed. Then, the experiments have been conducted on five different NASA data sets generated from PROMISE software engineering repository. Results from the experiments were evaluated using *balance* and *f-measure*. Finally, the analysis and comparisons have been carried out to demonstrate that the method creating defect

predictors based on *k*-nearest neighbor with MCML is superior to the baseline (*k*-nearest neighbor with the inverse of covariance matrix).

5.2 Practical Issues and Future Works

We are confident that the research makes positive contributions towards the goal of improving the performance of software defect prediction models. Nevertheless, there are still practical issues that should be addressed to increase the adoption of our approach in practice. This section is to discuss the issues, and to provide possible avenues for future works.

The first issue is class imbalance. It has been clearly indicated in section 4.3 that the class imbalance problem has a negative effect on the performance of predictors. Although the weighting technique based on the number of instances of each class can mitigate this problem, the defect predictors built using our method do not deliver impressive results compared to many others in the literature. In fact, class imbalance happens when a data set contains significantly fewer defective instances than defect-free ones. Therefore, it would be more helpful if data-resampling strategies were applied before learning predictive models. It is possible that software defect prediction models constructed by using the combination of our method and random under-sampling strategy (Pelayo & Dick, 2012) could give better results.

In reality, apart from class imbalance, data sets extracted from software archives often contain much correlated information, and pair with random errors and noise. These lead to a problem called over-fitting occurring when a defect prediction model becomes excessively complex (Gaber, Zaslavsky & Krishnaswamy, 2005). The problem of over-fitting data reduces power of the model because if the model attempts to conform itself to inaccurate data, it can be infected with substantial errors. This problem is not trivial. Even though applying cross validation can reduce over-fitting, in future, we need to take into consideration the problem and find an actual solution to it.

Another limitation of our method is that software defect prediction models are built based on the training data sets with labeled software components. One of the fundamental

conditions of creating a defect prediction model is to find similar components with the target component in terms of software metrics from software archives. For domains having less labeled components, it would be difficult to find those components due to the lack of data for training. This would also lead to the inaccuracy of the models in predicting defects. For dealing with the problem of limited availability of labeled components, the method needs to be provided with the ability of fully exploiting both labeled and unlabeled data from software repositories. This not only tackles a major problem in within-project prediction, but also is a good solution for cross-project software defect prediction.

REFERENCES

Abe, S. (2005). *Support vector machines for pattern classification* (Vol. 2). London: Springer.

Akiyama, F. (1971). An Example of Software System Debugging. In *Proceedings of the International Federation of Information Processing Societies Congress*, pages 353–359.

Alpaydin, E. (2004). *Introduction to machine learning*: MIT press.

Aranda, J., & Venolia, G. (2009). *The secret life of bugs: Going past the errors and omissions in software repositories.* Paper presented at the Proceedings of the 31st International Conference on Software Engineering.

Arauzo-Azofra, A., Benitez, J. M., & Castro, J. L. (2004). A feature set measure based on relief. In *Proceedings of the fifth international conference on Recent Advances in Soft Computing* (pp. 104-109).

Arisholm, E., Briand, L. C., & Fuglerud, M. (2007). *Data mining techniques for building fault-proneness models in telecom java software.* Paper presented at the Software Reliability, 2007. ISSRE'07. The 18th IEEE International Symposium on.

Armijo, L. (1966). Minimization of functions having Lipschitz continuous first partial derivatives. *Pacific Journal of mathematics*, *16*(1), 1-3.

Bacchelli, A., D'Ambros, M., & Lanza, M. (2010). Are popular classes more defect prone?. In *Fundamental Approaches to Software Engineering* (pp. 59-73). Springer Berlin Heidelberg.

Basili, V. R., Briand, L. C., & Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on, 22*(10), 751-761.

Bellet, A., Habrard, A., & Sebban, M. (2013). A survey on metric learning for feature vectors and structured data. *arXiv preprint arXiv:1306.6709*.

Belongie, S., Malik, J., & Puzicha, J. (2002). Shape matching and object recognition using shape contexts. *Pattern Analysis and Machine Intelligence, IEEE Transactions on, 24*(4), 509-522.

Bertsekas, D. P. (1976). On the Goldstein-Levitin-Polyak gradient projection method. *Automatic Control, IEEE Transactions on*, *21*(2), 174-184.

Bieman, J. M. (1997). Software Metrics: A Rigorous & Practical Approach. *IBM Systems Journal*, *36*(4), 594.

Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., & Devanbu, P. (2009). *Fair and balanced?: bias in bug-fix datasets.* Paper presented at the Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering.

Bird, C., Nagappan, N., Murphy, B., Gall, H., & Devanbu, P. (2011, September). Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (pp. 4-14). ACM.

Bishop, C. M. (2006). *Pattern recognition and machine learning*. springer.

Black, P. E. (2006). Manhattan distance. *Dictionary of Algorithms and Data Structures*, *18*, 2012.

Boetticher, G. D. (2005). Nearest neighbor sampling for better defect prediction. *ACM SIGSOFT Software Engineering Notes*, *30*(4), 1-6.

Brown, G. W. (1947). On Small-Sample Estimation. *The Annals of Mathematical Statistics, 18(4), 582–585.*

Burges, C. J. (1998). A tutorial on support vector machines for pattern recognition. Data mining and knowledge discovery, 2(2), 121-167.

Cem Kaner, J. D., Hendrickson, E., & Smith-Brock, J. (2001). MANAGING THE PROPORTION OF TESTERS TO (OTHER) DEVELOPERS.

Chen, Z., Menzies, T., Port, D., & Boehm, B. (2005). Finding the right data for software cost modeling. *Software, IEEE*, *22*(6), 38-46.

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on, 20*(6), 476-493.

Cover, T., & Hart, P. (1967). Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on, 13*(1), 21-27.

Cristianini, N., & Shawe-Taylor, J. (2000). *An introduction to support vector machines and other kernel-based learning methods*. Cambridge university press.

Cunningham, P., & Delany, S. J. (2007). k-Nearest neighbour classifiers. *Multiple Classifier Systems*, 1-17.

D'Ambros, M., Lanza, M., & Robbes, R. (2010). *An extensive comparison of bug prediction approaches.* Paper presented at the Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on.

D'Ambros, M., Lanza, M., & Robbes, R. (2012). Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering, 17*(4-5), 531-577.

Dai, W., Xue, G.-R., Yang, Q., & Yu, Y. (2007). *Transferring naive bayes classifiers for text classification.* Paper presented at the PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE.

Davis, J. V., Kulis, B., Jain, P., Sra, S., & Dhillon, I. S. (2007). *Information-theoretic metric learning.* Paper presented at the Proceedings of the 24th international conference on Machine learning.

De Maesschalck, R., Jouan-Rimbaud, D., & Massart, D. L. (2000). The mahalanobis distance. *Chemometrics and intelligent laboratory systems, 50*(1), 1-18.

Deza, M. M., & Deza, E. (2009). *Encyclopedia of distances* (pp. 1-583). Springer Berlin Heidelberg.

El Emam, K., Benlarbi, S., Goel, N., & Rai, S. N. (2001). Comparing case-based reasoning classifiers for predicting high risk software components. *Journal of Systems and Software, 55*(3), 301-320.

Elish, K. O., & Elish, M. O. (2008). Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, *81*(5), 649-660.

Fan, W., Davidson, I., Zadrozny, B., & Yu, P. S. (2005). *An improved categorization of classifier's sensitivity on sample selection bias.* Paper presented at the Data Mining, Fifth IEEE International Conference on.

Forman, G. (2003). An extensive empirical study of feature selection metrics for text classification. *The Journal of machine learning research*, *3*, 1289-1305.

Fukushima, T., Kamei, Y., McIntosh, S., Yamashita, K., & Ubayashi, N. (2014). *An empirical study of just-in-time defect prediction using cross-project models.* Paper presented at the Proceedings of the 11th Working Conference on Mining Software Repositories.

Gaber, M. M., Zaslavsky, A., & Krishnaswamy, S. (2005). Mining data streams: a review. *ACM Sigmod Record, 34*(2), 18-26.

Ganesan, K., Khoshgoftaar, T. M., & Allen, E. B. (2000). Case-based software quality prediction. *International Journal of Software Engineering and Knowledge Engineering, 10*(02), 139-152.

Giger, E., D'Ambros, M., Pinzger, M., & Gall, H. C. (2012). *Method-level bug prediction.* Paper presented at the Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement.

Goldberger, J., Hinton, G. E., Roweis, S. T., & Salakhutdinov, R. (2004). Neighbourhood components analysis. In *Advances in neural information processing systems* (pp. 513-520).

Graf, A. B., & Borer, S. (2001). Normalization in support vector machines *Pattern Recognition* (pp. 277-282): Springer.

Gunn, S. R. (1998). Support vector machines for classification and regression. *ISIS technical report*, *14*.

Hall, M. (2000). Correlation-based feature selection for discrete and numeric class machinelearning,‖ Proceedings of th7 IntentionalConference onMachine Learning,.Stanford University

Hall, M. A. (1999). *Correlation-based feature selection for machine learning* (Doctoral dissertation, The University of Waikato).

Halstead, M. H. (1977). *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA.

Hamon, J. (2013). *Optimisation combinatoire pour la sélection de variables en régression en grande dimension: Application en génétique animale* (Doctoral dissertation, Université des Sciences et Technologie de Lille-Lille I).

Han, J., Kamber, M., & Pei, J. (2012), *Data mining : concepts and techniques*, 3rd ed. Waltham, Mass.: Elsevier/Morgan Kaufmann.

Hassan, A. E. (2009). *Predicting faults using the complexity of code changes.* Paper presented at the Proceedings of the 31st International Conference on Software Engineering.

Hazewinkel, M. (2001). The algebra of quasi-symmetric functions is free over the integers. Advances in Mathematics, 164(2), 283-300.

He, H., & Garcia, E. A. (2009). Learning from imbalanced data. *Knowledge and Data Engineering, IEEE Transactions on, 21*(9), 1263-1284.

Henry, S., & Kafura, D. (1981). Software structure metrics based on information flow. *Software Engineering, IEEE Transactions on, SE-7*(5), 510-518.

Ilczuk, G., Mlynarski, R., Kargul, W., & Wakulicz-Deja, A. (2007, September). New feature selection methods for qualification of the patients for cardiac pacemaker implantation. In *Computers in Cardiology, 2007* (pp. 423-426). IEEE.

Ivanciuc, O. (2007). Applications of support vector machines in chemistry. *Reviews in computational chemistry*, *23*, 291.

Japkowicz, N., & Shah, M. (2011). Evaluating learning algorithms: a classification perspective. Cambridge University Press.

Jeong, G., Kim, S., & Zimmermann, T. (2009). *Improving bug triage with bug tossing graphs.* Paper presented at the Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering.

Jiang, Y., Cukic, B., & Ma, Y. (2008). Techniques for evaluating fault prediction models. *Empirical Software Engineering*, *13*(5), 561-595.

Jing, X.-Y., Ying, S., Zhang, Z.-W., Wu, S.-S., & Liu, J. (2014). *Dictionary learning based software defect prediction.* Paper presented at the Proceedings of the 36th International Conference on Software Engineering.

Jong, K., Marchiori, E., Sebag, M., & Van Der Vaart, A. (2004, October). Feature selection in proteomic pattern data with support vector machines. In *Computational Intelligence in Bioinformatics and Computational Biology, 2004. CIBCB'04. Proceedings of the 2004 IEEE Symposium on* (pp. 41-48). IEEE.

Juszczak, P., Tax, D., & Duin, R. P. W. (2002). Feature scaling in support vector data description. In Proc. ASCI (pp. 95-102).

Kamei, Y., Matsumoto, S., Monden, A., Matsumoto, K. I., Adams, B., & Hassan, A. E. (2010, September). Revisiting common bug prediction findings using effort-aware models. In *Software Maintenance (ICSM), 2010 IEEE International Conference on* (pp. 1-10). IEEE.

Khan, J., Gias, A. U., Siddik, M. S., Rahman, M. H., Khaled, S. M., & Shoyaib, M. (2014, May). An attribute selection process for software defect prediction. In *Informatics, Electronics & Vision (ICIEV), 2014 International Conference on* (pp. 1-4). IEEE.

Khoshgoftaar, T. M., & Seliya, N. (2003). Analogy-based practical classification rules for software quality estimation. *Empirical Software Engineering*, *8*(4), 325-350.

Khoshgoftaar, T. M., Ganesan, K., Allen, E. B., Ross, F. D., Munikoti, R., Goel, N., & Nandi, A. (1997). *Predicting fault-prone modules with case-based reasoning.* Paper presented at the Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on.

Kim, S., Whitehead, E. J., & Zhang, Y. (2008). Classifying software changes: Clean or buggy? *Software Engineering, IEEE Transactions on, 34*(2), 181-196.

Kim, S., Zhang, H., Wu, R., & Gong, L. (2011). *Dealing with noise in defect prediction.* Paper presented at the Software Engineering (ICSE), 2011 33rd International Conference on.

Knab, P., Pinzger, M., & Bernstein, A. (2006, May). Predicting defect densities in source code files with decision tree learners. In Proceedings of the 2006 international workshop on Mining software repositories (pp. 119-125). ACM.

Kotsiantis, S., Kanellopoulos, D., & Pintelas, P. (2006). Data preprocessing for supervised leaning. *International Journal of Computer Science, 1*(2), 111-117.

Kulis, B. (2012). Metric learning: A survey. *Foundations & Trends in Machine Learning, 5*(4), 287-364.

Kullback, S. (1968). *Information theory and statistics*. Courier Corporation.

Kullback, S., & Leibler, R. A. (1951). On information and sufficiency. *The annals of mathematical statistics*, 79-86.

Langley, P., & Sage, S. (1994, July). Induction of selective Bayesian classifiers. In *Proceedings of the Tenth international conference on Uncertainty in artificial intelligence* (pp. 399-406). Morgan Kaufmann Publishers Inc..

Laradji, I. H., Alshayeb, M., & Ghouti, L. (2015). Software defect prediction using ensemble learning on selected features. Information and Software Technology, 58, 388-402.

Lee, T., Nam, J., Han, D., Kim, S., & In, H. P. (2011). *Micro interaction metrics for defect prediction.* Paper presented at the Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering.

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.

Liljeson, M., & Mohlin, A. (2014). Software defect prediction using machine learning on test and source code metrics.

Liu, H. (2005). Building effective defect-prediction models in practice. *Software, IEEE*, *22*(6), 23-29.

Liu, M., Miao, L., & Zhang, D. (2014). Two-stage cost-sensitive learning for software defect prediction. Reliability, IEEE Transactions on, 63(2), 676-686.

Ma, Y., Luo, G., Zeng, X., & Chen, A. (2012). Transfer learning for cross-company software defect prediction. *Information and Software Technology, 54*(3), 248-256.

Mahalanobis, P. C. (1936). On the generalized distance in statistics. *Proceedings of the National Institute of Sciences (Calcutta), 2*, 49-55.

McCabe, T. J. (1976). A complexity measure. *Software Engineering, IEEE Transactions on*(4), 308-320.

McCabe, T. J., & Butler, C. W. (1989). Design complexity measurement and testing. *Communications of the ACM*, *32*(12), 1415-1425.

Mende, T. (2010). *Replication of defect prediction studies: problems, pitfalls and recommendations.* Paper presented at the Proceedings of the 6th International Conference on Predictive Models in Software Engineering.

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*(9), 637.

Menzies, T., DiStefano, J., Orrego, A., & Chapman, R. (2004). *Assessing predictors of software defects*. Paper presented at the Proc. Workshop Predictive Software Models.

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.

Mockus, A., & Votta, L. G. (2000). *Identifying reasons for software changes using historic databases*. Paper presented at the Software Maintenance, 2000. Proceedings. International Conference on.

Moser, R., Pedrycz, W., & Succi, G. (2008). *A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction.* Paper presented at the Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on.

Murphy, K. P. (2006). Naive bayes classifiers. *University of British Columbia*.

Myers, G. J., Sandler, C., & Badgett, T. (2011). The art of software testing. John Wiley & Sons.

Nagappan, N., & Ball, T. (2005). *Use of relative code churn measures to predict system defect density*. Paper presented at the Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on.

Nam, J. (2009). Survey on Software Defect Prediction. Master's Thesis.

Nam, J., Pan, S. J., & Kim, S. (2013). *Transfer defect learning.* Paper presented at the Proceedings of the 2013 International Conference on Software Engineering.

Pai, G. J., & Dugan, J. B. (2007). Empirical analysis of software fault content and fault proneness using Bayesian methods. *Software Engineering, IEEE Transactions on*, *33*(10), 675-686.

Pan, S. J., & Yang, Q. (2010). A survey on transfer learning. *Knowledge and Data Engineering, IEEE Transactions on, 22*(10), 1345-1359.

Pelayo, L., & Dick, S. (2012). Evaluating stratification alternatives to improve software defect prediction. *Reliability, IEEE Transactions on*, *61*(2), 516-525.

Peters, F., & Menzies, T. (2012). *Privacy and utility for defect prediction: Experiments with morph.* Paper presented at the Proceedings of the 2012 International Conference on Software Engineering.

Quinlan, J. R. (1986). Induction of decision trees. Machine learning, 1(1), 81-106.

Quinlan, J. R. (2014). *C4. 5: programs for machine learning*. Elsevier.

Rahman, F., & Devanbu, P. (2011). *Ownership, experience and defects: a fine-grained study of authorship.* Paper presented at the Proceedings of the 33rd International Conference on Software Engineering.

Rahman, F., & Devanbu, P. (2013). *How, and why, process metrics are better.* Paper presented at the Proceedings of the 2013 International Conference on Software Engineering.

Rahman, F., Posnett, D., & Devanbu, P. (2012). *Recalling the imprecision of cross-project defect prediction.* Paper presented at the Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering.

Rahman, F., Posnett, D., Hindle, A., Barr, E., & Devanbu, P. (2011). *BugCache for inspections: hit or miss?* Paper presented at the Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering.

Reena, P., & Rajan, B. (2014). A Novel Feature Subset Selection Algorithm for Software Defect Prediction. *International Journal of Computer Applications*, *100*(17).

Rigby, P. C., & Hassan, A. E. (2007). *What can oss mailing lists tell us? a preliminary psychometric text analysis of the apache developer mailing list.* Paper presented at the Proceedings of the Fourth International Workshop on Mining Software Repositories.

Rodríguez, D., Ruiz, R., Cuadrado-Gallego, J., & Aguilar-Ruiz, J. (2007, August). Detecting fault modules applying feature selection to classifiers. In *Information Reuse and Integration, 2007. IRI 2007. IEEE International Conference on* (pp. 667-672). IEEE.

Roh, J. J., Stoeckel, J., & Lee, J. (2015). Module defect prediction under the Eclipse platform: the quadratic effect of software size and the influence of prerelease defects. International Journal of Data Analysis Techniques and Strategies, 7(1), 96-109.

Sahana, D. C. (2013). *Software Defect Prediction Based on Classication Rule Mining* (Doctoral dissertation).

Sayad, S. (2015). *An introduction to data mining.* Retrieved from: http://www.saedsayad.com/data_mining_map.htm

Shannon, C. E. (1993). Collected papers. Edited by NJA Sloane and Aaron D. Wyner.

Shihab, E., Mockus, A., Kamei, Y., Adams, B., & Hassan, A. E. (2011). *High-impact defects: a study of breakage and surprise defects.* Paper presented at the Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering.

Shin, Y., Meneely, A., Williams, L., & Osborne, J. A. (2011). Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *Software Engineering, IEEE Transactions on, 37*(6), 772-787.

Shirabad, J. S., & Menzies, T. J. (2005). The PROMISE repository of software engineering databases. *School of Information Technology and Engineering, University of Ottawa, Canada*, *24*.

Shivaji, S., Whitehead, E. J., Akella, R., & Kim, S. (2013). Reducing features to improve code change-based bug prediction. *Software Engineering, IEEE Transactions on, 39*(4), 552-569.

Simard, P., LeCun, Y., & Denker, J. S. (1993). *Efficient pattern recognition using a new transformation distance.* Paper presented at the Advances in neural information processing systems.

Smola, A. J., & Schölkopf, B. (1998). *Learning with kernels* (p. 210). GMD-Forschungszentrum Informationstechnik.

Smola, A., & Vishwanathan, S. V. N. (2008). Introduction to machine learning. Cambridge University, UK, 32-34.

Song, Q., Jia, Z., Shepperd, M., Ying, S., & Liu, S. Y. J. (2011). A general software defect-proneness prediction framework. *Software Engineering, IEEE Transactions on*, *37*(3), 356-370.

Subramanyam, R., & Krishnan, M. S. (2003). Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *Software Engineering, IEEE Transactions on*, *29*(4), 297-310.

Tang, M. H., Kao, M. H., & Chen, M. H. (1999). An empirical study on object-oriented metrics. In Proceedings of the 1999 international workshop on Software metric symposium (pp. 242-249). IEEE.

Turhan, B., & Bener, A. (2007). *A multivariate analysis of static code attributes for defect prediction.* Paper presented at the Quality Software, 2007. QSIC'07. Seventh International Conference on.

Turhan, B., & Bener, A. (2009). Analysis of Naive Bayes' assumptions on software fault data: An empirical study. *Data & Knowledge Engineering*, *68*(2), 278-290.

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 Software Engineering, 14*(5), 540-578.

Turhan, B., Misirli, A. T., & Bener, A. (2013). Empirical evaluation of the effects of mixed project data on learning defect predictors. *Information and Software Technology, 55*(6), 1101-1118.

Wang, H. (2014). Software Defects Classification Prediction Based On Mining Software Repository.

Wang, S., & Yao, X. (2013). Using class imbalance learning for software defect prediction. Reliability, IEEE Transactions on, 62(2), 434-443.

Weinberger, K. Q., & Saul, L. K. (2009). Distance metric learning for large margin nearest neighbor classification. *The Journal of Machine Learning Research, 10*, 207-244.

Weinberger, K. Q., Blitzer, J., & Saul, L. K. (2005). *Distance metric learning for large margin nearest neighbor classification.* Paper presented at the Advances in neural information processing systems.

Whittaker, J. (2000). What is software testing? And why is it so hard?. Software, IEEE, 17(1), 70-79.

Witten, I. H., & Frank, E. (2005). *Data Mining: Practical machine learning tools and techniques*: Morgan Kaufmann.

Wu, R., Zhang, H., Kim, S., & Cheung, S. (2011). *Relink: recovering links between bugs and changes.* Paper presented at the Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering.

Xing, E. P., Jordan, M. I., Russell, S., & Ng, A. Y. (2002). Distance metric learning with application to clustering with side-information. In *Advances in neural information processing systems* (pp. 505-512).

Zhang, H., Zhang, X., & Gu, M. (2007, December). Predicting defective software components from code complexity measures. In Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on (pp. 93-96). IEEE.

Zheng, J. (2010). Cost-sensitive boosting neural networks for software defect prediction. *Expert Systems with Applications, 37*(6), 4537-4543.

Zheng, Z., Wu, X., & Srihari, R. (2004). Feature selection for text categorization on imbalanced data. *ACM SIGKDD Explorations Newsletter, 6*(1), 80-89.

Zhou, Z.-H., & Liu, X.-Y. (2006). Training cost-sensitive neural networks with methods addressing the class imbalance problem. *Knowledge and Data Engineering, IEEE Transactions on, 18*(1), 63-77.

Zimmermann, T., & Nagappan, N. (2008). *Predicting defects using network analysis on dependency graphs.* Paper presented at the Proceedings of the 30th international conference on Software engineering.

Zimmermann, T., Nagappan, N., Gall, H., Giger, E., & Murphy, B. (2009). *Cross-project defect prediction: a large scale experiment on data vs. domain vs. process.* Paper presented at the Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering.

APPENDIX A: MICRO INTERACTION METRICS

1. File-Level Metrics

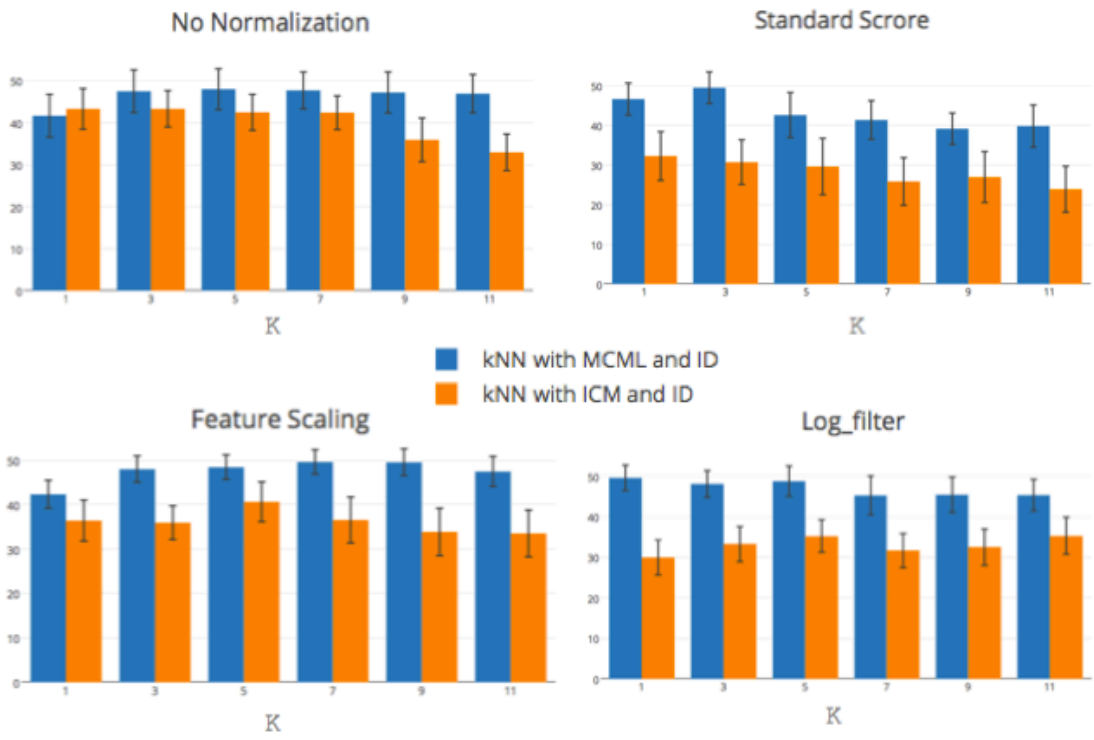
Category	Name	Description
Effort	NumSelectionEvent NumEditEvent NumManipulationEvent NumPropagationEvent	The number of each kind of events of the file.
Interest	AvgDOI VarDOI	Average and variance of Degree of Interest (DOI) values of the file.
Intervals	AvgTimeIntervalEditEdit	The average time interval between two edit events of the file.
	AvgTimeIntervalSelSel	The average time interval between two selection events of the file.
	AvgTimeGapSelEdit	The average time interval between selection and edit events of the file.

2. Task-Level Metrics

Category	Name	Description
Effort	NumUniqueSelFiles	The unique number of selected files in a task
	NumUniqueEdit	The unique number of edited files in a task.
	TimeSpent	The total time spent on a task.
Distraction	NumLowDOISel	The number of selection events with low DOI (DOI < median of all DOIs).
	NumLowDOIEdit	The number of edit events with low DOI (DOI < median of all DOIs).
	PortionNonJavaEdit	The percentage of non-java file edit events in a task.
Work Portion	TimeSpentBeforeEdit	The ratio of the time spent (out of total time spent) before the first edit.
	TimeSpendAfterEdit	The ratio of the time spent (out of total time spent) after the last edit.
	TimeSpentOnEdits	The ratio of the time spent (out of total time spent) between the first and last edits.
	NumSelBeforeEdit	The number of unique selections before the first edit event.
	NumSelAfterEdit	The number of unique selections after the last edit event.
Repetition	RepeatedlySelectedFileNum	The number of files selected more than once in one task.
	RepeatedlyEditedFileNum	The number of files edited more than once in one task.
Task Load	NumMultiTasks	The number of ongoing tasks at the same time.
	FileHierarchyDepth	The average depth of file hierarchy.

Category	Name	Description
Event Pattern	NumPatternSXSX NumPatternSXSY NumPatternSYSX NumPatternSYSY NumPatternSXEX NumPatternSXEY NumPatternSYEX NumPatternSYEY NumPatternEXSX NumPatternEXSY NumPatternEYSX NumPatternEYSY NumPatternEXEX NumPatternEXEY NumPatternEYEX NumPatternEYEY	The number of adjacent sequential event patterns in one task. S and E rep- resent selection and edit events respectively. For example, S?E? describes adjacent sequential interactions of selecting a file and continuously editing a file. We classify files into two groups: X and Y. If a file satisfies one of the following three conditions, we mark them as Group X. 1) If the file has been ever fixed before due to a defect. 2) If the file is one of frequently edited files in a task. (frequency threshold: top 50% of file editing frequencies). 3) If the file has been recently selected or edited within ±n time spent of a given task, where n = TotalTime/4 Otherwise, we put the file in the Y group. The group X implies high locality of file accessing with error-prone interaction.
	NumPatternSLSL NumPatternSLSH NumPatternSHSL NumPatternSHSH NumPatternSLEL NumPatternSLEH NumPatternSHEL NumPatternSHEH NumPatternELSL NumPatternELSH NumPatternEHSL NumPatternEHSH NumPatternELEL NumPatternELEH NumPatternEHEL NumPatternEHEH	As explained above, S and E share the same meaning. But instead of X or Y encoding, H or L encoding is used here. If the DOI value of the edited or selected file is higher than the median of all the DOI values in a task, it is denoted by H. Otherwise, it is L.

APPENDIX B: COMPARISON OF F-MEASURE OF OUR METHOD AND THE BASELINE FOR ID AND KC2



APPENDIX C: COMPARISON OF F-MEASURE OF OUR METHOD AND THE BASELINE FOR KC2 WITHOUT WEIGHTING