View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by The Research Repository @ WVU (West Virginia University)

Graduate Theses, Dissertations, and Problem Reports

2015

Semi-supervised and Active Learning Models for Software Fault Prediction

Huihua Lu

Follow this and additional works at: https://researchrepository.wvu.edu/etd

Recommended Citation

Lu, Huihua, "Semi-supervised and Active Learning Models for Software Fault Prediction" (2015). *Graduate Theses, Dissertations, and Problem Reports*. 6117.
https://researchrepository.wvu.edu/etd/6117

This Dissertation is protected by copyright and/or related rights. It has been brought to you by the The Research Repository @ WVU with permission from the rights-holder(s). You are free to use this Dissertation in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you must obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/ or on the work itself. This Dissertation has been accepted for inclusion in WVU Graduate Theses, Dissertations, and Problem Reports collection by an authorized administrator of The Research Repository @ WVU. For more information, please contact researchrepository@mail.wvu.edu.

Semi-supervised and Active Learning Models for Software Fault Prediction

Huihua Lu

Dissertation submitted
to the Benjamin M. Statler College of Engineering and Mineral Resources
at West Virginia University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in
Computer Engineering

Bojan Cukic, Ph.D., Chair
Mark Culp, Ph.D.
Afzel Noore, Ph.D.
Donald Adjeroh, Ph.D.
Vinod Kulathumani, Ph.D.

The Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2015

Keywords: software quality assurance, software fault prediction, semi-supervised learning, active learning, software metrics, dimension reduction
Copyright 2015 Huihua Lu

ABSTRACT

Semi-Supervised and Active Learning Models for Software Fault Prediction

Huihua Lu

As software continues to insinuate itself into nearly every aspect of our life, the quality of software has been an extremely important issue. Software Quality Assurance (SQA) is a process that ensures the development of high-quality software. It concerns the important problem of maintaining, monitoring, and developing quality software. Accurate detection of fault prone components in software projects is one of the most commonly practiced techniques that offer the path to high quality products without excessive assurance expenditures. This type of quality modeling requires the availability of software modules with known fault content developed in similar environment. However, collection of fault data at module level, particularly in new projects, is expensive and time-consuming. Semi-supervised learning and active learning offer solutions to this problem for learning from limited labeled data by utilizing inexpensive unlabeled data.

In this dissertation, we investigate semi-supervised learning and active learning approaches in the software fault prediction problem. The role of base learner in semi-supervised learning is discussed using several state-of-the-art supervised learners. Our results showed that semi-supervised learning with appropriate base learner leads to better performance in fault proneness prediction compared to supervised learning. In addition, incorporating pre-processing technique prior to semi-supervised learning provides a promising direction to further improving the prediction performance. Active learning, sharing the similar idea as semi-supervised learning in utilizing unlabeled data, requires human efforts for labeling fault proneness in its learning process. Empirical results showed that active learning supplemented by dimensionality reduction technique performs better than the supervised learning on release-based data sets.

Contents

Abstract i

Acknowledgements ii

List of Figures vi

1 Introduction 1
- 1.1 Software Fault Prediction Problem . . . 1
- 1.2 Machine Learning in Software Fault Prediction . . . 3
- 1.3 Practical Problems in Software Fault Prediction . . . 4
  - 1.3.0.1 Limited Fault Data . . . 4
  - 1.3.0.2 Imbalance in classes . . . 4
  - 1.3.0.3 Low quality in software data . . . 5
- 1.4 Semi-supervised learning in Software Fault Prediction problem . . . 5
- 1.5 Active learning for Software Fault Prediction problem . . . 9
- 1.6 Outline . . . 11

2 Literature Review 12
- 2.1 Distribution of Faults in Software Systems . . . 12
- 2.2 Software Metrics . . . 13
- 2.3 Software Fault Prediction Models . . . 14
- 2.4 Semi-supervised and Active learning in Software Fault Prediction problem 17

3 Semi-Supervised Learning for SFP problem 20
- 3.1 Semi-Supervised Learning approaches . . . 20
  - 3.1.1 Notation Definition . . . 21
  - 3.1.2 Fitting The Fits algorithm - FTF . . . 21
  - 3.1.3 Fitting The confident Fits algorithm - FTcF . . . 22
  - 3.1.4 Base Learner . . . 23
- 3.2 Experiment and Results . . . 25
  - 3.2.1 Experimental Data Sets . . . 25
  - 3.2.2 Experimental Setting . . . 26
  - 3.2.3 The Role of base learners . . . 29
    - 3.2.3.1 Convergence with Logistic Regression . . . 29
    - 3.2.3.2 Convergence with SVM . . . 31
    - 3.2.3.3 Convergence with Random Forest . . . 34

3.2.4 Results . . . 36
3.2.5 Discussion . . . 37
3.3 Conclusion . . . 41

4 Semi-Supervised Learning with dimensionality reduction approach 43
4.1 Multidimensional Scaling (MDS) . . . 43
4.2 Dimensionality Reduction based FTcF algorithm . . . 45
4.2.1 Notation Definition . . . 45
4.2.2 Methodology . . . 45
4.3 Experiment and Results . . . 46
4.3.1 Experimental Setting . . . 46
4.3.2 Results . . . 47
4.3.3 Statistical Analysis . . . 48
4.3.4 Robustness to Noise . . . 52
4.3.5 Discussion . . . 54
4.4 Conclusion . . . 55

5 Active Learning in SFP problem 57
5.1 Active Learning . . . 57
5.2 Active learning based Software Fault Prediction Model . . . 61
5.3 Experiments and Results . . . 63
5.3.1 Experimental Setting . . . 63
5.3.2 Results . . . 66
5.3.3 Discussion . . . 70
5.4 Conclusion . . . 71

6 Revisit Active Learning using different Data Sets 72
6.1 Feature Compression . . . 72
6.1.1 Feature Selection Techniques . . . 73
6.1.2 Dimensionality Reduction Techniques . . . 75
6.2 Experiments . . . 77
6.2.1 Software Data Sets . . . 77
6.2.2 Experimental Setting . . . 78
6.2.3 Results from Eclipse data sets . . . 80
6.2.4 Results from Camel and Ant data sets . . . 81
6.2.5 Discussion . . . 86
6.2.6 Statistical Analysis . . . 86
6.3 Threats to Validity . . . 92
6.4 Conclusions . . . 92

7 Summary and Future Work 94
7.1 Summary . . . 94
7.2 Scope and Limitations . . . 98
7.3 Future Work . . . 100

A Proof of convergence on FTF with LR 101

B Proof of convergence on FTcF with LR **104**

C Proof of convergence on FTF with SVM **107**

D Tables of Performance Comparison for Eclipse data sets **111**

Bibliography **117**

List of Figures

3.1 FTF algorithm . . . 22
3.2 FTcF algorithm . . . 23
3.3 Convergence plot on PC3(10%labeled set used). . . . 35
3.4 Convergence plot on PC3 with the measure of PD(10%labeled set used). . 36
3.5 Results of FTF algorithm on PC3(the numbers of modules initially labeled at 2%, 5%, 10%, 20%, 50% are 31, 78, 156, 313,782 respectively) . . . 37
3.6 Results of FTcF algorithm on PC3(the numbers of modules initially labeled at 2%, 5%, 10%, 20%, 50% are 31, 78, 156, 313,782 respectively) . . 38
3.7 Results of FTF algorithm on four data set with threshold=0.5. . . . 38
3.8 Results of FTcF algorithm on four data set with threshold=0.5. . . . 39

4.1 Dimension Reduction based FTcF Algorithm . . . 46
4.2 Performance plots for PC4 project. . . . 48

5.1 Active learningn process . . . 58
5.2 Comparison of two active learning sampling strategies with supervised learning approach. 10-cross-validation is used to evaluate the prediction performance of trained models at each iteration. . . . 59
5.3 Diagram of the Adaptive Fault Prediction process. . . . 60
5.4 Performance of AFP for 5% of initially labeled modules. . . . 64
5.5 Performance of AFP for 10% of initially labeled modules. . . . 64
5.6 Performance of AFP for 20% of initially labeled modules. . . . 65
5.7 Performance of AFP for 50% of initially labeled modules. . . . 65
5.8 Comparison between AFP approach and supervised learning. Both use Naive Bayes classifier with varied sizes of labeled data used in training. . 69

6.1 Comparison of different feature selection techniques with active learning using Eclipse 2.0 packages for training and 2.1 for evaluation. . . . 73
6.2 Comparison of Euclidean distance vs. RF similarity in multidimensional scaling (MDS) on Eclipse release 2.0. The plus sign represents defective modules, the minuses represent defect-free modules(at package level). . . . 75
6.3 Comparison of Euclidean distance vs. RF similarity in multidimensional scaling (MDS) on Eclipse release 2.0. The plus sign represents defective modules, the minuses represent defect-free modules(at file level). . . . 75
6.4 Defect prediction in release 2.1 from 2.0 (Eclipse - packages) . . . 82
6.5 Defect prediction in release 3.0 from 2.1 (Eclipse - packages) . . . 82
6.6 Defect prediction in release 3.0 from 2.0 and 2.1 (Eclipse - packages) . . . 83
6.7 Defect prediction in release 2.1 from 2.0 (Eclipse - files) . . . 83
6.8 Defect prediction in release 3.0 from 2.1 (Eclipse - files) . . . 84

6.9 Defect prediction in release 3.0 from 2.0 and 2.1(Eclipse - files) 84
6.10 Defect prediction in release 1.4 from 1.2 (Camel) 87
6.11 Defect prediction in release 1.6 from 1.4 (Camel) 87
6.12 Defect prediction in release 1.3 from 1.4 (Ant) 88
6.13 Defect prediction in release 1.5 from 1.6 (Ant) 88

7.1 Run-time (minutes) of semi-supervised learning for PC3 data set. 99

List of Tables

3.1 Datasets used in this study . . . 25
3.2 Number of modules in initially labeled data set . . . 28
3.3 Comparison between our results and Menzies's results with Probability of Detection(PD) at specified PF . . . 40
3.4 Comparison between our results and Lessmann's results with Area under ROC curve(AUC) . . . 41

4.1 AUC for the four data sets . . . 49
4.2 PD with threshold=0.75 for the four data sets . . . 50
4.3 PD with threshold=0.5 for the four data sets . . . 51
4.4 PD with threshold=0.1 for the four data sets . . . 51
4.5 One way ANOVA test for PD(threshold= 0.5) at 2% labeled data . . . 52
4.6 P-value of ANOVA test on varied size of labeled data for all performance measures . . . 52
4.7 Significance comparison of PD(0.5) . . . 52
4.8 Significance comparison of PD(0.1) . . . 52
4.9 Average decrease in AUC measure. . . . 53
4.10 Average percent decrease in PD measure (Threshold is 0.5). . . . 53
4.11 Comparison of results with [1] . . . 54
4.12 Comparison of results with [2] using AUC . . . 55

5.1 NASA software metrics . . . 62
5.2 Number of modules in initially labeled data set . . . 62
5.3 Percentage of change over the iterations of adaptive learning procedure. . . 67

6.1 Software data sets . . . 76
6.2 Metrics in Eclipse data set . . . 77
6.3 Metrics in Camel and Ant data sets . . . 77
6.4 One way ANOVA test for AUC measures at 10th iteration, when defects in release 2.1 are predicted from 2.0 . . . 90
6.5 P-values by ANOVA test, when defects in release 2.1 are predicted from 2.0 (package level) . . . 90
6.6 Post-hoc test for performance differences between the six active learning approaches at package level ($1 : MDS\_Act, 2 : MDS\_rand, 3 : IG\_Act, 4 : IG\_rand, 5 : Act, 6 : Rand$). " ✓" stands for statistically significant difference between two approaches. "✗" stands for no significant difference detected between the two approaches. . . . 90

6.7 Post-hoc test for performance differences between the six active learning approaches at file level ($1 : MDS\_Act, 2 : MDS\_rand, 3 : IG\_Act, 4 : IG\_rand, 5 : Act, 6 : Rand$). " ✓" stands for statistically significant difference between two approaches. "✗" stands for no signisficant difference detected between the two approaches. 91

7.1 Run-time (minutes) of semi-supervised learning when random forest is used as base learner 99

D.1 Performance comparison, from release 2.0 to 2.1 for Eclipse packages . . . 112
D.2 Performance comparison, from release 2.1 to 3.0 for Eclipse packages. . . . 112
D.3 Performance comparison, from releases 2.0 and 2.1 to 3.0 for Eclipse packages. 113
D.4 Performance comparison, from release 2.0 to 2.1 for Eclipse files. 113
D.5 Performance comparison, from release 2.1 to 3.0 for Eclipse files. 114
D.6 Performance comparison, from releases 2.0 and 2.1 to 3.0 for Eclipse files. 114
D.7 Performance comparison, from release 1.2 to 1.4 for Camel. 115
D.8 Performance comparison, from release 1.4 to 1.6 for Camel. 115
D.9 Performance comparison, from release 1.3 to 1.4 for Ant. 116
D.10 Performance comparison, from release 1.5 to 1.6 for Ant. 116

Chapter 1

Introduction

1.1 Software Fault Prediction Problem

As software continues to insinuate itself into nearly every aspect of our life, the quality of software has been an extremely important issue. Software Quality Assurance (SQA) consists of activities that ensure the development of high-quality software. It encompasses the development and implementation of methods and processes for quality software, regardless of the underlying software development model being used.

To ensure high quality products, software engineers need undertake significant efforts to ensure that software functions are intended while inspecting the risks of vulnerabilities that could bring harm to the end user. Software quality inspection and improvement can be detecting faulty software modules and reducing the number of faults occurring during system operations. A software fault usually refers to a defect or a flaw in an executable product that can cause system failures during operation. Faults in software systems are major problem that need to be resolved. Software module is the lowest level of software for which we have data, for example, java method or class.

It is critical to detecting where fault hides, as it allows verification and validation experts to concentrate their time, efforts, and resources on the potentially problematic modules under development, thus enables Verification and Validation (V&V) activities more effective [3–6]. On the other hand, learning the pattern how faults hide in code helps software engineering improve their design or development in the future project or release. Software fault prediction can identify faults in the current code base, but also warns about future fault-prone areas.

Over the past years, software fault prediction problem has been an important area of research [7–11]. Given the shorter development and release life cycles, accurate detection of software fault relies increasingly on automated techniques. Machine learning approaches are nature solutions to this type of problem. In a classic machine learning procedure, a predictive model can be trained to form a set of learning rules or patterned structures using training data set, such as historical software modules from previous releases where fault contents are known. The trained model can be then used to estimate the fault content of modules currently developed, for example, modules in a newly developed subsystem or in an upcoming project release. Depending on the learning problem, machine learning can be narrowed down into two categories, regression based methods and classification based methods. Target variable refers to 'response' variable in statistical language, or 'label' in machine learning literature. For regression problem, the target variable can be the number of faults associated with each software module. For classification problem, the target variable is a binary variable, fault proneness (*fp*) or non-fault proneness (*nfp*). Predicting the exact amount of faults is too risky, especially in the early development stage when only little information is available. Classifying software into *fp* or *nfp* can be more general and reliable. Throughout this dissertation, we focus on the binary classification problem in software fault prediction.

Besides the target variable, an important element in software fault prediction problem is the software metrics, or so-called features in machine learning. Most widely used software metrics includes static code metrics, Object-Oriented (OO) metrics, development process metrics, complexity metrics of modules, network based metrics and many others. The basic idea behind using software metrics is that, for example, more complex the code is more likely to have faults, or a software component is likely to be fault prone if it is similar to other faulty components in code structure or code complexity. Software metrics, for example the static code metrics, can be obtained using automated data collection tools. Typically, software metrics together with their fault contents form the basis of software fault prediction learning data, or training data.

For binary classification problem, software fault prediction models can be assessed using confusion matrix based criteria. The most widely used are accuracy, recall, precision, F-measure, G-mean, or the more recently used AUC measure. In addition, other criteria are also important when deploying fault prediction models in a development environment, including ease of use, computational efficiency, or model comprehensibility.

1.2 Machine Learning in Software Fault Prediction

Software fault prediction models have been studied since 1990s until now. There have been numerous efforts of applying various types of approaches in software fault prediction problem. Many of them aim to propose approaches to allocate limited SQA resources in a cost effective manner by utilizing machine learning approaches [12–14].

Machine learning practitioners have used unsupervised learning approaches and supervised learning approaches to estimate the fault contents of software modules, depending on whether labels are available or not in training data. Learning approaches with given labels in the training data set refers to supervised learning. Learning with no labels in the training data set refers to unsupervised learning. Both are important learning branches in machine learning and have been widely employed in software fault prediction problem. Intuitively, unsupervised learning approaches are good choices for new developed system which has no previous subsystem or release[15, 16], while supervised learning approaches are preferred when the previous subsystem or releases are tested and the corresponding fault contents are obtained[17–20].

In unsupervised learning setting, fault prediction models are built based on the natural structure and distribution of the data points. K-mean and hierarchical clustering are popular unsupervised learning approaches to software fault prediction practitioners. The underlying assumption is that software modules are likely to be labeled the same if they are closely connected to each other or highly grouped together. After the clustering analysis, software experts can label the clusters as either *fp* or *nfp* without inspecting the modules one at a time. This eases the labeling task and also saves budget consumed on labeling for each module. This could be significantly important in software development, especially when the software delivery date is urgent and the budgets are very limited. The challenge for unsupervised learning approaches is that the performance of the fault prediction is highly affected by the violation of the density (clustering) assumption, especially, for the situation when the data is strongly imbalanced or the clusters of minority class and majority class are significantly overlapped.

In supervised learning setting, software modules together with their fault contents form the training data set and the trained models can be used to predict fault contents of software modules currently under development. Logistic regression, Naive Bayes, tree-based methods, k-nearest neighbors and support vector machines are commonly used supervised learning approaches. The assumption for supervised learning approaches is that modules in training set and test set are from the same data space. Usually, supervised learning can provide relatively better performance in fault prediction comparing to unsupervised learning as it fits model with given fault contents. They thus are more

preferred in practice. However, to ensure high accurate in prediction supervised learning requires a reasonably large set of labeled modules (training set). The more fault contents in the training set the more accurate the trained model. A small set of fault contents will probably mislead the training and bias may arise. This requirement could be hard to meet the development schedule is tight or the budget is limited.

1.3 Practical Problems in Software Fault Prediction

Despite years of researches, the study of software fault prediction seems to have reached a plateau. According to recent studies, the probability of detection (PD) (71%) of fault prediction models may be higher than PD of software reviews (60%) if a robust model is built [21]. In this section, we discuss the practical problem or limits that causes such plateau.

1.3.0.1 Limited Fault Data

To build a desirable predictive model, it requires the training data set, i.e., software fault data as large as possible. Most of the past studies in literature assume that there are enough fault data to build the prediction models. Literature in the field indicates that researchers typically utilize at least 50% of software modules for training[1, 22]. However, sometimes we cannot have enough fault data to build accurate models. For example, a new project may have no previous release. On the other hand, labeling large amount of modules consumes time and human resources, which leads to a higher developing budget. This is problematic for most learning approaches, but particularly weighs in on supervised learning approaches.

1.3.0.2 Imbalance in classes

One characteristic of software fault data is that the amount of faulty modules is usually smaller than that of non-faulty modules. Sometimes, such imbalance can be significant. Part of experimental data used in our study are software projects from PROMISE repository, which are very typical examples reflecting the imbalance problem of software fault data (Table 3.1). Like most imbalance data prediction problem, the interest of study is on the “rare” class - the fault proneness class. Unfortunately, most commonly used classification approaches are designed to minimize the overall error rate rather than paying particular attention on the “rare” class[23]. Thus they may not work well for imbalance

data prediction problem. As a consequence, the effectiveness of predictive approaches can be vague if invalid performance measures are used.

1.3.0.3 Low quality in software data

Data Quality is another important issue for software fault prediction problem. Data may be low quality for the following reasons. First, a module may be contaminated by noise, that is, either the complexity metrics or the labels associated to modules are inaccurate. Second, a module may be an outlier in the class it belongs to, which is a typical observation or case. An outlier may or may not be contaminated by noise. The third, software data may be considered low quality if it contains missing values. Although several of imputation methods are available, additional bias or noise may be introduced into the data by missing data imputation. Finally, some other quality issues related to predictors could be constant predictors or inconsistent instances [24, 25]. As by now, very few studies report any data-processing scheme prior to fault prediction process.

The goal of our study is to innovate effective predictive approaches to address the limits in traditional predictive methods. Lacking of labeled data such as the limited fault data problem can be referred to as the $|L| << |U|$ case where the labeled size is significantly smaller than the unlabeled size, where L denotes the set of labeled data and U denotes the set of unlabeled data. Intuitively, the knowledge stored in unlabeled modules provides information that can achieve better performance for fault prediction. Thus, semi-supervised learning or active learning is an ideal solution to solve such problem due to their ability of incorporating information from unlabeled data.

1.4 Semi-supervised learning in Software Fault Prediction problem

Semi-supervised learning has received considerable attention in the machine learning literature due to its potentials in reducing the need for expensive labeled data software fault contents. It has proven successful in image recognition, speech recognition, text categorization, protein structure prediction, and many other domains. Semi-supervised learning falls somewhere between supervised learning and unsupervised learning. In fact, most semi-supervised learning approaches are based on the extension of either supervised learning or unsupervised learning approaches. In a semi-supervised learning setting, both labeled and unlabeled data are used as training data set. With the help of unlabeled data the amount of labeled data could be reduced which in turn reduces

the cost of labeling for training data [26, 27]. The underlying hypothesis for semi-supervised learning is that knowledge stored in unlabeled modules aids in improving the overall performance of classification.

In this section we will give a brief review on the history of semi-supervised learning. There has been a whole spectrum of interesting ideas on how to learn from both labeled and unlabeled data. It should be noted that semi-supervised learning is a rapidly evolving field, and the review is necessary incomplete. According to our knowledge, traditional semi-supervised learning algorithms can be roughly classfied into four categories:

1. Generative algorithms (such as EM algorithm [28]);
2. Iterative algorithms (such as self-training and co-training [29, 30]);
3. Density based algorithms (such as transductive-SVM [26]);
4. Graph based algorithm [26, 27].

The generative algorithms require the assumption of data distribution prior to learning. It is common to assume that the data is from multivariate normal distribution, so that the prediction of labels turns out to be the problem of estimating the missed parameters of a normal distribution (μ and Σ). The early work of generative algorithm in semi-supervised learning assumes that the complete data comes from a mixture Gaussian distribution. Let a full generative model be $p(D|\theta) = p(X, Y|\theta)$, thus the generative model for semi-supervised learning is:

$$P(D|\theta) = P(X_l, Y_l, X_u|\theta) = \sum_{Y_u} p(X_l, Y_l, X_u, Y_u|\theta) \tag{1.1}$$

where $\theta = \{w, \mu, \Sigma\}$ with Gaussian model $p(x, y|\theta) = p(y|\theta)p(x|y, \theta) = w_y N(x; \mu_y, \Sigma_y$. The goal is to find θ to maximize $P(D|\theta)$. The θ can be solved using maximum likelihood estimation (MLE). For simplicity, consider binary classification problem using MLE, the labeled data has: $logp(X_l, Y_l|\theta) = \sum_{i}^{l} logp(y_i|\theta)p(x_i|y_i, \theta)$. For labeled and unlabeled data, it becomes:

$$logp(X_l, Y_l, X_i|\theta) = \sum_{i=1}^{l} logp(y_i|\theta)p(x_i|u_i, \theta) + \sum_{i=l+1}^{l+u} log(\sum_{y=1}^{2} p(y|\theta)p(x_i|y, \theta)) \tag{1.2}$$

The Expectation-Maximization(EM) algorithm is a nature solution to find the optimum. Typically, EM algorithm contains two steps - E-step and M-step. The algorithm starts

from MLE θ by calculating $\theta = \{w, \mu, \Sigma\} on (X_l, Y_l)$. At E-step, the algorithm computes the expected label $p(y|x, \theta) = \frac{p(x,y|\theta)}{\sum_j p(x,y_j|\theta)}$ for all $x \in X_u$. At the M-step, the algorithm updates MLE θ with (now labeled) X_u. This procedure repeats the E and M steps until it converges to a local maximum of θ. Generative semi-supervised learning approaches are in a clear and well-studied probabilistic framework. It can be extremely effective if the model is close to correct. Unfortunately, it is often difficult to verify the correctness of the model assumption. The classification performance may be bad if generative model is wrong.

Unlike generative models, iterative semi-supervised learning algorithms, also called bootstrapping algorithms, do not reply on the knowledge of data distribution. They are basically wrapper methods that apply to existing classifiers. Self-training and co-training are two representatives in this category. The earliest self-training algorithm is called Yarowsky algorithm, which becomes widely known in computational linguistics. Later versions of self-training algorithms are about variants of the Yarowsky algorithm. The Yarowsky algorithm contains two loop. The inner loop is a supervised learning algorithm and called as base learner, consisting of a list of decision rules - If instance x contains feature f, then predict label j. The base learner selects those rules whose precision on the training data is highest. The Outer loop of self-training is given a seed set of rules to start with. In each iteration, it uses the current set of rules to assign labels to unlabeled data. Then, it selects those instances on which the base learners predictions are most confident. It then calls the inner loop to construct a new classifier and the cycle repeats.

Abney has introduced a modified Yarowskey algorithm which differs to the original one in two points: 1) once an unlabeled example gets labeled it stays labeled; 2)the labeling threshold is fixed to be $1/L$. In his study, he showed that the original Yarowsky algorithm aims to minimize an objective function. They proposed several variants of Yarowsky algorithm based the difference of objective function. The object function is the cross entropy between the prediction distribution of the model and the labeling distribution over all instances. For labeled instances the entropy of the labeling distribution is zero. Minimizing the objective function forces unlabeled data to be labeled, and forces the model to maximize the likelihood of the (old and newly) labeled data.

In contrast to self-training which iteratively trains a single base learner, co-training requires data attributes to be naturally separated into two views that are conditionally independent given the target label. It is showed that the classifier trained on one view has low generalization error if it agrees on unlabeled data with the classifier trained on the other view. There are also studies to extend two views into multi-views.

The main assumption for self-training and co-training is that the confidence prediction by base learner(s) is correct. That says both algorithms heavily rely on the base learner. This also implies that early mistake in iterative semi-supervised learning could reinforce themselves.

Next category of semi-supervised learning is density-based algorithm. The assumption regarding density-based algorithms is that the data can be naturally grouped into clusters according to the classes they belong to. Given the clustering assumption of the density-based algorithms, the semi-supervised learning problem can be viewed as the maximizing margin problem, i.e., the optimizing marginal technique based algorithms. With the rising popularity of support vector machine, transductive SVMs emerge as an extension of standard SVMs to semi-supervised learning. Transductive SVMs, also called as semi-supervised SVMs or S3VMs, is a method to improve the generalization accuracy of SVMs by using unlabeled data. S3TMs, like SVMs, learn a large marginal hyper-plane classifier using labeled training data, but simultaneously force this hyper-plane to be far away from the unlabeled data. More specifically, it aims to find a decision boundary that lies in the region of low density in terms of both labeled and unlabeled data. Therefore, it assumes that the underlying distribution of two classes is such that there is a low-density region between them.

The original S3VMs is based on an iterative algorithm. At the initial iteration, the standard SVMs is used to obtain an initial separating hyper-plane based on the labeled data. Then, pseudo labels are given to the unlabeled samples, which are thus called semi-labeled data. After that, transductive samples chosen from the semi-labeled patterns according to a given criterion are used to define a hybrid training set made up of these semi-samples together with original training samples. The resulting hybrid training set is used at the following iterations to find a more reliable discriminant hyper-plane. This hyper-plane can be derived as follows:

$$\begin{aligned} \min_{w,\xi_l,\xi_u} \{\frac{1}{2} w^T w + C \sum_{l=1}^{n} \xi_l + C^* \sum_{u=1}^{d} \xi_u\}, \\ s.t. y_l[w^T \phi(x_l) + b] \geq 1 - \xi_l, \xi_l > 0, \\ y_u[w^T \phi(x_u) + b] \geq 1 - \xi_u, \xi_u > 0 \end{aligned} \tag{1.3}$$

where C is the penalty term for misclassification vector ξ_l, and C^* is the penalty term for misclassification vector ξ_u. $\phi(.)$ is any mapping function. y_u or $\hat{y}_u$ is the prediction from previous iteration.

In order to handle non-separable training and transductive data, similarly to the SVMs, the slack variables ξ_l and ξ_u, and the associated penalty value C and C^* of both the training and transductive instances are introduced. $d(d < m)$ is the number of selected unlabeled samples for transductive learning. S3VMs provides a clear mathematical framework and applicable wherever SVMs are applicable. However, S3VMs has difficulty of optimization and can be trapped in bad local optima. On the other hand, it has more modest assumption than generative model or graph-based approaches.

For graph-based semi-supervised learning, the data are represented by a graph, where the edges are labeled with the pairwise similarities. All graph algorithms aim to compute a soft assignment of labels to the nodes of a graph $G = (V, E, W)$, where V is the set of nodes, E is the set of edges, and W is an edge weight matrix. If edge $(u; v) \notin E$, $W_{uv} = 0$. The assumption of the graph-based semi-supervised algorithms is that the points connected in a high-density region should belong to the same class.

1.5 Active learning for Software Fault Prediction problem

An alternative to semi-supervised learning to address the limited labeled data problem is active learning. The idea of active learning is to improve fault prediction performance by augmenting the training data set with intelligently sampled unlabeled data set. Active learning has many overlaps with iterative semi-supervised learning. For example, active learning requires a base learner iteratively pick the instances from the pool of unlabeled data set in the same way in semi-supervised learning. The key difference between the two learning schemes is that active learning requires the true labels by interacting with an outer oracle, who has expertise to provide ground truth of labels[31]. The general assumption behind active learning is that good prediction performance can be achieved by using only "essential data. This characteristic of active learning is desirable in situations where the availability of labeled data is limited.

Active learning approaches vary from different sampling mechanism. There is a class of strategies to sample the data from which to learn. The most popular ones are uncertainty sampling, query-by-committee (QBC), expected error reduction, and density weighted methods [32]. Of these, uncertainty sampling is the most widely used one in machine learning literature. The motivation behind uncertainty sampling is finding unlabeled instances that contain most uncertainty, and use them to clarify the decision boundary. One simplest strategy is to query the instances about which it is least confident how to label. This approach is straightforward for probabilistic learning models. For example, when using a probabilistic model for a binary classification problem, the instances with

most uncertainty are those with posterior probability closest to 0.5 typical decision cutoff for binary classification with balanced class sizes.

For multi-class problem this least confident criterion only considers information about the most probable label. It "throws away" information about the remaining label distribution. To correct for this shortcoming, a more general uncertainty sampling strategy is to use entropy as an uncertainty measure. In this case, one can consider it as selecting instances that maximize the Shannon entropy:

$$H(y|x) = \arg\max_x - \sum_i P(y_i|x) log P(y_i|x), \tag{1.4}$$

where *H(x)* is the uncertainty measurement function based on the entropy estimation of the classifier's posterior distribution. $P(y_i|x)$ is the posteriori probability. For binary classes, both least confidence and entropy-based strategy reduce to be equivalent. In our case, $y \in \{0, 1\}$ where 1 stands for defect prone packages and 0 stands for not defect prone packages. The highest uncertainty score implies that the current learner has the least confidence on its classification of this unlabeled component, thus should be selected first.

The QBC method construct an ensemble of learners induced over labeled data and request labels for instances in unlabeled data set for whose class ensemble members most disagree. This method is inspired by computational learning theory, that is, each committee member may be viewed as a hypothesis consistent with the instances in labeled data set. Acquiring a label for an instance about which two or more hypotheses disagree can be seen as a means of explicitly shrinking the version space, comprising of the hypotheses consistent with labeled data set.

Expected error reduction is decision-theoretic approach, which aims to measure not how much the model is likely to change, but how much its generalization error is likely to be reduced. Instances in unlabeled data set that directly minimize the expected model prediction error should be archived by acquiring a label. This expectation can be computed using the current model. Expected error minimization is appealing because it explicitly maximizes the prediction accuracy. However, in terms of software fault prediction, this type of approaches could be problematic, due to the accuracy is not a suitable performance measure for imbalance class problem in fault data.

Density-weight active learning is a relatively new approach in this subject. The idea behind is that informative instances should not only be those, which are uncertain, but also those which are representative of the underlying distribution. For example, one can query instances by maximize following equation:

$$\arg\max_{x} \phi_A(x) \times (\frac{1}{U} \sum_{u} sim(x, x^{(u)}))^{\beta}, \tag{1.5}$$

where $\phi_A(x)$ represents the informativeness of x according to a learner A. The second term weights the informativeness of x by its average similarity to all other instances in the vector space of unlabeled data, subject to a parameter β controlling the relative importance of the second term, i.e., the *weights*. Density-weighted active learning varies when different learners of A or different similarity approaches sim are used.

In this dissertation we investigate the uncertainty-sampling active learning for software fault prediction problem considering that it is straightforward for probabilistic learning models. We will come back to this with detail in chapter 5.

1.6 Outline

The outline of this proposal is as follows. Chapter 2 provides a literature review for software fault prediction problem. In Chapter 3, we present experimental results of investigating semi-supervised learning in SFP problem. Chapter 4 extends the similar experiments using semi-supervised learning approach with supplemented by dimensional reduction technique. In chapter 5, we present our experiments using active learning approaches. In Chapter 6, we revisit the active learning to release base software data set. Finally, conclusion of our study and future work are discussed in Chapter 7.

Chapter 2

Literature Review

Software fault prediction involves the identification of software locations what quality assurance efforts should focus on. It is one of the most important research areas in software engineering. Studies regarding on software fault prediction can be dated back to the mid 1970s. Each of these studies used their own unique data, features, and predictive techniques and evaluated their models differently. It is important to study the prior work in order to better understand the assumptions and implications of their work. In this Chapter we discuss the elements in software fault prediction models and review the prior literature in the area essential to understanding the role of machine learning in software fault prediction problem.

2.1 Distribution of Faults in Software Systems

A software fault is defined as flaws or imperfection found within code, which may cause the system or system component to fail to perform as required. Faults can be introduced into code at any phase of the software life cycle. In [33], it was discovered that faults in software product are not uniformly distributed throughout the code by investigating three evolutionary releases of a software product. They claimed that the fault rate increases when parts of the code for a new release are modified or newly developed code is added. [34] stated that nearly half of all faults in their telecommunication software system were related to coding faults, majority of which could have been prevented. A recently study based on data extracted from NASA mission stated that requirement faults, coding faults and data problems are the most common types of software faults [35]. They also suggested that observed common trends in software faults are likely intrinsic characteristics rather than project specific.

The analysis of fault distribution in the development of software systems is an area of interest to software developer as well as empirical researchers. [36] investigated four basic fault distribution hypotheses on two releases of a large commercial telecommunications system. One of the hypotheses showed that a small number of modules contain the majority of faults, that is approximately 20% of all faults are concentrated in about 80% of the modules which follows the Pareto principle. This observation was replicated and confirmed by [37–39]. Later on by Zhang [40], it was shown that the distribution of software faults can be more precisely described as the Weibull distribution, where they implemented Eclipse data and analyzed the distribution of its faults across models in package level. [41] discussed both Pareto and Weibull distribution and proposed a generalized pareto model to assess software fault distribution. Their results showed that the modified pareto model highly fit to the actual fault data.

Another important hypothesis in [36] concerned the similarities in fault densities within project phases or cross projects. This hypothesis was partly supported by their observation. The same hypotheses are replicated in [42] and [43] which confirmed Fenton's investigation by revisiting the same four hypotheses on different multi-releases software systems.

2.2 Software Metrics

The size metrics, such as the lines of code (LOC), are widely used prediction metrics that are the simplest and easiest to be extracted. Numerous studies investigated the relationship between size of modules and the number of faults, such as [36, 44, 45]. Some studies directly built linear regression models by using software module size as the predictor and fault count as the response. Others derived models analytically first and then fit the data to validate those models, such as Lipow's logarithmic model and Cox proportional hazard model [45, 46]. In [47], Koru applied cox models and further providing evidence that there is a power-law relationship between size and fault proneness with the latter increasing at a slower rate. This observation supports their hypothesis that smaller modules are proportionally more fault prone. They thus recommended focusing quality assurance resources on smaller modules, as they are more cost effective, i.e., more faults will be found in the same amount of code. This studies were further confirmed in [48] by the study of four large-scale object-oriented products.

In [36], a hypothesis regarding whether size metrics are good predictors of faults is also tested. Their observation shows that size metrics correlate with the number of faults, but there is no strong evidence that size metrics are a good predictor of faults. Limited supporting to the hypothesis was also observed in [42, 43].

In addition to size metrics, complexity metrics such as McCabe's cyclomatic complexity [49] and Halstead's metrics [50] are also widely used as fault prediction metrics. Some important studies using complexity metrics can be found in [22, 51, 52]. Fenton and Ohlsso [36] reported that complexity metrics are reasonable predictors but not the best. They observed that both the McCabe and halstead metrics are highly related to each other and to the lines of code. Zhou [53] also noted that size metric has strong confounding effect on association between complexity metrics and fault proneness and that the explanatory power of complexity metrics is limited.

In a recent study[54], the authors reviewed 106 paper published between 1991 to 2011 and concluded that Object-Oriented metrics and process metrics are more successful at fault prediction than traditional size metrics and complexity metrics. Their findings are similar to those of Hall's study [55] for size, complexity and OO metrics, but differ regarding process metrics. In Hall's paper, they reported that process metrics performed the worst among all metrics.

Although most of the research done in recent years focused on the impact of structural properties and process aspect of software component on fault-proneness, there are a few studies that investigated other types of prediction metrics. For example, Nagappan [56, 57] used code churn together with dependency metrics to predict fault-prone modules. In [58] counted the number of changes done in a module as well as the average age of the code.

2.3 Software Fault Prediction Models

Identifying fault in software components effectively is an economically important activity. Software fault prediction is a well-understood research field and has been studied for more than three decades. There exist a large number of modeling techniques to build fault prediction models in the literature. These techniques include statistical modeling techniques such as discriminant analysis [59–61], regression based models [62–66], and machine learning techniques like Naive Bayes [1, 67, 68], random forest [13, 69–71], C4.5 [12, 72, 73], neural network[74–78], and many others[79–85]. In [54], a systematic review on modeling techniques according 106 papers published from 2000 to 2010 indicated that statistical techniques, primarily logistic regression and linear regression, were used in 68% of the studies, while machine learning techniques were used in 24% of the studies. There are only 8% studies focusing on correlation analysis. However, even the large amount of studies, there is still no consensus on which modeling techniques perform the best when individual studies were viewed separately. Some researchers have

been conducting empirical overview of various software quality prediction techniques and analyze their performance in terms of various software datasets.

Khoshgoftaar and Seliya [72] compared seven fault prediction techniques that were built using a variety of tools. The models were built using different regression and classification trees including C4.5, CHAID, different versions of CART, logistic regression, and case-based reasoning. The techniques were evaluated against each other by comparing a measure of expected cost of misclassification. The differences between the techniques were at best moderate. They explained that the datasets and system characteristics affect the performance of prediction models.

Guo et al. (2004) compared 27 modeling techniques including random forest, logistic regression and other techniques available through the WEKA tool using five projects from NASA repository. The study compared the techniques using five different datasets from the NASA MDP program, and although the results showed that Random Forests perform better than many other classification techniques in terms of accuracy and specificity, the results were not significant in four of the five data sets.

In [86], the authors compared the performance of thirty predictive techniques on two datasets - JEditData and AR3 from PROMISE repository. This study showed that classification via regression technique and LWL performed better than the other techniques. However, this study was inconclusive as it only used two datasets.

Jiang and Cukic [87] claimed that comparison of fault prediction models is a multi-dimensional problem. Their results across multiple software projects as well as performance measures showed that there was rarely one model that can be proved to be the best for all possible uses in software quality assessment.

Elish and Elish [88] compared SVM against eight other modeling techniques. The modeling techniques were evaluated in terms of accuracy, precision, recall and the F-measure using four data sets from the NASA Metrics Data Program Repository. All techniques achieved an accuracy ranging from approximately 0.83 to 0.94. Their results showed that there were some differences, but no single modeling technique was significantly better than the others across data sets.

Vandecruys [89] compared Ant Colony Optimization against well-known techniques like C4.5, support vector machine (SVM), logistic regression, K-nearest neighbor, RIPPER and majority vote. In terms of accuracy, C4.5 was the best technique. However, the differences among the techniques in terms of accuracy, sensitivity and specificity were moderate.

Lessman [2] tried to benchmark classification techniques for software fault prediction problem. In his study, 22 techniques over 10 public domain datasets from NASA repository were compared. However, there are no significant performance differences detected among these techniques. He also argued that fault prediction techniques should not be judged on their predictive performance alone, but that other aspects such as computational efficiency, ease of use, and especially comprehensibility should also be paid attention to.

Menzies [1] achieved fault prediction performance of pd=71% and pf=25% on NASA projects using Naive Bayes leaner (with logNum filter) as predictive model, but they also admit that the conclusion may not still apply when the data sets are changed. Another study by Menzies [12] also suggested that to select a preferred learner for a particular domain.

Catal and Diri [90] collected 74 software fault prediction papers in 11 journals and several conferences. According to their review, they indicated that machine learning techniques have better features than statistical methods or expert opinion based approaches, and they suggested that the percentage usage of machine learning techniques should be increased. An extension to this study can be found in [91] where Catal investigated 90 software fault prediction papers published from 1990 to 2009 and provided review on each papers in terms of the year the papers published. Current trend in software fault prediction domain was discussed in their paper.

Arisholm [85] compared many data mining and machine learning methods to build predictive models in an industrial setting for a java system. They showed that the choice of predictive techniques has limited impact on the resulting classification accuracy or cost-effectiveness. They argued that fault prediction techniques that are ranked, as the best is highly dependent on the evaluation criteria applied. Thus, it is important that the evaluation criteria should be justified in the context in which the models are to be applied.

Tracy [55] reviewed 208 papers in term of software fault prediction from 2000 to 2010. They illustrated that simple technique, such as Naive Bayes and Logistic Regression perform comparatively well comparing to technique like SVM and C4.5. However, they also claimed that models seem to have performed best where the right techniques have been selected for the right set of data.

D'Ambros [92] provided a benchmark for software fault prediction models using publicly available datasets consisting of several software systems. They presented an extensive comparison of well-known prediction techniques as well as novel approaches. Their results showed that, while some approaches perform better than others in a statistically

significant manner, external validity in defect prediction is still an open problem, as generalizing results to different context/leaners proved to be a partially unsuccessful endeavor.

Dejaeger [93] investigated 15 different Bayesian Network algorithms and compared them to other popular machine learning techniques in terms of the AUC and H-measure. Their results showed that augmented Naive Bayes could perform similar or better than the commonly used Naive Bayes classifier. They also claimed that the development context is an item, which should be taken into account during modeling selection.

Recently, a new study [94] evaluated 179 machine learning classifiers, arising from 17 families, over 121 data sets. They concluded that "the classifiers most likely to be the best are the Random Forest (RF) versions". However, they also recognized that the best-performed classifier has no significantly different with the second best - SVM classifier.

2.4 Semi-supervised and Active learning in Software Fault Prediction problem

To our knowledge, semi-supervised learning has been marginally considered in the field of software fault-proneness prediction. The earliest study is the work from Khoshgoftaar on NASA MDP software projects[95]. In their study, an EM-based semi-supervised learning algorithm was implemented. As we've discussed in previous section, the EM algorithm is natural to this problem since one could view the labels of unlabeled instances as missing and thus semi-supervised learning can be reduced to be missing data problem. In their study, a case study is presented in which NASA software project JM1 is used as training data for software measurement modeling. A small size of labeled data is randomly selected from JM1, while remaining modules are treated as the unlabeled dataset. The performance of the EM-based semi-supervised algorithm is evaluated with multiple test datasets consisting of other NASA software projects. Their results demonstrated that the semi-supervised learning approach yielded better performance than a decision tree algorithm - C4.5 trained on program modules with known fault proneness data. Unfortunately, unlike random forest, $C4.5$ has not been identified as one of the top supervised learning algorithms on the MDP data set [2] making this result inconclusive. They also examined the modules remaining in the unlabeled dataset to be noisy from the perspective of data mining. They observed that roughly half of the modules that remain were in common as noisy.

Another interesting approach is semi-supervised clustering [96]. Unlike in self-training which extends supervised learning into semi-supervised learning, this approach extends traditional unsupervised learning (clustering) into semi-supervised context so that better partitions (or grouping) is achieved with the use of unlabeled data. However, this is not an entirely automated approach and requires software engineering experts in the loop. Semi-supervised clustering improves the performance compared to the corresponding unsupervised learning, but unsupervised learning does not perform as well as supervised learning. Hence, it is not likely that semi-supervised clustering is a good candidate for practical applications.

Catal[97] proposed an artificial immune system based semi-supervised learning approaches. In their proposed approach, a recent semi-supervised learning algorithm called YATSI (Yet Another Two Stage Idea) is used and in the first stage of YATSI, AIRS - Artificial Immune Recognition System - is applied. In addition, AIRS and Random Forest are benchmarked. Their experiments showed that the performance of AIRS based YATSi are comparable with Random Forest algorithm.

Kocaguneli [98] proposed an active learning solution to the problem of software effort estimation which relax the label data requirement. The proposed approach requires at most 40% of the original data and can perform as well as state-of-the-art supervised learners, which require all the available instances and labels. The reduced set of instances that can provide performance values as good as using all the instances is called as the essential content of the dataset.

Guangchun [99] implemented a two-stage active learning algorithm (TAL) for software defect prediction, in which clustering and support vector machine techniques are combined. Their results show that the proposed method improves the performance with a moderate labeling effort.

Li[100] proposed a software fault prediction approach which maps ensemble learning, random forest, into semi-supervised learning setting. Three methods of sampling were discussed in this study: random sampling with conventional machine learners, random sampling with semi-supervised learning learner and active sampling with active semi-supervised learning learner. The proposed semi-supervised learning methods - CoForest and ACoForest - then construct defect prediction models based on selected samples. In their CoForest method, random forest is trained using initially labeled modules. Each random tree is then iteratively refined with the original labels and the labels assigned to previously unlabeled modules from the other random trees. When the stop criterion is reached, the majority voting from the ensemble forms the prediction. CoForest is a disagreement-based semi-supervised learning algorithm, which exploits the advantage of both semi-supervised learning and ensemble learning. The ACoForest method extends

CoForest by actively selecting and labeling some previously unlabeled data from training the classifiers. Their results showed that the prediction models constructed using CoForest and ACoForest can achieve better performance than those using conventional machine learning techniques, such as logistic regression, decision tree and Naive Bayes.

Chapter 3

Semi-Supervised Learning for SFP problem

3.1 Semi-Supervised Learning approaches

Semi-supervised learning has tremendous practical value. In manay tasks, there is a dearth of labeled data. The labels Y may be difficult to obtain because they require human annotators, special devices, or expensive and slow experiments. Labeling fault data in software development falls in this category. Semi-supervised learning is attractive because it can potentially utilize both labeled and unlabeled data, assuming that information hidden in unlabeled data are useful in term of prediction.

In the past decade, semi-supervised learning has provided a class of classification approaches that can outperform corresponding supervised learning approaches, especially when $\mid L \mid << \mid U \mid$. Of particular note is self-training, which is the simplest and has less restriction on data compared to the others. One can take a supervised approach as base learner and extend it to semi-supervised learning by an iterative procedure. There are different variants of self-training. The classic one is to take the instances with the high confident scores from unlabeled data and then incorporate them (along with the corresponding predictions) into the initial labeled data to train a new leaner for subsequent iteration. The procedure repeats until converge or some stop criterion is met.

Yarowsky's algorithm [101] is the earliest version of such approach in which a simple decision list learner forms the "inner loop". If instance x contains feature f, then predict label j, and selects those rules whose precision on the training data is highest. The Outer loop of Yarowsky's algorithm is given a seed set of rules to start with. The initial Yarowsky algorithm is extended with important modifications such as those from

Abney et al. [102] and Haffari et al. [103]. The former argued that the best threshold could be fixed at $1/n$ where n is the size of initial labeled data, and an instance must stay labeled once it becomes labeled, but the label may change. The latter provided a general framework together with mathematical analysis on the variants of the Yarowsky's algorithm.

In this study we investigate two variants of traditional self-training based semi-supervised approaches in fault prone prediction problem: (i) the existing Fitting The Fits (FTF) approach [104] and (ii) a new variation called Fitting The confident Fits (FTcF).

3.1.1 Notation Definition

To begin, let X be the $(n+m) \times p$ matrix that denotes the given software data set. n is the size of labeled set l and m is the size of unlabeled set u. Rows in X are p-dimensional vectors defined as x, with $x \in \Re^p$. Specifically, $X = \{X_l, X_u\}$, where $X_l = \{x_1, x_2, \cdots, x_n\}$ and $X_u = \{x_{n+1}, x_{n+2}, \cdots, x_{n+m}\}$. Let $Y = \{Y_l, Y_u\}$ be response variable (or labels) where $Y_l = \{y_1, y_2, \cdots, y_n\}$ and $Y_u = \{y_{n+1}, y_{n+2}, \cdots, y_{n+m}\}$ is missing or unspecified. The observed labels are binary variables, $y_i \in \{0, 1\}$, where 0 denotes non-fault prone (*nfp*) module and 1 denotes fault prone (*fp*) module.

Our task with the investigated algorithms is to extend supervised learning into semi-supervised setting. Let $\phi(.)$ be any given supervised learner (base learner). Given a set of input-output pairs $D_l = (X_l, Y_l)$, the notation $\phi_{D_l}(X_u)$ indicates that the classifier trained from D_l is used to predict on unlabeled data set X_u. The probability class estimates (PCEs) for fault prone class, $\hat{p} = P(Y = 1|X_u)$, are returned. PCEs are a specific form of confident scores which is generally used in the literature. Commonly, we consider a module as fault prone when $\hat{p} > \tau$ and non fault prone otherwise, where $\hat{p} \in [0, 1]$ and τ is a specified threshold for making the decision.

3.1.2 Fitting The Fits algorithm - FTF

The FTF algorithm provides an interesting variant of traditional Yarowsky algorithm by extending learners from a supervised setting into a semi-supervised setting. It initially sets up the labels for unlabeled data to ensure that both the labeled set and the unlabeled set are labeled, and then a supervised procedure is implemented on the entire data set. The labels for the unlabeled data are gradually updated until a convergence criterion is met. This is different from Yarowsky's algorithm in which only a subset of unlabeled data is used to train a new classifier at each iteration. Also, FTF can be shown to globally converge which is a property that cannot be achieved with Yarowsky's algorithm.

Figure 3.1 gives the description of the FTF algorithm. The procedure of FTF starts with setting the initial labels for the unlabeled data at the $0th$ iteration. Specifically, a learner is trained from initial labeled data $D_l^{(0)} = (X_l, Y_l)$, and then the learner is used to predict the labels for unlabeled data $\hat{Y}_u^{(0)} = \phi_{D_l^{(0)}}(X_u)$. In the loop, labels for initial labeled data set are always reset to be original values $\hat{Y}_l^k = Y_l$ (step 3). The base learner which is built based on current status of entire data set $D^{(k)} = (X, \hat{Y}^{(k)})$ is used to predict new labels for entire data set $\hat{Y}^{(k+1)} = \phi_{D^{(k)}}(X)$ (step 4). This cycle continues until the statuses of labels converge. We observed that the convergence property is sensitive to the use of base learner, which will be discussed later. Note that the predictions of unlabeled data $\hat{Y}_u^{(k)}$ are the probability values (PCEs) in the sense that in the loop the learner trained is regression based.

Algorithm1: Fitting The Fits (FTF)
1: Initialization: $\hat{Y}_l^0 = Y_l$, $\hat{Y}_u^0 = \phi_{D_l^{(0)}}(X_u)$, $k = 0$;
2: loop until convergence*:
3: $\quad \hat{Y}_l^k = Y_l$
4: $\quad$ Fit $\hat{Y}^{(k+1)} = \phi_{D^{(k)}}(X)$, where $D^{(k)} = (X, \hat{Y}^{(k)})$
5: $\quad k = k + 1$
6: End loop

FIGURE 3.1: FTF algorithm

3.1.3 Fitting The confident Fits algorithm - FTcF

Next, we discuss another iterative self-training approach that can be considered as a variant of Yarowsky's algorithm: Fitting The Confident Fits (FTcF). Typically, a learner trained from current labeled data is used to classify the available unlabeled data. Predicted instances from unlabeled data with high confident scores are considered and added to the pool of labeled data set. Both sizes of labeled data and unlabeled data are updated due to the migration of instances from unlabeled data to labeled data at each iteration. As the procedure accesses to the end, the size of unlabeled data set goes to be zero, which means that all instances from unlabeled data set are labeled. The main difference between FTcF and Yarowsky's algorithm is that in Yarowsky's algorithm selected instances, which are used to train new learner are always given back to the pool of unlabeled data set so that the size of unlabeled data never changes. For FTcF, instances added to the pool of labeled data will stay in the pool with the fixed labels. Therefore, the size of unlabeled data decreases until exhausted. Figure 3.2 provides the details for FTcF algorithm.

Compared to the FTF algorithm, which constructs the learner based on all the modules set by giving an initial "guess" to the labels for unlabeled data, FTcF always learns from current labeled data. Typically, in the iterative phase of FTF the labels for unlabeled data set are updated based on the decision rules probabilities from previous iteration. In contrast, FTcF gradually pushes confident modules into labeled data set so that the size of labeled data set is increasing iteration by iteration. More specifically, in FTF it is considered that there is a "better" classifier by repeatedly fitting the predicted labels in unlabeled data set; in FTcF only a small amount of labels with highest confident scores are trusted. Both algorithms need the guidance of initial labeled data at the beginning.

Algorithm2: Fitting The Confident Fits (FTcF))
1: Initialization: $\hat{Y}_l = Y_l$
2: loop until $\|u\| \to 0$:
3: Fit $\hat{Y}_u = \phi_{D_l}(X_u)$
4: Take u' confident cases from X_u
5: updating: $X_l = X_{l+u'}, \hat{Y}_l = \hat{Y}_l + \hat{Y}'_u$, and $X_u = X_{u-u'}$
6: End loop

FIGURE 3.2: FTcF algorithm

3.1.4 Base Learner

Both FTF and FTcF procedures share the underlying concept that a supervised learner is trained repeated by using some form of unlabeled data. Basically, the supervised learner plays two important roles in our investigated algorithms: (i) provides initialization for iterative fitting, and (ii) trains new leaner at each iteration. Apparently, a well-chosen base learner can provide effective prediction for initially unlabeled portion of the data set and ensure a good starting point for tracking "better" learners. On the other hand, the ability that a base learner extracts useful information from unlabeled data at each iteration decides the behavior of entire algorithm. Most important, the selected base learner may lead to local or global convergence. Thus, we have to carefully choose the base learner.

In supervised learning literature, there are lots of choices. We have two constraints on the choices of base learner. First, the learner should have competitive performance in the fault prediction domain and its implementation should be available off-the-shelf. Second, it should produce well-calibrated probabilities, i.e., *PCEs*. In [105], an examination is provided on the relationship between the predictions made by different supervised algorithms and true posterior probabilities. They showed that some learning algorithms they examined, such as random forest and logistic regression, showed little or no bias

and predicted well-calibrated probabilities. Based on their work, we will explore random forest and logistic regression as supervised base learners. Also, support vector machine are worth for consideration due to their popularity and off-the-shelf status [106].s

- Logistic Regression (LR)
  Logistic regression is a standard off-the-shelf approach for building models for binary classification and has been widely used in software fault prediction problem. Let us assume that PCEs are modeled as a function of a linear combination, i.e, $\hat{p} = f(X\beta) \in [0, 1]$ where $f(.)$ is a link function . For logistic regression, the function is given as $f(X\beta) = \frac{e^{X\beta}}{1+e^{X\beta}}$, which is the logistic transform applied to $X\beta$. This transformation forces probabilities to be between 0 to 1. Parameters of a logistic regression model are usually estimated using the maximum likelihood method.

- Support Vector Machine (SVM)
  Support Vector Machine (SVM) is another popular classification approach which is motivated by the intuitive geometric interpretation of maximizing the margin. When two classes of points can be separated by a hyper-plane, it is natural to use the hyperplane that separates the two classes of points by the largest margin. This amounts to the hard margin support vector machine:

$$\begin{aligned} \min_{w,b} \quad & \frac{1}{2}||w||^2 \\ & y_i(w^T\phi(x_i) + b) \geq 1, \end{aligned} \tag{3.1}$$

  The goal is to find the hyper-plane described by $\{w, b\}$ that generates a maximal margin between two classes of points. One can also utilize a mapping function $\phi(.)$ to allow for the linear separation in non-linear classification problem. To allow the misclassification, one can incorporate a penalty vector ξ such that $\xi_i \geq 1$ indicates the corresponding point x_i is misclassified. This is well known as soft margin support vector machine.

$$\begin{aligned} \min_{w,b,\xi} \quad & \frac{1}{2}||w||^2 + C\sum_i \xi_i \\ & y_i(w^T\phi(x_i) + b) \geq 1 - \xi_i, \xi_i > 0 \end{aligned} \tag{3.2}$$

- Random Forest (RF)
  Random forest is an ensemble of individual tree predictors such that each tree is

randomly generated based on the values of a random sampled vector. The outputs of random forest are the majority votes by all individual trees. In random forest, each tree is grown on a bootstrap sample of the training data set, which ensures the independence of different trees. At each node of a tree, m out of M (M is the total number of attributes for each instance) attributes are randomly selected and the split is chosen from the m attributes for the node. Once the trees are built, the final classification is given by the majority votes within the ensemble.

TABLE 3.1: Datasets used in this study

Data	**Size#**	**% faulty**	**project description**	**language**
JM1	10,878	19.3%	Real time predictive ground system	C
KC1	2109	13.9%	Storage management for ground data	C++.
PC3	1563	10.43%	Flight software for earth orbiting satellite	C
PC4	1458	12.24%	Flight software for earth orbiting satellite	C
PC1	1109	6.59%	Flight software from an earth orbiting satellite	C

3.2 Experiment and Results

3.2.1 Experimental Data Sets

The experimental datasets used in this study are collected from mission critical projects at NASA, as a part of the Software Metrics Data Program (MDP). The repository provides metrics that describe the software artifacts from 13 NASA projects. We selected five of them that number of modules is larger than 1000. Table 3.1 provides the brief description of these datasets. Each module in these projects is measured in terms of the same set of software product metrics. A label associated with each module indicates if the module has been found to contain one or more faults (fault prone, *fp*) or no faults have been detected (not fault prone, *nfp*).

In order to provide a better understanding on our experimental data, we briefly describe each project:

- **JM1**: JM1 project is a real-time ground system that uses simulations to generate certain predictions for the space mission. It is coded in C language. There are eight years of fault data associated with the modules and their metrics. Modules in JM1 were characterized by 21 software measurement attributes. The data set contains $10,878$ modules, of which $2,102$ have one or more faults and $8,776$ have zero faults, the rate of 19.3%. The maximum number of faults in a module is 26.

- **PC1**: The PC1 project is flight software from an earth-orbiting satellite that is no longer operational. There are eight years of fault data associated with the metrics. It consists of more than $40,000$ lines of source code written in C. The software measurement data set contains $1,107$ modules characterized by 41 attributes / metrics. Only 76 modules have one or more faults and $1,031$ have zero faults, fault rate of 6.59%. The maximum number of faults in a module is 9.

- **PC3**: The PC3 project is also a flight software from an earth-orbiting satellite, but the mission is currently operational. It consists of approximately $40,000$ lines of source code written in C. The data set describes $1,563$ modules characterized by 41 attributes, of which 160 have one or more faults and $1,403$ have zero faults, the fault rate of 10.43%. The maximum number of faults in a module is 9.

- **PC4**: The PC4 project is flight software for an Earth orbiting satellite that is currently operational. It consists of approximately $36,000$ lines of source code written in C. The software measurement data set contains $1,458$ modules characterized by 41 attributes, of which 178 have one or more faults and $1,280$ have zero faults, with the fault rate of 12.24%. The maximum number of faults in a module is 25.

- **KC1**: The KC1 project is a computer software configuration item within a large ground system and consists of approximately $43,000$ lines of source code written in C++. The data set contains $2,107$ modules, of which 325 have one or more faults and $1,782$ have zero faults with the fault rate of 13.9%. The maximum number of faults in a module is 7.

It is important to note that these projects did not share a development process, they come from different government contracting organizations and, generally, at the time of their development, it would not have been possible to use one of them as a training data set for the fault prediction modeling on the other one. In situations like this, if an organization wants to deploy a fault prediction model, it needs to develop it using the metrics and fault labels associated with the modules emerging from the development early.

3.2.2 Experimental Setting

Before presenting our result, we need set up experimental parameters for our experiments. Four parameters need to be taken account:

1. *Rate of Faults*
   Although we know the fault rate for each of the data sets used, practically they are

not available until the fault data is obtained. We could not exactly obtain the fault rate of software prior to the development of a project. In our experiments, initial labeled data must be given so that semi-supervised learning can be achieved. Under this consideration, we provide a rough "guess" of the fault rate for the sampling of initial labeled data. It is taken as 10% across the five data sets used which is a reasonable setting as the overall fault rates in the five data sets is ranged from 6.59% to 19.3% (Table 3.1). Therefore, when a subset is sampled from the original data to generate the initial labeled set, the proportion between *fp* and *nfp* is 1 : 9 prior to our training procedure.

2. *Size of initially labeled data*
   In semi-supervised learning literature, it is aimed to exploit the unlabeled data set in order to improve the performance of the classification achieved by a relatively small size of labeled data ($| L | << | U |$). Intuitively, the size $| L |$ decides the effectiveness of labeled data to exploit the information from unlabeled data. Additionally, the lower $| L |$ is, the more reduction of human-labeling consumption is. To this end, we designed our experiments by exploring varied size of labeled data from the range of 2% to 50% (2%, 5%,10% 20%, 50%) of entire data on each data set used. Note that the size of data sets used are different, the size of the initial labeled data also differs from data to data. Table 3.2 shows the number of modules used as initially labeled data corresponding to the setting at 2%, 5%,10% 20%, and 50%.

3. *Number of iteration for FTF*
   In FTF, a trained learner will repeatedly update the status of predicted labels for unlabeled data until some stop criteria are met. The stop criteria could be the convergent status. However, the convergence property is not common for any given base learner in FTF algorithm. Some base learner, such as random forest, will never converge (will be discussed later). In this study, we force FTF algorithm to stop when it reaches 50 iterations.

4. *Growth size for FTcF*
   Unlike FTF, FTcF gradually increases the size of labeled set by adding the most confident candidates from unlabeled set until the size of unlabeled data set is exhausted. Therefore, we need to know how many candidates should be moved from unlabeled pool to labeled pool in each iteration, named as Growth Size (GS). Some of our data sets are large and the others are small. For example, PC1 only has 1107 modules, and JM1 has 10878 modules. Under this consideration, we used two different settings of Growth Size for these data sets. For PC1, PC3, PC4, KC1, which have module size smaller than 3,000, we pass 10 most confident candidates

to per iteration labeled set (GS=10). Meanwhile, we pass 100 most confident candidates to labeled set per iteration for the fifth data sets JM1 (GS=100), which has the modules size larger than 10,000. Number of iteration for FTcF can then be obtained by $\frac{|U|}{GS}$.

TABLE 3.2: Number of modules in initially labeled data set

	2%	**5%**	**10%**	**20%**	**50%**
JM1	218	544	1088	2176	5439
KC1	42	105	211	421	1054
PC3	31	78	156	313	782
PC4	29	73	146	292	729
PC1	22	55	111	221	554

To depict the performance of fault prediction experiments we will provide the Probability of Detection (PD) and the Area Under Receiver Operating Characteristic Curve (AUC) as the measures of binary classification. (PD), also called recall or specificity, is defined as the probability of correctly classifying a module as fault-prone, that is $P(\hat{C} = fp|C = fp)$. It is one of the most commonly used measures in this field. It is clearly suitable for representing the ability of an algorithm to correctly classify the instances of a minority class (fp). The PD value is obtained based on a specified threshold, which can be adjustable by user as needed. Receiver Operating Characteristic (ROC) curve is a plot of probability of detection (PD)as a function of the probability of false alarm (PF, the probability of misclassifying a fault free module as faulty) across all thresholds setting. AUC is the area under the ROC curve, another frequently used measure for performance evaluation in software fault prediction [87].

As discussed in previous sections, semi-supervised learning should improve software fault prediction models in situations where the number of modules with known fault content (labels) is limited. So, the goal is to improve the performance of prediction on unlabeled data (test data) using augmented training data set, referred as transductive semi-supervised learning. Our study aims to answer the following questions:

1. What is the role of base learner in FTF and FTcF?
2. Does FTF or FTcF outperform supervised learning?
3. How small can the size of the labeled data set be for FTF or FTcF to outperform supervised learning?
4. Are the behavior and performance of FTF or FTcF consistent over different data sets?

With these questions in mind, we developed experiments that compare the self-training algorithms and the corresponding supervised approach. Both the supervised and self-training approaches use the same base learner. Additionally, we will vary the sizes of labeled data instances: 2%, 5%, 10%, 20%, 50%, of the size of the five MDP projects. To track performance trends, we will present above described performance measures (PD and AUC) at each iteration of the FTF and FTcF, so that a performance curve can be derived. A total of 20 experiments were performed on each experimental setting. The instances of labeled data are randomly selected from each set for the first iteration of both self-training, with the remaining software modules being used for prediction as unlabeled data. Ideally, in the context of software development scenario, the components that first come out of development would be utilized for model building. Unfortunately, this information is not available for MDP projects. This is the reason for the 20 experiments over the same data set (and the proportion of labeled data). The repetition indicates our attempt to make the order of component delivery to the project less important. We understand this is one of the validity threats. Nevertheless, within the MDP data repository, there is no remedy that would offer its reduction or elimination.

3.2.3 The Role of base learners

Perhaps the easiest-to-apply semi-supervised learning, self-training is characterized by the fact that the learning process uses its own predictions to teach itself. For this reason, it is important to select a good guide - base learner. Before going to the experiments on FTF and FTcF, let us first take a look on the role of base learner in both approaches. In this section, the convergence property of three standard supervised learners - Logistic regression, Support vector machine and Random forest - in both FTF and FTcF will be analyzed. Whether or not the convergence property contributes to outperformance will be discussed as well.

3.2.3.1 Convergence with Logistic Regression

In supervised setting, the estimates of logistic regression model, $\widetilde{\beta}^{log}$, can be found via solving the equation of $X^T(Y - P) = \vec{0}$ and the prediction of X_l will be $\hat{Y}_l = p_l^{log} = P(X_l, \widetilde{\beta}^{log})$.

In FTF algorithm, a repeat procedure will be provided at each iteration:
i) $\hat{Y}_l^k = Y_l$
ii)$\hat{Y}_u^k = P(X_u, \hat{\beta^{k-1}}) = p_u^{k-1}$.

Thus, for k^{th} iteration, we can have $\hat{\beta^k}$ by solving equation of $X_l^T(p_l^{log} - p_l^k) + X_u^T(p_u^{k-1} - p_u^k) = \vec{0}$ which is derived from $X^T(Y - P) = \vec{0}$. Our solution then turns to be:

$$\hat{\beta}^k = \widetilde{\beta}^{log} + \left[(X^T W X)^{-1}(X_u^T W_u X_u)\right]^k (\hat{\beta}^0 - \widetilde{\beta}^{log}) \tag{3.3}$$

We can show that when $k \to \infty, \hat{\beta} \approx \widetilde{\beta}^{log}$. The detailed proof can be found in Appendix I. It implies that when logistic linear regression is used as a base learner, the unlabeled data is not helpful as we expected.

In FTcF algorithm, the size of labeled data set and unlabeled data set will be changed at each iteration. The algorithm will update the pool of labeled data set by gradually adding most confident labeled data from unlabeled data pool until the unlabeled data pool exhausted.

For a certain iteration, say k^{th} iteration, we will have:

$$\begin{aligned} X^{(k)} &= \begin{pmatrix} X_l \\ X_{lk} \end{pmatrix}, \\ \hat{Y}^{(k)} &= \begin{pmatrix} \hat{Y}^{(k-1)} \\ P(X_{lk}^T, \beta_{k-1}) \end{pmatrix}, \\ p^{(k)} &= \Big(P(X^{(k)}, \beta_k) \Big) \end{aligned}$$

By translating the equation $X^T(Y - P)$ at any give iteration, we found that above equation can be presented as the form of $Sup(X_l^T) + Sup(X_{l'}^T) + Sup(X_{lk}^T)$, where $l' = l1 + l2 + \cdots + l(k-1)$ is the cumulative most confident points, and $Sup(X)$ means supervised learning on X.Therefor, for any given iteration k, the equation can be write to be above form which contains three terms:
i) $Sup(X_l^T)$ Supervised learning on X_l;
ii) $Sup(X_{l'}^T)$ Supervised learning on $X_{l'}$;
iii) $Sup(X_{lk}^T)$ Supervised learning on X_{lk}.
It turns out that$\hat{\beta}_0 = \hat{\beta}_1 = \hat{\beta}_2 = \cdots = \hat{\beta}_k = \tilde{\beta}^{log}$ for any given iteration as long as the logistic linear regression exists. That is, unlabeled data cannot help to improve the prediction.

Until now, we learned that logistic regression is not a good candidate as base learner in self-training.

3.2.3.2 Convergence with SVM

In this section, we will analyze the convergence property of semi-supervised learning when SVM is used as base learner. Recall that a standard SVM algorithm for binary class problem can be defined as:

$$\begin{aligned} \min_{w,b,\xi} \quad & \frac{1}{2}||w||^2 + C\sum_{i=1}^{n}\xi_i \\ & y_i(w^T\phi(x_i)+b) \geq 1-\xi_i, \\ & \xi_i \geq 0, i = 1,\ldots,n. \end{aligned} \tag{3.4}$$

where $C > 0$ is a regularization constant and $\phi(.)$ is a kernal function that maps x_i into high dimensions. In supervised learning setting, by obtaining the optimal solution of $\{w, \xi, b\}$, one can predict the label of a point x_i by taking the sign of the equation $(w^T\phi(x_i)+b)$, i.e., $\hat{y}_i = sign(w^T\phi(x_i)+b)$. If $\xi_i > 1$, x_i is incorrectly classified.

In the first step of FTF algorithm, y_i are the true labels for data X_l, i.e., $Y_l = [y_1, y_2, ..., y_n]^T$ and $x_i \in R^n$. After solving the optimization problem (equation 3.4), we can have the optimal solution $\{w^{(0)}, \xi^{(0)}, b^{(0)}\}$. As for the iterative steps - step 2-6 in FIGURE 3.1, the FTF algorithm repeatedly solves the same optimization problem, but y_i are the true labels of X_l together with the predicted labels of X_u from previous iteration, i.e., $Y^{(k-1)} = [y_1, ..., y_n, \hat{y}_{n+1}^{(k-1)}, ..., \hat{y}_{n+m}^{(k-1)}]^T$ and $x_i \in R^{n+m}$. k stands for the k^{th} iteration. At each iteration k, the optimal solution is $\{w^{(k)}, \xi^{(k)}, b^{(k)}\}$. To show the convergence property of SVM in FTF algorithm, we write the objective function in the optimization problem at k^{th} iteration as:

$$f(w^{(k)}, \xi^{(k)}) = \frac{1}{2}||w^{(k)}||^2 + C\sum_{i=1}^{n+m}\xi_i^{(k)} \tag{3.5}$$

In Appendix C we've shown that at the first iteration in FTF algorithm, $\{w^{(0)}, \xi^{(0)}, b^{(0)}\}$ is feasible solution of following optimization problem:

$$\begin{aligned} \min_{w,b,\xi} \quad & \frac{1}{2}||w^{(1)}||^2 + C\sum_{i=1}^{n+m}\xi_i^{(1)} \\ & y_i^{(0)}((w^{(1)})^T\phi(x_i) + b^{(1)}) \geq 1 - \xi_i^{(1)}, \\ & \xi_i^{(1)} \geq 0, i = 1, \ldots, n+m. \end{aligned} \tag{3.6}$$

where $\{w^{(1)}, \xi^{(1)}, b^{(1)}\}$ is the optimal solution. Thus,

$$f(w^{(0)}, \xi^{(0)}) \geq f(w^{(1)}, \xi^{(1)}) \tag{3.7}$$

Similarly, at k^{th} iteration where $k > 1$, we showed that $\{w^{(k-1)}, \xi^{(k-1)}, b^{(k-1)}\}$ is feasible solution of following optimization problem:

$$\begin{aligned} \min_{w,b,\xi} \quad & \frac{1}{2}||w^{(k)}||^2 + C\sum_{i=1}^{n+m}\xi_i^{(k)} \\ & y_i^{(k-1)}((w^{(k)})^T\phi(x_i) + b^{(k)}) \geq 1 - \xi_i^{(k)}, \\ & \xi_i^{(k)} \geq 0, i = 1, \ldots, n+m. \end{aligned} \tag{3.8}$$

where $\{w^{(k)}, \xi^{(k)}, b^{(k)}\}$ is the optimal solution. Apparently,

$$f(w^{(k-1)}, \xi^{(k-1)}) \geq f(w^{(k)}, \xi^{(k)}) \tag{3.9}$$

Incorporating with equation 3.7, we can conclude that equation 3.9 holds for any given $k \geq 1$. Since $f(w^{(k)}, \xi^{(k)}) \geq 0$, we proved that $f(w^{(k)}, \xi^{(k)})$ is convergent when k increases. Thus FTF algorithm is convergent.

As for FTcF algorithm with SVM, it is easy to prove that the convergence property in FTF algorithm also holds in FTcF. Assume the $\{w^{(0)}, \xi^{(0)}, b^{(0)}\}$ is the optimal solution of optimization problem in equation 3.4 where only labeled data is used, and the $\{w^{(1)}, \xi^{(1)}, b^{(1)}\}$ is the optimal solution of the following problem:

$$\begin{aligned} \min_{w,b,\xi} \quad & \frac{1}{2}||w^{(1)}||^2 + C\sum_{i=1}^{n+u^{(1)}} \xi_i^{(1)} \\ & y_i^{(0)}((w^{(1)})^T\phi(x_i) + b^{(1)}) \geq 1 - \xi_i^{(1)}, \\ & \xi_i^{(1)} \geq 0, i = 1, \ldots, n + u^{(1)}. \end{aligned} \tag{3.10}$$

where $u^{(1)}$ is the most confident cases selected at step 4 in FIGURE 3.2. By expanding the vector $\xi^{(0)}$ to be the length of $n + u^{(1)}$ and using the prediction rule $y_i^{(0)}((w^{(0)})s^T\phi(x_i) + b^{(0)}) \geq 0$ for $i = n+1, .., n+u^{(1)}$, we can prove that $\{w^{(0)}, \xi^{(0)}, b^{(0)}\}$ is a feasible solution of optimization problem in equation 3.10. Thus,

$$f(w^{(0)}, \xi^{(0)}) \geq f(w^{(1)}, \xi^{(1)}) \tag{3.11}$$

Similarly, for any $k > 1$,$\{w^{(k-1)}, \xi^{(k-1)}, b^{(k-1)}\}$ is feasible solution of the following problem:

$$\begin{aligned} \min_{w,b,\xi} \quad & \frac{1}{2}||w^{(k)}||^2 + C\sum_{i=1}^{n+u^{(1)}+\ldots+u^{(k)}} \xi_i^{(k)} \\ & y_i^{(k-1)}((w^{(k)})^T\phi(x_i) + b^{(1)}) \geq 1 - \xi_i^{(k)}, \\ & \xi_i^{(l)} \geq 0, i = 1, \ldots, n + u^{(1)} + \ldots + u^{(k)}. \end{aligned} \tag{3.12}$$

where $\{w^{(k-1)}, \xi^{(k-1)}, b^{(k-1)}\}$ should be the optimal solution and $u^{(k)}$ is the most confident cases selected at $k^{(k)}$ iteration. Thus we have,

$$f(w^{(k-1)}, \xi^{(k-1)}) \geq f(w^{(k)}, \xi^{(k)}) \tag{3.13}$$

Given that $f(w^{(k)}, \xi^{(k)}) \geq 0$, we proved that $f(w^{(k)}, \xi^{(k)})$ is convergent. Hence, FTcF algorithm is convergent.

By now, we've showed that both FTF and FTcF algorithm will converge when the iteration indicator k increases. However, it is not clear whether this convergence property in FTF and FTcF will lead to increased classification performance. From our empirical study, we learned that there are several factors that affect the performance of FTF and

FTcF algorithm when SVM is use as base learner. First, the balance in two classes can affect the performance. As a density based method, SVM relies on recognizing low-density margin between two classes and then forming hyper-plane to maximize the margin with lowest misclassification rate. When the data in two classes are heavily imbalanced, the low-density margin will be hard to identify. Second, the selection of kernel function or mapping function in SVM optimization problem can be critical. Different kernel function with the same data may lead to significant different classification performance. Prior knowledge on the data can be helpful to make the decision of what kernel function to use. However, prior knowledge is not always available. Another factor that affects the performance is the parameters in SVM, for example, the penalty term C and the parameters in kernel function. Cross Validation is the most widely used approach to tune these parameters in SVM. However, this often leads to local optimization and semi-supervised learning usually won't benefit from it.

Our results from using MDP data showed that both FTF and FTcF algorithm converged to performing worse than the corresponding supervised learning when SVM is used as base learner. We are not surprised with this observation, one reason is that our data is heavily imbalanced in classes.

3.2.3.3 Convergence with Random Forest

Random forest creates single trees on many selected data subsets that are uniformly sampled from the original data. The outputs of random forest are vote-based. This is a non-parametric method, so that it would be mathematically hard to track the estimates of outputs when it is implemented in FTF and FTcF algorithm. On the other hand, the randomness of tree building in random forest also breaks the possibility of estimates tracking. To explore the convergence property of random forest in our semi-supervised approaches, we implemented an experiment to help us visualize the convergence trends. We used the stop criteria: $\|\hat{Y}_k - \hat{Y}_{k-1}\| \leq \delta$, where$\delta = 1e-6$, for both investigated semi-supervised approaches. Here $\hat{Y} = \hat{p}(Y|X = x)$. The approach is considered to be convergent when the change of predictions between any two successive iterations is less than the value δ.

Figure 3.3 shows the convergence curves for both approaches on PC3 data set with the initial labeled data is 10% of the entire data set. From the figure, logistic regression converges immediately for both approaches, which exactly confirms the proof we presented above. The curve of SVM decreases at beginning and then converges soon. Unlike logistic regression and SVM, random forest does not present the convergence property. Instead, the changes of prediction on random forest are gradually decreased

but never to be zero. Intuitively, it implies that, compared to logistic regression and SVM, random forest has more hope to provide improvement in the iterative procedure of semi-supervised approaches. To further visualize the convergence behavior, we presented a comparison among three base learners on the measure of PD ($\tau = 0.5$) as shown in Figure 3.4. From the figure, we observed the same convergence trend for logistic regression and SVM as in Figure 3.3. Random forest presents increased trends for both semi-supervised approaches.

By now it should be clear that the performance of self-training heavily relies on the selected base learner. It is worth pointing out that, given a data set, blindly selecting a base learner in self-training will not necessarily improves performance over supervised learning. For example, when logistic regression is used as base learner, self-training performs the same as supervised learning when logistic regression is used as base learner and when there is solution of optimization problem in logistic regression. In fact, unlabeled data can lead to worse performance with inappropriately selected base leaner. For example, based on our preliminary experiment self-training may perform worse than supervised learning when SVM is used as base learner, especially when the data is imbalanced in terms of fault proneness and non-fault proneness and there are noises in data. However, there is also chance that the self-training outperforms supervised learning when SVM is used as base learner. This is because, as we've mentioned earlier, that the performance of SVM itself is heavily replies on the underlying assumption of the data set - when the data has no significant noise and the data is well-balanced, SVM is a good solution for classification problem. Apparently, SVM is not a good candidate as base learner in self-training due to its inconsistent performance.

However, we have learned that when Random forest is used as base learner, self-training performs consistently better than the corresponding supervised learning. In next section, we mainly focus on the investigation of self-training when random forest is used as base learner.

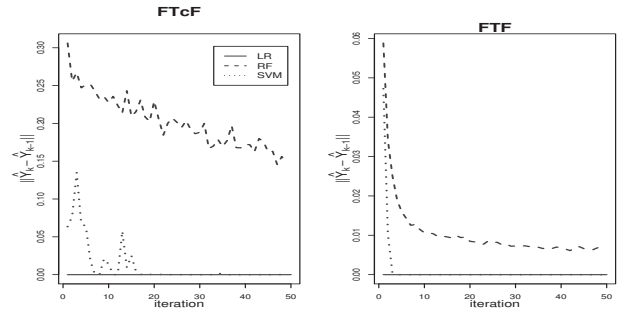

FIGURE 3.3: Convergence plot on PC3(10%labeled set used).

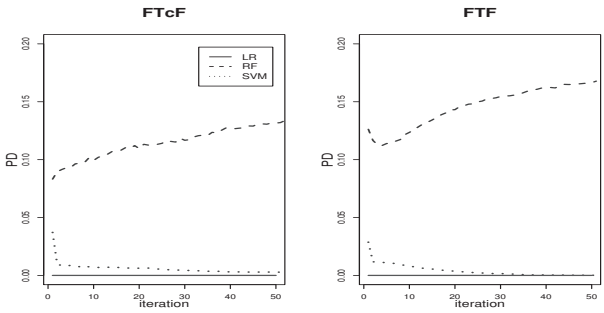

FIGURE 3.4: Convergence plot on PC3 with the measure of PD(10%labeled set used).

3.2.4 Results

This section we show the results of experiment for FTF and FTcF approaches. Figures 3.5 and 3.6 show the results of FTF and FTcF on PC3 data set respectively. Recall that we compute the PCEs ($\hat{p}$) to decide the class of a fault prone module, that is when $\hat{p} > \tau$ the module is classified as *fp* and as *nfp* otherwise. We measured PD based on three settings of threshold value (τ), i.e, 0.1, 0.5, 0.75, not a widely used setting at 0.5 considering the imbalance classes issue in the investigated data sets. Their results are presented as the first three plots in the figures. The fourth plot in the figures presents the comparisons with the measure of AUC. Note that the lines connecting shaped points represent the results of semi-supervised approaches. The single shaped points at $0th$ iteration and the extended straight lines represent the results of supervised random forest approach.

From Figure 3.5, we can see that when threshold is 0.1 (implies that modules with $\hat{p} > 0.1$ are classified as *fp*), the performance curves for FTF significantly exceed the corresponding straight lines(performance of supervised random forest) across all sizes of labeled data. When threshold is 0.5, most performance curves for FTF, except the lines of 2% and 5% labeled data, beyond the corresponding straight lines. When threshold is 0.75, the performance curves, except the line of 2% labeled data, still behave well but have relatively less improvement. The results based on AUC showed that our investigated semi-supervised approaches have the same performance as supervised learning throughout the iterative procedure. The similar behaviors are observed when FTcF is implemented on the same data set, which is shown in Figure 3.6.

Typically, when the threshold value is high (at 0.75), many modules are misclassified as non-fault proneness and then the lower PD value will be. Both FTF and FTcF algorithms help to identify more fault prone modules at threshold setting of 0.75 comparing to the supervised learning approach (check the first three plots in figure 3.5 and

3.6). However, when the threshold value decreases (to 0.1), more modules are correctly classified as fault proneness. That says there are only few fault prone modules are misclassified. This gives less space to improve for semi-supervised learning (check the fourth plot in both figures). When looking at the measure of AUC, as we mentioned in previous section it takes account of all possible threshold values. Our results based on the measure of AUC showed that the overall behavior by semi-supervised approaches across all PD and PF is as well as the corresponding supervised approach consistently. In other words, the improvement obtained by correctly identifying more fault prone modules with respect to PD can be considered as bonus by using semi-supervised approaches.

We observed that these trends of performance (PD and AUC) for both FTF and FTcF are quite common on all five data set used. Due to the consideration of paper size we only presented the results on the measure of PD at threshold is 0.5 (which is most commonly used in the literature) for other four data set (PC1, PC4, KC1, JM1), shown in Figure 3.7 and Figure 3.8. The numbers of modules used as initially labeled data can be found in table 3.2.

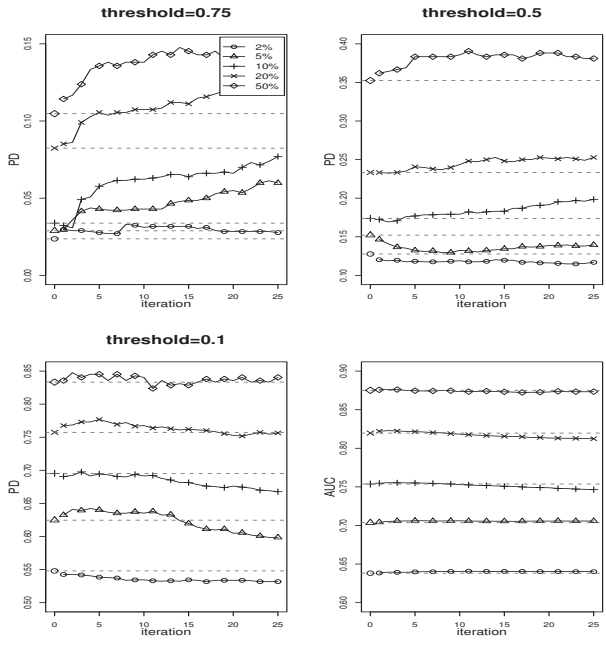

FIGURE 3.5: Results of FTF algorithm on PC3(the numbers of modules initially labeled at 2%, 5%, 10%, 20%, 50% are 31, 78, 156, 313,782 respectively)

3.2.5 Discussion

To our knowledge, semi-supervised learning has been marginally considered in the field of software fault-proneness prediction. One of the work on this topics was from Taghi

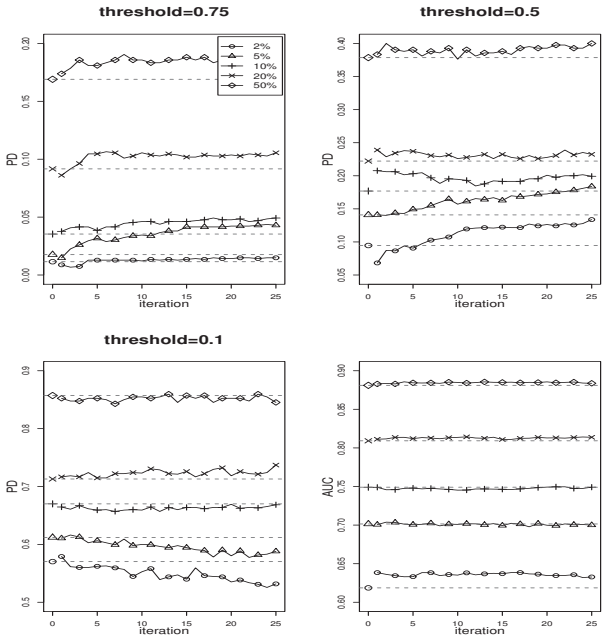

FIGURE 3.6: Results of FTcF algorithm on PC3(the numbers of modules initially labeled at 2%, 5%, 10%, 20%, 50% are 31, 78, 156, 313,782 respectively)

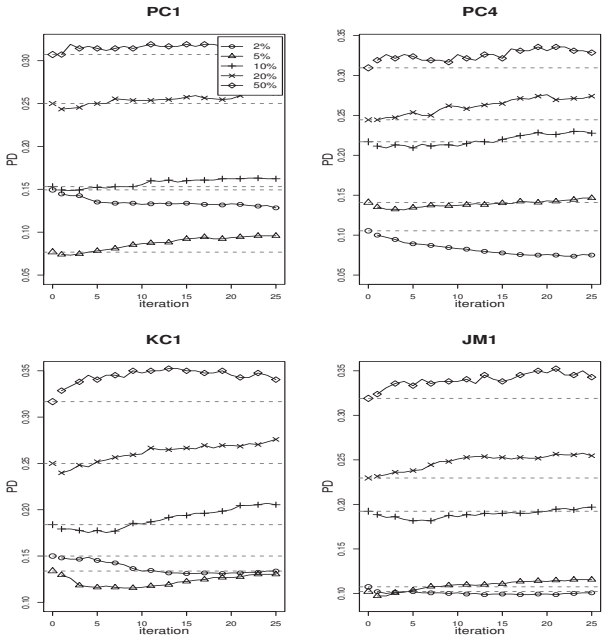

FIGURE 3.7: Results of FTF algorithm on four data set with threshold=0.5.

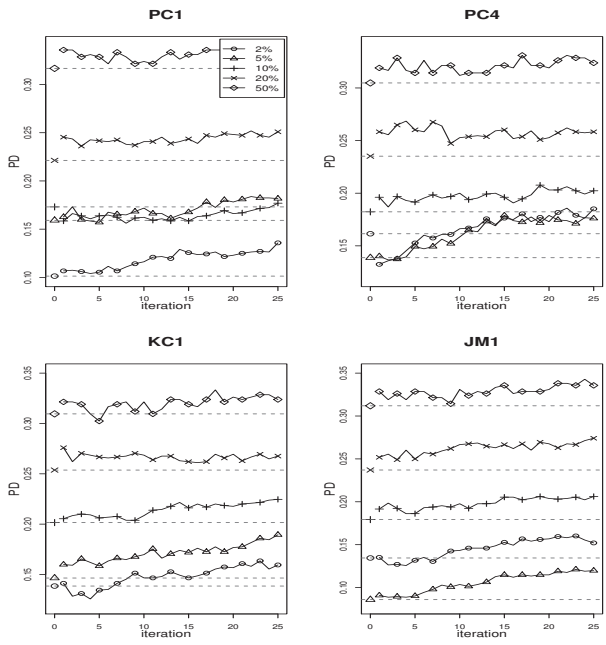

FIGURE 3.8: Results of FTcF algorithm on four data set with threshold=0.5.

Khoshgoftaar [95] on NASA MDP software projects. In their approach an EM-based semi-supervised learning algorithm [26] was implemented. Another interesting approach is to perform semi-supervised clustering [96]. Both of their studies are inductive learning based, that is, a model is built given a set of train data set (labeled and unlabeled data) and then is used to predict on test data set (similar projects). Performance is measured on the test data set. The goal of inductive semi-supervised learning is to find a model for entire data space. In our study, we implemented a transductive semi-supervised learning in which predicted labels of unlabeled data are the major concern (although unlabeled data are used for training as well). Performances are evaluated on the unlabeled data set. We refers the reader to [26] for detailed discussion between inductive and transductive semi-supervised learning. Considering the differences in the evaluation strategy used in inductive learning and transductive learning, it is impractical to compare the semi-supervised learning algorithms used in our study with the ones used in their work. However, we can compare our results with supervised learning algorithms from previous work, as in both cases only the predicted labels for unlabeled data (test data for supervised learning) are interested.

There has been large size of studies in software fault prediction using supervised learning approaches. Consider the consistency and comparability in the view of data implementation and performance evaluation, we are going to compare our results with the work by Menzies [1] and Lessmann [2]. Menzies' results (by Naive bayes algorithm with feature

filtering) has not been beaten since 2007 in which an average pairs ($pd = 71, pf = 25$) of eight data sets from MDP repository were observed. Therefore, it will be interesting to compare the results by our semi-supervised learning algorithms with the best results from Menzies. In table 4.11, we compared both of our semi-supervised algorithms with Menzies' results, in which the PD value at the fixed PF value are presented. There are three data sets (PC1, PC3 PC4) in common between Menzies' work and our study. From the table, the random Forest based learning algorithms (RF, FTF and FTcF) exceed the Naive Bayes algorithm on PC1 and PC3. They have the equivalent performance on PC4. The best one (PC1) starts to beat the results from Menzies at 5% (PD= 0.49 by FTcF). Especially, Menzies evaluated its algorithms by a $9:1$ separation (90% labeled data) while we experimented much lower sizes of labeled data (2% to 50% labeled data).

Later in [2], Lessmann designed a comparative experiment with multiple classification models on the same data repository, where AUC is measured as the performance metrics. In their work, 2/3 of the data are used for training, that is 67% labeled data. In table 4.12, we show the comparison between our work and Lessmann's results (the results by random forests). Our results exhibit insignificant different to their work at a little lower size of labeled data (50% labeled data).

TABLE 3.3: Comparison between our results and Menzies's results with Probability of Detection(PD) at specified PF

Data sets	Size of L	RF	FTF	FTcF	Menzies [1]
pc1	2%	0.45	0.45	0.45	
(PF=0.17)	5%	0.46	0.47	**0.49**	
	10%	**0.53**	**0.54**	**0.55**	
	25%	**0.66**	**0.66**	**0.67**	
	50%	**0.74**	**0.74**	**0.75**	0.48
pc3	2%	0.66	0.70	0.67	
(PF=0.35)	5%	0.71	0.73	0.72	
	10%	0.74	0.75	0.75	
	25%	**0.81**	**0.84**	**0.82**	
	50%	**0.85**	**0.87**	**0.87**	0.8
pc4	2%	0.62	0.63	0.65	
(F=0.29)	5%	0.80	0.80	0.81	
	10%	0.86	0.87	0.88	
	25%	0.94	0.95	0.97	
	50%	**0.98**	**0.98**	**0.98**	0.98

TABLE 3.4: Comparison between our results and Lessmann's results with Area under ROC curve(AUC)

Data sets	Size of L	RF	FTF	FTcF	Lessmann [2]
pc1	2%	0.67	0.67	0.66	
	5%	0.71	0.65	0.66	
	10%	0.77	0.78	0.79	
	25%	0.85	0.87	0.87	
	50%	0.87	0.86	0.86	0.9
pc3	2%	0.64	0.64	0.64	
	5%	0.7	0.71	0.7	
	10%	0.75	0.76	0.75	
	25%	0.82	0.82	0.82	
	50%	0.88	0.88	0.89	0.82
pc4	2%	0.72	0.71	0.73	
	5%	0.82	0.78	0.77	
	10%	0.87	0.88	0.89	
	25%	0.91	0.9	0.9	
	50%	0.93	0.94	0.95	0.97
kc1	2%	0.74	0.73	0.73	
	5%	0.74	0.73	0.73	
	10%	0.76	0.78	0.79	
	25%	0.78	0.79	0.79	
	50%	0.8	0.82	0.82	0.78
jm1	2%	0.64	0.67	0.67	
	5%	0.66	0.68	0.68	
	10%	0.68	0.69	0.7	
	25%	0.7	0.72	0.72	
	50%	0.73	0.74	0.74	0.76

3.3 Conclusion

Semi-supervised approaches have been successful applied in many problems. In this chapter, we investigated two variants of self-training approaches to the problem of software fault-prone prediction. Both approaches are variants of Yarowsky's algorithm with the difference of the way they update the labels for unlabeled data set. We illustrated the comparison between each investigated self-training approaches - FTF and FTcF algorithms - and the corresponding supervised approach.

Base learner as a critical component in our self-training approaches is discussed. To investigate the impact of base learner in both self-training approaches, we selected three widely used supervised learning algorithms - Logistic regression, Support vector machine and Random Forest due to their popularities in software fault prediction problem. We demonstrated that both semi-supervised approaches are sensitive to base learner. We proved that logistic regression as base learner causes both self-training approaches convergent to a supervised learning setting. When SVM is used as base learner, both

FTF and FTcF are convergent. However, their performances are worse than the supervised SVM. In contrast, Random Forest never converges and exhibits outstanding performance. Our results showed that both self-training approaches benefit from using random forest as base learner. In particular, FTcF algorithm exhibits more superiority on small set of initial labeled data than FTF algorithm.

Chapter 4

Semi-Supervised Learning with dimensionality reduction approach

In previous chapter, we demonstrated that FTcF algorithm outperforms FTF algorithm as it is capable of reducing the size of initial labeled data to achieve the better performance than supervised approach. We also learned that when random forest is used as base learner, both FTcF and FTF perform consistently better than the corresponding supervised learning. However, we admit that the improvement is limited and not statistically significant. In this chapter, we continue on improving the prediction performance by augmenting the FTcF algorithm with dimensionality reduction technique.

4.1 Multidimensional Scaling (MDS)

An inherent problem with self-training algorithms is that an existing noise or some wrong information are repeatedly rounded up in each iteration of self-training. This inherent problem also threatens the use of self-training algorithms applied on SE data sets, since SE data sets are likely to contain:

- Noise due to different reasons such as human error or difficulty of measurement and so on;
- As well as highly correlated variables, redundant and/or irrelevant variables.

Our intuition is that the dimensionality reduction techniques are good candidates to address this inherent problem of semi-supervised learning by reducing the complexity

of dimensionality and extracting the essential information without affecting the data structure. In previous chapter, we claimed that the FTcF algorithm performs better than the FTF algorithm, although it does not exhibit any significant outperformance. Therefore, we augment FTcF with a pre-processing strategy - dimensionality reduction based technique called Multidimensional scaling (MDS). The MDS algorithm is used to reduce the dimensionality of the space of independent variables (metrics) before the semi-supervised learning iterations are initiated. Throughout the dissertation, we call this version of self-training approach as FTcF.MDS algorithm.

MDS is one type of dimension reduction, which has been widely used. In [107], multidimensional scaling as a dimension reduction technique is applied to the mapping of computer usage data. In [108, 109], dimension reduction is extended to manifold learning. Typically, MDS is considered for the analysis of proximity data to reveal the hidden structure. The main assumption of MDS is that instances of data can be placed as points in a multidimensional space and the relation between instances is inversely related to the similarities of the corresponding points in the multidimensional space. In machine learning and statistical learning, MDS is used for exploratory data analysis in which instances of data can be placed as points in a low dimensional space so that the observed complexity in the original data is reduced while preserving the essential information.

The proximity metrics in MDS play an important role as they contain the similarity or structural information of the data studied. Usually the proximity metrics can come from distance metrics, similarity metrics, identification confusion metrics, grouping data, etc. In practice, the Euclidian distance metric is often used because of its mathematical convenience. In this paper, we use the similarity metrics obtained from random forest [110].

In random forest, we construct a set of decision trees such that each tree contains a randomly selected subset of the features. Next, instances of data are propagated down the trees and a similarity matrix based on terminal leaf occupancy is calculated for all instances. If two instances land in the same terminal node their similarity increases. More specifically, let $x1$ and $x2$ be two instances in the data. Let T1, T2 be the terminal positions for $x1$ and $x2$, respectively. k is the number of trees in the forest. Then the similarity $S(.)$ between $x1$ and $x2$ is set to:

$$S(x_1, x_2) = \frac{1}{k} \sum_{i=1}^{k} I(T_{1i} == T_{2i}) \tag{4.1}$$

where I(.) is the indicator of closeness of terminal positions; T_{1i} and T_{2i} are the terminal positions of $x1$ and $x2$ in the i^{th} tree.

4.2 Dimensionality Reduction based FTcF algorithm

4.2.1 Notation Definition

Notation defined in previous chapter can be extensively used in here. For dimension-reduced data, we need to define new notations. Again, let X be an $(n+m) \times p$ matrix that denotes software metrics, where n is the size of the labeled set (represented with l) and m is the size of unlabeled set (represented with u), i.e., $x \in \Re^p$. Let Z be an $(n+m) \times q$ matrix that denotes the dimension-reduced data derived from X, where q is the number of principle dimensions in the reduced data representation, i.e., $z \in \Re^q$. Specifically, $Z = \{Z_l, Z_u\}$, where $Z_l = \{z_1, z_2, \cdots, z_n\}$ and $Z_u = \{z_{n+1}, z_{n+2}, \cdots, z_{n+m}\}$. In addition, let $Y = \{Y_l, Y_u\}$ be the response variable (labels) where $Y_l = \{y_1, y_2, \cdots, y_n\}$ are known and $Y_u = \{y_{n+1}, y_{n+2}, \cdots, y_{n+m}\}$ are missing or unknown. The observed labels are binary variables, $y_i \in \{0, 1\}$, where *0* denotes the absence of faults (i.e. *nfp*) and *1* denotes faulty modules (i.e. *fp*).

Recall that $\phi(.)$ is defined as any given supervised learner - base learner. Consider the step of incorporating dimension reduction procedure in the FTcF algorithm, we denote $D_Z = D = (Z, Y)$, the dimension reduced data associating the corresponding fault data. Given a set of input-output pairs $D_l = (Z_l, Y_l)$, the notation $\phi_{D_l}(Z_u)$ indicates that the classifier trained from D_l is used to predict on unlabeled data set Z_u.

4.2.2 Methodology

The pseudo code of augmented FTcF algorithm is defined in Figure 4.1. In the *Pre-processing Step* part of the pseudo code, X and Y_l are the inputs. The *tune.MDS* function is a tuning procedure, searching for the best d. The tuning criteria of d is the evaluation function of generalized cross validation:

$$GCV(d) = \frac{\sum_{i=0}^{n} (y_{i(d)} - \hat{y}_{i(d)})^2}{(1 - \frac{df_{(d)}}{n})^2} \tag{4.2}$$

Here $df_{(d)}$ represents generalized degrees of freedom for Random Forest [111]. The best d takes the value that minimizes the GCV(.), i.e., $d = argmin_d(GCV(d))$. The tuned dimension number (d), together with X, is then used as the parameter of $MDS(.)$ function. The output of the $MDS(.)$ function is the dimension reduced data that is represented with Z.

Pre-processing Step:MDS
1: Input: X, Y_l, d_m
2: $d = tune.MDS(X_l, Y_l)$
3: $Z = MDS(X, d)$
4: Output: Z

SSL Step: FTcF
1: Input: Z, Y_l
2: Initialization: $D_l = (Z_l, Y_l), \mathsf{u} = u$
3: loop until $|u| \to 0$:
4: Fit $\hat{Y}_u = \phi_{D_l}(Z_u)$
5: Take u' confident cases from Z_u
6: Updating: $Z_l = Z_{l+u'}$, $Z_u = Z_{u-u'}$, $Y_l = Y_l + \hat{Y}_{u'}$, and $D_l = (Z_l, Y_l)$
7: End loop
8:Output: $\hat{Y}_{\mathsf{u}}$

FIGURE 4.1: Dimension Reduction based FTcF Algorithm

In the SSL step of the proposed algorithm in Figure 4.1: FTcF algorithm takes the available information Z and Y_l as inputs and starts with preserving the size of the initial unlabeled data ($\mathsf{u} = u$). After initialization, it enters a loop in which a model trained from current labeled data. $\phi_{D_l}(.)$ is used to classify the current unlabeled data Z_u (Step 4). The prediction $\hat{Y}_u$ with high confidence scores are incorporated into the pool of labeled data (Step 5). Both the size of labeled data and unlabeled data are updated at each iteration due to the migration of instances between unlabeled data and labeled data (step 6). As the procedure nears completion, the size of unlabeled data set becomes zero $|u| \to 0$, which means that all instances from unlabeled data set are labeled. The output of the FTcF algorithm is the prediction for the unlabeled data set, that is $\hat{Y}_{\mathsf{u}}$.

4.3 Experiment and Results

4.3.1 Experimental Setting

To understand the impact of the dimension reduction technique (MDS) to semi-supervised learning, we will compare the performance through the following fault prediction experiments: a) Supervised learning without MDS (acronym SL), b) Supervised learning with MDS (SL.MDS); c) Semi-supervised learning without MDS (FTcF); d) Semi-supervised learning with MDS (FTcF.MDS). Moreover, we explore multiple sizes of labeled data to demonstrate one more benefit of semi supervised fault prediction: the ability to learn from smaller sets of labeled data. The size of the labeled data set will range between 2% and 50% of the overall number of modules in the project. More specifically, we will train

from randomly selected module subsets that contain 2%,5%,10%,25%,50% of modules in the project. The same projects from NASA MDP repository in Chapter 2 will be used for performance evaluation, except the project JM1. Table 3.2 shows the numbers of modules used in initially labeled data set for each project.

Again, the Area Under ROC Curve (AUC) and the Probability of Detection (PD) with different thresholds (i.e., $\tau = \{0.1, 0.5, 0.75\}$) are computed as performance measures. Ten independent runs were performed for each of the settings and reported results represent the average classification outcomes. Considering the consistently outstanding performance of random forest in our previous work (chapter 2), we will remain it as base learner in this study.

4.3.2 Results

Figure 4.2 depicts the performance of the four fault prediction models on the PC4 project data. The four graphs depict the AUC, and the true positive rate (PD) at three different thresholds: 0.1, 0.5 and 0.75. The X axes represent the size of the labeled data used for model training. We observe that the FTcF algorithm with MDS (FTcF.MDS) has the best overall performance across all performance measures. It demonstrates a dramatic improvement over other algorithms especially for the probability of detection at lower size settings of labeled data. For example, when threshold is 0.1, random forest as a supervised algorithm detects about 67% of fault prone modules correctly when learning from 2% of data that has labels. The semi-supervised algorithm with dimension reduction, which uses the same random forest algorithm as the base learner, returns over 91% correctly detected fault prone modules.

The second ranked classification approach is the supervised learning with MDS (SL.MDS), which exceeds the results of FTcF and SL for most of the measures. Obviously, both FTcF.MDS and SL.MDS have significant advantage compared to the same learning algorithms with out dimensionality reduction. Meanwhile, FTcF is better than SL at PD with threshold 0.1 and 0.5, while they are very similar at threshold 0.75 and in terms of AUC. This figure provides us with an indication that the dimension reduction and semi-supervised learning algorithm provide strong benefits for detecting fault proneness modules in the software project, PC4. In addition, the dimension reduction technique improves all learning algorithms. SL.MDS and FTcF.MDS outperform the corresponding supervised learning algorithm (SL) and the original semi-supervised learning (FTcF).

The results of the other data sets (KC1, PC1, PC3) are similar to the PC4. In Tables 4.1 through 4.4, we summarize the performance of fault prediction models over all four data sets and performance measures. The best performance point for each training size

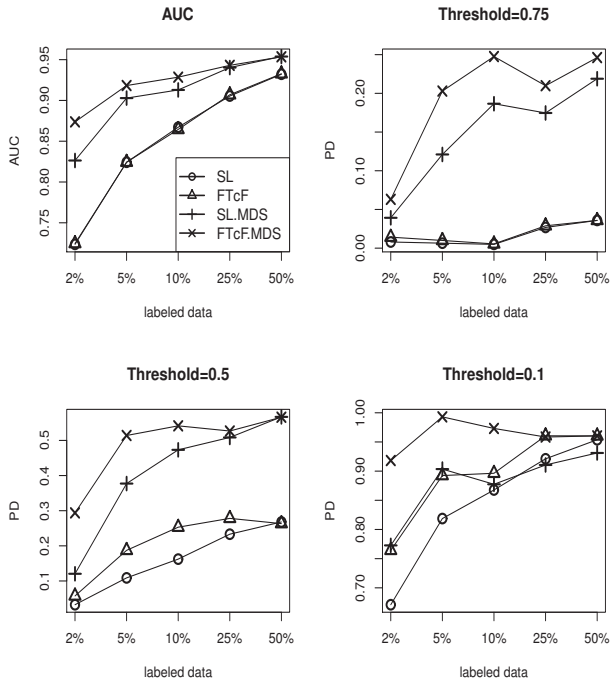

FIGURE 4.2: Performance plots for PC4 project.

setting (the size of the labeled data set) is highlighted in a bold font. Clearly, most of the highlighted values are consistently located in the column for FTcF.MDS algorithm. It is also not hard to observe that the supervised learning with MDS (SL.MDS) has better performance then SL and FTcF. By itself, FTcF offers only a slight improvement over SL for the probability of detection measure while their behaviors measured by the AUC are almost the identical.

4.3.3 Statistical Analysis

To test statistical significance, we conducted one way ANOVA test for the experimental outcomes reported above. The ANOVA test examines whether the level of the differences in algorithm performance is significant. The hypotheses of the test are:

H_o: There is no difference among the four algorithms across the data sets used;
H_a: The performance of at least one the algorithms is significantly different (better) than the others.

An example of one way ANOVA test for the PD at threshold 0.5 with 2% labeled data is given in Table 4.5. Classification results between different algorithms to the variability

TABLE 4.1: AUC for the four data sets

Data	size of L	SL	FTcF	SL.MDS	FTcF.MDS
PC1	2%	0.6733	0.6677	0.8379	**0.8536**
	5%	0.7122	0.7087	0.8719	**0.8889**
	10%	0.7721	0.7806	0.9166	**0.9253**
	25%	0.8484	0.8464	0.9353	**0.9356**
	50%	0.8687	0.8728	0.9425	**0.9434**
PC3	2%	0.7053	0.7096	0.7550	**0.7841**
	5%	0.7386	0.7355	0.8494	**0.8860**
	10%	0.7512	0.7573	0.8829	**0.9024**
	25%	0.7922	0.7981	0.9103	**0.9183**
	50%	0.8199	0.8246	**0.9267**	0.9260
PC4	2%	0.7235	0.7246	0.8264	**0.8737**
	5%	0.8242	0.8243	0.9029	**0.9183**
	10%	0.8672	0.8644	0.9129	**0.9285**
	25%	0.9054	0.9069	0.9403	**0.9430**
	50%	0.9321	0.9327	**0.9538**	0.9535
KC1	2%	0.7374	0.7295	0.6793	**0.7382**
	5%	0.7404	0.7476	0.7437	**0.7477**
	10%	0.7635	0.7693	0.7728	**0.7831**
	25%	0.7794	0.7897	0.**7938**	0.7850
	50%	0.8043	0.8108	**0.8134**	0.8030

observed within the outcomes of experiments that use only one algorithm. A large value of F indicates that the outcomes of different algorithms vary more that the outcomes of the single algorithm. The P-value is a probability of observing a test statistic as extreme as the one actually observed. The smaller the P-value, the more strongly the test rejects the null hypothesis. Choosing the significance criteria (α) of 0.05, we can conclude that the differences in observed classification performance between SL, SL.MDS, FTcF, and FTcF.MDS are significant as the p-value 0.000517 is much smaller than α.

Overall results of ANOVA test on all size settings of labeled data are presented in Table 4.6. Since p-value measures how much evidence we have against the null hypothesis (*H0*), reporting p-values is sufficient. In our case, a p-value smaller than 0.05 indicates that there is statistically significant difference among the algorithms. In the table, we highlighted the significant outcomes. Only in cases when the size of the labeled data is 2% the AUC results significantly different. While we cannot argue that AUCs of different modeling approaches are significantly different (across all threshold settings), for the probability of detection the story is different. For thresholds 0.75 and 0.5, at least one of the proposed approaches significantly outperforms in the ability to correctly detect fault prone modules across all labeled data size settings. For threshold 0.1, performance of models significantly differs when we have less than 25% of modules with known fault content (labeled).

TABLE 4.2: PD with threshold=0.75 for the four data sets

Data	size of L	SL	FTcF	SL.MDS	FTcF.MDS
PC1	2%	0.0000	0.0014	0.0514	**0.1365**
	5%	0.0014	0.0070	0.0408	**0.0789**
	10%	0.0185	0.0185	0.1400	**0.1769**
	25%	0.0612	0.0612	0.2306	**0.2327**
	50%	0.1190	0.1190	0.2571	0.2571
PC3	2%	0.0006	0.0019	0.1287	**0.1994**
	5%	0.0026	0.0026	0.1131	**0.1595**
	10%	0.0021	0.0021	0.1634	**0.1800**
	25%	0.0058	0.0058	0.2760	**0.3050**
	50%	0.0061	0.0061	**0.2439**	0.2427
PC4	2%	0.0080	0.0142	0.0392	**0.0631**
	5%	0.0064	0.0099	0.1211	**0.2029**
	10%	0.0049	0.0055	0.1866	**0.2476**
	25%	0.0268	0.0289	0.1746	**0.2099**
	50%	0.0358	0.0358	0.2189	**0.2462**
KC1	2%	0.0081	0.0246	0.0558	**0.1533**
	5%	0.0181	0.0343	0.0917	**0.1673**
	10%	0.0263	0.0359	0.1145	**0.1602**
	25%	0.0121	0.0125	0.1484	**0.1747**
	50%	0.0214	0.0223	0.1195	**0.1286**

Next, we conducted the post-hoc test to determine which algorithms differ from each other. For this question, we use Tukey's Honestly Significant Difference (HSD) [112]. For AUC, we did not obtain significant difference except the 2% labeled data setting. Therefore, we will concentrate on the probability of detection with different thresholds. The rates of detection of fault prone modules with threshold 0.75 are very low and likely not interesting for software quality engineers. Our pairwise comparison will, therefore, only consider PD with thresholds 0.5 and 0.1. Tables 4.7 and 4.8 show the Tukey's HSD pairwise comparison among discussed algorithms. If in the intersection between the two modeling approaches indicates result "none" this means that no mater the size of the labeled data, there are no significant prediction performance differences. The result *"all"* has the inverse meaning.

For example, table 4.7 indicates that FTcF.MDS significantly outperforms supervised Random Forest (SL) and FTcF for all size settings of labeled data. It also "wins over" SL.MDS for the lower sizes of labeled data (2% and 5%). SL.MDS beats SL for all labeled data size settings and beats FTcF for all labeled data size setting except the 2%. Table 4.8 makes similar comparisons for threshold 0.1. We infer FTcF.MDS still consistently outperform SL, FTcF, and SL.MDS at the lowest labeled data size settings. The dimension reduction based semi-supervised algorithm (FTcF.MDS), therefore, offers significant advantages when very few project modules have known fault content.

TABLE 4.3: PD with threshold=0.5 for the four data sets

Data	size of L	SL	FTcF	SL.MDS	FTcF.MDS
PC1	2%	0.1027	0.1365	0.3351	**0.4797**
	5%	0.1169	0.1690	0.3141	**0.4577**
	10%	0.1508	0.1892	0.4554	**0.5031**
	25%	0.2694	0.3041	0.4939	**0.4898**
	50%	0.3476	0.3429	0.5143	0.5143
PC3	2%	0.0541	0.1510	0.2459	**0.3930**
	5%	0.0895	0.1458	0.3458	**0.4072**
	10%	0.0828	0.1103	0.3690	**0.4193**
	25%	0.0934	0.1174	0.4636	**0.4793**
	50%	0.1268	0.1390	**0.4854**	0.4817
PC4	2%	0.0324	0.0574	0.1205	**0.2938**
	5%	0.1088	0.1871	0.3772	**0.5140**
	10%	0.1622	0.2530	0.4732	**0.5415**
	25%	0.2331	0.2782	0.5085	**0.5268**
	50%	0.2679	0.2632	**0.5670**	0.5660
KC1	2%	0.0710	0.1583	0.1695	**0.2732**
	5%	0.1022	0.1597	0.2384	**0.3603**
	10%	0.1434	0.1970	0.3293	**0.4303**
	25%	0.1348	0.1546	0.3300	**0.3718**
	50%	0.1636	0.1695	0.3159	**0.3186**

TABLE 4.4: PD with threshold=0.1 for the four data sets

Data	size of L	SL	FTcF	SL.MDS	FTcF.MDS
PC1	2%	0.6365	0.7108	0.8027	**0.8770**
	5%	0.6662	0.7394	0.8648	**0.9592**
	10%	0.7138	0.8092	0.8800	**0.9631**
	25%	0.8204	0.8571	0.8796	**0.9449**
	50%	0.8476	0.8667	0.9095	**0.9238**
PC3	2 %	0.6395	0.7758	0.7248	**0.8771**
	5%	0.6693	0.7497	0.8196	**0.9725**
	10 %	0.6910	0.7903	0.8366	**0.9538**
	25%	0.7851	0.8587	0.8579	**0.9240**
	50%	0.8085	0.8622	0.8780	**0.9098**
PC4	2 %	0.6710	0.7642	0.7727	**0.9182**
	5%	0.8187	0.8924	0.9035	**0.9930**
	10%	0.8677	0.8963	0.8774	**0.9732**
	25%	0.9211	0.9606	0.9106	**0.9585**
	50%	0.9538	0.9604	0.9311	**0.9604**
KC1	2%	0.5489	0.7249	0.5938	**0.7969**
	5%	0.6067	0.6876	0.7130	**0.8190**
	10%	0.6773	0.7461	0.7421	**0.8043**
	25%	0.6773	0.7473	0.7260	**0.7623**
	50%	0.7368	0.7695	0.7505	**0.7618**

Table 4.5: One way ANOVA test for PD(threshold= 0.5) at 2% labeled data

	d.f	Sum Sq	Mean Sq	F value	p-value
algorithm	3	0.197433	0.065811	12.569	**0.000517**
Residuals	12	0.062832	0.005236		

Table 4.6: P-value of ANOVA test on varied size of labeled data for all performance measures

size of L	AUC	PD(0.75)	PD(0.5)	PD(0.1)
2%	**0.037951**	**0.000541**	**0.000517**	**0.001004**
5%	0.056876	**0.000105**	**1.09E-06**	**0.010114**
10%	0.08185	**5.72E-07**	**3.97E-06**	**0.02952**
25%	0.338097	**1.44E-05**	**0.000332**	0.531505
50%	0.491754	**0.000623**	**0.004326**	0.853906

Table 4.7: Significance comparison of PD(0.5)

	SL	FTcF	SL.MDS
FTcF	none	–	–
SL.MDS	all	5%,10%,25%,50%	–
FTcF.MDS	all	all	2%, %5

Table 4.8: Significance comparison of PD(0.1)

	SL	FTcF	SL.MDS
FTcF	none	–	–
SL.MDS	none	none	–
FTcF.MDS	2%, 5%, 10%	2%	2%

The algorithms that have MDS embedded (FTcF.MDS and SL.MDS) outperform the corresponding supervised / semi supervised counterparts (SL, FTcF) for some of the size settings. Both tables indicate that just changing the training method from supervised to semi-supervised, both using random forest at the core, does not offer significant differences.

4.3.4 Robustness to Noise

On software fault identification, it is nature to contain noise in the labels due to many practice issue such as mislabeling by software developers, or fault information reported by untrustworthy part. This type of noise can dramatically affects the results of classification. It is then an issue that software developers should consider when selecting the classification strategies. Robustness test is an effective technique to characterize the behavior of an algorithm in the presence of mislabeling condition.

To investigate the robustness of our semi-supervised approach with the presence of noisy response, we randomly selected partial of the modules from each data set and permuted

the response. The noisy rate we explored is $2\%, 5\%$, and 10%. Similar to previous experiments, varied size of labeled data are sampled ($2\%, 5\%, 10\%, 20\%$,and50%) in each of the four data sets. For this experiment, we are not interested in the performance of our semi-supervised approaches on noisy data directly, because the performance will of course be worse than original data (assume the original data are pure). Instead, we are interested in the stabilization of our semi-supervised approach on noisy data.

Table 4.9 and 4.10 show the average decrease on the measure of AUC and PD (threshold is 0.5) respectively when noisy response appears in four data sets. The positive value implies the decrease of performance when noise appears and the negative value implies the increase of performance. From the table, the changes on performance caused by noise response are fairly small for both measures across all size setting of labeled data. This implies that the dimension reduction based semi-supervised learning algorithm is stable to noise..

Table 4.9: Average decrease in AUC measure.

Rate of Noise	data set	2%	5%	10%	25%	50%
2%	pc1	0.088	0.041	0.034	0.016	0.015
	pc3	-0.04	0.022	0.018	0.012	0.006
	pc4	0.029	0.024	0.046	0.034	0.019
	kc1	0.009	0.025	0.025	0.029	0
5%	pc1	0.112	0.104	0.072	0.058	0.031
	pc3	-0.022	0.053	0.046	0.051	0.039
	pc4	0.12	0.056	0.069	0.053	0.044
	kc1	0.024	0.014	0.072	0.059	0.066
10%	pc1	0.205	0.159	0.132	0.077	0.039
	pc3	0.042	0.073	0.068	0.077	0.041
	pc4	0.094	0.102	0.086	0.069	0.052
	kc1	0.05	0.098	0.098	0.096	0.09

Table 4.10: Average percent decrease in PD measure (Threshold is 0.5).

Rate of Noise	data set	2%	5%	10%	25%	50%
2%	pc1	0.224	0.197	0.043	0.079	0.094
	pc3	-0.035	0.005	0.009	0.066	0.034
	pc4	0.061	0.038	0.139	0.073	0.094
	kc1	-0.191	0.014	0.017	0.023	-0.002
5%	pc1	0.222	0.204	0.136	0.2	0.168
	pc3	0.026	0.084	0.035	0.077	0.096
	pc4	0.085	0.172	0.178	0.091	0.148
	kc1	-0.083	-0.002	0.063	0.06	0.029
10%	pc1	0.354	0.319	0.309	0.177	0.15
	pc3	0.18	-0.005	0.028	0.139	0.105
	pc4	0.036	0.099	0.178	0.202	0.203
	kc1	-0.064	0.003	0.113	0.095	0.046

4.3.5 Discussion

To study the outperformance of the augmented FTcF algorithm, we repeated the comparison of semi-supervised learning with Menzies and Lessmann's studies that we conducted in previous chapter. Table 4.11 compares the results by providing the value of the probability of correctly detecting (unlabeled) fault prone modules, PD, at fixed probability of false detection, PF. We set the values of PF to 0.17, 0.35 and 0.29 for PC1, PC3, and PC4, respectively, to match the performance reported by Menzies. For PC1 project, Random Forest (SL) starts to beat Menzies' results at 10% labeled data and FTcF.MDS algorithm starts to beat the same result when only 5% of modules have labels. Both SL and FTcF.MDS exceed Menzies' result with 25% labeled data for PC3 and at 50% labeled data for PC4.

In Table 4.12, we compare FTcF.MDS and Lessmann's results, both using Random Forest algorithm. Since Random Forest exhibited the best performance in Lessmann's study (although not significantly better than other classifiers), the comparison of these results is fair. The performance of FTcF.MDS exceeds Lessmann's Random Forest algorithm when only 10% modules have labels in PC1 and KC1, and when 5% modules have labels in PC3. We could not exceed the performance reported by Lessmann on PC4, but we did not try to use 67% labeled data for training either.

TABLE 4.11: Comparison of results with [1]

Data sets	Size of L	SL	FTcF.MDS	Menzies [1]
PC1	2%	0.45	0.73	
(PF=0.17)	5%	0.46	**0.80**	
	10%	**0.53**	**0.85**	
	25%	**0.66**	**0.88**	
	50%	**0.74**	**0.91**	0.48
PC3	2%	0.66	0.77	
(PF=0.35)	5%	0.73	**0.90**	
	10%	0.74	**0.92**	
	25%	**0.81**	**0.94**	
	50%	**0.85**	**0.95**	0.8
PC4	2%	0.62	0.89	
(F=0.29)	5%	0.79	0.81	
	10%	0.86	0.97	
	25%	0.94	0.98	
	50%	**0.98**	**0.99**	0.98

TABLE 4.12: Comparison of results with [2] using AUC

Data sets	Size of L	SL	FTcF.MDS	Lessmann [2]
PC1	2%	0.67	0.85	
	5%	0.71	0.89	
	10%	0.77	**0.93**	
	25%	0.85	**0.94**	
	50%	0.87	**0.94**	0.9
PC3	2%	0.71	0.78	
	5%	0.74	**0.88**	
	10%	0.75	**0.90**	
	25%	0.79	**0.91**	
	50%	0.82	**0.93**	0.82
PC4	2%	0.72	0.87	
	5%	0.82	0.92	
	10%	0.87	0.93	
	25%	0.91	0.94	
	50%	0.93	0.95	0.97
KC1	2%	0.74	0.74	
	5%	0.74	0.74	
	10%	0.76	**0.78**	
	25%	**0.78**	**0.79**	
	50%	**0.80**	**0.80**	0.78

4.4 Conclusion

In Chapter 3 we demonstrated that self-training typically improves the corresponding supervised learning, when both use the same learning algorithm, in our case random forest. Nevertheless, prediction performance improvement is not significant. In this chapter, we then added a pre-processing strategy, MDS, to the semi-supervised learning algorithm, FTcF, and obtained statistically significant improvements. Statistically significant performance improvements are rarely seen in fault prediction modeling. A combination of semi-supervised learning and the dimension reduction technique provides important benefits to software quality prediction. The robustness test is examined to test the stabilization of our approach to noisy response when Random Forest is used as base learner. Our results showed that the dimension reduction FTcF algorithm exhibits stable performance.

The very good performance of FTcF.MDS at the 2% and 5% labeled data sizes at threshold 0.1 are of particular interest to software engineers. To our knowledge, no one so far reported success in developing fault prediction models that offer reasonable performance with such an "extremely" small number of available modules with known fault content. This is the situation in which semi supervised learning approach shines because it incorporates unlabeled data in the learning process. Our results indicate that

the metrics extracted from modules with unknown fault content can compensate for the shortcoming of supervised learning. However, this advantage, while still present, is not as significant when a larger portion of software modules is labeled.

This study, we believe, indicates that empirical software engineering needs to move from the use of generic off-the-shelf machine learning algorithms towards the ones that take into account the specificity of the domain in which we work. We also believe that dimension reduction technique developed in this chapter looks promising as a preprocessing strategy for many software prediction problems with large dimensionality of independent (predictor) variables.

Chapter 5

Active Learning in SFP problem

In previous chapters, we've investigated the semi-supervised learning approaches in software fault prediction problem and demonstrated that semi-supervised learning, such as self-training approaches, augmented with appropriate pre-processing technique can outperform the corresponding supervised learning when the same base learner is used. As we proposed in Introduction chapter, an alternative to semi-supervised learning is active learning, which has similar learning procedure but differs from the way they select candidates to label. Typically, semi-supervised learning approach boosts the accuracy of a classifier, which learns from a few labeled instances selected according to some confidence criteria. Active learning selects unlabeled instances according to their informativeness. On the other hand, active learning requires labels from oracle interactively, while semi-supervised learning requires no human efforts. In this chapter, we aim to investigate an adaptive software fault prediction model. The core of the adaptive approach is active learning.

5.1 Active Learning

Active learning, in statistics literature also called optimal experimental design or query learning, is a class of strategies to choose the data from which to "learn", in our case, a fault prediction model. In principle, good prediction performance can be achieved by using only "essential data", that is, use only the selected data for training. This characteristic of active learning is desirable in situations when labeled data items are not abundantly available [113]. Traditional supervised learning approach is a two-step procedure in which a learner is first trained using labeled data and the model then predicts on unlabeled data. Unlike supervised learning, active learning requires interactions

Active Learning with uncertainty sampling

Input :
Labeled instances X_l with fault content Y_l;
Unlabeled instances X_u;
A base learner C

loop :
1: Train C using current labeled data X_l, Y_l;
2: Use C to predict unlabeled instances X_u;
3: Calculate the uncertainty score for each module in X_u;
4: Select u' most uncertainty instances;
5: Obtain labels for selected instances;
6: Updating: $X_l = X_{l+u'}$, $X_u = X_{u-u'}$, and $Y_l = Y_{l+u'}$;

End when stop criteria is met;
Output : The learnr C;

FIGURE 5.1: Active learningn process

between a learner and an oracle simultaneously. A typical active learning approach begins with a small labeled set. Base learner(s) is/are then trained using the small labeled data set, such that labels within the considered unlabeled data set can be predicted. Next, the most "informative" instances are carefully selected and sent to the oracle to check the correctness of the prediction of their labels. After the labels are activated (confirmed), these selected instances will be incorporated into the pool of labeled data. In the next round, the learner(s) will be trained using data in the currently labeled data pool. The cycle repeats until a stopping criterion is met.

In a general framework, active learning can be referred to as selective sampling. There are many active learning strategies proposed with respect to a sampling method. The most popular ones are uncertainty sampling, query-by-committee, error reduction, and density-weighted methods [31]. Of particular note is uncertainty sampling [114–117], the most widely used in machine learning literature. The motivation behind uncertainty sampling is finding unlabeled instances that contain most uncertainty, and use them to clarify the decision boundary. This approach is straightforward for probabilistic learning models. For example, when using a probabilistic model for a binary classification problem, such as Naive Bayes, the instances with most uncertainty are those with posterior probability closest to 0.5 – typical decision cutoff for binary classification with balanced class sizes. Fiture 5.1 provides the process of active learning approach with uncertainty sampling technique. C refers to a learner.

However, two issues can hamper uncertainty sampling: outliers and imbalanced classes.

Usually, outliers in data have high uncertainty, but cannot provide much help for classification. In active learning, outliers raise the risk of introducing wrong predictive information and failing to learn. On the other hand, when classes are of dramatically different size, the selection of cutoffs, such as 0.5 for binary classification problems, is indeterminate. This observation holds whenever the prior knowledge about the data is not available. In our prior experiments, uncertainty sampling with mid-range cutoffs (e.g. 0.5) does not show any benefits in software fault prediction, since the number of not-fault-prone modules is significantly larger than the number of fault-prone ones.

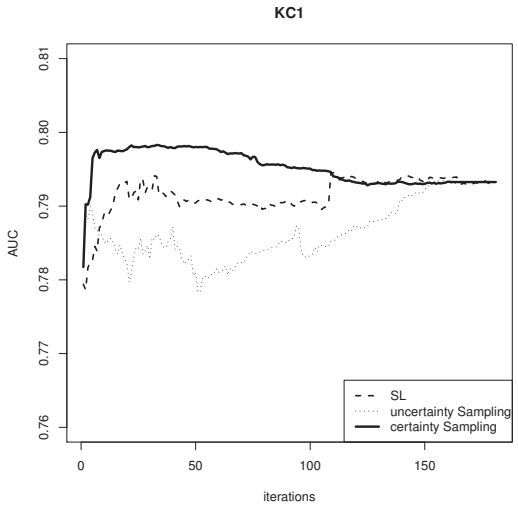

FIGURE 5.2: Comparison of two active learning sampling strategies with supervised learning approach. 10-cross-validation is used to evaluate the prediction performance of trained models at each iteration.

Instead of focusing on "uncertainty" at each adaptive round, we will be more concerned with "representativeness". More specifically, we will try to clarify the position of the decision boundary between prediction classes using the most representative modules from each pool, fault-prone and not-fault-prone. This borrows the basic idea from self-training. Self-training and active learning approaches follow similar iterative procedures. In the self-training approach, the instances with the highest confidence scores are selected from unlabeled data set and then incorporated into the labeled data set. The confidence score is a probabilistic prediction of a base learner wrapped in self-training procedure on the particular unlabeled data item. The underlying assumption of self-training is that the most informative instances are the ones with the highest confidence of prediction. Iteratively augmenting labeled data set using the most informative instances guides a prediction model towards gradual improvement. In our study, the active learning mimics the same assumption about software data sets. The main difference between active learning and self-training approach is that the former needs oracle to confirm the

labels of selected modules, thus providing an additional level of software verification checks, while the latter uses the predicted labels without an explicit confirmation.

To justify the choice and demonstrate the advantages of certainty sampling, we set up an experiment to compare certainty sampling and uncertainty sampling on software fault prediction data. 10-way cross validation procedure is utilized for evaluation, in which 9 folds are used as training data and the instances in the last fold are used for testing. For the training data, we separate modules into two pools - labeled data pool and unlabeled data pool. Active learning proceeds until unlabeled pool is empty. We tracked the performance of the trained model at each active learning iteration. The performance of two active learning strategies (certainty vs. uncertainty sampling) is depicted in Figure 5.2. The details of the experiment will be clarified in later section. We use this experiment to better motivate the work that follows. In comparison to random sampling of training instances with Naive Bayes (marked SL for Supervised Learning in the Figure), we observe that uncertainty sampling performs worse while certainty sampling performs better. The finding is reasonably consistent across all the projects we analyzed.

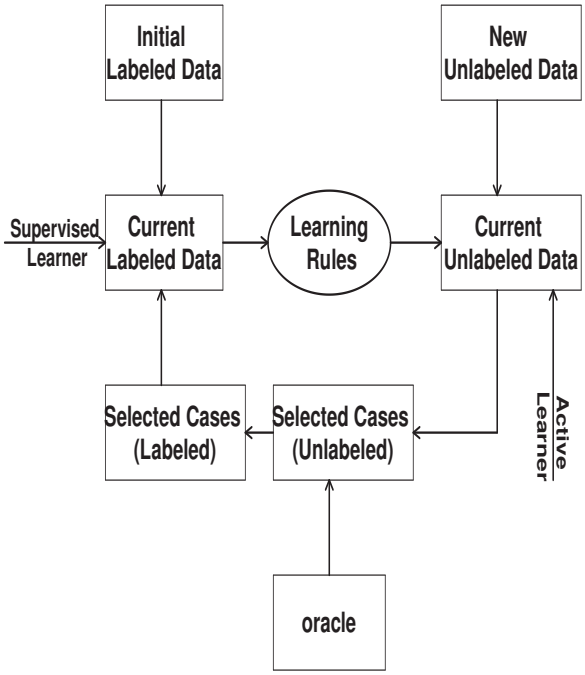

FIGURE 5.3: Diagram of the Adaptive Fault Prediction process.

5.2 Active learning based Software Fault Prediction Model

In this section, we describe the Adaptive Fault Prediction approach - AFP. The idea of utilizing an adaptive prediction mechanism in software engineering is not new. A few adaptive failure detection mechanisms have been proposed to detect quality degradations in computer networks and various other applications [118–121]. However, most of these studies focus on on-line prediction and real-time processing. To the best of our knowledge, there are no studies of adaptive learning in software fault prediction using complexity metrics. In particular, this seems to be the first attempt to deal with the prediction of fault prone modules in emerging software projects, those with no history or previous releases. In these projects, only the modules developed and verified early in the development can be used to predict fault content of those developed later.

Figure 5.3 shows the procedure of the proposed classification approach. In the beginning, a small set of modules needs to be labeled (Initial Labeled Data). These modules would be the thoroughly inspected and / or unit tested, allowing the developers to assign the faulty or not faulty labels. These instances create the initial pool of currently labeled data. At each adaptive iteration, the unlabeled data set is formed from the software modules developed up until that point in time. They form the pool of currently unlabeled data. In the adaptive loop, a supervised learner trains from currently labeled modules and the learned rules are then applied to predict fault content in currently unlabeled modules.

In this process, modules in the pool of currently unlabeled data receive confidence scores, i.e., probabilistic predictions. An active learner then selects the modules with respect to the confidence score which, it believes, are the most informative or the most likely to be correct. The selected modules, marked in Figure 5.3 as *Selected Cases (Unlabeled)*, undergo a detailed labeling procedure by an expert (oracle). The expert ensures that the modules are labeled correctly. These selected cases are then removed from the pool of currently unlabeled data and incorporated into the pool of currently labeled modules, together with those that had their labels assigned at the very beginning. The modules not selected in the current round stay in the pool of currently unlabeled data. This cycle repeats until no additional, new unlabeled data are available or a stop criterion is met.

In our experiments, we know the ground truth about the fault content of all software modules from the NASA MDP data repository. Therefore, in our experiments, this information replaces the oracle. In actual application of the proposed approach, the modules selected for checking of label values would be passed to thorough inspections. Since we know the labels of all the modules in the unlabeled pool from the same source too, these will be used to evaluate prediction performance.

Two types of learners, a supervised learner and active learner, are coupled together in the adaptive procedure. First, the supervised learner triggers the adaptive cycle by providing a prediction model on initially labeled data. This learning algorithm is then wrapped into the adaptive loop and repeatedly retrains from the updated labeled data set. Each time, a new prediction model emerges. The supervised learner can be any supervised learning algorithm. In this study, we use Naive Bayes, one of the most popular learning algorithms. The studies of software fault prediction [1, 2] recommend Naive Bayes due to its computational efficiency, simplicity and the ability to sum up the information from multiple attributes. Computational efficiency is important, as the learning algorithm will be invoked within the loop, repeatedly. This being the first experiment with adaptive fault prediction, we want to preserve simplicity in the core learning algorithm too, and concentrate on the utility of the proposed adaptive learning framework. The predictions created by Naive Bayes are the posterior probabilities ($Pr(y = fp|x)$), i.e, the scores of certainty.

The other learning mechanism essential in our approach is the active learning with its certainty sampling strategy. It plays a role in selecting of unlabeled modules whose newly acquired or confirmed labels provide the "best" guidance for the separation of classes in binary classification. In our experiments, the modules with the highest score in each adaptive iteration are considered good class representatives and thus selected.

Table 5.1: NASA software metrics

Data	**Modules**	**% Faulty**	**features**	**Project description**
KC1	2109	13.9%	22	Storage management for ground data
PC3	1563	10.43%	41	Flight software for earth orbiting satellite
PC4	1458	12.24%	41	Flight software for earth orbiting satellite
PC1	1109	6.59%	41	Flight software from an earth orbiting satellite
CM1	505	16.04%	41	Spacecraft instrument
KC3	458	6.3%	41	Storage management for ground data

Table 5.2: Number of modules in initially labeled data set

	5%	**10%**	**20%**	**50%**
KC1	105	211	421	1054
PC3	78	156	313	782
PC4	73	146	292	729
PC1	55	111	221	554
CM1	25	50	101	252
KC3	23	46	92	229

5.3 Experiments and Results

5.3.1 Experimental Setting

Our goal is to address the difficulties raised by traditional supervised learning approaches, mentioned in Introduction chapter. Thus, following assumptions are the starting point in our work:

- Only a modest number of labeled modules are initially available;
- Cost of human intervention (the V&V activities for modules selected in the iterations of adaptive procedure) to support the algorithm is high and, therefore, can be afforded at a limited scale.

The latter can be achieved by limiting the number of modules to which V&V activities that check the validity of presumed labels (the oracle function) are applied. If we expand the labeled data set with a very few modules in each iteration, verification activities should have an acceptable cost.

In this study, we used six projects from NASA MDP repository for our experiments (Table 5.1). We start the Adaptive Fault Prediction (AFP) process with the proportions of initially labeled modules being 5%, 10%, 20% and 50% of all the project modules. Table 5.2 shows the numbers of modules in each project at the setting of 5%, 10%, 20% and 50%. We are particularly interested in the small sizes of initially labeled pools, 5% or 10% of project modules. Going through the iterations of AFP, we will typically run the next iteration by including u unlabeled modules that have not been considered in the previous iteration. In other words, our experiments simulate fault prediction iterations whenever u new modules are released by the development team. For KC1, PC3, PC4 and PC1, the number of new unlabeled modules added in each iteration of AFP, i.e., u, is set to 100; for CM1 and KC3, we set the number to 50.

At the end of each iteration, for all data set, the algorithm selects only 5 modules for oracle assessment. For example, PC3 data set has $1,563$ modules. Excluding the 5% initially labeled modules (78), $1,485$ are left unlabeled for prediction, enough for 15 adaptive iterations. Therefore, selecting 5 modules in each iteration adds $15 * 5 = 75$ modules into the labeled data set. Together, only $78 + 75 = 153$ modules will ever be exposed to verification activities to determine their fault content. That is less than 10% of project's modules that need to have ground truth established, a far smaller number than any approach in the research literature using supervised learning. To evaluate the classification performance of our approach, we use Area Under Receiver Operating Characteristic Curve (AUC) as the goodness measure.

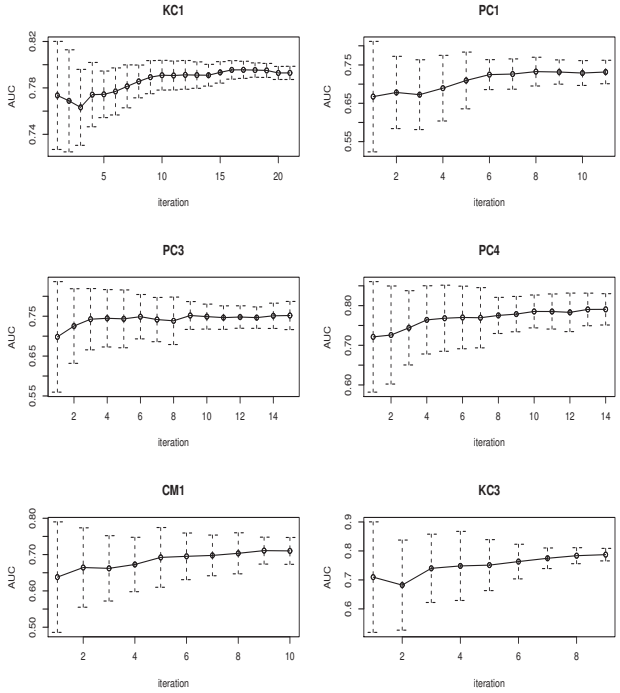

FIGURE 5.4: Performance of AFP for 5% of initially labeled modules.

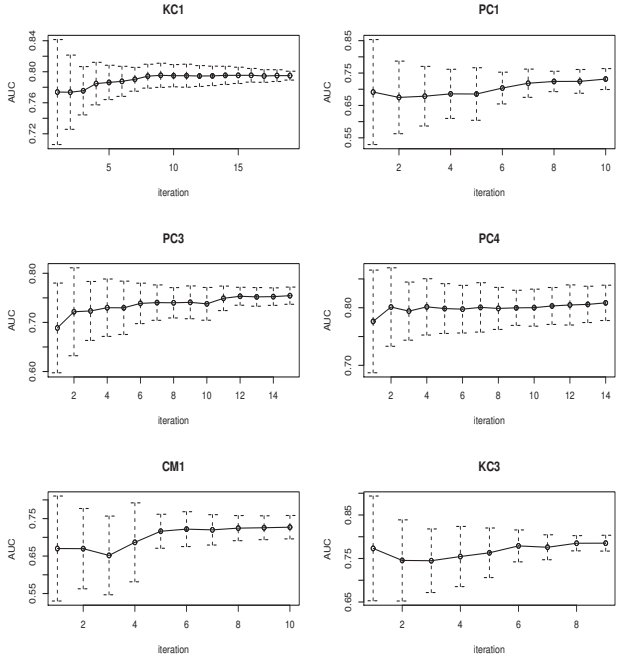

FIGURE 5.5: Performance of AFP for 10% of initially labeled modules.

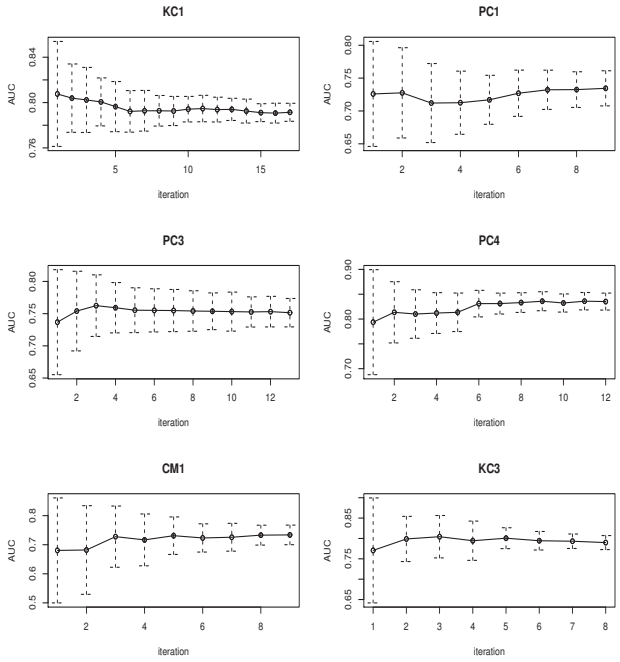

FIGURE 5.6: Performance of AFP for 20% of initially labeled modules.

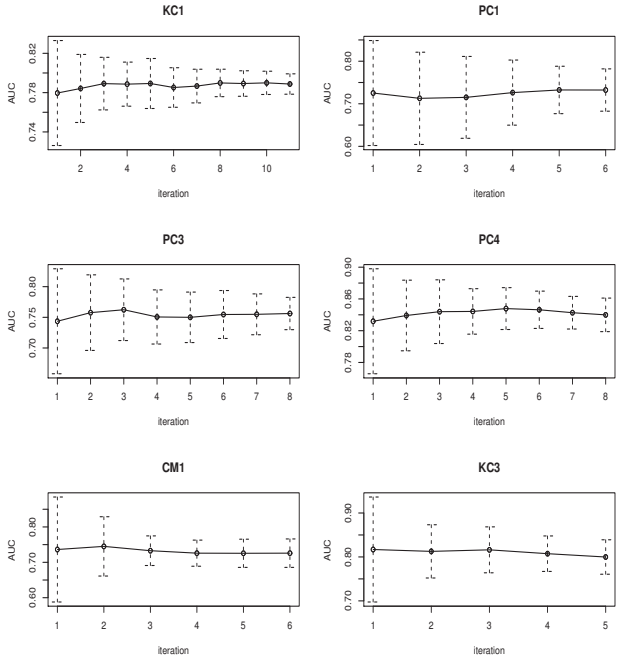

FIGURE 5.7: Performance of AFP for 50% of initially labeled modules.

5.3.2 Results

Figures 5.4 to 5.7 depict classification performance of our approach with the four relative sizes of initially labeled modules: 5%, 10%, 20% and 50%. Points in the figures represent the performance of the current model on the unlabeled instances in each adaptive iteration. Each point is an average of 20 runs, allowing us to express variance. In the figures, the number of iterations reflects the size of data set (larger data sets undergo more iterations, since 100 unlabeled modules are added in each) and the proportion of initially labeled modules. To visualize the variability of results, we show the error bars. A bar measures one standard deviation from the average, at each iteration.

Figures 5.4 depict the performance trends of the AFP on six data sets when the number of initially labeled software modules is 5% of the project size. The measure of interest is the area under the ROC curve (AUC). From the first plot, we can observe that the average prediction performance on KC1 in the first iteration is 0.77 with the corresponding error band ranging from 0.72 to 0.82. This is a large variance. Through additional iterations, the error band gets narrowed. At the end of the adaptive procedure, the error band is reduced to the range $(0.78, 0.80)$. Meanwhile, the performance of the model improves. The average performance at last iteration is 0.79, an increase of 2.6% over the first iteration. In the last iteration, the number of unlabeled modules is much larger than in the first iteration. In case of project KC1, we observe that the performance of fault prediction models gains stability throughout the adaptive procedure, i.e., the standard deviation shrinks.

The same phenomenon can be observed with all the other data sets in Figures 5.4: PC1, PC3, PC4, CM1, and KC3. For PC1, the average performance increases from 0.67 to 0.73 while the width of the error range is reduced from $(0.53, 0.81)$ to $(0.70, 0.76)$. Therefore, over the iterations of adaptive learning, average prediction performance improves 9% and the standard deviation decreases by 78.6%. The average prediction improvement rates for the other four data sets, PC3, PC4, CM1, and KC1 are 7.1%, 9.7%, 10.9% and 11.3%, respectively. The corresponding rates by which the standard deviation decreases are 71.4%, 71.4%, 73.3% and 89.5%. The detailed statistics for the improvement rate of average performance and the reduction rate of standard deviation for all data sets is presented in Table 5.3.

We experimented with 10%, 20% and 50% of initially labeled modules too and the results are shown in Figures 5.5 to 5.7, respectively. In these figures similar trends emerge. Through consecutive iterations of adaptive learning, error bars consistently shrink on all data sets across all initial labeled data size settings. However, the average performance

Init. Labeled	% of Changes	KC1	PC1	PC3	PC4	CM1	KC3	AVE.
5%	Ave. Performance	2.6	9.0	7.1	9.7	10.9	11.3	8.4
	Std.	80.0	78.6	71.4	71.4	73.3	89.5	77.4
10%	Ave. Performance	2.6	5.8	8.7	3.8	9.0	2.6	5.4
	Std.	85.7	81.3	77.8	66.7	78.6	83.3	78.9
20%	Ave. Performance	-2.5	0.0	1.4	6.3	7.4	2.6	2.5
	Std.	80.0	62.5	75.0	81.8	83.3	84.6	77.9
50%	Ave. Performance	1.3	0.0	2.7	1.2	-1.4	-2.4	0.2
	Std.	80.0	58.3	66.7	71.4	73.3	66.7	69.4

TABLE 5.3: Percentage of change over the iterations of adaptive learning procedure.

levels out as the size of initially labeled data set increases. For some experimental settings, the average performance slightly decreased.

The rates at which average performance improves and variance decreases across all experimental settings are clearly shown in Table 5.3. The values reflect performance improvement parameters derived from predictions taken at the end of the first iteration and after the last iteration. The percentage of increase for average performance and the percentage of decrease for standard deviation are calculated. $Ave.performance$ stands for average AUC improvement and $Std.$ stands for the reduction in standard deviation.

For example, PC1 exhibits a 9% improvement in average performance between the first and last adaptive iteration when 5% of the data set is labeled initially. The same rate of average performance improvement is reduced to 5.8% when 10% of the project modules are initially labeled. When the initially labeled data size increases to 20% and 50%, the rate of performance improvement through iterations is closer to zero. This demonstrates that with the larger number of initially labeled modules the improvement obtained through the adaptive procedure is minimal.

An average increase in performance, measured by AUC, of $(8.4, 5.4, 2.5, 0.2)$ across all data sets is presented in the last column of Table 5.3). Meanwhile, the reduction of standard deviation is relatively stable for all data sets. As the last column of Table 2 indicates, the average rate of decrease of standard deviation through algorithm iterations is at or above 70%. This is a significant observation, as variability in the application of fault prediction models is a strong concern.

Overall, we observe that through the iterations the performance of fault prediction model in the adaptive procedure slightly improves while the corresponding standard deviation is significantly reduced. This observation is consistent through most data sets, across varied sizes of initially labeled modules. However, as the proportion of initially labeled modules increases, average performance obtained through adaptive iterations of AFP approach flattens out.

Up until now, we analyzed the iterative performance of the AFP approach. One might wonder about performance comparison between this approaches when compared with the traditional supervised learning. Supervised learning is more popular and straightforward. Next, we compare the performance of AFP with the corresponding supervised learning experiment, which uses Naive Bayes classifier. This is a fair comparison, as Naive Bayes is used within each iteration of AFP too.

In Figure 5.8 , we compare AFP algorithm and the supervised learning approach - Naive Bayes. Again, we vary the size settings for the initially labeled data set. The comparison is fair from one more perspective: we use the same total number of labeled modules for training in both cases. For example, PC1 contains $1,107$ modules. When 5% of modules are initially labeled for AFP approach, there are 11 adaptive iterations. Within each iteration the fault content is predicted for 100 new modules. Having five modules added to labeled data set and used in training, we arrive to the total of $5 * 11 = 55$ modules newly labeled by the oracle. Together with the initial 5% $(0.05 * 1107 = 55)$ labeled modules, there are $(55 + 55 = 110)$ modules that require the application of V&V activities to establish their faultiness. In the corresponding supervised learning case, we randomly select 110 modules as training instances. The performance for both approaches is evaluated on all the modules not used in training. Each experiment is repeated 20 times to balance randomness in the selection of the training data. In Figure 7, the filled points represent the performance of AFP and the unfilled points represent the AUC achieved by supervised learning. Again, the average performance and standard errors are obtained from 20 runs.

The first plot in Figure 5.8 compares the performance of AFP and supervised learning (SL) on all data sets when 5% of the modules are initially labeled for training in AFP and the appropriate size of the training set is used by SL. In the plot, we observe that AFP approach outperforms the corresponding supervised learning on all but one data set (PC4). On the other hand, the error bars for AFP are narrower than those coming from the application of SL. This observation is consistent across all the data sets.

In the second plot (10% of modules initially labeled) of Figure 5.8, the trends are similar as in the first plot, although average performance improvements are less pronounced. When the proportion of initially labeled modules increases to 20% and 50% in the two bottom plots in Figure 5.8, the average performance of supervised learning approach overtakes the performance of AFP for most of the data sets, although by a slight margin and within error bars.

It is interesting to note that the difference of standard error between AFP and AL is large when 5% labeled data set is used - AFP has narrower error bar than AL. The difference in standard error tends to disappear when the size of labeled data increases

up to 50%. This is because that when a small size of data is sampled and used as initially labeled set, SL classifier is more likely to get stuck at local solution, thus its classification performance is less stable. Data quality issues, such as outliers or noise in labels, can also significantly affect the performance of SL when sampled training data is small. AFP, repeatedly and intelligently selecting representative additional samples, tends to train more generalized model, thus receives less dispersion in standard error. When the size of initially labeled data is large (50% of entire data set), the instability in classification by SL gets reduced and the impact of data quality issues to SL is less significant. Thus, the standard deviations of SL in these experiments are close to that observed from AFP.

It is apparent that the advantages of AFP can be expected with smaller number of initially labeled modules. When the number of labeled modules approaches 20% or more relative to the project size, supervised learning is a simpler and, likely, a better choice.

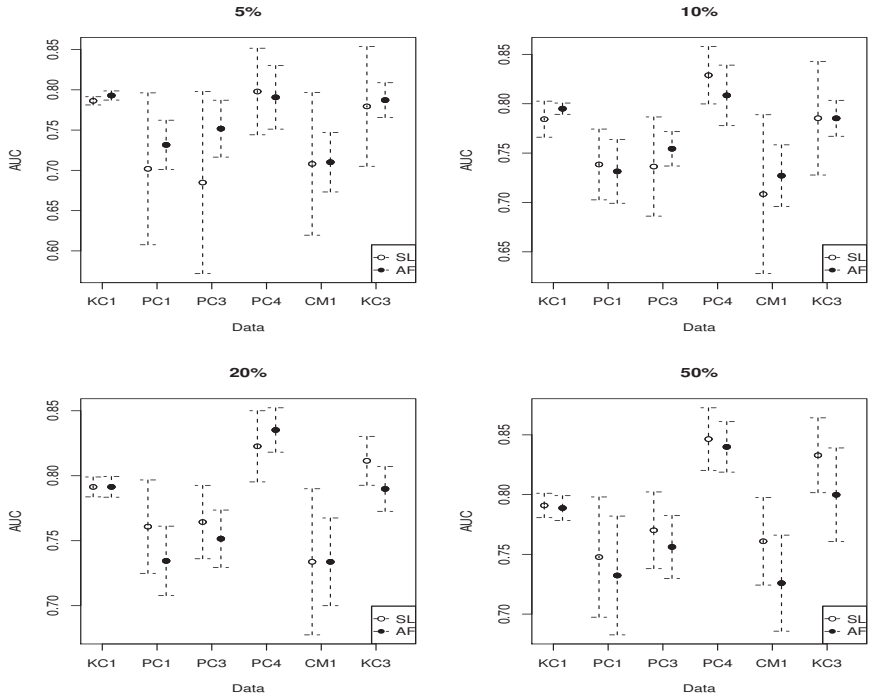

FIGURE 5.8: Comparison between AFP approach and supervised learning. Both use Naive Bayes classifier with varied sizes of labeled data used in training.

5.3.3 Discussion

In our experiments, we varied the number of modules in the initially labeled group between 5% and 50% of the overall data set size. At 5% setting, there are 105, 78, 73, 55, 25, and 23 modules that are randomly selected from projects KC1, PC3, PC4, PC1, CM1 and KC3 respectively, and used for training. For 10% setting, the number of initially labeled modules is 211, 156, 146, 111, 50 and 46 (Table 5.2). Such small sets of training data have rarely been used in software fault prediction literature and have been deemed impractical for most practical purposes. The rationale has always been that fault prediction is typically carried out to predict faulty modules in successive versions of the project. When the fault content of modules in earlier versions is available and easy to obtain, due to well-organized problem reporting and maintenance, this makes supervised learning a practical method for modeling. However, every project has version one. Within the initial development, the performance of existing fault prediction techniques has never been studied, because supervised learning algorithms offer poor prediction performance in absence of a substantial number of training instances.

Our adaptive approach offers better fault prediction classification results in the early development context, when no more than 5% or 10% of modules have been inspected for faults. Compared with the corresponding supervised learning approach at the same size of training instances, AFP exhibits better average performance with less dispersion in the standard error. Nevertheless, a large number of modules, 20% or 50%, are not likely to be available for model training of Version 1 in practice. Consequently, we are not as concerned with the limited benefits of AFP at large proportions of initially labeled modules. Supervised learning would likely be the modeling technique of choice in such cases.

We can summarize our observations as follows. For projects in which the number of modules with known fault content is small, AFP approach:

1. Efficiently adapts fault prediction to the dynamics of software development in which modules are developed over time;
2. Reduces the cost of quality assurance techniques;
3. Has better performance and less variance than traditional supervised learning when the same number of modules are available for model training;
4. Works better when used to track emerging system's quality, especially when the fault rate of newly developed modules is uneven, i.e., not a constant.

For observations 1 to 3, the merits are explicit, derived directly from experiments. The last observation needs elaboration. Let us restrict attention to new software projects, those without past performance data. If we want to use supervised leaning for fault prediction, only the modules developed and released early can undergo verification activities, be labeled and used as training data. In some projects, the fault introduction rates can vary over time for a variety of reasons. For example, a project might have low fault introduction rate in the beginning because early modules tend not to be as complex as the ones released later. Or, for whatever reasons, if the most experienced developers leave the project in its infancy and are replaced by less experienced ones, the fault introduction rate may vary too.

Thereby, the proportion of faulty modules may increase with time. In a situation like this, supervised learning approach will obviously lack the capability to capture the overall quality distribution and trends. Similar argument can be made for situations in which fault introduction rate decreases over time. AFP approach, in principle, predicts the fault content of modules as they emerge from the development. The fault trends of project can be "sampled" intelligently. In this study, we randomize the order of modules in the experiments because the experimental data sets lack time order information. However, studying fault prediction using the chronological development order would be valuable.

5.4 Conclusion

Predicting and tracking software quality at the time of development is important for new projects, which have no previous releases. Fault prediction literature mostly focuses on supervised learning approaches. However, supervised learning is not practical in absence of large training data set, the availability of which raises the cost of fault prediction. Supervised learning approaches also lack the capability to capture variations of software quality in development, over time.

In this chapter, we propose an adaptive learning approach, which resolves some of the limitations of supervised learning. In our approach, we wrap supervised learning into an adaptive procedure. Active learning is part of the adaptive learning procedure. It supports intelligent automated selection of modules, which best represent faulty and non-faulty classes. Our results show that the proposed adaptive approach provides improved performance when the number of initially labeled modules is small - lower than 20% of the project size. Compared to the corresponding supervised learning approach, our algorithm provides slightly better performance with significantly reduced variability of prediction results.

Chapter 6

Revisit Active Learning using different Data Sets

By far, all of our experimental data sets are projects from NASA MDP repository and we assume there are once-a-time projects that have no previous subsystems or versions we can learn from. To validate our findings in active learning, in this chapter we extend the investigation of active learning on version-based projects. Several active learning approaches are investigated using releases of Eclipse, Camel and Ant projects from PROMISE. From our precious studies, we also learned that the performance of active learning, in general, depends on the combination of features and the quality of the data. Hence, in this chapter we also investigated the effect of feature transformations in active learning by exploring two dimensionality reduction techniques and four feature selection techniques.

6.1 Feature Compression

Software metrics used as features in defect prediction problems can be described by aggregating complexity metric values (maximum, average and total)[122]. Such aggregation may introduce irrelevant or redundant information relative to defect prediction. Compressing features into a small set may not only compact the information but remove the noise too. On the other hand, active learning with uncertainty sampling typically selects outliers for the assessment by the oracle. In many cases these are modules with noise, for example incorrectly computed metrics, wrong labels, highly unusual code content or similar problems. Such outliers in the unlabeled data set have high uncertainty, but may not provide much help in building the model. Feature selection and dimensionality reduction are methods designed to reduce the number of dimensions and thus

describe the objects of interest more succinctly [123, 124]. Feature selection and dimensionality reduction techniques are, therefore, reasonable solutions to minimize noise related problems in software defect data. We postulate that these techniques enable more effective and rapid learning process.

In this section we discuss the performance of multiple feature selection and dimensionality reduction techniques applied to modules in software version control repositories prior to active learning for defect prediction.

6.1.1 Feature Selection Techniques

Feature selection techniques [125, 126] broadly fall into two categories: a) the filter model and b) the wrapper model. The filter model is based on general/intrinsic characteristics of training data and it does not consider learning algorithms. Usually, it ranks each feature according to some univariate metric such as uncertainty or correlation and

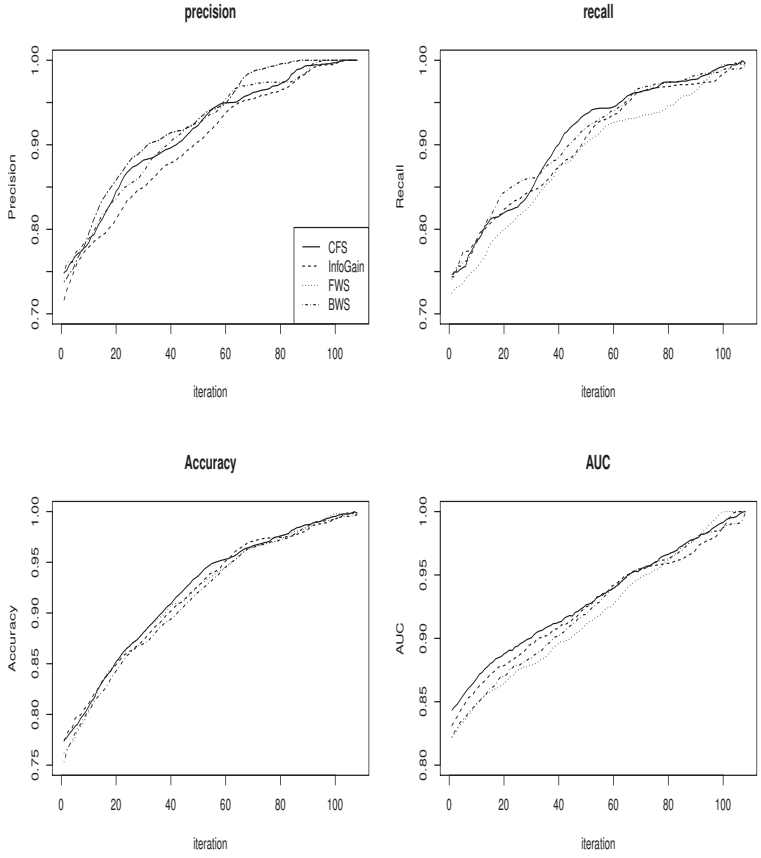

FIGURE 6.1: Comparison of different feature selection techniques with active learning using Eclipse 2.0 packages for training and 2.1 for evaluation.

selects the highest ranked features. Note that the scoring should reflect the discriminative power of each feature. The wrapper model computes feature scores relative to the learning algorithm. For each subset of features, the wrapper needs to create a predictive model. The performance of selected features is evaluated with respect to the learned model. Wrappers tend to find features better suited for the learning algorithm, boosting its performance. Incorporating feature selection techniques prior to defect prediction process is not new in the literature [127, 128].s In this study, we utilize four feature selection methods, two from filter and two from wrapper feature selection models:

- *Information Gain (InfoGain)*: A measure based entropy, it measures the decrease of the weighted average impurity of the partitions compared to the impurity of the complete set of data.
- *Correlation-based feature selection (CFS)*: A heuristic, which scores feature subsets and trades off the average relevance of the class - the dependent variable - against the average inter-correlation. CFS selects features that are relevant to the class, but are not redundant to any of the other relevant features.
- *Forward selection (FWS)*: Selects features iteratively based on a certain criteria. Newly selected features (in each sequence) boost the performance of previously selected metrics.
- *Backward selection (BWS)*: Similar to FWS, but it starts with all features and iteratively removes the least relevant one. J48 is the predetermined learner in both the FWS and BWS methods.

To compare above feature selection techniques, we applied each one of them in subsequent releases of the projects, when feature selection technique was followed by the active learning process. The prediction model is built using, for example, Eclipse packages from release 2.0 and each iteration adds 1% of the modules from release 2.1, following the determination of their true labels. Trends in Figure 6.1, using Eclipse packages level data, reflect the performance of trained models at each iteration, marked along the x axis. While the details of our experimental methodology are explained later (Section 6.2.2), it is interesting to note the uniformity of observations in Figure 6.1. Essentially, there is no observable difference in defect prediction performance that can be attributed to the use of any of the four feature selection techniques. We observed a similar result when the units of prediction were files in Eclipse, Camel and Ant.

6.1.2 Dimensionality Reduction Techniques

As we've learned in Chapter 4, dimensionality reduction refers to the process of replacing the original features of a high-dimensional space with a set of derived features in a lower-dimensional space. Unlike feature selection in which subsets of original features are selected, dimensionality reduction combines original features to extract essential information [129].

With the success of Multidimensional Scaling (MDS) in our previous study, we continue to use the MDS technique in this study. Recall that MDS is a non-linear optimization

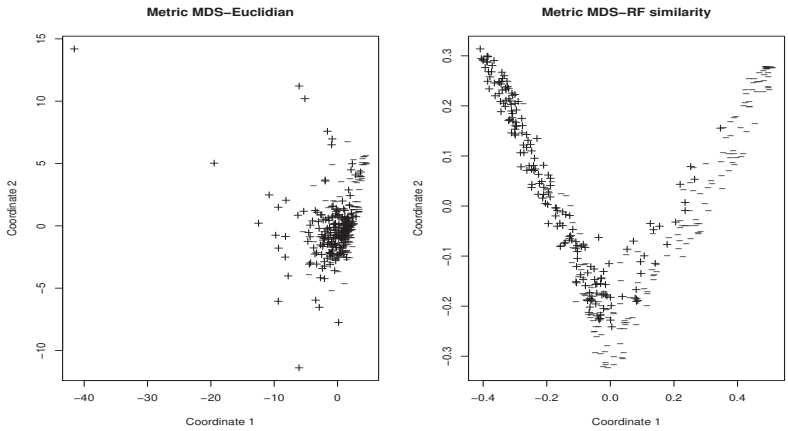

FIGURE 6.2: Comparison of Euclidean distance vs. RF similarity in multidimensional scaling (MDS) on Eclipse release 2.0. The plus sign represents defective modules, the minuses represent defect-free modules(at package level).

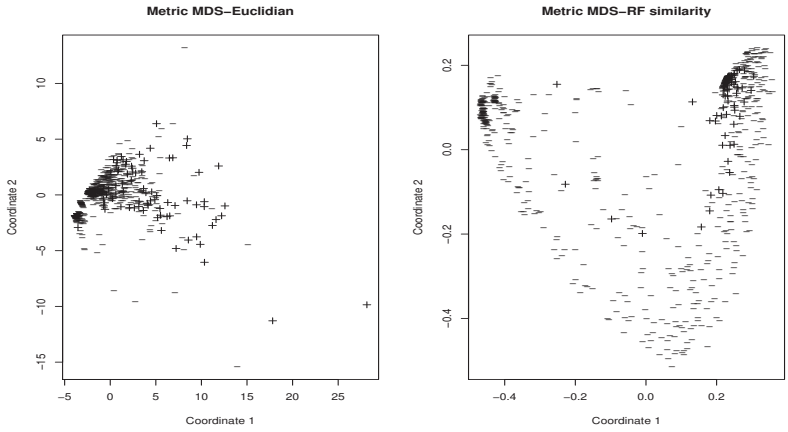

FIGURE 6.3: Comparison of Euclidean distance vs. RF similarity in multidimensional scaling (MDS) on Eclipse release 2.0. The plus sign represents defective modules, the minuses represent defect-free modules(at file level).

approach creating a lower dimensional mapping of high dimensional data with respect to the proximity information. It attempts to find a data embedding such that the similarities or dissimilarities are preserved. Typically, MDS starts with a proximity matrix and then assigns a location to each data instance in a lower dimensional space. Similar instances are represented by points that are close together.

We investigated two proximity measures in the context of Eclipse project data: Euclidean distance and Random Forest's (RF) similarity (check Chapter 4 for the details about Random Forest similarity). Note that when Euclidean distance matrix is used as the proximity matrix, the multidimensional scaling is equivalent to the Principal Component Analysis (PCA) - a linear dimensionality reduction technique. In the case of RF similarity, the matrix is built via the Random Forest algorithm.

To understand the difference between the two MDS proximity measures, we mapped the Eclipse data, the packages and files from Version 2.0, into a two-dimensional distance space. Figure 6.3 shows the 2-D scatter plots. The left plots in a) and b) reflect the Euclidean distance matrix and the right plots come from the RF similarity as the proximity matrix. In the plots, instances denoted by a $+$ and a $-$ are the indications of defective and defect-free modules, respectively. This exercise shows the clear advantage of RF similarity in spatial separation between the defective and defect-free modules. Learning from the transformed space of software metrics, in which defective and defect-free modules appear separated, is likely to lead to better classification performance.

TABLE 6.1: Software data sets

Release	**Size**	**Defects (%)**	**Metrics**
Eclipse 2.0 (pkg)	377	50.4%	41
Eclipse 2.1 (pkg)	434	44.7%	41
Eclipse 3.0 (pkg)	661	47.4%	41
Eclipse 2.0 (file)	6,729	14.5%	32
Eclipse 2.1 (file)	7,888	10.8%	32
Eclipse 3.0 (file)	10,593	14.8%	32
Camel 1.2	608	35.5%	20
Camel 1.4	872	16.6%	20
Camel 1.6	965	19.5%	20
Ant 1.3	125	16.0%	20
Ant 1.4	178	22.5%	20
Ant 1.6	351	26.2%	20

Table 6.2: Metrics in Eclipse data set

Metric	**Details**
FOUT	Number of method calls(fan out)
MLOC	Method lines of code
NBD	Nested black depth
PAR	Number of parameters
VG	McCabe cyclomatic complexity
NOF	Number of fields
NOM	Number of methods
NSF	Number of static fields
NSM	Number of static methods
ACD	Number of anonymous type declarations
NOI	Number of interface
NOT	Number of classes
TLOC	Total lines of code
NOCU	Number of files(compilation units)

Table 6.3: Metrics in Camel and Ant data sets

Metric	**Details**
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 in Methods
CA	Afferent Couplings
CE	Efferent Coupling
NPM	Number of Public Methods
LOC	Line of Codes
DAM	Data Access Metric
MOA	Measure of Aggregation
MFA	Measure of Function Abstraction
CAM	Cohesion Among Methods
IC	Inheritance Coupling
CBM	Coupling Between method
AMC	Average Method Complexity
Max_CC	Maximum values of methods in the same class
Avg_CC	Mean values of methods in the same class

6.2 Experiments

6.2.1 Software Data Sets

The data sets used in this study comes from four projects - Eclipse, Camel, and Ant. Eclipse is a multi-language software development environment consisting of the base workspace and extensible plug-ins that customize the environment. The environments include the Eclipse Java development tools (JDT) for Java and Scala, Eclipse CDT for C/C++ and Eclipse PDT for PHP, among others. We unitlized the defect content in three successive releases of Eclipse, 2.0, 2.1, 3.0, at two levels of granularity: files and

packages. Several versions of Eclipse data sets have been in use to study defect prediction [130, 131]. In our study, we use the Eclipse data sets introduced by Zimmerman et al. [122], which are publicly available. Zimmerman et al. used the Java parsers for Eclipse - *visitors* and *aggregators* - to aggregate the metrics to file and package levels. More specifically, the *visitors* is implemented to compute standard metrics for methods, classes, or files (compilation units), while the *aggregators* is used to compute single values for each level. They computed the average (avg), maximum (max) and total values (total) for each metric, except the NOCU - the Number of files. The complexity metrics for each package/file can be computed from the archived builds of Eclipse. These data sets have been used in several recent studies as well [100, 132, 133]. Table **??** presents the metrics included in the Eclipse data sets.

In addition, we also applied active learning on data sets from two projects - Camel and Ant which are publicly available in the PROMISE repository. Both of Camel and Ant consist of three releases. Each instance in the three projects represents a class (.java) file and consists of twenty software metrics.

A summary of the data sets used in our study is reported in Table 6.1. The table lists the number of instances, actual defect rate and the number of metrics used in each release of the four projects. We note that the sample sizes of Ant releases are particularly low. For example, Ant 1.3 consists of 125 files and only 20 are defective. Table 6.2 and Table 6.3 provide the annotation for metrics in the three projects, respectively. To seek more details of the projects, please refer to [122, 134].

6.2.2 Experimental Setting

Defect prediction between successive releases of the same product is practical because we expect minimal changes in the development environment and, consequently, similar defect characteristics. Further, the defect content of modules from an earlier release is known as a consequence of defect reporting. If the community of users is sufficiently large, the reports are likely to cover a big portion of the existing defects. Suggesting that humans serve as "oracles" for some modules in the upcoming release does not represent an extraordinary burden on the development team. For example, in Eclipse the defects reported in the six months prior to the release date are called *pre-defects*. Generally, development teams perform pre-release assessment, debugging and defect removal through unit testing, code walk-through, inspection and other forms of software verification.

Active learning defect prediction approach investigated in this study simply introduces a discipline in the selection of modules that need to be exposed to more thorough verification. Depending on project practices, this requirement may induce additional development cost. However, if defect prediction model performs well, the cost of post-release maintenance should be lower. Whether this value proposition is valid or not remains an open question not only for the proposed defect prediction approach but for the entire research area [135]. However, it is clear that our approach (like any other active learning method) should use the oracle sparingly, requesting as few pre-release module assessments as possible.

In this section, we report experimental results from active learning defect prediction on four projects, totally nine releases, using visual analysis such as graphs and tables. In Section 6.2.6, we supplement the visual analysis with appropriate statistical tests. The experiments will help us understand:

- The defect prediction performance of active learning between subsequent releases;
- The impact of active learning variants - random selection vs. uncertainty-based selection of modules that need oracle's assessment;
- The impact of feature selection techniques when applied prior to active learning;
- The impact of dimensionality reduction techniques when applied prior to active learning
- The impact of data size and defect rate on the prediction performance of active learning;

In Section 6.1.1 we showed that all feature selection techniques perform similarly (see Figure 6.1). Hence, for further experiments we selected only one of them - the information gain feature selection (*InfoGain*). We also learned that the RF similarity coupled with dimensionality reduction technique MDS outperforms Euclidean proximity. Therefore, we will experiment with MDS, which uses RF similarity only. The six experimental approaches we analyzed and their abbreviations are:s

1. *Act*: Active learning with uncertainty-based selection;
2. *Rand*: Active learning with random-based selection;
3. *IG_Act*: Information Gain feature selection, *IG*, followed by *Act*;
4. *IG_Rand*: Information Gain feature selection, *IG*, followed by *Rand*;
5. *MDS_Act*: *MDS* with RF similarity followed by *Act*;
6. *MDS_Rand*: *MDS* with RF similarity followed by *Rand*;

Each release is experimented for each of the above six active learning approaches. For example, Release 2.0 in Eclipse is used to build defect prediction model predicting defect prone modules in release 2.1. Next, release 2.1 is used for training and release 3.0 for prediction. Every experiment is run 10 times and average values are reported for experimental comparison.

Random Forest (RF) is selected as the base algorithm in active learning experiments due to its consistent performance in[71, 136]. Our previous studies also showed that random forest outperforms other supervised learning when the data is imbalanced and noisy.

At *each iteration* of active learning, a fixed number of modules (∼1% of the unlabeled modules) are selected for the assignment of their true defect labels. For example, with the Eclipse data, 4 packages (79 files) are selected at each iteration when predicting on release 2.1, and 7 packages (106 files) when predicting on release 3.0.

To track the prediction performance at each iteration of active learning, we do not set an apriori stopping criterion. The algorithm continues until it runs out of unlabeled modules (i.e. all unlabeled modules are labeled). Of course, in practice we are interested in the prediction performance of models that use as few modules analyzed by the oracle as possible, likely no more than 20%. A classic supervised learning experiment with random forest (RF) is the same as the 1^{st} iteration of our experiment, before active learning process starts. At that point, modules from previous release(s) are used as training data and all modules from the current release represent test data.

Performance measures for active learning can be derived by tracking the predictions, i.e, $P(Y_u = 1|X_u)$, at each iteration. Following the best practices in [87] and [122], we computed AUC, Precision, Recall and Accuracy measures. The fault prediction at each iteration reflects the performance of the trained model on all the unlabeled modules of the current release.

6.2.3 Results from Eclipse data sets

In this section we discuss experimental results using Eclipse at package and file levels. In this section we discuss experimental results using Eclipse data sets. Figure 6.4 compares the six active learning approaches using the release 2.0 for training and the release 2.1 for prediction. By all measures (precision, recall, accuracy, AUC), dimensionality reduction followed by active learning with uncertainty based selection (MDS_Act) offers the best performance. Active learning with uncertainty based selection (Act) works consistently better than active learning with random based selection ($Rand$), regardless of the feature

compression technique. The *MDS_Rand* approach works better than the *IG_Rand*. The *Rand* approach performs the worst. In the AUC plot, which combines all the other performance plots we observe that *MDS_Act* and *MDS_Rand* both outperform the other approaches. The *MDS_Act* is slightly better than *MDS_Rand*. Although differences are small, the *IG_Act* slightly outperforms *IG_Rand*, *Act* and *Rand*.

Figures 6.5 and 6.6 depict the performance of the six approaches when release 2.1 and 2.0 & 2.1 are used for training respectively and release 3.0 for prediction at the package level. In the Precision plot, it is apparent that the active learning with uncertainty sampling related approaches (*MDS_Act*, *IG_Act* and *Act*) outperform active learning with random sampling (*MDS_Rand*, *IG_Rand*, and *Rand*). Regarding Recall and Accuracy, the *MDS_Act* outperforms the other approaches. In the AUC plot, there are no considerable differences between *MDS_Act* and *MDS_Rand*, but both approaches are better than the others. Prediction on release 3.0 at the package level is also interesting because initially dimensionality reduction based methods need a few extra cycles to adjust their performance.

Figures 6.7, 6.8 and 6.9 depict defect prediction experiments with Eclipse files. Similar to the package level prediction, the *Act* approach wins over the *Rand* approach across all measures. The *MDS_Act* approach with files seems to be the one with the best likelihood of providing the best prediction performance overall.

Tables D.1 to D.3 in Appendix D enable a closer look into the early cycles (1^{st}, 10^{th}, 20^{th}, and 30^{th} iteration) of active learning variants from Figures 6.4 to 6.6, respectively. For example, in Table D.1, the *MDS_Act* approach receives a Recall value of 0.924 at 30^{th} iteration, when the *Rand*, *Act*, *IG_Rand*, *IG_Act*, and *MDS_Rand* approaches reach 0.815, 0.861, 0.797, 0.845, and 0.853, respectively. Tables D.4 to D.6 similarly quantify the performance of active learning approaches depicted in Figures 6.7 to 6.9 for the files in Eclipse. Recall that the measures at 1^{st} iteration are obtained from classic supervised learning where the previous release(s) is/are used as training data and the current release as test data. One can easily observe that random forest as a supervised learning approach performs better than logistic regression by simply comparing our results with those by Zimmerman [122]. Considering that the focus of this study is primarily on active learning and its variants, we will not delve in the comparison among supervised learning approaches, which has been addressed in the literature.

6.2.4 Results from Camel and Ant data sets

Next, we explore the prediction performances of six approaches using Camel and Ant projects. The results from Camel project are presented in Figure 6.10 and Figure

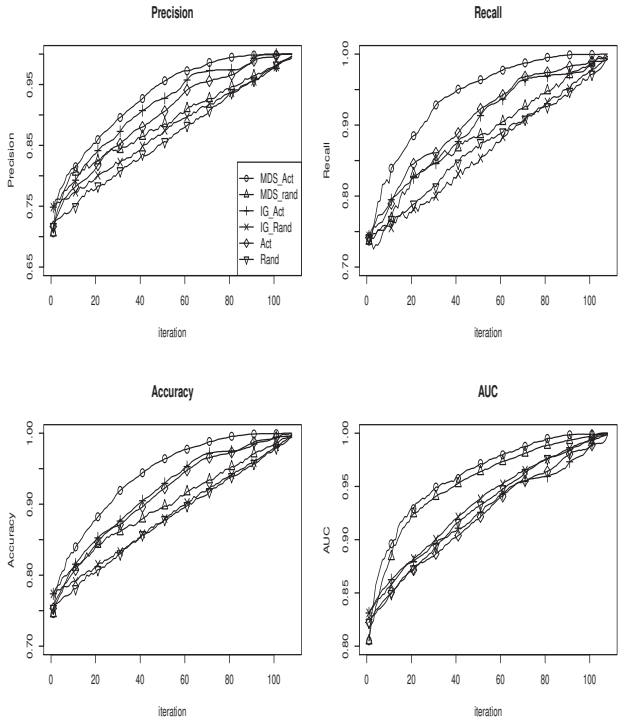

FIGURE 6.4: Defect prediction in release 2.1 from 2.0 (Eclipse - packages)

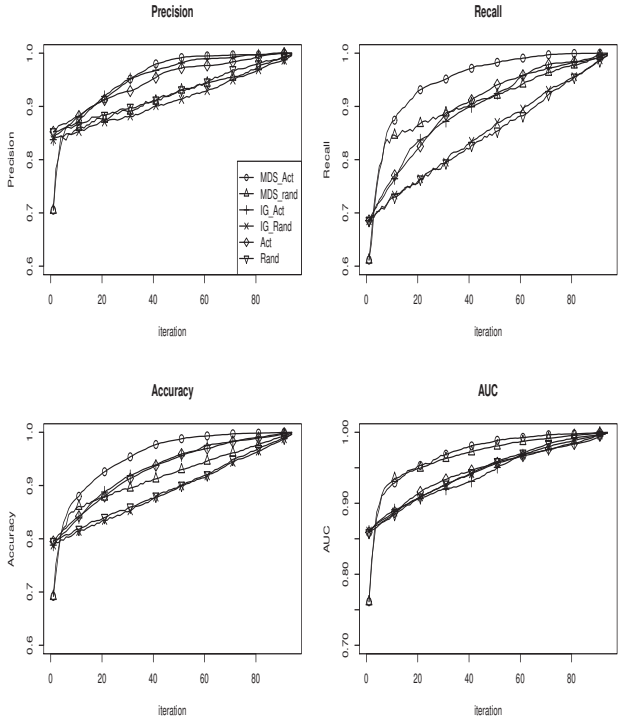

FIGURE 6.5: Defect prediction in release 3.0 from 2.1 (Eclipse - packages)

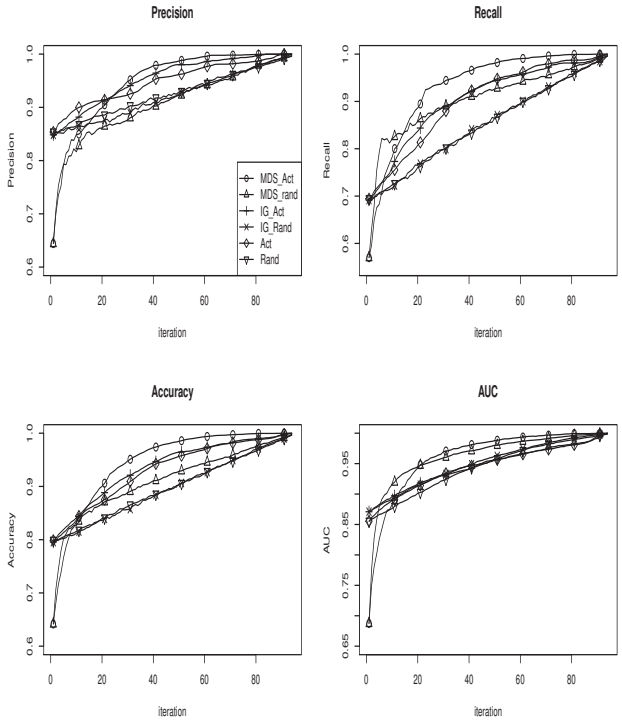

FIGURE 6.6: Defect prediction in release 3.0 from 2.0 and 2.1 (Eclipse - packages)

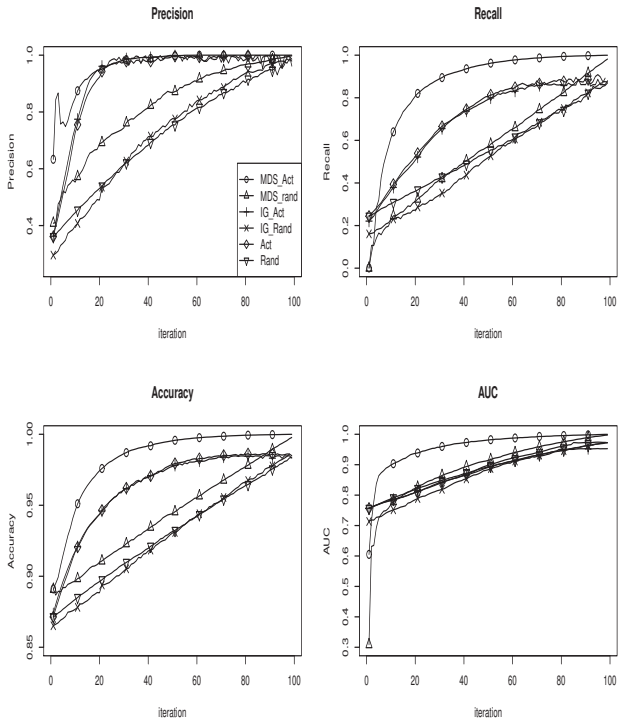

FIGURE 6.7: Defect prediction in release 2.1 from 2.0 (Eclipse - files)

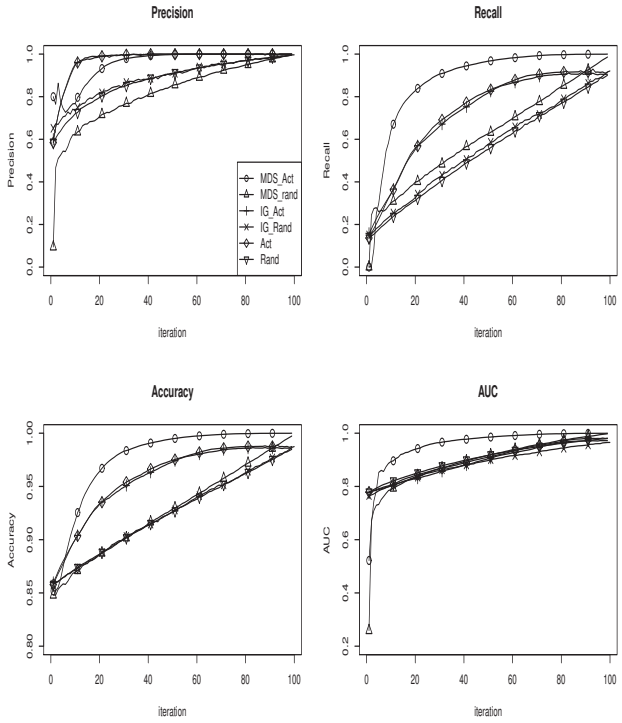

FIGURE 6.8: Defect prediction in release 3.0 from 2.1 (Eclipse - files)

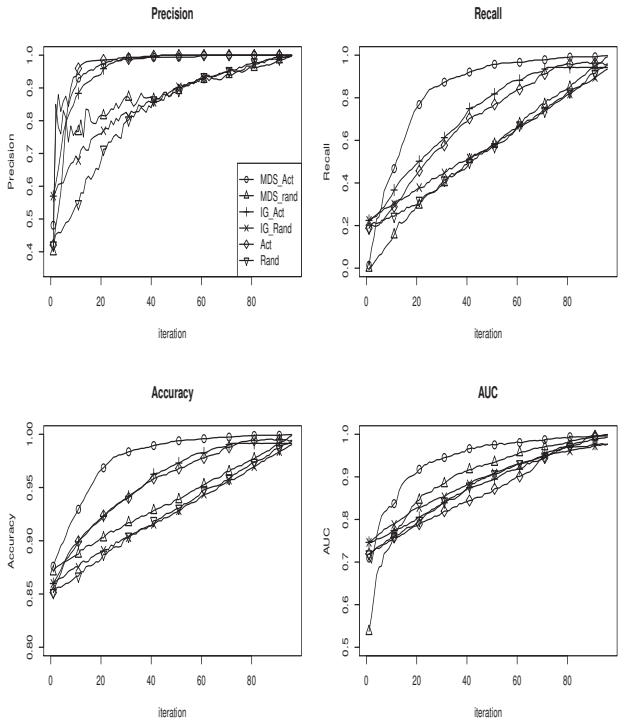

FIGURE 6.9: Defect prediction in release 3.0 from 2.0 and 2.1(Eclipse - files)

6.11. In Figure 6.10, we compared the difference of six approaches using the release 1.2 for training and release 1.4 for prediction. In the plot of precision measure, the *MDS_Act* outperforms the other five approaches. The *MDS_rand* performs better than the *IG_Act*, *IG_Rand*, *AL* and *Rand* before the 40^{th} iteration. *Act* and *IG_Act* have the similar performance trends, and both are significantly better than the *IG_rand* and *Rand*. The same performance ranking of the six approaches is observed in the plot of Accuracy. With the recall measure, the *MDS_Act* has the worst performance at beginning, but it exceeds the others soon near 20^{th} iteration. *Act* performs slightly better than the *IG_Act*, *IG_Rand*, *MDS_Rand* and *Rand*, primarily between 20^{th} and 40^{th} iterations. The *MDS_Act* and *MDS_Rand* perform approximately the same, consistently, both being better than the other approaches in terms of the AUC measure, while all the other approaches are not distinguishable.

In Figure 6.11, we use Camel 1.4 for training and Camel 1.6 for prediction. In the Precision plot, we observed that *Act* related approaches (*MDS_Act*, *IG_Act*, and *Act*) outperform non-*Act* approaches. *MDS_Act* stands out against the other five approaches at the 15^{th} iteration, where the *MDS_Rand* starts to outperform the *Act*, *IG_Act*, *IG_Rand* and *Rand*. In the plot of Accuracy, *MDS_Act* significantly performs better than the others. *MDS_Rand*, *IG_Act*, and *Act* have very similar performance. With regard to the AUC measure, *MDS_Act* and *MDS_Rand* perform significantly better than all others.

Figure 6.12 illustrates the prediction performance of the six approaches using Ant project, where the release 1.4 is predicted from the release 1.3. In the first plot, *MDS_Act*, *IG_Act* and *Act* perform better than *MDS_Rand*, *IG_Rand*, and *Rand*. *MDS_Rand* exceeds *IG_Rand* and *Rand*. With Recall and AUC measures, the *MDS_Act* approach slightly outperforms the others. It is hard to distinguish the difference among the six approaches in the plot of AUC. The prediction performances of Precision, Recall and Accuracy are very similar when release 1.5 is used for training and release 1.6 is for prediction in Figure 6.13, except that the superiority of *MDS_Act* tends to be significant compared to those in Figure 6.12. In the last plot of Figure 6.13, *MDS_Rand* slightly outperforms the others.

Tables D.7 and D.10 in Appendix D quantify the performance measures of six approaches in Figure 6.10 to Figure 6.13. Considering the small size of data in Camel and Ant, we capture the measures at $1st$, 10^{th} 20^{th} and 40^{th} iterations.

6.2.5 Discussion

We mentioned above that in the case of active learning accompanied by dimensionality reduction (MDS), prediction in early iterations may be degraded. For example, in Figures 6.5 and 6.10, it is interesting to note that in the early iterations (i.e., before the 10^{th} iteration) the active learning with dimensionality reduction technique (MDS_Act and MDS_Rand) attained a much lower performance than the others. After a few iterations, the dimensionality reduction based approaches quickly recover and exceed the other approaches. Rapid performance improvements by MDS_Act and MDS_Rand shows that dimensionality reduction techniques adjust prediction performance as the features from the new release enter the training. However, this is a cautionary tale against the use of dimensionality reduction techniques such as MDS in supervised learning - learning at 1st iteration in active learning process. If there is no intention to deploy active learning, dimensionality reduction methods may lower the performance of models built by the random forest algorithm (see the performance in iteration one when predicting release 3.0).

For readers interested in comparing the performance of active learning with supervised learning, we want to point out that the performance of active learning with random sampling (*rand*) model closely resembles supervised learning using random forest. The difference between *rand* and *Act* approaches is in the guidance towards the choice of "the most informative" modules for labeling. Active models that use uncertainty principle for selection generally perform better than those with a random selection, when training data sets of the same size are provided to both methods.

Performance gains by active learning comes in part from utilizing more software modules in training. The utilization of software verification engineers as "oracles" comes at a cost of their time and effort. In order to reduce that cost, we should use the oracle sparingly. Therefore, from a practical point of view it is important to focus on the performance of active learning in earlier iterations.

6.2.6 Statistical Analysis

In previous section we analyzed the experimental results using graphs and tables. To validate our observations from visual analysis, in this section we conduct a statistical significance test (ANOVA) to compare the prediction performance among six investigated active learning approaches. Figures 6.4 to 6.13 along with tables in Appendix D offer the necessary information for the test. The hypotheses of the test are:

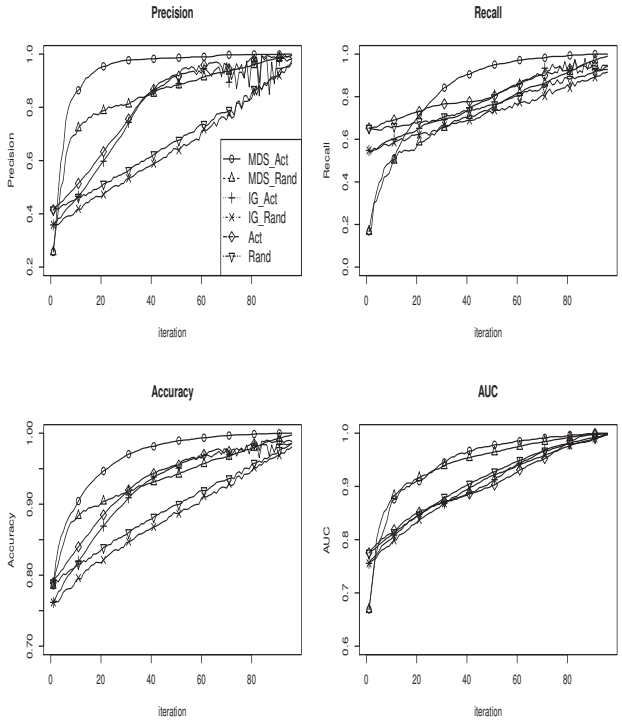

FIGURE 6.10: Defect prediction in release 1.4 from 1.2 (Camel)

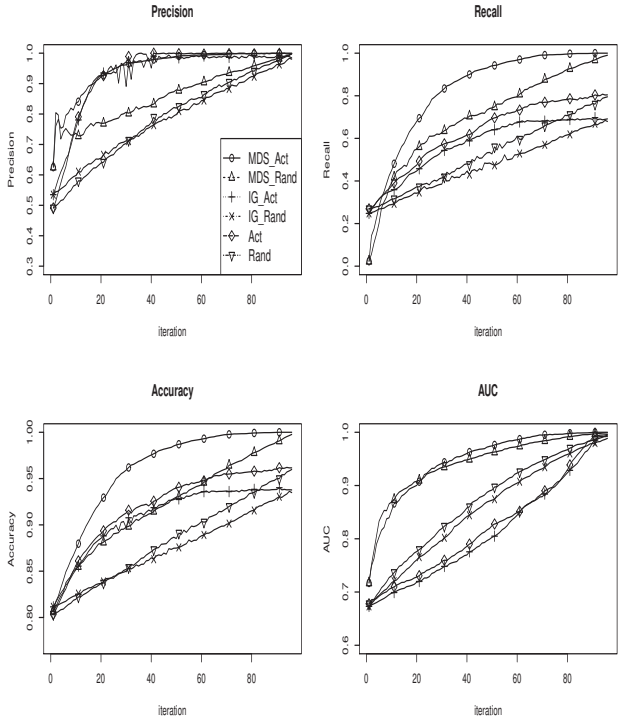

FIGURE 6.11: Defect prediction in release 1.6 from 1.4 (Camel)

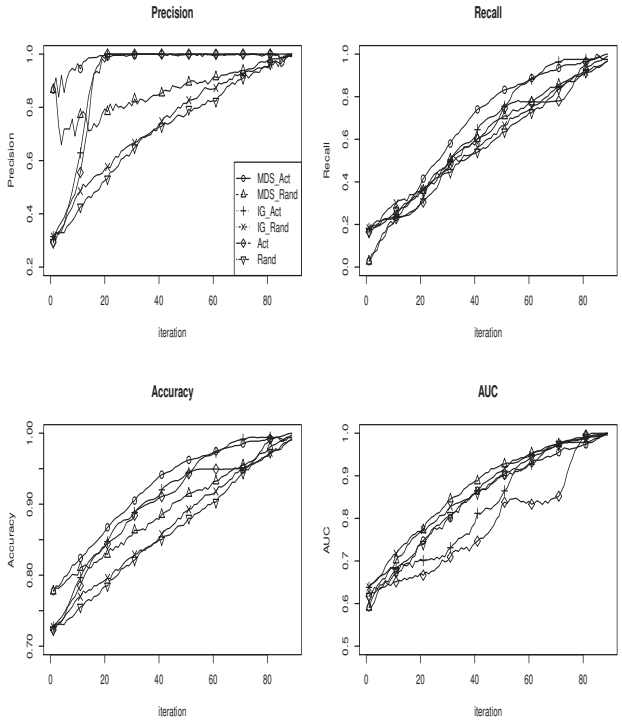

FIGURE 6.12: Defect prediction in release 1.3 from 1.4 (Ant)

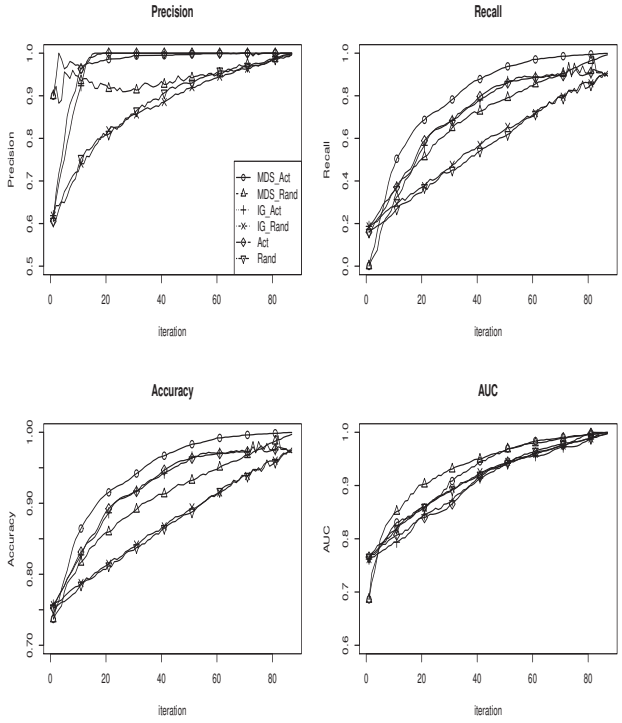

FIGURE 6.13: Defect prediction in release 1.5 from 1.6 (Ant)

H_O: There are no performance differences amongst the six active learning defect prediction approaches;
H_A: At least one approach offers a different performance.

An example of one way ANOVA test is given in Table 6.4. A large F-value indicates that the outcomes of different approaches vary more than the outcomes of the 10 experiments performed for any specific algorithm. The P-value offers a probability of observing a test statistic as extreme as the one actually observed. The smaller the P-value, the more strongly the test rejects the null hypothesis. Let significant level α be 0.05. In our example, the P-value of 0.0188 is smaller than the α value, resulting a rejection to the null hypothesis H_o. We can therefore conclude that there is at least one approach amongst those evaluated, MDS_Act, MDS_Rand, IG_Act, IG_Rand, Act and $Rand$, that significantly outperforms the others.

The P-values of ANOVA for experiments at the package level are shown in Table 6.5. The significant outcomes are those where the P-value is smaller than the significance level, set at $\alpha = 0.05$. They are presented in bold font. We observe that significant differences exist amongst the six approaches, but not for all the performance measures, at 10^{th} iteration. Referring to Table 6.5 the P-values for Precision, Accuracy and AUC at 10^{th} iteration are all smaller than α. The significance does not hold for the Recall, with a P-value of 0.1364. At the 20^{th} and the 30^{th} iterations, the statistical significance between the six approaches is consistently observed for all the four measures. These observations are consistent for the other experiments with Eclipse packages and files. Hence, given the page limit, we did not provide the P-value tables for all the experiments.

Next, we conducted the post-hoc test to determine which of the six approaches differs from the others using Tukey's Honestly Significant Difference (HSD) [112]. This is a simple and frequently used pairwise comparison technique. Table 6.6 and 6.7 present the results from post-hoc tests for all the experiments at package level and file level respectively. In the tables, we compare the performance of the six active learning approaches at 20^{th}, 30^{th} and 40^{th} iterations. Active learning by itself, with dimensionality reduction and feature selection are of particular interest in this paper. For this reason, some pairwise comparisons (MDS_Rand vs IG_Act or MDS_Rand vs IG_Rand, etc.) are not included. Looking into Table 6.6, for example, for Eclipse 2.1 predicted from Eclipse 2.0 at the package level, we can observe that the MDS_Act significantly outperforms all the other approaches across all performance measures. The IG_Act performs significantly better than the IG_Rand and $Rand$, but does not significantly outperform Act except for AUC. Act outperforms $Rand$ for all the measures except the AUC.

TABLE 6.4: One way ANOVA test for AUC measures at 10th iteration, when defects in release 2.1 are predicted from 2.0

	d.f	Sum Sq	Mean Sq	F value	p-value
algorithm	5	0.01560	0.003120	2.988	**0.0188**
Residuals	54	0.05639	0.001044		

TABLE 6.5: P-values by ANOVA test, when defects in release 2.1 are predicted from 2.0 (package level)

Iteration	Precision	Recall	Accuracy	AUC
10th	**7.78E-10**	0.1364	**3.39E-08**	**0.0188**
20th	**1.77E-09**	**0.0005**	**3.98E-11**	**4.54E-18**
30th	**3.67E-09**	**3.00E-07**	**5.71E-11**	**1.36E-18**

TABLE 6.6: Post-hoc test for performance differences between the six active learning approaches at package level ($1 : MDS\_Act, 2 : MDS\_rand, 3 : IG\_Act, 4 : IG\_rand, 5 : Act, 6 : Rand$). " ✓" stands for statistically significant difference between two approaches. "x" stands for no signisficant difference detected between the two approaches.

Labeled	**Unlabeled**	**Exp.**	**Package level**											
			Precision			Recall			ACC			AUC		
			20th	*30th*	*40th*	*20th*	*30th*	*40th*	*20th*	*30th*	*40th*	*20th*	*30th*	*40th*
Eclipse 2.0	Eclipse 2.1	1–2	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	x
		1–3	x	✓	x	✓	✓	✓	✓	✓	✓	✓	✓	✓
		1–4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		1–5	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	✓
		1–6	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		3–4	✓	✓	✓	x	✓	✓	✓	✓	✓	x	x	✓
		3–5	x	x	x	x	x	x	x	x	x	x	✓	x
		3–6	✓	✓	✓	x	x	x	✓	✓	✓	x	x	x
		5–6	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	x	✓
Eclipse 2.1	Eclipse 3.0	1–2	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	x	✓
		1–3	x	x	x	✓	✓	✓	✓	✓	✓	✓	✓	✓
		1–4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		1–5	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		1–6	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		3–4	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	x	✓
		3–5	x	✓	x	x	x	x	x	x	x	✓	✓	✓
		3–6	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	x	✓
		5–6	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	x
Eclipse 2.0 & 2.1	Eclipse 3.0	1–2	x	✓	✓	x	✓	✓	x	✓	✓	x	✓	✓
		1–3	x	x	✓	x	✓	✓	x	✓	✓	✓	✓	✓
		1–4	x	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		1–5	x	✓	✓	✓	✓	✓	x	✓	✓	✓	✓	✓
		1–6	x	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		3–4	x	✓	✓	✓	✓	✓	✓	✓	✓	x	x	✓
		3–5	x	x	x	x	x	x	x	✓	x	x	x	✓
		3–6	x	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	x
		5–6	x	✓	✓	x	✓	✓	x	✓	✓	x	✓	✓

TABLE 6.7: Post-hoc test for performance differences between the six active learning approaches at file level ($1 : MDS\_Act, 2 : MDS\_rand, 3 : IG\_Act, 4 : IG\_rand, 5 : Act, 6 : Rand$). "✓" stands for statistically significant difference between two approaches. "x" stands for no signisficant difference detected between the two approaches.

Labeled	**Unlabeled**	**Exp.**	**File level**											
			Precision			Recall			ACC			AUC		
			20th	*30th*	*40th*	*20th*	*30th*	*40th*	*20th*	*30th*	*40th*	*20th*	*30th*	*40th*
Eclipse 2.0	Eclipse 2.1	1–2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		1–3	x	x	x	✓	✓	✓	✓	✓	✓	✓	✓	✓
		1–4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		1–5	x	x	x	✓	✓	✓	✓	✓	✓	✓	✓	✓
		1–6	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		3–4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		3–5	x	x	x	x	x	x	x	x	x	x	x	x
		3–6	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	x	x
		5–6	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	x	x
Eclipse 2.1	Eclipse 3.0	1–2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		1–3	✓	x	x	✓	✓	✓	✓	✓	✓	✓	✓	✓
		1–4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		1–5	✓	x	x	✓	✓	✓	✓	✓	✓	✓	✓	✓
		1–6	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		3–4	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	x	x
		3–5	x	x	x	x	x	x	x	x	x	✓	✓	✓
		3–6	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		5–6	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	✓
Eclipse 2.0 & 2.1	Eclipse 3.0	1–2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		1–3	x	x	x	✓	✓	✓	✓	✓	✓	✓	✓	✓
		1–4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		1–5	x	x	x	✓	✓	✓	✓	✓	✓	✓	✓	✓
		1–6	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		3–4	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	x	x
		3–5	x	x	x	x	x	x	x	x	x	x	x	✓
		3–6	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	x	x
		5–6	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	x	✓
Camel 1.2	Camel 1.4	1–2	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	x	✓
		1–3	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	✓
		1–4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		1–5	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	✓
		1–6	✓	✓	✓	x	✓	✓	✓	✓	✓	✓	✓	✓
		3–4	✓	✓	✓	x	x	✓	✓	✓	✓	x	x	x
		3–5	✓	x	x	✓	✓	✓	✓	✓	x	x	x	x
		3–6	✓	✓	✓	x	x	x	✓	✓	✓	x	x	✓
		5–6	✓	✓	✓	x	✓	✓	✓	✓	✓	x	x	✓
Camel 1.4	Camel 1.6	1–2	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	x	x
		1–3	x	✓	x	✓	✓	✓	✓	✓	✓	✓	✓	✓
		1–4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		1–5	x	x	x	✓	✓	✓	✓	✓	✓	✓	✓	✓
		1–6	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		3–4	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		3–5	x	x	x	x	x	x	x	✓	✓	x	✓	✓
		3–6	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		5–6	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Ant 1.3	Ant 1.4	1–2	✓	✓	✓	x	x	✓	✓	✓	✓	x	x	x
		1–3	x	x	x	x	x	✓	✓	x	✓	x	x	✓
		1–4	✓	✓	✓	x	x	✓	✓	✓	✓	x	x	x
		1–5	x	x	x	x	x	✓	✓	✓	✓	x	✓	✓
		1–6	✓	✓	✓	x	✓	✓	✓	✓	✓	x	x	x
		3–4	✓	✓	✓	x	x	x	✓	✓	✓	x	✓	✓
		3–5	x	x	x	x	x	x	x	x	x	x	x	x
		3–6	✓	✓	✓	x	x	x	✓	✓	✓	x	x	✓
		5–6	✓	✓	✓	x	x	x	✓	✓	✓	x	✓	✓
Ant 1.4	Ant 1.6	1–2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	x
		1–3	x	x	x	✓	✓	✓	✓	✓	✓	x	✓	✓
		1–4	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	✓
		1–5	x	x	x	✓	✓	✓	✓	✓	✓	✓	✓	✓
		1–6	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	x	✓
		3–4	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	x
		3–5	x	x	x	x	x	x	x	x	x	x	x	x
		3–6	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	x
		5–6	✓	✓	✓	✓	✓	✓	✓	✓	✓	x	✓	x

6.3 Threats to Validity

In this section we discuss the threats to validity of our study. We believe in the integrity of the data set used in this research, i.e., it honestly represents the defect content stemming from project development. However, as it is also the case with other empirical studies, unintentional noise and mistakes during data collection are out of our control. Dimensionality reduction techniques applied to software metrics prior to the active learning process overcome noise or outliers to a certain extent. However, if the data contains significant amounts of noise, internal validity may be compromised.

In addition, model parameters in active learning are set on the basis of the data size of four projects. For example, we fixed the growth size of training data set at a small amount, and we used the default settings in the random forest tool. Changes in these parameters may lead to biased outcomes. For those interested in replicating our work, we run all the experiments in R, version 2.15.1, with the RandomForest package.

Although we demonstrate the capability of active learning with dimensionality reduction on predicting software, like most empirical studies, our observations reflect the findings for release based data from three open-source projects. Software quality data from different applications may not achieve the same performance outcomes with active learning approaches. To test the robustness of active learning in release based defect prediction problem, it is necessary to extend our work on other release based software data sets.

Another possible validity threat pertains to the use of dimensionality reduction techniques. These techniques may be reflecting the characteristics of the specific software metrics used by the development team. The software metrics used in our study are complexity metrics. We cannot claim that dimensionality reduction applied to other software measures, such as OO metrics, process metrics or their combinations [54], will show similar outcomes.

6.4 Conclusions

We analyzed the performance of six active learning approaches on defect prediction between the successive releases of four projects. The prediction performance of active learning improves as the model adapts to the characteristics of the new release. Our experiments demonstrate that the guided selection of modules from the new release into the model training achieves better prediction performance than supervised learning. Given the same size of the training data set, the difference in performance is due to

uncertainty selection in active learning, as opposed to the random selection in supervised learning.

Improved performance through the inclusion of oracle-labeled instances from the new software release into the training data set comes at the expense of additional verification activities needed to identify the true defect content of the modules (which are selected via active learning) in the new software release. Our research offers evidence that significant performance advances can be observed in the early iterations of active learning.

We also demonstrate that feature selection techniques and dimensionality reduction techniques complement active learning for defect prediction. Active learning preceded by the dimensionality reduction algorithm MDS which uses Random Forest similarity measure on the software metrics (independent prediction variables) outperforms other active learning approaches. However, some experiments indicate that dimensionality reduction may not be appropriate with supervised learning in cases when major differences exist between successive software releases.

In the future, we plan to expand this research with the analysis of defect prediction performance on other software defect data sets and other software metrics. We want to further elaborate on the validity issues and possible limitations concerning active learning approaches. Active learning approaches exhibit great benefits for defect prediction and have the potential of being applicable to industrial practice. We believe that they will become tools of choice in practice, provided that they become supported by adequately automated tools.

Chapter 7

Summary and Future Work

7.1 Summary

Although there is diversity in the definition of software quality, it is widely acknowledged that software with many faults lacks quality. Accurate detection and removal of faulty modules ensure high quality software product. A low cost method to detect software fault proneness is to learn from past failures to prevent future ones. The assumption is that if certain types of software modules were likely to fail in the past they are also likely to do so in the future. Machine learning approaches are nature solutions to this problem.

Research efforts to predicting where faults are likely to hide have been substantial. Although a large number of machine learning based software fault prediction approaches have been investigated, none of them has proven to be consistently accurate. As we discussed in the Introduction section, the limitations stem from the lack of long fault history, failure in using appropriate predictive approaches, and low data quality.

In many real-world learning scenarios, acquiring a large amount of labeled training data is usually expensive and time-consuming. However, unlabeled data is a powerful resource and is easy to obtain. The key question is how to gather useful information out of unlabeled resources in a wide range of learning environments. In software fault prediction problem, it could be the situation of identifying fault prone modules when no previous subsystems or earlier versions are available for model training. In this dissertation we proposed machine learning solutions to address this problem and also discussed the efficacy of proposed approaches using multiple sources of software fault data. Two machine learning paradigms we studied are semi-supervised learning and active learning.

Semi-supervised learning is a principled framework for training predictive models using both labeled and unlabeled data. In many applications, it has been proved that semi-supervised learning performs better than supervised learning where there is only a small amount of labeled data. In the past decades, many semi-supervised learning approaches have been proposed. In this dissertation we focus on the iterative based semi-supervised learning approaches self-training.

In chapter 3 we implemented two variants of self-training approaches using software data sets from NASA projects repository. Both variants follow the similar learning procedure in which a supervised learner is repeatedly trained using newly labeled data. They differ in the way they adapt information for unlabeled data at each iteration. The supervised learner, also called base learner, could be any supervised learning algorithm. In our study we explored three base learners Logistic Regression, Support Vector Machine, and Random Forest. Convergence properties of both self-training approaches with different supervised learners were studied. Our results showed that the performance of self-training heavily relies on which base learner is used. We proved that when logistic regression is used as base learner, both self-training approaches converge to the supervised learning. In other word, the performance of semi-supervised learning is exactly equivalent to that of a supervised logistic regression.

When support vector machine is used as base learner, both self-training approaches are convergent too. However it is not certain that whether it converges to performing better or worse than traditional supervised learning. Our understanding is that the performance could be determined largely by the parameters and mapping function used in the support vector machine. In most cases, additional effort may be needed to achieve satisfied prediction performance. For example, one may need to seek for the optimal set of parameters needed in support vector machine by taking advantage of unlabeled data. Based on our experimental results, we observed that both self-training approaches perform worse than the corresponding supervised support vector machine. We are not surprised to see this. Support vector machine is designed to do classification by maximizing the hyper-plane among classes across low-density area. It lacks capability of handling imbalanced data or data with noises, that is, the situation when the low-density area is not clear or less reliable. However, we do believe that with well-balanced high-quality data set, self-training with support vector machine can be strong candidate in classification problem.

With random Forest as base learner, we observed that self-training is never convergent. We are unable to prove this theoretically, as there is uncontrolled randomness in the process of building random forests. Our experiments showed that self-training with

random forest outperformed supervised random forest. In addition, with random Forest as base learner, FTcF approach performs slightly better than FTF approach.

To answer the question that how small the labeled data could be for semi-supervised learning to outperform the corresponding supervised learning, we have explored the performance of both semi-supervised learning approaches FTF and FTcF approaches - by setting the initial labeled data at 2%, 5%, 10%, 20% and up to 50%. We demonstrated that semi-supervised learning with random forest starts exceeding the supervised learning approach at a low 5% initial labeled data. However, we recognized that the improvements by semi-supervised learning approaches are marginal.

In chapter 4 we proposed a new variant of semi-supervised learning approach, which incorporates a data pre-processing technique dimensionality reduction technique into the self-training approach. Dimensionality reduction is an important machine learning technique that helps to reduce the misleading in prediction that derives from noises, irrelevant or redundant predictors, or software metrics. It is a plausible addition to semi-supervised learning. When implementing dimensionality reduction technique we used random forest similarity as proximity matrix instead of using Euclidean metrics. We observed that random forest similarity captures proximity in a better manner compared to Euclidean metrics. Our results showed that the dimensional reduction based semi-supervised learning approach performed statistically significantly better than that without dimensionality reduction.

With the spirit of the semi-supervised learning, it is interested to investigate another machine learning paradigm - active learning. Similar to semi-supervised learning, active learning aims to train predictive model using both labeled and unlabeled data. In chapter 5 we proposed an adaptive fault prediction approach where active learning is the core. The prerequisite is a set of labeled data and at least one oracle. An oracle can be a software engineer or software developer who is able to interactively communicate with the learning machine and to label machine-selected modules as fault proneness or non-fault proneness. The key difference between active learning and semi-supervised learning is that the former requires the labors from oracle(s), while the latter is based upon an automatic learning process. With controlling the size of modules sending to an oracle, we observed that the active learning approach provides better performance and less variance than the corresponding supervised learning approach.

To test the validation of active learning approach on other data sets, in Chapter 6 we extended our study using version-based data sets. The intuition is that mistakes in software modules can be dynamic in successive software versions. For example, coding habit in one version may be largely changed in the next. Supervised learning, training model using only previous version(s), may lead biased prediction as it may learn from

outdated patterns. We thus proposed active learning approach for version-based software fault prediction. Our results on multiple data sources showed that active learning augmented by dimensionality reduction significantly outperformed the corresponding supervised learning by using only a small set of labeled data in current version together with labeled modules from previous version(s).

Below, we summarized the contribution of our studies in this dissertation:

- The investigation of two semi-supervised learning approaches for software fault prediction problem, particularly for the situation where software complexity metrics are abundant but fault history data is limited. This includes the analysis of convergence property for both semi-supervised learning when different base learners are used. We observed that semi-supervised learning perform pretty well when random forest is used as the base learner;
- The creation of an augmented semi-supervised learning in which dimensionality reduction technique is utilized on data prior to semi-supervised learning procedure. This includes the comparison of random forest similarity and Euclidean distance when both are used as the proximity matrix in dimensionality reduction technique. We learned that semi-supervised learning with dimensionality reduction statistically significantly outperforms supervised learning approach as well as the ones with no dimensionality reduction.
- The development of an adaptive approach for defect prediction with the core is an active learning. We showed that active learning approach could accommodate changing defect dynamics better than supervised learning approach using once-a-type projects. We observed that active learning approach achiever better performance in defect prediction and lower variance comparing to supervised learning. However we recognized that the improvement in prediction performance is not statistically significant.
- The implementation of dimensionality reduction based active learning using version based software data sets. This includes a thorough analysis of feature selection and dimensionality reduction techniques. We observed that dimensionality reduction based active learning tends to outperform feature selection based one, while feature selection based active learning approach performs no significantly better than that without data compression techniques.

7.2 Scope and Limitations

The scope of this dissertation is confined to the investigation of advanced machine learning approaches, more specifically semi-supervised learning and active learning, for identifying software fault prone modules when limited fault contents are available. The investigation of the presented approaches primarily focuses on open source domain projects, for example, the NASA projects or Eclipse defect data. Findings identified in this dissertation hold across the investigated projects. It may be possible to extend the findings of our study to similar software projects. However, further validations are necessary to help us draw stronger conclusions. Rather, we suggest that, when limited fault data are available, the semi-supervised learning or active learning approach could be a better choice compared to supervised learning approaches. However, we recognized that semi-supervised learning would not work well when base learner is not appropriately selected. Semi-supervised learning may lead to working the same or worse than supervised learning.

Due to the lack of information regarding when each module was generated, both of the semi-supervised learning and active learning in our studies assume that the order of software modules sequences generated in time is not a significant issue and has no effects on the performance of the learning procedure. That means each module is generated under the same environment without dramatic change in either the project requirement or the plan of system design. Practically, this assumption could be violated. This could threaten the validity of proposed approaches.

One may also concern the practical issues when proposed approaches are implemented in the real world. In order to be widely adopted, fault prediction approaches should be easy-to-use and applicable across different domain. However, both semi-supervised learning and active learning require extra efforts on selecting suitable base learner or tuning algorithmic parameters. Rather, software developer may also need to consider the trade-offs between 1) the growth size at each iteration and the run-time; 2) base learner and run-time; 3) dimensions to drop and run-time. Apparently, 2) is the key deciding the time complexity of the approaches. Figure 7.1 shows the time used when semi-supervised learning is conducted on PC3 project. Random forest is used as base learner. At each iteration, 10 modules are labeled and added into the labeled data set. Throughout the learning process, 10 dimensions extracted from original dimensional space are used for model training. The figure depicts that it requires twenty minutes for the semi-supervised learning to run 10^{th} iteration, when there are 2% initial labeled data. The time exponentially increases when the learning process continues. To reach at 50^{th} iteration, semi-supervised learning costs more than one hour. Less time is consumed if more data is initially labeled. Table 7.1 shows the association between the number

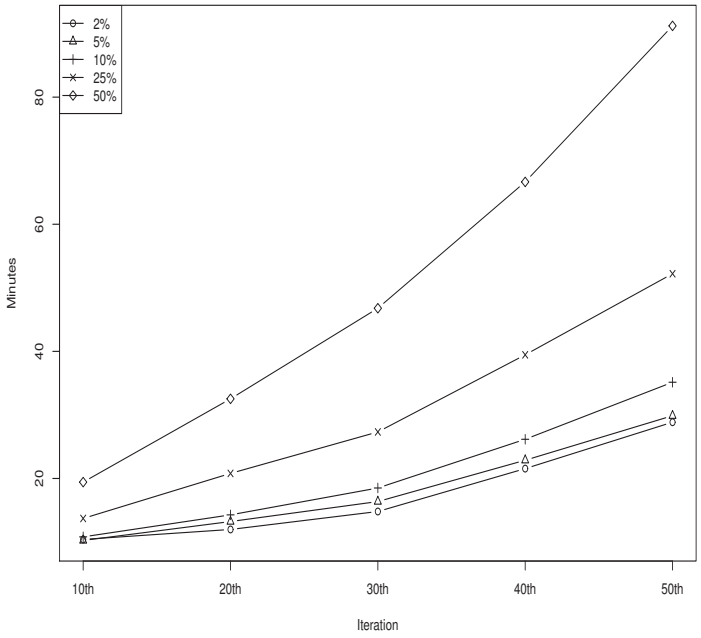

FIGURE 7.1: Run-time (minutes) of semi-supervised learning for PC3 data set.

TABLE 7.1: Run-time (minutes) of semi-supervised learning when random forest is used as base learner

Data sets	Iteration	2%	5%	10%	25%	50%
PC1	iter10	4.566	5.033	5.639	7.448	11.113
	iter20	6.508	8.197	9.052	12.248	20.015
	iter30	9.529	10.424	12.063	18.143	30.236
	iter40	14.619	15.892	17.871	27.056	43.819
	iter50	21.845	22.949	25.826	37.245	59.939
PC3	10th	10.415	10.249	10.815	13.711	19.427
	20th	11.978	13.226	14.276	20.798	32.531
	30th	14.804	16.371	18.505	27.322	46.786
	40th	21.553	22.872	26.167	39.456	66.659
	50th	28.855	29.874	35.141	52.217	91.212
PC4	10th	8.029	8.61	9.116	11.769	15.63
	20th	10.05	11.341	12.689	17.764	25.91
	30th	13.378	14.699	16.751	24.388	39.655
	40th	21.467	21.809	24.805	35.279	57.465
	50th	26.433	28.522	34.027	50.638	76.507
KC1	10th	18.504	19.684	20.093	23.723	32.08
	20th	21.072	21.859	24.393	31.588	50.108
	30th	24.143	25.45	29.91	40.542	68.609
	40th	32.048	35.588	39.574	55.147	95.12
	50th	37.916	41.65	47.77	71.61	122.079

of iteration and the run-time for semi-supervised learning across four NASA projects. For active learning, it may require longer run-time due to the additional idle time when waiting for response from oracle.

7.3 Future Work

This dissertation showed that semi-supervised learning and active learning approaches have tremendous practical value in identifying fault prone modules, especially when prior knowledge of fault content is limited. Software managers or software engineers can take our approaches as alternatives when they tackle difficult tasks of software quality prediction problems in real-world practice. However, we understand that there remains more work for us to do in the future.

First, to validate proposed approaches, it is meaningful to expand our work to more data sets from different data sources. This includes the validation study using different software metrics, other than static metrics, for example the process metrics and dynamic metrics.

The approaches investigated in our study are the simplest forms of some widely used ones. Different types of semi-supervised learning and active learning along with their improved versions can be examined in the future. Despite that sophisticated learning approaches do not always work better than simple ones, it is always worthwhile to compare the approaches in the market and find the ones we can benefit the most from for software fault prediction. One direction of the future work can be investigating other type of semi-supervised learning, such as clustering based semi-supervised learning which extending unsupervised learning to semi-supervised learning, or graph-based semi-supervised learning that applies structural software metrics.

It is not uncommon that researchers build applicable tools to transfer their knowledge. For example, R is an open source with tons of free build-in packages contributed by people all around the world. To our knowledge, majority of the learning algorithms included in R packages are either supervised learning based or unsupervised learning based. R packages that provide function of semi-supervised learning approaches are very scarce. One future direction can be building software packages or tools for semi-supervised learning or active learning, so our research can be available to other machine learning practitioners.

Appendix A

Proof of convergence on FTF with LR

In this section, we will prove that FTF algorithm will converge to supervised learning with logistic regression as base learner.

Proof: For logistic regression, we have logit function: $logit(p_i) = x_i^T\beta$, with $p_i = P(y_i = 1|x_i)$, where $y_i \in (0, 1)$. To have the optimal estimation, we can use MLE method to minimize the likelihood function of β ($L(\beta)$). Then the log-likelihood function will be: $l(\beta) = logL(\beta) = \sum_{i=1}^{n}[y_i log(p_i) + (1 - y_i)log(1 - p_i)]$. Take the derivation of $l(\beta)$, we will have:

$$\frac{\partial l(\beta)}{\partial \beta} = \sum_{i=1}^{n} x_i(y_i - p_i) = X^T(Y - P). \tag{A.1}$$

To solve equation $X^T(Y - P) = \vec{0}$, we can use Newton's Raphson method. By some mathematical calculation, we will have the supervised logistic regression estimates $\widetilde{\beta}^{log} = (X_l^T W_l X_l)^{-1} X_l^T W_l Z$, where $Z = X\beta_{old} + W^{-1}(Y - P)$ and W is a diagonal matrix with $w_{ii} = p_i(1 - p_i)$. Therefore, given labeled training modules X_l, the prediction will be $\hat{Y}_l = p_l^{log} = P(X_l, \widetilde{\beta}^{log})$.

In FTF algorithm, remind that there are two main steps at each iteration:
i) $\hat{Y}_l^k = Y_l$
ii)$\hat{Y}_u^k = P(X_u, \hat{\beta^{k-1}}) = p_u^{k-1}$.

For k^{th} iteration, equation (A.1) will have the form:

$$\begin{aligned} X^T(Y-P) &= \begin{pmatrix} X_l^T & X_u^T \end{pmatrix} \left[\begin{pmatrix} Y_l \\ p_u^{k-1} \end{pmatrix} - \begin{pmatrix} p_l^k \\ p_u^k \end{pmatrix} \right] \\ &= X_l^T(p_l^{log} - p_l^k) + X_u^T(p_u^{k-1} - p_u^k) \\ &= \vec{0} \end{aligned} \tag{A.2}$$

To solve above equation, we will use Taylor expansion techniques. The detailed calculation is shown as following:

$$\begin{aligned} & X_l^T(p_l^{log} - p_l^k) + X_u^T(p_u^{k-1} - p_u^k) \\ & = X_l^T(p(X_l, \widetilde{\beta}^{log}) - p(X_l, \hat{\beta}^k)) \\ & + X_u^T(p(X_u, \hat{\beta}^{k-1}) - p(X_u, \hat{\beta}^k)) \\ & \cong X_l^T(p(X_l, \widetilde{\beta}^{log}) - p(X_l, \widetilde{\beta}^{log}) - \frac{\partial p(X_l, \widetilde{\beta}^{log})}{\partial \widetilde{\beta}^{log}}(\hat{\beta}^k - \widetilde{\beta}^{log})) \\ & + X_u^T(p(X_u, \widetilde{\beta}^{log}) + \frac{\partial p(X_u, \widetilde{\beta}^{log})}{\partial \widetilde{\beta}^{log}}(\hat{\beta}^{k-1} - \widetilde{\beta}^{log}) \\ & - p(X_u, \widetilde{\beta}^{log}) + \frac{\partial p(X_u, \widetilde{\beta}^{log})}{\partial \widetilde{\beta}^{log}}(\hat{\beta}^k - \widetilde{\beta}^{log})) \\ & = X_l^T(\frac{\partial p(X_l, \widetilde{\beta}^{log})}{\partial \widetilde{\beta}^{log}}(\hat{\beta}^k - \widetilde{\beta}^{log})) \\ & + X_u^T(\frac{\partial p(X_u, \widetilde{\beta}^{log})}{\partial \widetilde{\beta}^{log}}(\hat{\beta}^k - \widetilde{\beta}^{log}) \\ & - \frac{\partial p(X_u, \widetilde{\beta}^{log})}{\partial \widetilde{\beta}^{log}}(\hat{\beta}^{k-1} - \widetilde{\beta}^{log})) \\ & = X_l^T W_l X_l(\hat{\beta}^k - \widetilde{\beta}^{log}) + X_u^T W_u X_u(\hat{\beta}^k - \beta^{k-1}) \\ & = \vec{0} \end{aligned} \tag{A.3}$$

After translation, above equation can be written to be:

$$\begin{aligned} & \hat{\beta}^k - \widetilde{\beta}^{log} = \left[(X^T W X)^{-1}(X_u^T W_u X_u)\right](\hat{\beta}^{k-1} - \widetilde{\beta}^{log}) \\ & \Rightarrow \hat{\beta}^k = \widetilde{\beta}^{log} + \left[(X^T W X)^{-1}(X_u^T W_u X_u)\right]^k(\hat{\beta}^0 - \widetilde{\beta}^{log}) \end{aligned} \tag{A.4}$$

Since X_u is contained in the space spanned by X, we get that $X_u^T W_u X_u < X^T W X$. In the loewner ordering, it implies that, $(X^T W X)^{-1}(X_u^T W_u X_u) < I$, and therefore, the radial spectrum of the left hand side is less than 1 (i.e. $\rho((X^T W X)^{-1}(X_u^T W_u X_u)) < 1$).

Thus, $\left[(X^T W X)^{-1}(X_u^T W_u X_u)\right] = \overrightarrow{0}$ when $k \to \infty$. Finally we can show $\hat{\beta}^k \approx \widetilde{\beta}^{log}$ when $k \to \infty$.

Appendix B

Proof of convergence on FTcF with LR

In this section, we will prove that FTcF algorithm will converge to supervised learning with logistic regression as base learner.

Proof:In FTcF algorithm, at $0th$ iteration, we will have:

$$X_l^{(0)} = X_l, \hat{Y}_l^{(0)} = Y_l, p_l^{(0)} = P(X_l^T, \hat{\beta}_0)$$

and at kth itheration($k \neq 0$), we will have:

$$\begin{aligned} X_l^{(k)} &= \begin{pmatrix} X_l^{(k-1)} \\ X_{l_k} \end{pmatrix}, \\ \hat{Y}_l^{(k)} &= \begin{pmatrix} \hat{Y}_l^{(k-1)} \\ P(X_{l_k}^T, \hat{\beta}_{k-1}) \end{pmatrix}, \\ p_l^{(k)} &= \left(P(X_l^{T(k)}, \hat{\beta}_k) \right) \end{aligned}$$

Here, k is the number of iteration, l_k is the confident samples that added to the labeled data set at k^{th} iteration. Now, we want to estimate $\hat{\beta}^k$ by solving $X^T(Y - P) = \vec{0}$ at k^{th} iteration. We've already have $\hat{\beta}_0 = \widetilde{\beta}^{log}$, since at 0^{th} iteration, it is just a supervised learning.

For 1^{st} iteration, we will have:

$$
\begin{aligned}
& X_l^{T(1)}(\hat{Y}_l^{(1)} - p_l^{(1)}) \\
& = (X_l^T \quad X_{l1}^T) \left[\begin{pmatrix} Y_l \\ P(X_{l1}^T, \hat{\beta}_0) \end{pmatrix} - \begin{pmatrix} P(X_l^T, \hat{\beta}_1) \\ P(X_{l1}^T, \hat{\beta}_1) \end{pmatrix} \right] \\
& = X_l^T(Y_l - P(X_l^T, \hat{\beta}_1)) + X_{l1}^T(P(X_{l1}^T, \hat{\beta}_0) - P(X_{l1}^T, \hat{\beta}_1)) \\
& = \vec{0}
\end{aligned}
\tag{B.1}
$$

We assume that the solution of equation (B.1), $\hat{\beta}_1$, is unique. The first term of the equation is $X_l^T(Y_l - P(X_l^T, \hat{\beta}_1)) = \vec{0}$, whenever $\hat{\beta}_1 = \widetilde{\beta}^{log} = \hat{\beta}_0$ by the definition of supervised learning. In this case, the second term will be $X_{l1}^T(P(X_{l1}^T, \hat{\beta}_0) - P(X_{l1}^T, \hat{\beta}_0)) = \vec{0}$, which causes $X_l^{T(1)}(\hat{Y}_l^{(1)} - p_l^{(1)}) = X_l^T(Y_l - P(X_l^T, \hat{\beta}_1)) - X_{l1}^T(P(X_{l1}^T, \hat{\beta}_0) - P(X_{l1}^T, \hat{\beta}_1)) = \vec{0} + \vec{0} = \vec{0}$. Therefore, $\hat{\beta}_1 = \hat{\beta}_0$ is the solution of 1^{st} iteration.

For 2^{nd} iteration, we will have:

$$
\begin{aligned}
& X_l^{T(2)}(\hat{Y}_l^{(2)} - p_l^{(2)}) \\
& = (X_l^{T(1)} \quad X_{l2}^T) \left[\begin{pmatrix} \hat{Y}_l^{(1)} \\ P(X_{l2}^T, \hat{\beta}_{(1)}) \end{pmatrix} - \begin{pmatrix} P(X_l^{T(1)}, \hat{\beta}_2) \\ P(X_{l2}^T, \hat{\beta}_2) \end{pmatrix} \right] \\
& = (X_l^T \quad X_{l1}^T \quad X_{l2}^T) \left[\begin{pmatrix} Y_l \\ P(X_{l1}^T, \hat{\beta}_0) \\ P(X_{l2}^T, \hat{\beta}_1) \end{pmatrix} - \begin{pmatrix} P(X_l^T, \hat{\beta}_2) \\ P(X_{l1}^T, \hat{\beta}_2) \\ P(X_{l2}^T, \hat{\beta}_2) \end{pmatrix} \right] \\
& = X_l^T(P(X_l^T, \hat{\beta}_0) - P(X_l^T, \hat{\beta}_2)) \\
& + X_{l1}^T(P(X_{l1}^T, \hat{\beta}_0) - P(X_{l1}^T, \hat{\beta}_2)) \\
& + X_{l2}^T(P(X_{l2}^T, \hat{\beta}_1) - P(X_{l2}^T, \hat{\beta}_2))
\end{aligned}
\tag{B.2}
$$

From 1^{st} iteration, we've have $\hat{\beta}_1 = \hat{\beta}_0$, it leaves equation (B.2) to be:

$$
\begin{aligned}
& X_l^{T(2)}(\hat{Y}_l^{(2)} - p_l^{(2)}) \\
& = X_l^T(P(X_l^T, \hat{\beta}_0) - P(X_l^T, \hat{\beta}_2)) \\
& + X_{l1}^T(P(X_{l1}^T, \hat{\beta}_0) - P(X_{l1}^T, \hat{\beta}_2)) \\
& + X_{l2}^T(P(X_{l2}^T, \hat{\beta}_0) - P(X_{l2}^T, \hat{\beta}_2)) = \vec{0}
\end{aligned}
\tag{B.3}
$$

It is apparently that the solution of equation (B.3) is $\hat{\beta}_2 = \hat{\beta}_0$.

For generalizaiton, at k^{th} iteration, assume $\hat{\beta}_0 = \hat{\beta}_1 = \hat{\beta}_2 = \cdots = \hat{\beta}_{k-1}$, we will have:

$$
\begin{aligned}
& X_{(k)}^T(\hat{Y}^{(k)} - p^{(k)}) \\
& = (X_{(k-1)}^T \quad X_{lk}^T) \left[\begin{pmatrix} \hat{Y}^{(k-1)} \\ P(X_{lk}^T, \hat{\beta}_{(k-1)}) \end{pmatrix} - \begin{pmatrix} P(X_{(k-1)}^T, \hat{\beta}_k) \\ P(X_{lk}^T, \hat{\beta}_k) \end{pmatrix} \right] \\
& = (X_l^T \quad X_{l'}^T \quad X_{lk}^T) \left[\begin{pmatrix} P(X_l^T, \hat{\beta}_0) \\ P(X_{l'}^T, \hat{\beta}_0) \\ P(X_{lk}^T, \hat{\beta}_0) \end{pmatrix} - \begin{pmatrix} P(X_l^T, \hat{\beta}_k) \\ P(X_{l'}^T, \hat{\beta}_k) \\ P(X_{lk}^T, \hat{\beta}_k) \end{pmatrix} \right] \\
& = X_l^T(P(X_l^T, \hat{\beta}_0) - P(X_l^T, \hat{\beta}_k)) \\
& + X_{l'}^T(P(X_{l'}^T, \hat{\beta}_0) - P(X_{l'}^T, \hat{\beta}_k)) \\
& + X_{lk}^T(P(X_{lk}^T, \hat{\beta}_0) - P(X_{lk}^T, \hat{\beta}_k)) \\
& = Sup(X_l^T) + Sup(X_{l'}^T) + Sup(X_{lk}^T)
\end{aligned}
\tag{B.4}
$$

which shows that $\hat{\beta}_0 = \hat{\beta}_1 = \hat{\beta}_2 = \cdots = \hat{\beta}_k = \tilde{\beta}^{log}$ for any given iteration as long as the logistic linear regression exists.

Appendix C

Proof of convergence on FTF with SVM

In this section, we will analyze the convergence property of semi-supervised learning when support vector machine is used as base learner. A standard SVM algorithm for binary class problem can be defined as:

$$\begin{aligned} \min_{w,b,\xi} \quad & \frac{1}{2}||w||^2 + C\sum_{i=1}^{n}\xi_i \\ & y_i(w^T\phi(x_i)+b) \geq 1-\xi_i, \\ & \xi_i \geq 0, i = 1,\ldots,n. \end{aligned} \tag{C.1}$$

where $C > 0$ is a regularization constant. In the first step of FTF algorithm, y_i are the true initial labels for labeled data X_l, i.e., $Y_l = [y_1, y_2, ..., y_n]^T$ and $x_i \in R^n$. As for the iterative steps - step 2-6 in FIGURE 3.1, the FTF algorithm repeatedly solves the same optimization problem, but y_i are the true initial labels for X_l together with the predicted labels for X_u, i.e., $Y^{(k-1)} = [y_1, ..., y_n, \hat{y}_{n+1}^{(k-1)}, ..., \hat{y}_{n+m}^{(k-1)}]^T$ and $x_i \in R^{n+m}$. Here, k stands for the k^{th} iteration. To show the convergence property of SVM in FTF algorithm, we write the objective function in the optimization problem at k^{th} iteration as:

$$f(w^{(k)}, \xi^{(k)}) = \frac{1}{2}||w^{(k)}||^2 + C\sum_{i=1}^{n+m}\xi_i^{(k)} \tag{C.2}$$

According to the equation C.1, we can obtain the optimal solution $\{w^{(0)}, \xi^{(0)}, b^{(0)}\}$ using only labeled data. To have the prediction $\hat{Y}_u^{(0)}$, we can solve the following inequalities:

$$y_i^{(0)}((w^{(0)})^T\phi(x_i) + b^{(0)}) \geq 0, i = n+1, \ldots, n+m. \tag{C.3}$$

This says that, given an unlabeled input x_i, if $((w^{(0)})^T\phi(x_i) + b^{(0)}) \geq 0$ then $\hat{y}_i^{(0)} = 1$, otherwise $\hat{y}_i^{(0)} = -1$. It equals $\hat{y}_i^{(0)} = sign((w^{(0)})^T\phi(x_i) + b^{(0)})$. To expand the vector $\xi^{(0)}$ to be the length of $n+m$, we can define:

$$\xi_i^{(0)} = \begin{cases} 0 & \text{if } y_i^{(0)}((w^{(0)})^T\phi(x_i) + b^{(0)}) \geq 1 \\ 1 - y_i^{(0)}((w^{(0)})^T\phi(x_i) + b^{(0)}) & \text{otherwise} \end{cases} \tag{C.4}$$

where $i = n+1, ..., n+m$. This complies with the definition of SVM.

At the first iteration in FTF algorithm, we need to solve the following optimization problem:

$$\begin{aligned} \min_{w,b,\xi} \quad & \frac{1}{2}||w^{(1)}||^2 + C\sum_{i=1}^{n+m}\xi_i^{(1)} \\ & y_i^{(0)}((w^{(1)})^T\phi(x_i) + b^{(1)}) \geq 1 - \xi_i^{(1)}, \\ & \xi_i^{(1)} \geq 0, i = 1, \ldots, n+m. \end{aligned} \tag{C.5}$$

which provides the optimal solution $\{w^{(1)}, \xi^{(1)}, b^{(1)}\}$. Learned from equation C.3 and C.4, we know that $\{w^{(0)}, \xi^{(0)}, b^{(0)}\}$ is a feasible solution of equation C.5. We concluded it from: 1) for $i = 1, ..., n$, $\xi_i^{(0)} \geq 0$ and $y_i((w^{(0)})^T\phi(x_i) + b^{(0)}) \geq 1 - \xi_i^{(0)}$; 2) for $i = n+1, ..., n+m$, the same constraints hold due to the prediction rule (equation C.3) and the expanded $\xi_i^{(0)}$ (equation C.4). Thus, we have:

$$f(w^{(0)}, \xi^{(0)}) \geq f(w^{(1)}, \xi^{(1)}) \tag{C.6}$$

Now, let us look at the solution when $k > 1$. With $k > 1$, we can have the $\{w^{(k-1)}, \xi^{(k-1)}, b^{(k-1)}\}$ as the optimal solution of following optimization problem:

$$
\begin{aligned}
s\min_{w,b,\xi} \quad & \frac{1}{2}||w^{(k-1)}||^2 + C\sum_{i=1}^{n+m}\xi_i^{(k-1)} \\
& y_i^{(k-2)}((w^{(k-1)})^T\phi(x_i) + b^{(k-1)}) \geq 1 - \xi_i^{(k-1)}, \\
& \xi_i^{(k-1)} \geq 0, i = 1, \ldots, n+m.
\end{aligned} \tag{C.7}
$$

where the vector $Y^{(k-2)} = [y_1, ..., y_n, \hat{y}_{n+1}^{(k-2)}, ..., \hat{y}_{n+m}^{(k-2)}]^T$. Similarly, $\{w^{(k)}, \xi^{(k)}, b^{(k)}\}$ is the optimal solution of following optimization problem:

$$
\begin{aligned}
\min_{w,b,\xi} \quad & \frac{1}{2}||w^{(k)}||^2 + C\sum_{i=1}^{n+m}\xi_i^{(k)} \\
& y_i^{(k-1)}((w^{(k)})^T\phi(x_i) + b^{(k)}) \geq 1 - \xi_i^{(k)}, \\
& \xi_i^{(k)} \geq 0, i = 1, \ldots, n+m.
\end{aligned} \tag{C.8}
$$

where the vector $Y^{(k-1)} = [y_1, ..., y_n, \hat{y}_{n+1}^{(k-1)}, ..., \hat{y}_{n+m}^{(k-1)}]^T$.

We note that there are two cases between any two iterations ($k > 1$):

1) If $y_i^{(k-1)} = y_i^{(k-2)}$, i.e., predictions at $(k-1)^{th}$ are the same as those from previous iteration. This always holds for $i = 1, ..., n$. From the constraint of equation C.7,

$$
\begin{aligned}
y_i^{(k-1)}((w^{(k-1)})^T\phi(x_i) + b^{(k-1)}) &= y_i^{(k-2)}((w^{(k-1)})^T\phi(x_i) + b^{(k-1)}) \\
&\geq 1 - \xi_i^{(k-1)}, \xi_i^{(k-1)} \geq 0
\end{aligned} \tag{C.9}
$$

2) If $y_i^{(k-1)} \neq y_i^{(k-2)}$, i.e., predictions at $(k-1)^{th}$ are not the same as those from previous iteration. This may hold for $i = n+1, ..., n+m$. We know that $y_i^{(k-1)}((w^{(k-1)})^T\phi(x_i) + b^{(k-1)}) \geq 0$ for $i = n+1, ..., n+m$ based on the definition of $y_i^{(k-1)}$. Here, $y_i^{(k-1)}$ can be written as $\hat{y}_i^{(k-1)}$ and $\hat{y}_i^{(k-1)} = sign((w^{(k-1)})^T\phi(x_i) + b^{(k-1)})$.

Thus, we can have:

$$
y_i^{(k-2)}((w^{(k-1)})^T\phi(x_i) + b^{(k-1)}) \leq 0 \tag{C.10}
$$

This implies that:

$$y_i^{(k-1)}((w^{(k-1)})^T\phi(x_i)+b^{(k-1)}) > y_i^{(k-2)}((w^{(k-1)})^T\phi(x_i)+b^{(k-1)}) \geq 1-\xi_i^{(k-1)}, \xi_i^{(k-1)} \geq 0 \quad \text{(C.11)}$$

Until now, we have showed that $\{w^{(k-1)}, \xi^{(k-1)}, b^{(k-1)}\}$ satisfies the constraints of equation C.8. This says that $\{w^{(k-1)}, \xi^{(k-1)}, b^{(k-1)}\}$ is a feasible solution of the optimization problem in equation C.8. Noting that $\{w^{(k)}, \xi^{(k)}, b^{(k)}\}$ is the optimal solution of C.8, hence we have the inequality of

$$f(w^{(k-1)}, \xi^{(k-1)}) \geq f(w^{(k)}, \xi^{(k)}) \quad \text{(C.12)}$$

Incorporating equation C.6, we can conclude that C.12 holds for any given $k \geq 1$, i.e., $f(w^{(k)}, \xi^{(k)})$ is a monotonic decreasing function, where $k = 1, ..., +\infty$. Since $f(w^{(k)}, \xi^{(k)}) \geq 0$, we proved that $f(w^{(k)}, \xi^{(k)})$ is convergent when k increases. Thus FTF algorithm is convergent.

Appendix D

Tables of Performance Comparison for Eclipse data sets

In this section, we present the tables of performance comparison for Eclipse data sets between successive releases which correspond to the figures from 6.4 to 6.9.

TABLE D.1: Performance comparison, from release 2.0 to 2.1 for Eclipse packages

Measurement	Iteration	Rand	Act	IG_Rand	IG_Act	MDS_Rand	MDS_Act
Precision	1st	0.717	0.717	0.748	0.748	0.706	0.706
	10th	0.751	0.779	0.772	0.789	0.805	0.812
	20th	0.778	0.812	0.797	0.836	0.823	0.853
	30th	0.805	0.849	0.819	0.868	0.839	0.894
Recall	1st	0.741	0.741	0.744	0.744	0.737	0.737
	10th	0.764	0.785	0.758	0.788	0.765	0.818
	20th	0.786	0.844	0.777	0.824	0.823	0.882
	30th	0.815	0.861	0.797	0.845	0.853	0.924
Accuracy	1st	0.753	0.753	0.774	0.774	0.746	0.746
	10th	0.781	0.804	0.791	0.811	0.81	0.832
	20th	0.804	0.843	0.812	0.849	0.84	0.878
	30th	0.829	0.87	0.831	0.874	0.86	0.916
AUC	1st	0.822	0.822	0.831	0.831	0.805	0.805
	10th	0.847	0.848	0.855	0.86	0.879	0.891
	20th	0.871	0.87	0.879	0.879	0.922	0.93
	30th	0.891	0.884	0.898	0.895	0.939	0.947

TABLE D.2: Performance comparison, from release 2.1 to 3.0 for Eclipse packages.

Measurement	Iteration	Rand	Act	IG_Rand	IG_Act	MDS_Rand	MDS_Act
Precision	1st	0.852	0.852	0.838	0.838	0.705	0.705
	10th	0.865	0.881	0.851	0.878	0.857	0.869
	20th	0.884	0.911	0.873	0.916	0.876	0.91
	30th	0.895	0.928	0.88	0.949	0.891	0.947
Recall	1st	0.686	0.686	0.685	0.685	0.612	0.612
	10th	0.729	0.764	0.735	0.756	0.846	0.871
	20th	0.756	0.82	0.759	0.834	0.864	0.928
	30th	0.792	0.88	0.793	0.87	0.883	0.949
Accuracy	1st	0.795	0.795	0.788	0.788	0.692	0.692
	10th	0.818	0.839	0.813	0.835	0.858	0.876
	20th	0.837	0.877	0.834	0.885	0.877	0.922
	30th	0.857	0.911	0.851	0.917	0.893	0.951
AUC	1st	0.859	0.859	0.861	0.861	0.761	0.761
	10th	0.882	0.884	0.885	0.888	0.931	0.927
	20th	0.906	0.913	0.907	0.907	0.949	0.952
	30th	0.925	0.932	0.924	0.92	0.962	0.967

TABLE D.3: Performance comparison, from releases 2.0 and 2.1 to 3.0 for Eclipse packages.

Measurement	Iteration	Rand	Act	IG_Rand	IG_Act	MDS_Rand	MDS_Act
Precision	1st	0.854	0.854	0.848	0.848	0.644	0.644
	10th	0.865	0.898	0.863	0.878	0.817	0.849
	20th	0.885	0.913	0.873	0.911	0.861	0.899
	30th	0.9	0.923	0.888	0.938	0.876	0.949
Recall	1st	0.694	0.694	0.692	0.692	0.57	0.57
	10th	0.724	0.75	0.717	0.761	0.82	0.782
	20th	0.76	0.807	0.765	0.837	0.858	0.886
	30th	0.798	0.871	0.799	0.888	0.89	0.941
Accuracy	1st	0.799	0.799	0.795	0.795	0.642	0.642
	10th	0.816	0.841	0.812	0.837	0.826	0.834
	20th	0.84	0.872	0.836	0.884	0.867	0.9
	30th	0.862	0.905	0.857	0.919	0.888	0.948
AUC	1st	0.855	0.855	0.871	0.871	0.687	0.687
	10th	0.876	0.889	0.891	0.896	0.916	0.882
	20th	0.901	0.913	0.914	0.916	0.945	0.943
	30th	0.922	0.933	0.931	0.93	0.959	0.97

TABLE D.4: Performance comparison, from release 2.0 to 2.1 for Eclipse files.

Measurement	Iteration	Rand	Act	IG_Rand	IG_Act	MDS_Rand	MDS_Act
Precision	1st	0.361	0.361	0.296	0.363	0.408	0.633
	10th	0.445	0.727	0.398	0.745	0.553	0.85
	20th	0.531	0.923	0.49	0.948	0.688	0.948
	30th	0.613	0.974	0.615	0.994	0.744	0.983
Recall	1st	0.244	0.244	0.161	0.221	0.001	0.001
	10th	0.302	0.377	0.23	0.364	0.227	0.611
	20th	0.361	0.524	0.272	0.507	0.313	0.808
	30th	0.423	0.656	0.344	0.644	0.418	0.889
Accuracy	1st	0.871	0.871	0.865	0.874	0.891	0.891
	10th	0.884	0.917	0.877	0.918	0.896	0.946
	20th	0.896	0.944	0.888	0.944	0.91	0.974
	30th	0.909	0.961	0.904	0.961	0.921	0.986
AUC	1st	0.756	0.756	0.713	0.757	0.309	0.605
	10th	0.787	0.782	0.748	0.783	0.769	0.899
	20th	0.818	0.81	0.785	0.809	0.822	0.937
	30th	0.847	0.84	0.816	0.838	0.86	0.957

Table D.5: Performance comparison, from release 2.1 to 3.0 for Eclipse files.

Measurement	Iteration	Rand	Act	IG_Rand	IG_Act	MDS_Rand	MDS_Act
Precision	1st	0.584	0.584	0.652	0.601	0.095	0.8
	10th	0.72	0.952	0.735	0.948	0.622	0.768
	20th	0.796	0.993	0.815	0.987	0.705	0.922
	30th	0.847	0.994	0.86	0.993	0.761	0.978
Recall	1st	0.136	0.136	0.15	0.158	0.001	0
	10th	0.227	0.342	0.243	0.345	0.301	0.64
	20th	0.312	0.552	0.326	0.559	0.399	0.828
	30th	0.399	0.683	0.422	0.661	0.48	0.906
Accuracy	1st	0.858	0.858	0.858	0.86	0.848	0.852
	10th	0.873	0.9	0.871	0.9	0.869	0.917
	20th	0.886	0.933	0.886	0.934	0.886	0.964
	30th	0.9	0.952	0.901	0.949	0.9	0.983
AUC	1st	0.779	0.779	0.766	0.778	0.259	0.522
	10th	0.818	0.805	0.801	0.802	0.788	0.893
	20th	0.848	0.841	0.832	0.83	0.836	0.94
	30th	0.874	0.867	0.859	0.855	0.867	0.964

Table D.6: Performance comparison, from releases 2.0 and 2.1 to 3.0 for Eclipse files.

Measurement	Iteration	Rand	Act	IG_Rand	IG_Act	MDS_Rand	MDS_Act
Precision	1st	0.421	0.421	0.57	0.57	0.4	0.481
	10th	0.54	0.936	0.692	0.873	0.769	0.905
	20th	0.677	0.985	0.76	0.941	0.817	0.972
	30th	0.797	0.988	0.826	0.992	0.874	0.993
Recall	1st	0.188	0.188	0.224	0.224	0	0.014
	10th	0.245	0.276	0.294	0.34	0.132	0.443
	20th	0.298	0.439	0.369	0.49	0.295	0.753
	30th	0.4	0.566	0.438	0.599	0.388	0.866
Accuracy	1st	0.851	0.851	0.86	0.86	0.871	0.876
	10th	0.866	0.897	0.876	0.895	0.886	0.925
	20th	0.883	0.921	0.889	0.92	0.903	0.967
	30th	0.902	0.939	0.903	0.94	0.916	0.983
AUC	1st	0.719	0.719	0.746	0.746	0.537	0.709
	10th	0.758	0.751	0.787	0.771	0.745	0.829
	20th	0.792	0.785	0.827	0.802	0.843	0.91
	30th	0.837	0.815	0.852	0.836	0.882	0.942

Table D.7: Performance comparison, from release 1.2 to 1.4 for Camel.

Measurement	Iteration	Rand	Act	IG_Rand	IG_Act	MDS_Rand	MDS_Act
Precision	1st	0.257	0.257	0.358	0.358	0.415	0.415
	10th	0.856	0.698	0.455	0.411	0.504	0.455
	20th	0.950	0.785	0.586	0.465	0.621	0.503
	40th	0.982	0.848	0.854	0.583	0.861	0.615
Recall	1st	0.168	0.168	0.548	0.548	0.652	0.652
	10th	0.475	0.503	0.598	0.579	0.688	0.654
	20th	0.699	0.567	0.643	0.622	0.728	0.675
	40th	0.904	0.703	0.735	0.688	0.777	0.728
Accuracy	1st	0.786	0.786	0.762	0.762	0.789	0.789
	10th	0.897	0.879	0.814	0.792	0.835	0.812
	20th	0.944	0.901	0.865	0.818	0.881	0.835
	40th	0.981	0.930	0.935	0.866	0.942	0.879
AUC	1st	0.669	0.669	0.756	0.756	0.775	0.775
	10th	0.871	0.876	0.802	0.794	0.812	0.809
	20th	0.910	0.914	0.844	0.832	0.850	0.844
	40th	0.964	0.953	0.886	0.894	0.883	0.903

Table D.8: Performance comparison, from release 1.4 to 1.6 for Camel.

Measurement	Iteration	Rand	Act	IG_Rand	IG_Act	MDS_Rand	MDS_Act
Precision	1st	0.625	0.625	0.535	0.535	0.491	0.491
	10th	0.836	0.737	0.754	0.598	0.768	0.567
	20th	0.922	0.770	0.928	0.655	0.926	0.639
	40th	0.977	0.830	0.951	0.760	0.985	0.772
Recall	1st	0.023	0.023	0.248	0.248	0.269	0.269
	10th	0.450	0.391	0.354	0.289	0.377	0.314
	20th	0.680	0.549	0.447	0.343	0.474	0.372
	40th	0.895	0.694	0.596	0.446	0.615	0.480
Accuracy	1st	0.807	0.807	0.812	0.812	0.803	0.803
	10th	0.873	0.852	0.852	0.824	0.856	0.820
	20th	0.926	0.880	0.885	0.837	0.890	0.837
	40th	0.975	0.912	0.915	0.864	0.923	0.871
AUC	1st	0.717	0.717	0.673	0.673	0.678	0.678
	10th	0.853	0.869	0.698	0.717	0.708	0.730
	20th	0.900	0.908	0.719	0.760	0.728	0.776
	40th	0.960	0.948	0.770	0.842	0.786	0.858

Table D.9: Performance comparison, from release 1.3 to 1.4 for Ant.

Measurement	Iteration	Rand	Act	IG_Rand	IG_Act	MDS_Rand	MDS_Act
Precision	1st	0.867	0.867	0.314	0.314	0.293	0.293
	10th	0.964	0.743	0.582	0.474	0.515	0.410
	20th	0.994	0.794	0.969	0.568	0.985	0.525
	40th	1.000	0.848	0.996	0.736	1.000	0.727
Recall	1st	0.028	0.028	0.183	0.183	0.165	0.165
	10th	0.205	0.203	0.235	0.295	0.228	0.250
	20th	0.388	0.358	0.298	0.353	0.298	0.363
	40th	0.728	0.573	0.610	0.543	0.595	0.533
Accuracy	1st	0.778	0.778	0.727	0.727	0.723	0.723
	10th	0.819	0.800	0.790	0.767	0.778	0.751
	20th	0.862	0.831	0.840	0.793	0.841	0.784
	40th	0.939	0.879	0.912	0.852	0.909	0.849
AUC	1st	0.591	0.591	0.638	0.638	0.617	0.617
	10th	0.666	0.687	0.680	0.714	0.650	0.673
	20th	0.741	0.769	0.704	0.769	0.671	0.738
	40th	0.858	0.887	0.788	0.858	0.742	0.860

Table D.10: Performance comparison, from release 1.5 to 1.6 for Ant.

Measurement	Iteration	Rand	Act	IG_Rand	IG_Act	MDS_Rand	MDS_Act
Precision	1st	0.900	0.900	0.619	0.619	0.606	0.606
	10th	0.973	0.946	0.902	0.722	0.946	0.734
	20th	0.984	0.925	1.000	0.809	1.000	0.807
	40th	0.995	0.934	1.000	0.884	1.000	0.902
Recall	1st	0.001	0.001	0.187	0.187	0.160	0.160
	10th	0.466	0.303	0.352	0.279	0.352	0.252
	20th	0.678	0.503	0.546	0.370	0.573	0.346
	40th	0.873	0.718	0.772	0.560	0.785	0.533
Accuracy	1st	0.737	0.737	0.757	0.757	0.752	0.752
	10th	0.856	0.813	0.820	0.783	0.825	0.780
	20th	0.913	0.859	0.881	0.812	0.888	0.807
	40th	0.966	0.913	0.940	0.865	0.944	0.862
AUC	1st	0.686	0.686	0.762	0.762	0.766	0.766
	10th	0.824	0.847	0.794	0.815	0.803	0.814
	20th	0.859	0.901	0.841	0.855	0.839	0.858
	40th	0.941	0.949	0.909	0.920	0.914	0.917

Bibliography

[1] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33:2–13, 2007.

[2] Stefan Lessmann, Bart Baesens, Christopher Mues, and Swantje Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *Software Engineering, IEEE Transactions on*, 34(4):485 –496, july-aug. 2008. ISSN 0098-5589. doi: 10.1109/TSE.2008.35.

[3] Steven R. Rakitin. *Software Verification and Validation for Practitioners and Managers, Second Edition*. Artech House, 2001. ISBN 1-58053-296-9.

[4] A.P. Nikora and J.C. Munson. Developing fault predictors for evolving software systems. In *Ninth International Software Metrics Symposium (METRICS'03)*, 2003.

[5] Tim Menzies, Justne S. Di Stefano, and Mike Chapman. Learning early lifecycle IV&V quality indicators. In *IEEE Metrics '03*, 2003. Available from http://menzies.us/pdf/03early.pdf.

[6] Forrest Shull, Vic Basili ad Barry Boehm, A.Winsor Brown, Patricia Costa, Mikael Lindvall, Dan Port, Ioana Rus, Roseanne Tesoriero, and Marvin Zelkowitz. What we have learned about fighting defects. In *Proceedings of 8th international software metrics symposium,Ottawa, Canada*, pages 249–258, 2002.

[7] Yan Ma. *An Empirical Investigation of Tree Ensembles in Biometrics and Bioinformatics*. PhD thesis, January 2007.

[8] V. U. Challagulla, F. B. Bastani, I-Ling Yen, and R.A.Paul. Empirical assessment of machine learning based software defect prediction techniques. pages 263–270, 2005.

[9] Taghi M. Khoshgoftaar and Naeem Seliya. Fault prediction modeling for software quality estimation: Comparing commonly used techniques. *Empirical*

Softw. Engg., 8:255–283, September 2003. ISSN 1382-3256. doi: 10.1023/A:1024424811345. URL http://portal.acm.org/citation.cfm?id=820004.820032.

[10] Norman F. Schneidewind. Investigation of logistic regression as a discriminant of software quality. In *METRICS '01: Proceedings of the 7th International Symposium on Software Metrics*, page 328, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1043-4.

[11] Krishnamoorthy Srinivasan and Douglas Fisher. Machine learning approaches to estimating software development effort. *IEEE Trans. Soft. Eng.*, pages 126–137, February 1995.

[12] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engg.*, 17(4):375–407, December 2010. ISSN 0928-8910. doi: 10.1007/s10515-010-0069-5. URL http://dx.doi.org/10.1007/s10515-010-0069-5.

[13] Lan Guo, Yan Ma, Bojan Cukic, and Harshinder Singh. Robust prediction of fault-proneness by random forests. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 417 – 428, 2004. doi: 10.1109/ISSRE.2004.35.

[14] Swapna Gokhale and Michael R. Lyu. Regression tree modeling for the prediction of software quality. In *In Proc. of ISSAT'97*, pages 31–36, 1997.

[15] Shi Zhong, Taghi M Khoshgoftaar, and Naeem Seliya. Analyzing software measurement data with clustering techniques. *Intelligent Systems, IEEE*, 19(2):20–27, Mar-Apr 2004. ISSN 1541-1672. doi: 10.1109/MIS.2004.1274907.

[16] Shi Zhong, T.M. Khoshgoftaar, and N. Seliya. Unsupervised learning for expert-based software quality estimation. *High Assurance Systems Engineering, 2004. Proceedings. Eighth IEEE International Symposium on*, pages 149–155, March 2004. ISSN 1530-2059. doi: 10.1109/HASE.2004.1281739.

[17] M. Mertik, M. Lenic, G. Stiglic, and P. Kokol. Estimating software quality with advanced data mining techniques. *Software Engineering Advances, International Conference on*, pages 19–19, Oct. 2006. doi: 10.1109/ICSEA.2006.261275.

[18] Burak Turhan and Ayse Bener. A multivariate analysis of static code attributes for defect prediction. In *QSIC '07: Proceedings of the Seventh International Conference on Quality Software*, pages 231–237, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3035-4.

[19] S. Gokhale and M. Lyu. Regression tree modeling for the prediction of software quality, March 1997.

[20] Taghi M. Khoshgoftaar, Bojan Cukic, and Naeem Seliya. An empirical assessment on program module-order models. *Quality Technology of Quantitative Management*, 4(2):171–190, 2007.

[21] Tim Menzies, Alex Dekhtyar, Justin Distefano, and Jeremy Greenwald. Problems with precision. *IEEE Transactions on Software Engineering*, 33(9):637–640, Sept. 2007.

[22] Yue Jiang, Bojan Cukic, Tim Menzies, and Nick Bartlow. Comparing design and code metrics for software quality prediction. In *Proceedings of the 4th international workshop on Predictor models in software engineering*, PROMISE '08, pages 11–18, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-036-4.

[23] C. Drummond and R. C. Holte. Severe class imbalan, i(.) is the indicator ce: Why better algorithms aren't the answer? In *Proc. of the 16th European Conference of Machine Learning,Porto,Portugal*, Oct. 2005.

[24] Martin Shepperd, Qinbao Song, Zhongbin Sun, and Carolyn Mair. Data quality: Some comments on the nasa software defect data sets. In *Personal Communication*, July 2012.

[25] David Gray, David Bowes, Neil Davey, Yi Sun, and Bruce Christianson. The misuse of the nasa metrics data program data sets for automated software defect prediction. In *Evaluation Assessment in Software Engineering (EASE 2011), 15th Annual Conference on*, pages 96 –103, april 2011.

[26] O. Chapelle, B. Schölkopf, and A. Zien, editors. *Semi-Supervised Learning*. MIT Press, MA, 2006. URL http://www.kyb.tuebingen.mpg.de/ssl-book.

[27] Xiaojin Zhu. Semi-supervised learning literature survey, 2006.

[28] Kamal Nigam, Andrew Kachites Mccallum, Sebastian Thrun, and Tom Mitchell. Text classification from labeled and unlabeled documents using em. In *Machine Learning*, pages 103–134, 1999.

[29] Chuck Rosenberg, Martial Hebert, and Henry Schneiderman. Semi-supervised self-training of object detection models. In *Seventh IEEE Workshop on Applications of Computer Vision*, pages 29–36, 2005.

[30] Avrim Blum and Tom Mitchell. Combining labeled and unlabeled data with co-training. pages 92–100. Morgan Kaufmann Publishers, 1998.

[31] Burr Settles. Active Learning Literature Survey. Technical Report 1648, University of Wisconsin–Madison, 2009. URL http://pages.cs.wisc.edu/~{}bsettles/active-learning.

[32] Burr Settles. Active learning literature survey. Technical report, 2010.

[33] Karl-Heinrich Moller and Daneil Paulish. An empirical investigation of software fault distribution. In *Software Metrics Symposium, 1993. Proceedings., First International*, pages 82–90, 1993. doi: 10.1109/METRIC.1993.263798.

[34] Weider D. Yu. A software fault prevention approach in coding and root cause analysis. *Bell Labs Technical Journal*, 3(2):3–21, 1998. ISSN 1538-7305. doi: 10.1002/bltj.2101.

[35] Maggie Hamill and Katerina Goseva-Popstojanova. Common trends in software fault and failure data. *Software Engineering, IEEE Transactions on*, 35(4):484–496, 2009. ISSN 0098-5589. doi: 10.1109/TSE.2009.3.

[36] N. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8):797–814, August 2000.

[37] M. Kaaniche and K. Kanoun. Reliability of a commercial telecommunications system. In *Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on*, pages 207–212, 1996. doi: 10.1109/ISSRE.1996.558807.

[38] G. Denaro and M. Pezze? An empirical evaluation of fault-proneness models. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 241–251, 2002.

[39] Thomas J. Ostrand and Elaine J. Weyuker. The distribution of faults in a large industrial software system. *SIGSOFT Softw. Eng. Notes*, 27(4):55–64, July 2002. ISSN 0163-5948.

[40] Hongyu Zhang. On the distribution of software faults. *Software Engineering, IEEE Transactions on*, 34(2):301–302, 2008. ISSN 0098-5589. doi: 10.1109/TSE.2007.70771.

[41] Chih-Song Kuo and Chin-Yu Huang. A study of applying the bounded generalized pareto distribution to the analysis of software fault distribution. In *Industrial Engineering and Engineering Management (IEEM), 2010 IEEE International Conference on*, pages 611–615, 2010. doi: 10.1109/IEEM.2010.5674517.

[42] Carina Andersson and Per Runeson. A replicated quantitative analysis of fault distributions in complex software systems. *IEEE Transactions on Software Engineering*, 33(5):273–286, 2007. ISSN 0098-5589.

[43] Tihana Galinac Grbac, Per Runeson, and Darko Huljeni? A second replicated quantitative analysis of fault distributions in complex software systems. *Software Engineering, IEEE Transactions on*, 39(4):462–476, 2013. ISSN 0098-5589. doi: 10.1109/TSE.2012.46.

[44] Norman E. Fenton and Martin Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25:675–689, 1999. ISSN 0098-5589.

[45] Les Hatton. Reexamining the fault density-component size connection. *IEEE Softw.*, 14(2):89–97, 1997. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/52.582978.

[46] Gunes Koru, Dongsong Zhang, and Hongfang Liu. Modeling the effect of size on defect proneness for open-source software. PROMISE '07, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2954-2.

[47] Gunes Koru, Khaled Emam, Dongsong Zhang, Hongfang Liu, and Divya Mathew. Theory of relative defect proneness. *Empirical Software Engineering*, 13:473–498, 2008. ISSN 1382-3256. URL http://dx.doi.org/10.1007/s10664-008-9080-x. 10.1007/s10664-008-9080-x.

[48] Gunes. Koru, Dongsong Zhang, K. El Emam, and Hongfang Liu. An investigation into the functional form of the size-defect relationship for software modules. *Software Engineering, IEEE Transactions on*, 35(2):293–304, 2009. ISSN 0098-5589. doi: 10.1109/TSE.2008.90.

[49] Thomas J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320, 1976. ISSN 0098-5589. doi: 10.1109/TSE.1976.233837.

[50] Maurice H. Halstead. *Elements of Software Science*. Elsevier, North-Holland, 1975.

[51] Norman E. Fenton and Martin Neil. Software metrics and risk. In *FESMA 1999, the second European Software Measurement Conference*, pages 39–55, 1999.

[52] Victor R. Basili, Lionel Briand, and Walclio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22:751–761, 1995.

[53] Yuming Zhou, Baowen Xu, and Hareton Leung. On the ability of complexity metrics to predict fault-prone classes in object-oriented systems. *Journal of Systems and Software*, 83(4):660 – 674, 2010. ISSN 0164-1212.

[54] Danijel Radjenovic, Marjan Hericko, Richard Torkar, and Ales Zivkovic. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55(8):1397 – 1418, 2013. ISSN 0950-5849.

[55] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *Software Engineering, IEEE Transactions on*, 38(6):1276–1304, 2012. ISSN 0098-5589. doi: 10.1109/TSE.2011.103.

[56] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. ICSE '05, New York, NY, USA, 2005. ACM. ISBN 1-58113-963-2.

[57] N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 364–373, 2007. doi: 10.1109/ESEM.2007.13.

[58] T.L. Graves, A.F. Karr, J.S. Marron, and H. Siy. Predicting fault incidence using software change history. *Software Engineering, IEEE Transactions on*, 26(7):653–661, 2000. ISSN 0098-5589. doi: 10.1109/32.859533.

[59] F. Lanubile, A. Lonigro, and G. Visaggio. Comparing models for identifying fault-prone software components. In *In proceeding of the 7th international conference on software engineering and knowledge engineering*, pages 312–319, 1995.

[60] Niclas Ohlsson, Ming Zhao, and Mary Helander. Application of multivariate analysis for software fault prediction. *Software Quality Journal*, 7(1):51–66, 1998. ISSN 0963-9314. doi: 10.1023/B:SQJO.0000042059.16470.f0.

[61] Taghi Khoshgoftaar, Naeem Seliya, and Kehan Gao. Assessment of a new three-group software quality classification technique: An empirical case study. *Empirical Software Engineering*, 10(2):183–218, 2005. doi: doi:10.1007/s10664-004-6191-x.

[62] G. Denaro. Estimating software fault-proneness for tuning testing activities. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 704–706, 2000. doi: 10.1109/ICSE.2000.870474.

[63] T.J. Ostrand, E.J. Weyuker, and R.M. Bell. Predicting the location and number of faults in large software systems. *Software Engineering, IEEE Transactions on*, 31(4):340–355, 2005. ISSN 0098-5589. doi: 10.1109/TSE.2005.49.

[64] P. Tomaszewski, J. Hakansson, L. Lundberg, and H. Grahn. The accuracy of fault prediction in modified code - statistical model vs. expert estimation. In *Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on*, pages 10 pp.–343, 2006. doi: 10.1109/ECBS.2006.68.

[65] Kehan Gao and T.M. Khoshgoftaar. A comprehensive empirical study of count models for software fault prediction. *Reliability, IEEE Transactions on*, 56(2): 223–236, 2007. ISSN 0018-9529. doi: 10.1109/TR.2007.896761.

[66] H.M. Olague, L.H. Etzkorn, S. Gholston, and S. Quattlebaum. Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *Software Engineering, IEEE Transactions on*, 33(6):402–419, 2007. ISSN 0098-5589. doi: 10.1109/TSE.2007.1015.

[67] Yuming Zhou and Hareton Leung. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Transactions on Software Engineering*, 32(10):771–789, 10 2006. URL http://search.proquest.com/docview/195572642?accountid=2837. Copyright - Copyright IEEE Computer Society Oct 2006; Last updated - 2011-07-20; CODEN - IESEDJ.

[68] Burak Turhan and Ayse Bener. Analysis of naive bayes? assumptions on software fault data: An empirical study. *Data and Knowledge Engineering*, 68(2):278 – 290, 2009. ISSN 0169-023X. doi: http://dx.doi.org/10.1016/j.datak.2008.10.005. URL http://www.sciencedirect.com/science/article/pii/S0169023X08001535.

[69] A. Gunes Koru and Hongfang Liu. Building defect prediction models in practice. *IEEE Softw.*, 22(6):23–29, November 2005. ISSN 0740-7459. doi: 10.1109/MS.2005.149. URL http://dx.doi.org/10.1109/MS.2005.149.

[70] Yan Ma, Lan Guo, and Bojan Cukic. A statistical framework for the prediction of fault-proneness. In *ADVANCES IN MACHINE LEARNING APPLICATIONS IN SOFTWARE ENGINEERING*. Idea Group, 2007.

[71] N. Gayatri, S. Nickolas, A.V. Reddy, and R. Chitra. Performance analysis of datamining algorithms for software quality prediction. In *Advances in Recent Technologies in Communication and Computing, 2009. ARTCom '09. International Conference on*, pages 393 –395, 2009. doi: 10.1109/ARTCom.2009.12.

[72] Taghi M. Khoshgoftaar and Naeem Seliya. Comparative assessment of software quality classification techniques: An empirical case study. *Empirical Software Engineering*, 9(3):229–257, 2004. ISSN 1382-3256. doi: 10.1023/B:EMSE.0000027781.18360.9b.

[73] M. Mertik, M. Lenic, G. Stiglic, and P. Kokol. Estimating software quality with advanced data mining techniques. In *Software Engineering Advances, International Conference on*, pages 19–19, 2006. doi: 10.1109/ICSEA.2006.261275.

[74] Taghi M. Khoshgoftaar, E.B. Allen, J.P. Hudepohl, and S.J. Aud. Application of neural networks to software quality modeling of a very large telecommunications system. *Neural Networks, IEEE Transactions on*, 8(4):902–909, 1997. ISSN 1045-9227. doi: 10.1109/72.595888.

[75] Mie Mie Thet Thwin and Tong-Seng Quah. Application of neural networks for software quality prediction using object-oriented metrics. *Journal of Systems and Software*, 76(2):147 – 156, 2005. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/j.jss.2004.05.001.

[76] Qi Wang, Bo Yu, and Jie Zhu. Extract rules from software quality prediction model based on neural network. In *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence*, ICTAI 04, pages 191–195, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2236-X.

[77] S. Kanmani, V. Rhymend Uthariaraj, V. Sankaranarayanan, and P. Thambidurai. Object oriented software quality prediction using general regression neural networks. *SIGSOFT Softw. Eng. Notes*, 29(5):1–6, September 2004. ISSN 0163-5948.

[78] Atchara Mahaweerawat, Peraphon Sophatsathit, Chidchanok Lursinsap, and Petr Musilek. Fault prediction in object-oriented software using neural network techniques. In *Center (AVIC), Department of Mathematics, Faculty of Science, Chulalongkorn University*, pages 27–34, 2004.

[79] Lan Guo, Bojan Cukic, and Harshinder Singh. Predicting fault prone modules by the dempster-shafer belief networks. In *Proc. 18th International Conference on Automated Software Engineering (ASE)*, 2003.

[80] Kim Kaminsky and Gary D. Boetticher. How to predict more with less defect prediction using machine learners in an implicitly data starved domain.

[81] Fei Xing, Ping Guo, and M.R. Lyu. A novel method for early software quality prediction based on support vector machine. In *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*, pages 10 pp.–222, 2005.

[82] V.U.B. Challagulla, F.B. Bastani, and I-Ling Yen. A unified framework for defect data analysis using the mbr technique. In *Tools with Artificial Intelligence, 2006. ICTAI '06. 18th IEEE International Conference on*, pages 39–46, 2006.

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

[84] A. Marcus, D. Poshyvanyk, and R. Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *Software Engineering, IEEE Transactions on*, 34(2):287–300, 2008. ISSN 0098-5589.

[85] Erik Arisholm, Lionel C. Briand, and Eivind B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2 – 17, 2010. ISSN 0164-1212.

[86] S. Shafi, S.M. Hassan, A. Arshaq, M.J. Khan, and S. Shamail. Software quality prediction techniques: A comparative analysis. In *Emerging Technologies, 2008. ICET 2008. 4th International Conference on*, pages 242–246, 2008.

[87] Yue Jiang, Bojan Cukic, and Yan Ma. Techniques for evaluating fault prediction models. *Empirical Software Engineering*, 13:561–595, 2008. ISSN 1382-3256. URL http://dx.doi.org/10.1007/s10664-008-9079-3. 10.1007/s10664-008-9079-3.

[88] Karim O. Elish and Mahmoud O. Elish. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software*, 81(5):649 – 660, 2008. ISSN 0164-1212.

[89] Olivier Vandecruys, David Martens, Bart Baesens, Christophe Mues, Manu De Backer, and Raf Haesen. Mining software repositories for comprehensible software fault prediction models. *Journal of Systems and Software*, 81(5):823 – 839, 2008. ISSN 0164-1212.

[90] Cagatay Catal and Banu Diri. A systematic review of software fault prediction studies. *Expert Systems with Applications*, 36(4):7346 – 7354, 2009. ISSN 0957-4174.

[91] Cagatay Catal. Software fault prediction: A literature review and current trends. *Expert Systems with Applications*, 38(4):4626 – 4636, 2011. ISSN 0957-4174.

[92] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012. ISSN 1382-3256.

[93] Karel Dejaeger, Thomas Verbraken, and Bart Baesens. Toward comprehensible software fault prediction models using bayesian network classifiers. *IEEE Transactions on Software Engineering*, 39(2):237–257, 2013. ISSN 0098-5589.

[94] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. Do we need hundreds of classifiers to solve real world classification problems? *Journal of Machine Learning Research*, 15:3133–3181, 2014. URL http://jmlr.org/papers/v15/delgado14a.html.

[95] Naeem Seliya and Taghi Khoshgoftaar. Software quality estimation with limited fault data: a semi-supervised learning perspective. *Software Quality Journal*, 15:327–344, 2007. ISSN 0963-9314. URL http://dx.doi.org/10.1007/s11219-007-9013-8. 10.1007/s11219-007-9013-8.

[96] Naeem Seliya and Taghi M. Khoshgoftaar. Software quality analysis of unlabeled program modules with semisupervised clustering. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 37(2):201 –211, 2007. ISSN 1083-4427. doi: 10.1109/TSMCA.2006.889473.

[97] Cagatay Catal and Banu Diri. A fault prediction model with limited fault data to improve test process. In Andreas Jedlitschka and Outi Salo, editors, *Product-Focused Software Process Improvement*, volume 5089 of *Lecture Notes in Computer Science*, pages 244–257. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-69564-6. doi: 10.1007/978-3-540-69566-0_21. URL http://dx.doi.org/10.1007/978-3-540-69566-0_21.

[98] Ekrem Kocaguneli, Tim Menzies, Jacky Keung, David Cok, and Ray Madachy. Active learning and effort estimation: Finding the essential content of software effort estimation data. *IEEE Transactions on Software Engineering*, 99.

[99] Guangchun Luo, Ying Ma, and Ke Qin. Active learning for software defect prediction. *IEICE Transactions*, 95-D(6):1680–1683, 2012.

[100] Ming Li, Hongyu Zhang, Rongxin Wu, and Zhi-Hua Zhou. Sample-based software defect prediction with active and semi-supervised learning. *Automated Software Engineering*, 19:201–230, 2012. ISSN 0928-8910. URL http://dx.doi.org/10.1007/s10515-011-0092-1.

[101] David Yarowsky. Unsupervised word sense disambiguation rivaling supervised methods. In *preceedings of the 33rd annual meeting of the association for computational linguistics*, pages 189–196, 1995.

[102] Steven Abney. Understanding the yarowsky algorithm. *Computational Linguistics*, 30:2004, 2004.

[103] Gholamreza Haffari and Anoop Sarkar. Analysis of semi-supervised learning with the yarowsky algorithm.

[104] Mark Culp and George Michailidis. An iterative algorithm for extending learners to a semisupervised setting. In *The 2007 Joint Statistical Meetings (JSM)*, 2007.

[105] Alexandru Niculescu-mizil and Rich Caruana. Predicting good probabilities with supervised learning. In *In Proc. Int. Conf. on Machine Learning (ICML*, pages 625–632, 2005.

[106] Yogesh Singh, Arvinder Kaur, and Ruchika Malhotra. Software fault proneness prediction using support vector machines.

[107] Andreas, Michael, 5, Hofmann, and H. Interactive Data Visualization with Multidimensional Scaling. Technical report, March 2004.

[108] Matthieu Brucher, Christian Heinrich, Fabrice Heitz, and Jean-Paul Armspach. A metric multidimensional scaling-based nonlinear manifold learning approach for unsupervised data reduction. *EURASIP J. Adv. Sig. Proc.*

[109] Yanjun Qi, J. Klein-seetharaman, and Z. Bar-joseph. Random forest similarity for protein-protein interaction prediction. *Pac Symp Biocomput*, 2005:531–542, 2005.

[110] Leo Breiman. Random forests. *Machine Learning*, 45:5–32, 2001.

[111] Jianming Ye. On measuring and correcting the effects of data mining and model selection. *Journal of the American Statistical Association*, 93(441):pp. 120–131, 1998. ISSN 01621459. URL http://www.jstor.org/stable/2669609.

[112] Shirley Dowdy, Stanley Wearden, and Daniel Chilko. *Statistics for research*. John Wiley & Sons, third edition, 2004.

[113] Burr Settles and Mark Craven. An analysis of active learning strategies for sequence labeling tasks. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '08, pages 1070–1079, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics. URL http://dl.acm.org/citation.cfm?id=1613715.1613855.

[114] David D. Lewis and Jason Catlett. Heterogeneous uncertainty sampling for supervised learning. In *In Proceedings of the Eleventh International Conference on Machine Learning*, pages 148–156. Morgan Kaufmann, 1994.

[115] Jingbo Zhu, Huizhen Wang, Tianshun Yao, and Benjamin K. Tsou. Active learning with sampling by uncertainty and density for word sense disambiguation and

text classification. In *Proceedings of the 22nd International Conference on Computational Linguistics - Volume 1*, COLING '08, pages 1137–1144, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics. ISBN 978-1-905593-44-6. URL http://dl.acm.org/citation.cfm?id=1599081.1599224.

[116] Nicholas Roy and Andrew Mccallum. Toward optimal active learning through sampling estimation of error reduction. In *In Proc. 18th International Conf. on Machine Learning*, pages 441–448. Morgan Kaufmann, 2001.

[117] Min Tang, Xiaoqiang Luo, and Salim Roukos. Active learning for statistical natural language parsing. In *In Proceedings of ACL 2002*, pages 120–127, 2002.

[118] C. Fetzer, M. Raynal, and F. Tronel. An adaptive failure detection protocol. In *Dependable Computing, 2001. Proceedings. 2001 Pacific Rim International Symposium on*, pages 146 –153, 2001. doi: 10.1109/PRDC.2001.992691.

[119] Dong Tian, Kaigui Wu, and Xueming Li. A novel adaptive failure detector for distributed systems. In *Networking, Architecture, and Storage, 2008. NAS '08. International Conference on*, pages 215 –221, june 2008. doi: 10.1109/NAS.2008.37.

[120] Liu Datong, Peng Yu, and Peng Xiyuan. Online adaptive status prediction strategy for data-driven fault prognostics of complex systems. In *Prognostics and System Health Management Conference (PHM-Shenzhen), 2011*, pages 1 –6, may 2011. doi: 10.1109/PHM.2011.5939530.

[121] Jie Ma, Di Li, Shaohong Wang, and Xiaoli Xu. Data-based adaptive fault prediction method and its application. In *Electronic Measurement Instruments, 2009. ICEMI '09. 9th International Conference on*, pages 4–1011 –4–1016, aug. 2009. doi: 10.1109/ICEMI.2009.5274148.

[122] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for eclipse. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 9–, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2954-2. doi: 10.1109/PROMISE.2007.10. URL http://dx.doi.org/10.1109/PROMISE.2007.10.

[123] L.C. Molina, L. Belanche, and A. Nebot. Feature selection algorithms: a survey and experimental evaluation. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pages 306–313, 2002. doi: 10.1109/ICDM.2002.1183917.

[124] Sam T. Roweis and Lawrence K. Saul. Nonlinear dimensionality reduction by locally linear embedding. *SCIENCE*, 290:2323–2326, 2000.

[125] Luis Talavera. An evaluation of filter and wrapper methods for feature selection in categorical clustering. In A.Fazel Famili, JoostN. Kok, JosM. Pea, Arno Siebes, and Ad Feelders, editors, *Advances in Intelligent Data Analysis VI*, volume 3646 of *Lecture Notes in Computer Science*, pages 440–451. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-28795-7. doi: 10.1007/11552253_40. URL http://dx.doi.org/10.1007/11552253.40.

[126] Rakkrit Duangsoithong and Terry Windeatt. Relevance and redundancy analysis for ensemble classifiers. In *Machine Learning and Data Mining in Pattern Recognition*, volume 5632 of *Lecture Notes in Computer Science*, pages 206–220. 2009.

[127] Huanjing Wang, Taghi M. Khoshgoftaar, and Amri Napolitano. A comparative study of ensemble feature selection techniques for software defect prediction. In *ICMLA'10*, pages 135–140, 2010.

[128] Shivikumar Shivaji, E.James Whitehead, Ram Akella, and Sunghun Kim. Reducing features to improve code change-based bug prediction. *Software Engineering, IEEE Transactions on*, 39(4):552–569, April 2013. ISSN 0098-5589.

[129] F.S. Tsai and Kap-Luk Chan. Dimensionality reduction techniques for data exploration. In *Information, Communications Signal Processing, 2007 6th International Conference on*, pages 1–5, Dec 2007. doi: 10.1109/ICICS.2007.4449863.

[130] Emanuel Giger, Martin Pinzger, and Harald Gall. Using the gini coefficient for bug prediction in eclipse. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL '11, pages 51–55, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0848-9. doi: 10.1145/2024445.2024455. URL http://doi.acm.org/10.1145/2024445.2024455.

[131] Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *Proceedings of MSR 2010 (7th IEEE Working Conference on Mining Software Repositories)*, pages 31 – 41. IEEE CS Press, 2010.

[132] Kehan Gao and Taghi M. Khoshgoftaar. Software defect prediction for high-dimensional and class-imbalanced data. In *SEKE*, pages 89–94. Knowledge Systems Institute Graduate School, 2011. ISBN 1-891706-29-2.

[133] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. Analysis of the reliability of a subset of change metrics for defect prediction. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, ESEM '08, pages 309–311, New York, NY, USA,

2008. ACM. ISBN 978-1-59593-971-5. doi: 10.1145/1414004.1414063. URL http://doi.acm.org/10.1145/1414004.1414063.

[134] Marian Jureczko and Lech Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, pages 9:1–9:10, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0404-7.

[135] ElaineJ. Weyuker, ThomasJ. Ostrand, and RobertM. Bell. Comparing the effectiveness of several modeling methods for fault prediction. *Empirical Software Engineering*, 15(3):277–295, 2010. ISSN 1382-3256. doi: 10.1007/s10664-009-9111-2. URL http://dx.doi.org/10.1007/s10664-009-9111-2.

[136] Lan Guo, Yan Ma, Bojan Cukic, and Harshinder Singh. Robust prediction of fault-proneness by random forests. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 417 – 428, 2004. doi: 10.1109/ISSRE.2004.35.