SOFTWARE DEFECT PREDICTION USING BAYESIAN NETWORKS AND KERNEL METHODS

AHMET OKUTAN
B.S., Computer Engineering, Boğaziçi University, 1998
M.S., Computer Engineering, Işık University, 2002

Submitted to the Graduate School of Science and Engineering
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in
Computer Engineering

IŞIK UNIVERSITY
2012

IŞIK UNIVERSITY
GRADUATE SCHOOL OF SCIENCE AND ENGINEERING

SOFTWARE DEFECT PREDICTION USING BAYESIAN NETWORKS AND KERNEL METHODS

AHMET OKUTAN

APPROVED BY:

Assoc. Prof. Olcay Taner Yıldız (Thesis Supervisor)	Işık University	....
Prof. H. Levent Akın	Boğaziçi University	....
Prof. Selahattin Kuru	Kemerburgaz University	....
Assist. Prof. Emine Ekin	Işık University	....
Assist. Prof. Ali İnan	Işık University	....

APPROVAL DATE: ../../....

SOFTWARE DEFECT PREDICTION USING BAYESIAN NETWORKS AND KERNEL METHODS

Abstract

There are lots of different software metrics discovered and used for defect prediction in the literature. Instead of dealing with so many metrics, it would be practical and easy if we could determine the set of metrics that are most important and focus on them more to predict defectiveness. We use Bayesian modeling to determine the influential relationships among software metrics and defect proneness. In addition to the metrics used in Promise data repository, we define two more metrics, i.e. NOD for the number of developers and LOCQ for the source code quality. We extract these metrics by inspecting the source code repositories of the selected Promise data repository data sets. At the end of our modeling, we learn both the marginal defect proneness probability of the whole software system and the set of most effective metrics. Our experiments on nine open source Promise data repository data sets show that response for class (RFC), lines of code (LOC), and lack of coding quality (LOCQ) are the most effective metrics whereas coupling between objects (CBO), weighted method per class (WMC), and lack of cohesion of methods (LCOM) are less effective metrics on defect proneness. Furthermore, number of children (NOC) and depth of inheritance tree (DIT) have very limited effect and are untrustworthy. On the other hand, based on the experiments on Poi, Tomcat, and Xalan data sets, we observe that there is a positive correlation between the number of developers (NOD) and the level of defectiveness. However, further investigation involving a greater number of projects, is needed to confirm our findings.

Furthermore, we propose a novel technique for defect prediction that uses plagiarism detection tools. Although the defect prediction problem has been researched for a long time, the results achieved are not so bright. We use kernel programming to model the relationship between source code similarity and defectiveness. Each value in the kernel matrix shows how much parallelism exist between the corresponding files in the software system chosen. Our experiments on 10 real world datasets indicate that support vector machines (SVM) with a precalculated kernel matrix performs better than the SVM with the usual linear and RBF kernels and generates comparable results with the famous defect prediction methods like linear logistic regression and J48 in terms of the area under the curve (AUC). Furthermore, we observed that when the amount of similarity among the files of a software system is high, then the AUC found by the SVM with a precalculated kernel matrix is also high. Furthermore, we show that SVM with precomputed kernel can be used to predict the number of defects in the files or classes of a software system, because we observe a relationship between source code similarity and the number of defects. Based on the results of our analysis, the developers can focus on more defective modules rather than on less or non defective ones during testing activities. The experiments on 10 Promise datasets indicate that while predicting the number of defects, SVM with a precomputed kernel performs as good as the SVM with the usual linear and RBF kernels, in terms of the root mean square error (RMSE). The method proposed is also comparable with other regression methods like linear regression and IBK. The results of these experiments suggest that source code similarity is a good means of predicting both defectiveness and the number of defects in software modules.

BAYESIAN AĞLARI VE ÇEKİRDEK YONTEMLERİ İLE YAZILIM HATA TAHMİNİ

Özet

Literatürde kullanılan çok çeşitli yazılım ölçütleri mevcuttur. Çok sayıda ölçütle hata tahmini yapmak yerine, en önemli ölçüt kümesini belirleyip bu kümedeki ölçütleri hata tahmininde kullanmak daha pratik ve kolay olacaktır. Bu tezde yazılım ölçütleri ile hataya yatkınlık arasındaki etkileşimi ortaya çıkarmak için Bayesian modelleme yöntemi kullanılmıştır. Promise veri deposundaki yazılım ölçütlerine ek olarak, yazılım geliştiricisi sayısı (NOD) ve kaynak kodu kalitesi (LOCQ) adlı 2 yeni ölçüt tanımlanmıştır. Bu ölçütleri çıkarmak için Promise veri deposundaki veri kümelerinin açık kaynak kodları kullanılmıştır. Yapılan modelleme sonucunda, hem sınanan sistemin hatalı olma ihtimali, hem de en etkili ölçüt kümesi bulunmaktadır. 9 Promise veri kümesi üzerindeki deneyler, RFC, LOC ve LOCQ ölçütlerinin en etkili ölçütler olduğunu, CBO, WMC ve LCOM ölçütlerinin ise daha az etkili olduğunu ortaya koymuştur. Ayrıca, NOC ve DIT ölçütlerinin sınırlı bir etkiye sahip olduğu ve güvenilir olmadığı gözlemlenmiştir. Öte yandan, Poi, Tomcat ve Xalan veri kümeleri üzerinde yapılan deneyler sonucunda, yazılım geliştirici sayısı (NOD) ile hata seviyesi arasında doğru orantı olduğu sonucuna varılmıştır. Bununla birlikte, tespitlerimizi doğrulamak için daha fazla veri kümesi üzerinde deney yapmaya ihtiyaç vardır.

Ayrıca bu tezde, hata tahmini için intihal tespit araçlarını kullanan yeni bir yöntem önerilmiştir. Hata tahmin problemi çok uzun zamandan beri araştırılmaktadır, fakat ortaya çıkan sonuçlar çok parlak değildir. Farklı bir bakış açısı getirmek üzere, kaynak kod benzerliği ve hataya yatkınlık arasındaki ilişkiyi modelleyen çekirdek metodu yöntemi kullanılmıştır. Bu yöntemde, üretilen çekirdek matrisindeki her bir değer, matrisin satır ve sütununda bulunan kaynak kodu dosyaları arasındaki paralelliği göstermektedir. 10 veri kümesi üzerindeki deneyler, önceden hesaplanmış çekirdek matrisi kullanan SVM yönteminin, doğrusal veya RBF çekirdek kullanan SVM yöntemlerine göre daha başarılı olduğunu ayrıca mevcut hata tahmin yöntemleri doğrusal lojistik regresyon ve J48 ile benzer sonuçlar ürettiğini göstermiştir. Ayrıca, bir yazılım sistemi içerisinde bulunan dosyalar arasındaki kod benzerliğinin daha fazla olduğu durumlarda, ROC eğrisi altındaki alan (AUC) ölçütünün de daha yüksek olduğu görülmüştür. Ayrıca, önceden hesaplanmış çekirdek matris kullanan SVM yönteminin, hata sayısı ile kaynak kodu benzerliği arasında gözlemlenen ilişkiden ötürü, bir yazılım sistemindeki hata sayısının tahmin edilmesinde de kullanılabileceği gösterilmiştir. Yapılan analiz sonucunda, yazılım geliştiriciler hatasız veya daha az hatalı modüllere odaklanmak yerine, daha fazla hata içeren modüllere odaklanabilirler. 10 Promise veri kümesi üzerinde yapılan deneyler, hata sayısını tahmin ederken, önceden hesaplanan çekirdek matris kullanan SVM yönteminin ortalama karesel hata (RMSE) açısından doğrusal ve RBF çekirdek kullanan SVM yöntemi kadar başarılı olduğunu göstermiştir. Uygulanan yöntem, doğrusal regresyon ve IBK gibi diğer regresyon yöntemleri ile benzer sonuçlar üretmiştir. Yapılan deneylerin sonuçları, kaynak kodu benzerliğinin hataya yatkınlık ve hata sayısını tahmin etmede iyi bir araç olduğunu ortaya koymuştur.

Acknowledgments

Completing my PhD degree is one of the most challenging activities in my life. First of all, I would like to thank to my advisor, Assoc. Prof. Olcay Taner Yıldız for his guidance, patient and support through this rough road to finish the thesis. I really felt that I was able to call him whenever I had a question, no matter what time it was.

I would like to thank to my thesis committee, Prof. H. Levent Akın and Assist. Prof. Emine Ekin for their guidance and helpful suggestions. Many thanks to Prof. Selahattin Kuru and Assist. Prof. Ali İnan for their support and accepting to be a member of the thesis jury.

I want to thank my parents, Safiye Okutan and Ramiz Okutan. I owe them everything and wish I could show them just how much I love and appreciate them.

Very special thanks to my family; my wife Nur, and my children Esma and Ömer for their understanding and support all the time. I know that I could not finish this thesis without their understanding, support, and patient. So, I dedicate this work to my family and I hope that this work makes them proud.

Table of Contents

List of Tables

List of Figures

List of Symbols/Abbreviations

BN	Bayesian Networks
CBO	Coupling Between Objects
DIT	Depth of Inheritance Tree
LCOM	Lack of Cohesion of Methods
LCOM3	Lack of Cohesion in Methods
LinR	Linear Regression
LOC	Line of Code
LOCQ	Lack of Coding Quality
LR	Linear Logistic Regression
L-SVM	SVM with Linear Kernel
MOSS	Measure Of Software Similarity
NOC	Number of Children
NOD	Number of Developers
P-SVM	SVM with Precalculated Kernel Matrix Kernel
RBF-SVM	SVM with RBF Kernel
RFC	Response for Class
SE	Software Engineering
SVM	Support Vector Machines
WMC	Weighted Method per Class
x	Scalar value
X	Random variable
$\mathbf{x}$	Vector
$\mathbf{x}^T$	Transpose
$\mathbf{x}^i$	Vector $\mathbf{x}$ with subscript i

Chapter 1

Introduction

Many software engineering projects run out of budget and schedule. This is one of the biggest problems that the software development industry has met so far and so many attempts have been made to increase the success rate of software projects. It has been observed that as the project budget gets larger that is the project becomes more complex, the success rate decreases very fast. Humphrey explores the reasons for the project failures and reviews the issues to consider to improve the performance of large-scale software projects [9]. He observes that when the project budget is below $750,000, the success rate is around 55 percent, but as the project size increases the success rate decreases very fast as it is shown in Figure 1.1. When the size becomes more than $10,000,000 the success rate becomes almost zero [9]. This is a fascinating reality and a big obstacle in front of the software development companies to handle.

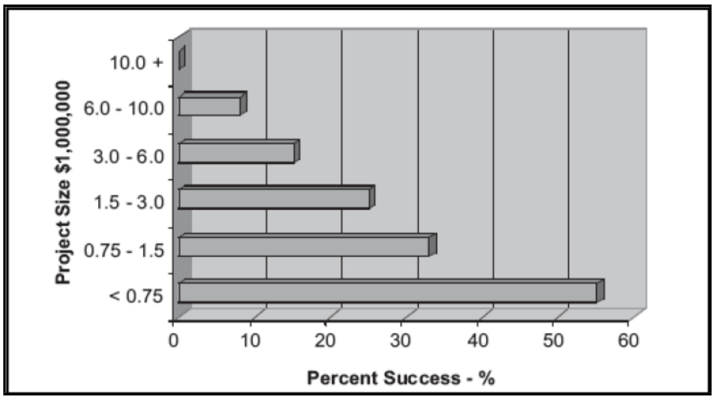


Figure 1.1. Success rate vs. project size.

Technical, financial or social reasons of software project failures have been discussed and several preventive measures were proposed by many researchers so far [10, 11, 12, 13, 14, 15, 16, 17, 18]. According to the Standish Group 31.1% of projects are canceled before they ever get completed and 52.7% of projects cost 189% of their original cost estimation [10]. Jones focuses on the factors that differentiate successful projects from the unsuccessful ones and observes that large software systems are also more likely to be canceled or to experience schedule delays compared to the small projects [11]. Brooks discusses several factors that affect scheduling failures and observes that adding man power to a late software project makes it later [12]. Verner *et al.* analyze the most frequent factors for project failures on 70 projects and observe that in 93% of the failed projects the delivery date impacted the development process, in 81% of the failed projects the project was underestimated and in 76% of the failed projects risks were not re-assessed, controlled, or managed through the project [13].

In this thesis we focus on the prediction of the defect proneness of a software system to decrease the project failures and lessen the total cost during the development and maintenance phases. Software Engineering is providing engineers with necessary means to standardize the development process and to increase productivity. But unfortunately the implementation of the same software design may produce software systems with different quality, depending on how much the defined processes applied and how experienced or creative the project team is.

Developing a defect free software system is very difficult and most of the time there are some unknown bugs or unforeseen deficiencies even in software projects where the principles of the software development methodologies were applied carefully. Due to some defective software modules, the maintenance phase of software projects could become really painful for the users and costly for the enterprises.

Therefore, it is very important and critical to predict the defectiveness of software modules (or files), to plan and manage the testing and maintenance phases of a project better. Defect prediction will give one more chance to the development team to retest the modules or files for which the defectiveness probability is high. First, in testing period, it will be possible to focus more on the defect prone modules or modules where there are more errors comparatively. As a result of a well-advised testing, the probability of fixing the residual defects would increase and more qualified software products would be delivered to the end users. Second, since more defects would be fixed during the test period, the maintenance cost of the project will decrease and this will cause a decrease in the total cost of the project automatically.

When we talk about defect prediction one can ask the question how? From engineering point of view, it is very clear that the prediction must depend on some measurements about the software. In the literature, many different types of software engineering metrics exist and they can be used to evaluate or measure the worth of a software system. According to the Daskalantonakis, software metrics can be categorized as product, process and project metrics [19]:

- Product metrics are related to the software product itself measuring the product features like line of code metric for instance.
- Process metrics are based on the software development processes and measure them. For example, design or testing efficiency of a project or its budget can be used as process metrics.
- Project metrics are focusing on the productivity of software project life cycle. The number of developers or the effort spent per phase can be regarded as a project metric.

Other than these three types of metrics, especially in recent years, developer metrics are also taken into consideration by some researchers. These type of metrics measure the education, experience or productivity level of the software development team. So far, most of the research is done on product metrics rather than process and developer metrics. In Chapter 2, we give a detailed overview of product (software), process and developer metrics.

In defect prediction literature, there are many defect prediction algorithms studied like regression [20, 21, 22], rule induction [22], decision tree approaches like C4.5 [23], case-based reasoning (CBR) [24, 25, 22], artificial neural networks [26, 27, 28, 22], linear discriminant analysis [29], K-nearest neighbor [30], K-star [31], Bayesian networks [32, 1, 33] and support vector machine based classifiers [34, 35, 2, 3]. According to Shepperd *et al.* the accuracy of a specific defect prediction method is very much dependent on the attributes of the data set like its size, number of attributes and distribution [22].

In the literature, defect prediction methods are proposed either to classify the software modules as defective or non defective or guess the number of defects in the software classes or modules. In classification based studies, the aim is to determine whether a software module is defective or not, whereas in regression studies, the number of unexplored bugs are predicted. In this dissertation, we propose a novel technique which is based on SVM with a precomputed kernel, to predict both the defectiveness and the number of resident bugs in the classes or files of a software system.

Menzies *et al.* show that, how the code metrics are used to build predictors is much more important than which particular metrics are used. Furthermore, they also suggest that McCabe [36] and Halstead [37] metrics are intra module metrics and new metrics showing the interactions among different modules (inter module metrics) shall be used that yield better defect predictors [38]. Similarly, Zimmermann and Nagappan suggest that, understanding of the dependencies that exist between different pieces of the code is very important to predict the defect-proneness of software systems [39].

Machine learning based techniques are used more compared to other methods in defect prediction studies. Shepperd *et al.* compare regression, rule induction, nearest neighbor (a form of case-based reasoning), and neural nets and suggest that there are significant differences among the performance of these techniques based on the characteristics of their underlying data set. That is why, they propose to ask the question of which method is the best in a particular context, rather than which method is the best in general [22]. Ekanayake *et al.* use four open source data sets to understand why the quality of defect prediction methods fluctuate across data sets. They suggest that as the number of different developers editing a file and the number of defects fixed by them change, the defect prediction quality is influenced. So, the benefit of bug prediction in general must be seen as volatile over time and, therefore, should be used cautiously [21]. Khoshgoftaar *et al.* use case-based reasoning (CBR) [24, 25] to predict defectiveness and state that a software module currently under development is probably fault prone if a similar module in a previous release was fault-prone. They show that CBR is a preferable technique compared to the nonparametric discriminant analysis in terms of Type-1 and Type-2 errors [24]. In another defect prediction study, Qinbao *et al.* propose association rule mining to predict software defect associations and defect correction effort, using NASA SEL defect data set. They conclude that for defect correction effort prediction, the method they propose performs relatively better than machine learning methods, like PART, C4.5, and Naive Bayes in terms of accuracy [23].

In this thesis, we use source code metrics together with some process and developer metrics to predict defect proneness. By learning from the data sets, we model the interactions among the metrics and determine the most important metrics using Bayesian networks. As a result of our modeling, we not only reveal the influential relationships among the metrics, but also the set of most important metrics that are relatively more effective on defectiveness. Furthermore, it is possible to make probabilistic reasoning about the dependencies between the metrics and defect proneness.

Furthermore, we focus on the similarities of code patterns among different classes (or files) of a software system, to predict the defect proneness and the number of defects. We wonder if the extent of similarity between two source codes is related with the extent of similarity in their defectiveness where the similarity is measured in terms of both syntactic and semantic features. We generate a kernel matrix representing the similarities among the files or classes of the software system and give this kernel matrix as input to support vector machine learning algorithm. To extract similarities among the files or classes, we use the outputs of the plagiarism detection tool MOSS. At the end of our experiments, we show that support vector machines (SVM) with a precalculated kernel matrix performs better than the SVM with the usual linear and RBF kernels and generates comparable results with the famous defect prediction methods like linear logistic regression and J48 in terms of the area under the curve (AUC).

This thesis is organized as follows: In Chapter 2, we give an overview of software metrics that are used in the literature so far. In Chapter 3, we discuss defect prediction studies in general, why they are needed and what they bring? In Chapters 4 and 5, we explain Bayesian networks and support vector machines respectively. In Chapter 6, we briefly explain the methods we propose to determine the set of the most effective metrics and to explore the probabilistic relationships among the metrics and the defectiveness. Furthermore, we explain how we generate the source code similarity based kernel matrix and use it as a precomputed kernel for SVM to predict defect proneness and the number of defects. In Chapter 7, we present our experiments and their results. In Chapter 8, we conclude our work and show the main contributions of our research.

Chapter 2

Software Metrics

In order to predict defectiveness of a software system, we need to be able to measure it. A software metric can be defined as a measure of some property of a piece of software that can be used for defect prediction. There are numerous different types of metrics studied so far and we explain these metrics in detail below.

2.1. Static Code Metrics

Static code metrics are directly calculated from the source code and give an idea about the complexity and the size of the source code. For example, the number of branches or boolean statements or the size of the source code itself is important while calculating these type of metrics. Menzies *et al.* are classifying static code metrics as line of code (LOC) metrics, McCabe metrics, Halstead metrics in general [38]. (See Figure 2.1)

Compared to other types of metrics, static code metrics are easy to understand and help us to see how much complex a software system is. For the purpose of defect prediction, these metrics are just inputs to the defect prediction algorithms.

2.1.1. McCabe Metrics

McCabe metrics were first developed by Thomas J. McCabe to measure the complexity of the source code. McCabes's main argument is that loops and branches make the source code more complex. Three types of McCabe metrics exist:

- Cyclomatic Complexity: Cyclomatic complexity measures the decision logic in a

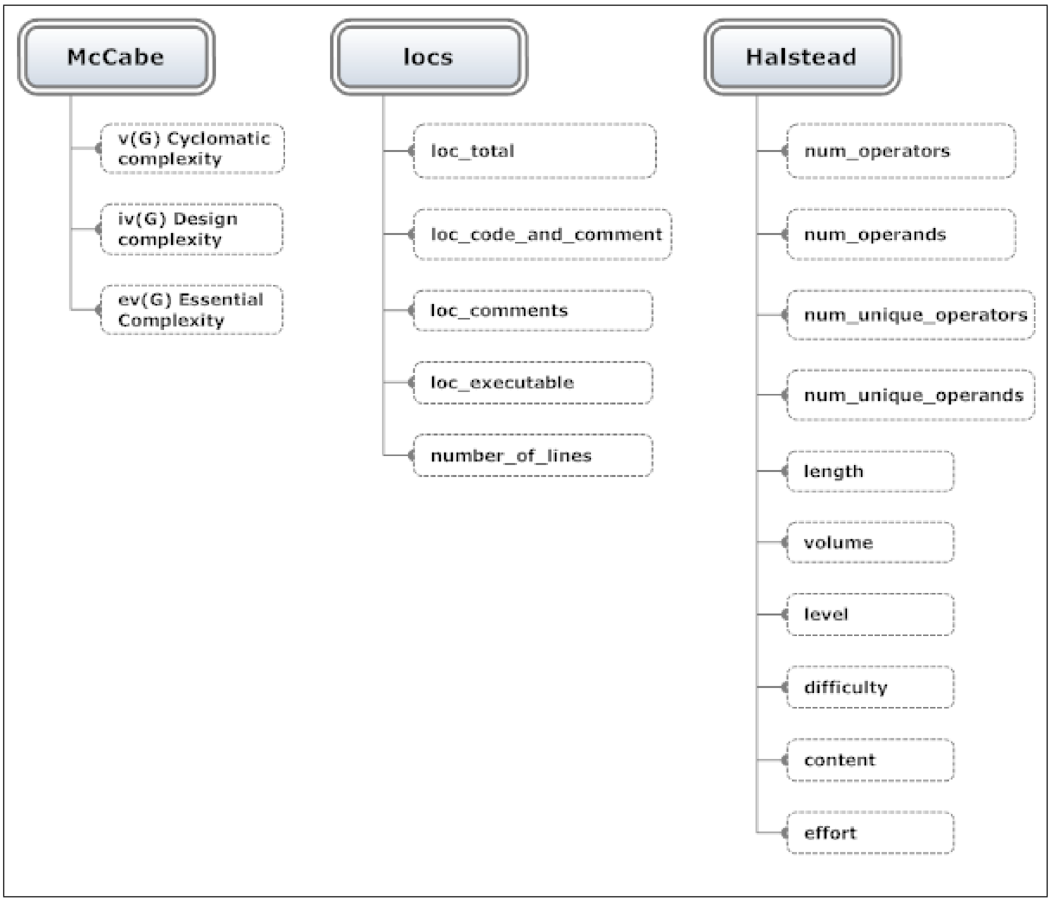
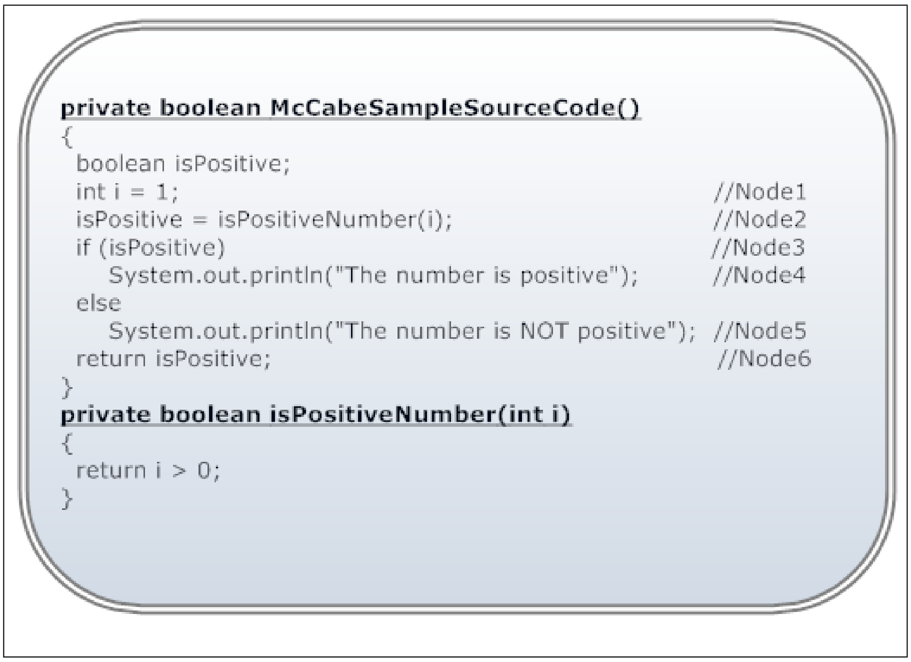


Figure 2.1. McCabe, LOC, and Halstead metrics studied by Menzies *et al.*



```
private boolean McCabeSampleSourceCode()
{
  boolean isPositive;
  int i = 1;                                              //Node1
  isPositive = isPositiveNumber(i);                       //Node2
  if (isPositive)                                         //Node3
     System.out.println("The number is positive");        //Node4
  else
     System.out.println("The number is NOT positive");    //Node5
  return isPositive;                                      //Node6
}
private boolean isPositiveNumber(int i)
{
  return i > 0;
}
```

Figure 2.2. Sample source code to illustrate McCabe metrics.

software program [40]. In order to extract cyclomatic complexity attributes of a source code, we need to extract its flow graph first. We generate the flow graph

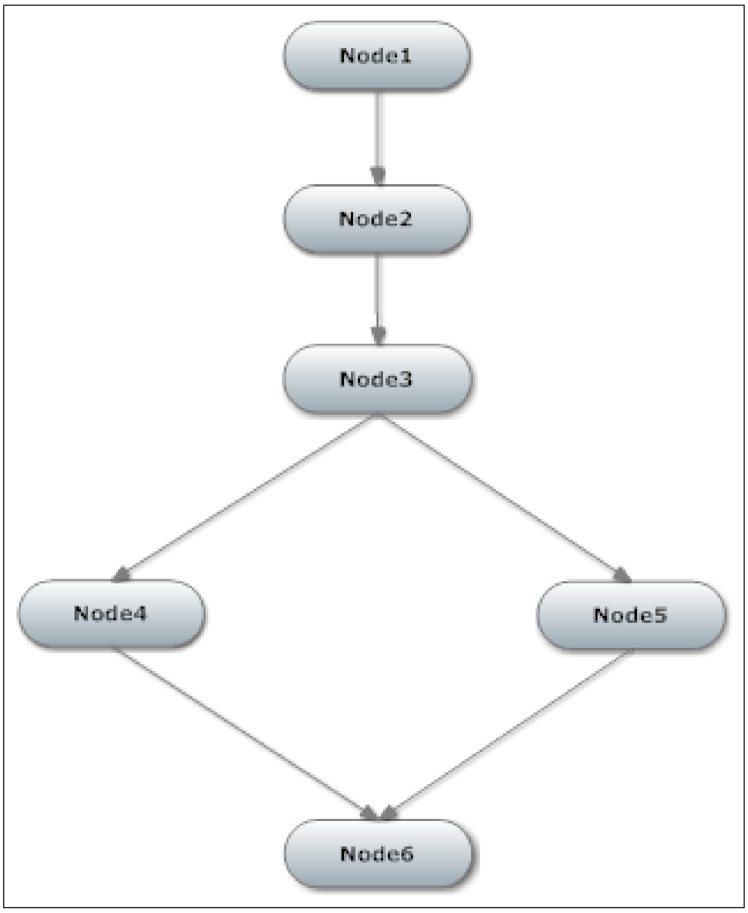


Figure 2.3. Flow graph of the sample source code given in Figure 2.2.

by traversing the source code. All statements are regarded as nodes and there is an edge between two nodes if they are successive in the program. Let say we generate a graph G with n nodes and e edges. Then the cyclomatic complexity of the source code is

$$v(G) = e - n + 2. \tag{2.1}$$

Figure 2.3 shows the flow graph of the sample source code given Figure 2.2. There are 6 nodes and 6 edges, so the cyclomatic complexity would be $V(G) = 6 - 6 + 2 = 2$.

- Design Complexity: In order to calculate design complexity, the flow graph generated for cyclomatic complexity is reduced to include a node if it calls another module and to remove all other nodes together with their edge connections. For example, in addition to entry node $Node_1$ and exit node $Node_6$, only $Node_2$ is included to generate the flow graph for design complexity (See Figure 2.4). According to McCabe and Butler [41], the design complexity of a graph G is shown with $iv(G)$ and is the cyclomatic complexity of its reduced graph.

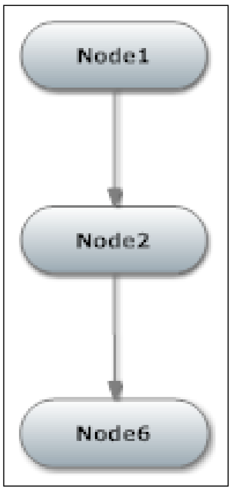


Figure 2.4. Reduced flow graph for design complexity of the sample source code given in Figure 2.2.

- Essential Complexity: According to McCabe [36], essential complexity indicates the extent to which a flow graph can be reduced. In order to calculate the essential complexity, the flow graph is reduced to remove all nodes in a structure. A structure is defined either as a loop or a sequence of nodes such that all incoming edges are to the first node and all outgoing edges are to the last node. For example, part of the flow graph between $Node_3$ and $Node_6$ in Figure 2.3 is a structure and was removed together with the node $Node_2$, to generate the flow graph shown in Figure 2.5. Essential complexity is represented by $ev(G)$ and equals to the cyclomatic complexity of the reduced graph.

2.1.2. Line of Code Metrics

Line of code metrics are directly related to the number of source code lines. These metrics are:

- loc_total: Number of lines in the source code.
- loc_blank: Number of blank lines in the source code.

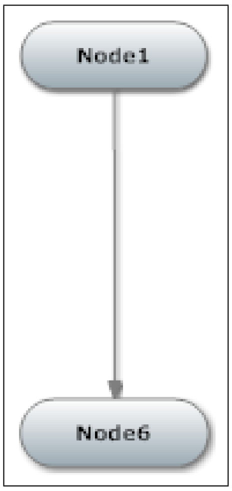


Figure 2.5. Reduced flow graph for essential complexity of the sample source code given in Figure 2.2.

- loc_code_and_comment: Number of source code lines and comment lines.
- loc_comments: Number of lines of comment in the source code.
- loc_executable: Number of lines of executable source. (Not including declarations)

2.1.3. Halstead Metrics

Halstead complexity metrics were proposed by Maurice Howard Halstead [37]. He define four key attributes and derive new metrics from them where the key attributes are:

1. N_1: Number of operators.
2. N_2: Number of operands.
3. n_1: Number of unique operators.
4. n_2: Number of unique operands.

He derive the following attributes from these key attributes:

1. Program length (N):

$$N = N_1 + N_2 \tag{2.2}$$

2. Program vocabulary (n):

$$n = n_1 + n_2 \tag{2.3}$$

3. Volume (V):

$$V = Nlog_2(n) \tag{2.4}$$

4. Difficulty (D) :

$$D = \frac{n_1}{2}\frac{N_2}{n_2} \tag{2.5}$$

5. Effort (E):

$$E = DV \tag{2.6}$$

2.1.4. Object Oriented Metrics

With the introduction of the object oriented design methodology, new metrics proposed by Chidamber and Kemerer [42] to measure the quality of software systems that are developed using object oriented programming languages.

Similar to the traditional metrics explained above, these metrics model both inner module complexity and the interactions among the classes of the system ([42, 43]) and are extensively used in the defect prediction literature [44]. These metrics are:

- Weighted methods per class (WMC): The number of methods and operators defined in each class excluding inherited ones.
- Depth of inheritance tree (DIT): The number of ancestors of a class.
- Response for a class (RFC): The number of functions (or methods) executed when a method or operator of that class is invoked. It is expected that when the value of RFC metric is higher, the class is more complex.

- Number of children (NOC): The number of direct descendants of class excluding grandchildren classes.
- Coupling between object classes (CBO): The number of coupled classes in a software system. A class is coupled with another class if it uses operators or methods of that class.
- Lack of cohesion on methods (LCOM): The number of methods in a class that use a variable shared with other classes minus the number of methods that uses variables that are not shared with any other class [43]. When LCOM is negative it is set to zero.
- Lines of code (LOC): The number of executable source code lines in a class. Note that comment and blank lines are excluded.

2.2. Developer Metrics

Contrary to the software product metrics, developer metrics have not been considered that much in the literature. Recently researchers have started to study developer metrics. Developer metrics focus on the experience, education and quality of the software developers. Matsumoto *et al.* [45] use the eclipse project dataset and conclude that if a software module is touched by more than one developer that module is more defect prone compared to the modules developed by just one developer.

He propose the following metrics:

- The number of code churns made by each developer.
- The number of commitments made by each developer.
- The number of developers for each module.

2.3. Process Metrics

Besides static code measurements, most of the time it is also important to measure and model the software development process. Arisholm *et al.* believe that if process metrics are used together with source code metrics, the prediction performance could be improved significantly [46]. For example the answers of the following questions which are not related with static code, are important to measure the quality or the worth of a software system:

- What kind of software development methodology is used? Is it followed efficiently?
- What is the average experience of the developer team?
- What is the development cost for each process and the total development cost of the project itself?
- Are the processes well documented?

According to Mills, process metrics measure the software development process, such as overall development time, type of methodology used, or the average level of experience of the programming staff [47]. Furthermore, Daskalantonakis defines process metrics as those that can be used to improve the software development and maintenance processes such as the efficiency of these processes, and their corresponding costs [19].

Chapter 3

Defect Prediction

A software system is regarded defective if it does not meet the expectation of its customers i.e. it does not behave as it is specified in its requirement analysis document. This situation might be due to a miscalculation in a formula, lack of some features or a bug. All of these deficiencies can be accepted as defects. In these cases, the development team should take action and make some corrections. That is why, we can define a defective software as a system where further development involvement is needed.

There are two very essential points that we should emphasize to decide the level of defectiveness. First, not all defects have the same priority considering their effect in the maintenance phase. For example, a bug in an accounting software is very critical and has a high priority, because it causes wrong payments. On the other hand, the caption of a button or the tool tip text of some screen control is a deficiency with low priority. Second, the number of defects in a software system is important to decide how much defective a software is. A software system with one non-critical bug is much more preferable to a system with too many bugs. In brief, the number of bugs together with their severity are two very important parameters to decide the extent of defectiveness.

3.1. Defect Prediction Data

The metrics of a software system must be ready prior to applying any defect prediction method. Since it is hard to measure a software system, extracting its metrics is not a straight forward task. Most of the time, companies do not spend time for metrics collection and analysis, due to the famous fact i.e. rush for market. Although analyzing software metrics saves time and provides more efficient project management, it is generally very difficult to make managers believe in this. Furthermore, even if the managers are convinced, most of the time metrics are not shared because metrics data are regarded confidential and sharing is not desired.

On the other hand, there are some open data sets shared by NASA [48] and Promise data repository [49] and the number of data sets in such repositories is increasing day by day. It is possible to make comparative analysis using these public data sets. Several defect prediction methods could be compared using these open data sets and there are examples of such benchmarking studies in literature [34]. During our literature review, we observed that most of the defect prediction papers use private data sets (around 55 %) whereas around 38 % of them use public data sets from either NASA metric data repository [48] or Promise data repository [49].

3.2. Performance Measure

Considering defect prediction as a binary classification problem, let S $= \{(\mathbf{x_i}, y_i)\}$ be a set of N examples where $\mathbf{x_i}$ represents software metrics and y_i represents the target class i.e. whether the software is defective or not. Let us assume that defective modules (positive instances) are represented with + and not defective modules (negative instances) are represented with -, so we have $y_i \in \{+, -\}$. Then defect predictor is a function $f(x) : \{\mathbf{x_i} \to \{+, -\}\}$. Depending on the outcome of the defect predictor, there are four possible cases:

- True positive (TP): If a software module is defective and is classified as defective.
- False negative (FN): If a module is defective and is classified as nondefective.
- True negative (TN): If a module is nondefective and is classified as nondefective.
- False positive (FP): If a module is nondefective and is classified as defective

A sample confusion matrix using metrics defined above is shown in Table 3.1. It is desirable for a defect predictor to be able to mark as many defective modules as possible while avoiding false classifications FN and FP.

Table 3.1. A Sample Confusion Matrix.

Target class	Defective	Non defective
Defective	TP	FN
Non defective	FP	TN

Some classifiers are evaluated using their TP Rate (also called sensitivity)

$$TPRate = \frac{TP}{TP + FN} \tag{3.1}$$

or FP rate (also called false alarm rate)

$$FPRate = \frac{FP}{FP + TN} \tag{3.2}$$

It is a common way to look at the error rates of classifiers while making comparisons. But, this is not true in real life, because first, the proportions of classes are not equal. For instance, in defect prediction, the proportion of defective modules are much more smaller compared to the nondefective ones. Furthermore, the cost of FP and FN are not the same i.e. FN is more costly than FP. Considering a defective module as not defective is more costly than considering a not defective module as defective. As a result, instead of making comparisons using sensitivity or false alarm rate, it is a better and convenient way to consider them together.

ROC analysis is used in the literature and considers TP rate (sensitivity) and FP rate (hit) together [44]. ROC curve is a two dimensional graphical representation where sensitivity (TP) is the y-axis and false alarm rate (FP) is the x-axis. It is desirable to have high sensitivity and small false alarm rate. So, as the area under the ROC curve gets larger, the classifier gets better. Consider the ROC curves in Figure 3.1, where $f2$ is better classifier than $f1$ since the area under the curve of $f2$ is larger.

We use precision measure besides ROC curves, but specificity and accuracy is not used in our thesis. Although accuracy and specificity are used as performance measure

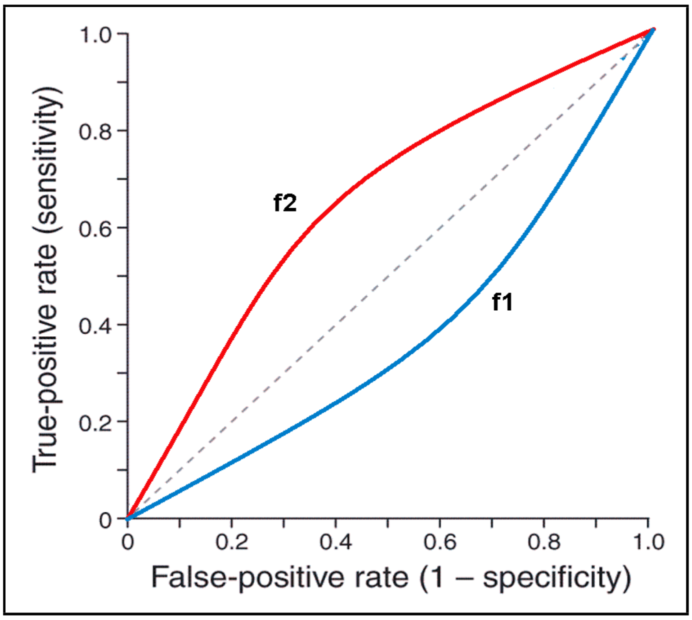


Figure 3.1. A sample ROC Curve.

in some studies, accuracy is not preferred in general since similar accuracy values have been found for different defect predictors although they had different precisions [51]. In addition to the ROC curves, several other performance indicators exist in the literature:

- Specificity:

$$\frac{TN}{TN+FP} \tag{3.3}$$

- Accuracy:

$$\frac{TP+TN}{TP+FN+TN+FP} \tag{3.4}$$

- Precision:

$$\frac{TP}{TP+FP} \tag{3.5}$$

3.3. An Overview of the Defect Prediction Studies

Software defect prediction studies are grouped into two main categories namely classification and regression. In the classification case, at the end of the analysis, the software modules (classes or files) are marked as defective or not, that is, a binary classification problem is solved. The focus is on defect proneness rather than its extent. However in the regression case, at the end of the analysis, the number of faults in each module is found and the emphasis is on the number of faults rather than defect proneness. In terms of the type of the defect predictors used, classification methods can be grouped into three categories as statistical methods, machine learning methods and mixed ones where statistical and machine learning methods are used together.

3.3.1. Defect Prediction Using Statistical Methods

There are studies where a specific statistical method is researched or the performance of several statistical methods are compared. For example, Pickard *et al.* [52] compare the efficiency of residual analysis, multivariate regression and classification and regression trees (CART) for the analysis of the software data. Succi and Stefanovic [53] compare Poisson regression model with binomial regression models to deal with software defect data that is not distributed normally. They observe that the zero-inflated negative binomial regression model, designed to explicitly model the occurrence of zero counts in the dataset, shows the best ability to describe the high variability in the dependent variable.

Schneidewind [54] show that logistic regression method is not very successful alone. But when used together with Boolean discriminant functions (BDF) it gives more accurate results. Basili *et al.* use logistic regression method to explore the relationships between object oriented metrics and the defect proneness and observe that object oriented metrics are better predictors than traditional static code metrics, since they can be collected at earlier phases of the software development process [55]. Munson and Khoshgoftaar use the statistical technique of discriminant analysis to predict detect prone software and observe that the technique is successful in classifying programs with a relatively low error rate [29].

3.3.2. Defect Prediction Using Machine Learning Methods

Machine learning is a discipline that deals with the development of algorithms to make computers exhibit intelligent behavior. Its focus is to make computers learn from the training data and to behave intelligently when a new data set is given. Machine learning accomplishes this by modeling the complex relationships among the data entries in the training set.

Defect prediction is a very appropriate area to apply machine learning algorithms. There are some automated tools in the industry that extracts software metrics like Prest [50]. Furthermore, based on the maintenance experience of users, if there is a bug database, it would also be possible to say whether a software module is defective or not or even how many defects it has. Then, using the extracted metrics together with the defectiveness information, machine learning algorithms can model the complex relationships between the metrics and the defectiveness.

In recent years, compared to other methods, the amount of research done on defect prediction using machine learning algorithms has increased significantly. Many algorithms have been studied and as a consequence, some of these algorithms have been marked to be superior to others. According to the No Free Lunch theorem [56], there is no algorithm which is better than other algorithms on all data sets. That is why, most of the time it is difficult to generalize the results. According to Myrtveit *et al.*, "we need to develop more reliable research procedures before we can have confidence in the conclusion of comparative studies of software prediction models" [57].

Rule induction [22], regression [22, 21], case-based reasoning (CBR) [24, 25, 22], decision tree approaches like C4.5 [23], random forest [58, 59], linear discriminant analysis [29], artificial neural networks [26, 27, 28, 22], K-nearest neighbor [30], K-star [31], Bayesian networks [32, 1, 33] and support vector machine based classifiers [34, 35, 2, 3, 60] are machine learning algorithms that are used in the fault prediction literature. For the purpose of this thesis, our focus is on Bayesian networks and support vector machines based defect prediction methods.

3.4. Previous Work on Defect Prediction

When we look at the publications about defect prediction, we see that in early studies static code features were used more. But afterwards, it was shown that together with static code metrics, other measures like process metrics are also effective on defect prediction and should be investigated. For example, Fenton and Neil [4] argue that static code measures alone are not able to predict software defects accurately. To support this idea we argue that, if a software is defective this might be related to one of the following:

- The specification of the project may be wrong either due to contradictory requirements or missing features. It may be too complex to realize or even not very well documented.
- The design might be poor, it may not consider all requirements or it may reflect some requirements wrongly.
- Developers are not qualified enough for the project.
- There might be a project management problem and the software life cycle methodologies might not be followed very well.
- The software may not be tested enough, so some defects might not be fixed during the test period.

None of the above factors are related to code metrics and all of them may very well affect defect proneness. So, the question is which factors or metrics are effective on defectiveness and how can we measure their effect? Looking at the defect prediction problem from the perspective that all or an effective subset of software or process metrics must be considered together besides static code measures, it is crystal clear that Bayesian network model is a very good candidate for taking into consideration several process and product metrics at the same time.

3.4.1. Critics About Studies

While making a critique of the software defect prediction studies, Fenton and Neil argue that although there are so many studies in the literature, defect prediction problem is far from solution [4]. There are some wrong assumptions about how defects are defined or observed and this has caused misleading results. Their claim can be understood better when we notice that some papers define defects as observed deficiencies while some others define them as residual ones. Fenton and Neil also state that the Goldilocks principle is a false claim. Goldilocks principle states that there is an optimum software module size that is not too big or too small. Hatton [61] claims that "compelling empirical evidence from disparate sources suggests that in any software system, larger components are proportionally more reliable than smaller components". To be able to better interpret the claim of Fenton and Neil, we show how Goldilocks principle supported or not supported by different authors. In studies by Gaffney [62] and Basili and Perricome [63], it is accepted that defect density decreases as module size increase (See Figure 3.2.A). On the other hand, in studies by Hatton [61] and Moller and Paulish [64], one sees that there is an optimum module size that is "not too big or too small" (See Figure 3.2.B). Lastly, Neil [65] and Fenton and Ohlsson [66] observe that there is no apparent relationship between the defect density and module size as shown in Figure 3.2.C.

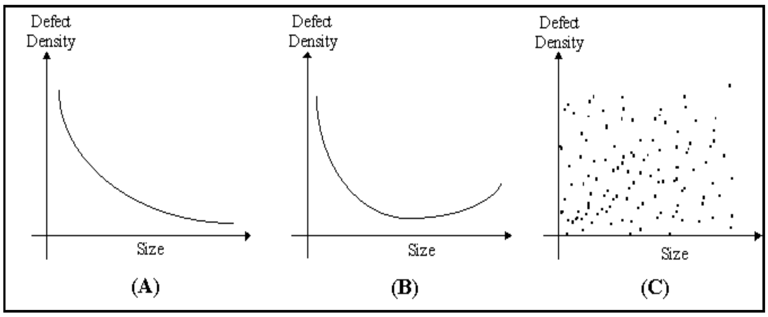


Figure 3.2. Defect densities reported in different studies [4].

Fenton and Neil claim that if the Goldilocks principle was correct then the program decomposition as a solution method would not work. We also think that it can not be the case that a reliable software can have less reliable or more defective sub modules. As a result, they conclude that the relationship between defect density and module size is too complex to model, and in general, it is difficult to use straightforward curve fitting models [4].

Gyimothy *et al.* use regression and machine learning methods (decision tree and neural networks) to see the importance of object oriented metrics for fault proneness prediction [43]. They formulate a hypothesis for each object oriented metric and test the correctness of these hypotheses using open source web and email tool Mozilla. For comparison they use precision, correctness and completeness. It was found that CBO is the best predictor and LOC is the second. On the other hand, the prediction capability of WMC and RFC is less than CBO and LOC but much better than LCOM, DIT, and NOC. According to the results, DIT is untrustworthy and NOC can not be used for fault proneness prediction. Furthermore, the correctness of LCOM is good although it has a low completeness value.

Menzies and Shepperd explain the possible reasons behind the conclusion instability problem [67]. In their analysis, they state that there are two main sources of conclusion instability, (i) bias showing the distance between the predicted and actual values and (ii) variance measuring the distance between different prediction methods. The bias can be decreased by using separate training and validation data sets and the variance can be decreased by repeating the validation many times.

In another research, Menzies *et al.* show what appears to be useful in a global context is often irrelevant for particular local contexts in effort estimation or defect prediction studies. They suggest to test if the global conclusions found are valid for the subsets of the data sets used [68].

Posnett *et al.* explains the ecological inference risk which arises when one builds a statistical model at an aggregated level (e.g., packages), and infers that the results of the aggregated level are also valid for the disaggregated level (e.g., classes), without testing the model in the disaggregated level [69]. They show that modeling defect prediction in two different aggregation levels can lead to different conclusions.

3.4.2. Benchmarking Studies

Although there are many defect predictors in the literature, there are not so many extensive benchmarking studies. Comparing the accuracy of the defect predictors is very important since most of the time the results of one method is not consistent across different data sets [34]. There are a couple of reasons for this. First of all, especially early studies in defect prediction used only a small number of data sets. Furthermore, the performance indicators used across studies were different, so making a comparison was really difficult. That is why, good benchmarking studies are always welcome to see which defect prediction methods produce more accurate results.

Lessmann *et al.* compare the performance of 22 defect prediction methods across 10 data sets from the NASA metric data repository and Promise data repository. Methods used in this study are grouped into six categories, namely statistical methods, nearest neighbour methods, neural networks methods, SVM based methods, decision tree methods, and ensemble methods. Then methods that belong to each of these categories (See Table 3.2) were benchmarked.

The prediction methods are analyzed using 10 data sets and the results for each data set are compared according to the area under ROC curve (AUC). The last column (AR) shows the average ranking of the method over all defect prediction methods.

When we look at the AUC and AR in Table 3.2, we see that LS-SVM, RndFor and Bayes Net are one of the most accurate methods in defect prediction. However, we also see that the difference among the AUC values are not so significant. To check the significance of differences, post-hoc Nemenyi's test was applied to pairwise compare the 22 methods. Nemenyi's test is used to check the null hypothesis that the mean ranks of a set of classifiers are equal. The classifiers are pairwise tested and if the difference among their mean ranks exceeds a critical value, then the null hypothesis is rejected. In Figure 3.3, all classifiers are sorted according to their mean ranks where the line segment on the right of a classifier shows the amount of its critical difference. For instance, for LS-SVM, it's line segment is on the left of RBF, VP, CART and this

Table 3.2. The AUC values and the average ranking (AR) of the 22 defect prediction methods on 10 data sets.

Algorithms	Data Sets										
	CM1	KC1	KC3	KC4	MW1	JM1	PC1	PC2	PC3	PC4	AR
LDA	0.77	0.78	0.62	0.73	0.82	0.73	0.82	0.87	0.82	0.88	9.7
QDA	0.7	0.78	0.74	0.8	0.83	0.7	0.7	0.8	0.78	0.86	13.1
LogReg	0.8	0.76	0.61	0.74	0.82	0.73	0.82	0.86	0.82	0.89	10
NB	0.72	0.76	0.83	0.68	0.8	0.69	0.79	0.85	0.81	0.85	12.9
Bayes Net	0.79	0.75	0.83	0.8	0.82	0.73	0.84	0.85	0.8	0.9	8.7
LARS	0.84	0.75	0.8	0.76	0.74	0.72	0.7	0.3	0.79	0.9	13.3
RVM	0.82	0.76	0.74	0.74	0.75	0.72	0.84	0.91	0.82	0.89	10.4
K-NN	0.7	0.7	0.82	0.79	0.75	0.71	0.82	0.77	0.77	0.87	14.5
K*	0.76	0.68	0.71	0.81	0.71	0.69	0.72	0.62	0.74	0.83	17.1
MLP-1	0.76	0.77	0.79	0.8	0.77	0.73	0.89	0.93	0.78	0.95	6.9
MLP-2	0.82	0.77	0.83	0.76	0.76	0.73	0.91	0.84	0.81	0.94	6.9
RBF net	0.58	0.76	0.68	0.73	0.65	0.69	0.64	0.79	0.78	0.79	17.8
SVM	0.7	0.76	0.86	0.77	0.65	0.72	0.8	0.85	0.77	0.92	13
L-SVM	0.8	0.76	0.82	0.76	0.76	0.73	0.86	0.83	0.84	0.92	7.7
LS-SVM	0.75	0.77	0.83	0.81	0.6	0.74	0.9	0.85	0.83	0.94	6.8
LP	0.9	0.75	0.74	0.83	0.74	0.72	0.73	0.88	0.82	0.92	9.3
VP	0.72	0.76	0.74	0.73	0.73	0.54	0.75	0.5	0.74	0.83	18.2
C4.5	0.57	0.71	0.81	0.76	0.78	0.72	0.9	0.84	0.78	0.93	11.6
CART	0.74	0.67	0.62	0.79	0.67	0.61	0.7	0.68	0.63	0.79	19.3
ADT	0.78	0.69	0.74	0.81	0.76	0.73	0.85	0.7	0.76	0.94	11.8
RndFor	0.81	0.78	0.86	0.85	0.81	0.76	0.9	0.82	0.82	0.97	4
LMT	0.81	0.76	0.78	0.8	0.71	0.72	0.86	0.83	0.8	0.92	10.4

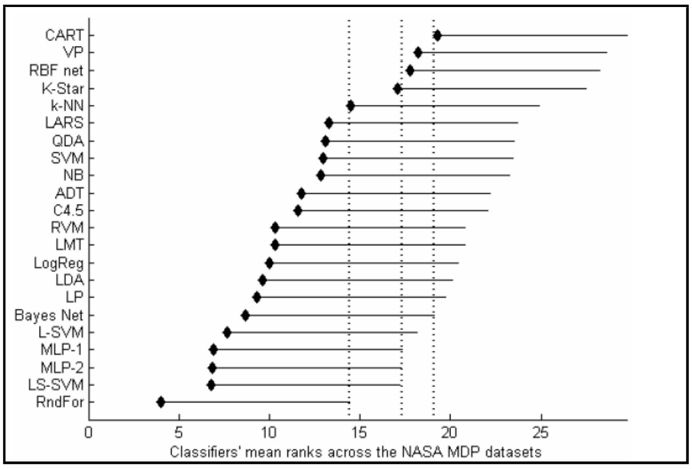


Figure 3.3. Results of the pairwise comparisons of all classifiers using Nemenyi's post hoc test with $\alpha = 0.05$.

shows that LS-SVM is significantly different from these three classifiers. Interpreting the results from this point of view, it is clear that excluding a couple of classifiers like RndFor, LS-SVM, and Bayes Net, there is not a significant difference among the 22 methods. Almost all of them are able to model the linear dependency among the software metrics and the defectiveness. This might be due to the fact that MDP data sets are not accurate enough to model the nonlinear relationship among the metrics and the defectiveness. One can also interpret this result as the discriminative attribute of the data sets is more important than the classifier chosen in the defect prediction.

Qinbao *et al.* propose a general software defect-proneness prediction framework that includes unbiased and comprehensive comparison of competing prediction systems and conclude that one should choose different learning techniques for different data sets which means that no method can dominate in all data sets since small details changed during experiment design may generate different results [70]. We believe that this approach is more realistic as the nature of the data, for instance whether it is balanced or not, may affect the performance of the learning method. If the data is skewed i.e. there are not enough defective instances to learn, one technique may perform poor, although it performs well on a balanced data set.

Hall *et al.* make a literature review comprising 208 fault prediction studies published from January 2000 to December 2010 and suggest that simple, and easy to use modeling techniques like Naive Bayes or Logistic Regression perform well. On the other hand, more complex modeling techniques, such as Support Vector Machines perform relatively less well [71]. Regarding to the two conflicting results on SVM, one by Lessmann *et al.* [34] (SVM performs well) and the other by Hall *et al.* [71] (the performance of SVM is not well) we can say that:

- The performance of SVM depends on the type of the kernel used. Linear kernels are simple and usually perform well on easy data sets, but they may underfit on non-linear data sets. On the other hand, RBF kernels are more complex and can learn non-linear relationships better, but they may overfit on linear and easy data sets.
- If the data is skewed, RBF kernels might not perform well although they can generate bright results with balanced data sets.
- The number of studies reviewed is 33 for Naive Bayes and 56 for Logistic Regression, whereas it is only 4 for SVM and we believe that it may not be safe to make a generalization with such a small number of cases.

Arisholm *et al.* use a Java middle-ware system from a telecommunication company to compare several data mining and machine learning techniques like SVM, Neural network, Logistic regression, C4.5, and Boost based on different evaluation criteria like accuracy, precision/recall, AUC, and cost-effectiveness measure (CE). They use process metrics like change or fault history besides structural source code metrics [46]. Their results suggest that the effect of different prediction techniques on the classification accuracy is limited, however if process metrics are included together with source code metrics, the prediction performance is significantly improved, compared to the case when only structural source code metrics are used. However, we believe that there are two points that one can question about the findings in this study. First the experiments were carried out using several releases of one data set, so it is difficult to generalize the results for all data sets. Second, all the techniques were run with default parameter sets and no tuning was done. If the default parameters of the prediction methods were tuned, the performance of the techniques could change.

Gondra [72] consider defect proneness prediction as a binary classification problem and compare the performance of Artificial Neural Network (ANN) and SVM using a data set from NASA metrics data repository. He concludes that SVM is a better defect prediction technique when compared with ANN wince its accuracy is higher. (The accuracy of SVM was 87.4% whereas ANN had an accuracy of 72.6%) Elish and Elish [73] compare the performance of SVM with eight statistical and machine learning methods in terms of accuracy, precision, recall and the F-measure using four data sets from NASA metrics data repository. They suggest that the performance of SVM is better than or at least competitive with other statistical and machine learning methods. Arisholm *et al.* [74] compare the performance of SVM with eight data mining techniques like C4.5, neural networks, and logistic regression in terms of defect classification accuracy (precision, recall, and ROC) using an industrial Java system. Although they suggest that C4.5 classification tree method performs better than other techniques in general, the results are comparable in terms of AUC and the AUC for SVM is 83.7% which is better than the AUC of the six out of eight techniques.

As a summary of the literature review, we can say that there are really so many studies about defect prediction but the results are not so bright. That is, the defect prediction problem is still unsolved or we can say it is far from solution. There are some misleading or contradictory findings in the literature most of which are related to the wrong assumptions and data set special issues. However in recent years, the research direction is towards some novel approaches rather than older statistical methods. Using Bayesian networks, it is possible to take into consideration the broad possible reasons of defect prediction. Furthermore, using kernel methods, it is possible to model the interaction between the source code similarity and defectiveness. We base our research on these two approaches and propose novel methods that we believe make the solution of the defect prediction problem easier.

Chapter 4

Bayesian Networks

In this chapter we first provide the necessary technical background on Bayesian networks and then explain how they are used in defect prediction. Afterwards, we present a literature review and give examples of previous studies about Bayesian networks.

4.1. Background on Bayesian Networks

Bayesian network is a graphical representation that shows the probabilistic causal or influential relationships among a set of variables that we are interested in. There are a couple of practical factors for using Bayesian networks. First, Bayesian networks are able to model probabilistic causal influence of a set of variables on another variable in the network. Given the probability of parents, the probability of their children can be calculated. Second, Bayesian networks can cope with the missing data problem. This aspect of Bayesian networks is very important for defect prediction since some metrics might be missing for some modules in defect prediction data sets.

A Bayesian network is a directed acyclic graph (DAG), composed of edges E and vertices V which represent the joint probability distribution of a set of variables. In this notation, each vertex represents a variable and each edge represents the causal or associational influence of one variable to its successor in the network.

Let $X = \{X_1, X_2, ...X_n\}$ be n variables taking continuous or discrete values. The probability distribution of X_i is shown as $P(X_i|a_{x_i})$ where a_{x_i}'s represent parents of X_i if any. When there are no parents of X_i, then it is a prior probability distribution and can be shown as $P(X_i)$.

The joint probability distribution of X can be calculated using chain rule. That is,

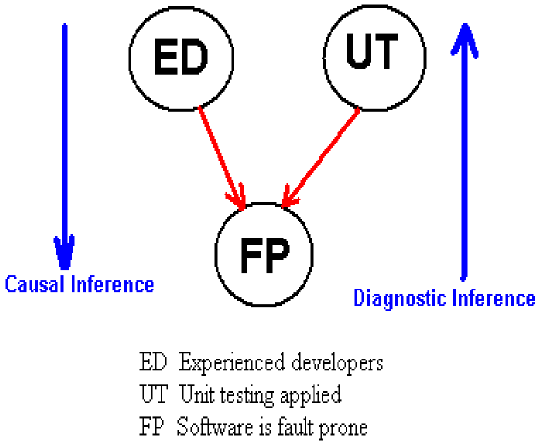
$$\begin{aligned}
P(X) &= P(X_1|X_2, X_3, ..., X_n)P(X_2, X_3, ..., X_n) \\
&= P(X_1|X_2, ..., X_n)P(X_2|X_3, ..., X_n)P(X_3, ..., X_n) \\
&= P(X_1|X_2, ..., X_n)P(X_2|X_3, ..., X_n)...P(X_{n-1}|X_n)P(X_n) \\
&= \prod_{i=1}^{n} P(X_i|X_{i+1}, ..., X_n) \qquad (4.1)
\end{aligned}$$

Given the parents of X_i, other variables are independent from X_i, so we can write the joint probability distribution as

$$P(X) = \prod_{i=1}^{n} P(X_i|a_{x_i}) \qquad (4.2)$$

On the other hand, Bayes' rule is used to calculate the posterior probability of X_i in a Bayesian network based on the evidence information present. We can calculate probabilities either towards from causes to effects ($P(X_i|E)$) or from effects to causes ($P(E|X_i)$). Calculating probability of effects from causes is called causal inference whereas calculating probability of causes from effects is called diagnostic inference [5]. A sample Bayesian network and the conditional probability table are shown in Figure 4.1 to illustrate the causal and diagnostic inference. Assume that we would like to investigate the effect of using experienced developers (ED) and applying unit testing methodology (UT) on defectiveness (FP). Furthermore, each variable can take discrete values of ON/OFF, that is developers are experienced or not, unit testing is used or not used. Suppose we would like to make a causal inference and calculate the probability of having a fault prone software if we know that the developers working on the project are experienced. We shall calculate

$$\begin{aligned}
P(FP|ED) = P(FP|ED, UT)P(UT|ED) +& \\
P(FP|ED, \sim UT)P(\sim UT|ED)&
\end{aligned}$$



Conditional Probabilities
P(ED) = 0.8
P(UT) = 0.4
P(FP|UT,ED)=0.1
P(FP|UT,~ED)=0.8
P(FP| ~UT,ED)=0.5
P(FP| ~UT,~ED)=0.95

Figure 4.1. A sample Bayesian network to illustrate Bayesian inference.

We can write $P(UT|ED) = P(UT)$ and $P(\sim UT|ED) = P(\sim UT)$ since the variables ED and UT are independent. Then we have,

$$\begin{aligned} P(FP|ED) &= P(FP|ED, UT)P(UT) \\ &+ P(FP|ED, \sim UT)P(\sim UT) \end{aligned}$$

Feeding up the values in the conditional probability table, $P(FP|ED)$ is calculated as 0.34. Assume that we are asked to calculate the probability of having experienced developers given the software is fault prone. So, now we shall calculate $P(ED|FP)$. Using Bayes' rule we write

$$P(ED|FP) = \frac{P(FP|ED)P(ED)}{P(FP)} \tag{4.3}$$

We can also write

$$\begin{aligned} P(FP) &= P(FP|UT, ED)P(UT)P(ED) \\ &+ P(FP|UT, \sim ED)P(UT)P(\sim ED) \\ &+ P(FP| \sim UT, ED)P(\sim UT)P(ED) \\ &+ P(FP| \sim UT, \sim ED)P(\sim UT)P(\sim ED) \end{aligned} \tag{4.4}$$

Since $P(FP|UT, ED)$, $P(FP|UT, \sim ED)$, $P(FP| \sim UT, ED)$, and $P(FP| \sim UT, \sim ED)$ can be read from the conditional table, the diagnosis probability $P(ED|FP)$ can also be calculated.

As it can be seen in these examples of causal and diagnostic inferences, it is possible to propagate the effect of states of variables (nodes) to calculate posterior probabilities. Propagating the effects of variables to the successors, or analyzing the probability of some predecessor variable based on the probability of its successor is very important in defect prediction since software metrics are related to each other and that is why the effect of a metric might be dependent on another metric based on some relationship.

4.1.1. K2 Algorithm

In Bayesian network structure learning, the search space is composed of all of the possible structures of directed acyclic graphs based on the given variables (nodes). Normally, it is very difficult to enumerate all of these possible directed acyclic graphs without a heuristic method. Because, when the number of nodes increases, the search space grows exponentially and it is almost impossible to search the whole space. Given a data set, the K2 algorithm proposed by Cooper and Herskovits, heuristically searches for the most probable Bayesian network structure [75]. Based on the ordering of the nodes, the algorithm looks for parents for each node whose addition increases the score of the Bayesian network. If addition of a certain node X_j to the set of parents of node X_i does not increase the score of the Bayesian network, K2 stops looking for parents of node X_i further. Since the ordering of the nodes in the Bayesian network is known, the search space is much more smaller compared to the entire space that needs to be searched without a heuristic method. Furthermore, a known ordering ensures that there will be no cycles in the Bayesian network, so there is no need to check for cycles too.

K2 algorithm takes a set of n nodes, an initial ordering of the n nodes, the maximum number of parents of any node denoted by u and a database D of m cases

```
for $i \leftarrow 1$ to $n$ do
    $\pi_i := 0$;
    $P_{old} := f(i, \pi_i)$ (See Equation 4.5);
    $OKToProceed :=$ true;
    while $OKToProceed$ and $|\pi_i| < u$ do
        let $z$ be the node in $Pred(x_i) - \pi_i$ that maximizes $f(i, \pi_i \cup \{z\})$;
        $P_{new} := f(i, \pi_i \cup \{z\})$;
        if $P_{new} > P_{old}$ then
            $P_{old} := P_{new}$;
            $\pi_i := \pi_i \cup \{z\}$
        else
            $OKToProceed :=$ false
        end
    end
    write(Node: $x_i$, Parent of $x_i : \pi_i$)
end
```

Algorithm 1: The K2 algorithm [75].

as input and outputs a list of parent nodes for every node in the network. The pseudo code of the K2 algorithm is given in Algorithm 1. For every node in the network, the algorithm finds the set of parents with the highest probability taking into consideration the upper bound u for the maximum number of parents a node can have. During each iteration, the function $Pred(x_i)$ is used to determine the set of nodes that precede x_i in the node ordering. The algorithm calculates the probability that the parents of x_i are π_i using the following equation:

$$f(i, \pi_i) = \prod_{j=1}^{q_i} \frac{(r_i - 1)!}{(N_{ij} + r_i - 1)!} \prod_{k=1}^{r_i} N_{ijk}! \tag{4.5}$$

where π_i is the set of parents of x_i, $q_i = |\phi_i|$ where ϕ_i is the set of all possible instances of the parents of x_i for database D. Furthermore, $r_i = |V_i|$ where V_i is the set of all possible values of x_i. On the other hand, N_{ijk} is the number of instances in database D for which x_i is instantiated with its k^{th} value, and the parents of x_i in the set π_i are

Table 4.1. Correlational analysis of KC1 metrics [1].

METRIC	CBO	LCOM	NOC	RFC	WMC	DIT	SLOC	DC
CBO	1.000							
LCOM	0.041	1.000						
NOC	-0.129	0.119	1.000					
RFC	0.542	0.383	-0.032	1.000				
WMC	0.388	0.378	0.101	0.666	1.000			
DIT	0.470	0.249	-0.112	0.718	0.215	1.000		
SLOC	0.818	0.010	-0.136	0.523	0.496	0.385	1.000	
DC	0.520	-0.006	-0.156	0.245	0.352	0.036	0.560	1.000

instantiated with the j^{th} instantiation in the set ϕ_i. And lastly,

$$N_{ij} = \sum_{k=1}^{r_i} N_{ijk}$$

gives the number of instances in D where the parents of x_i are instantiated with the j^{th} instantiation in ϕ_i.

4.2. Previous Work on Bayesian Networks

Pai and Dugan use Bayesian network to analyze the effect of object oriented metrics [42] on the number of defects (fault content) and defect proneness using KC1 project from the NASA metrics data repository [1]. They build a Bayesian network where parent nodes are the object oriented metrics (also called C-K metrics) and child nodes are the random variables fault content and fault proneness.

They make a Spearman correlation analysis to check whether the variables of the model (metrics) are independent or not. They show that almost all metrics have a correlation at a moderate level. On the other hand, almost all metrics are related with the size metric whereas only CBO, RFC, WMC, and SLOC metrics are highly related to the fault content (See Table 4.1). Having determined that CBO, RFC, WMC, and

Table 4.2. Prediction performance of variable set v_2 is higher [1].

Predictor	Specificity	Sensitivity	Precision	FN Rate	FP Rate
v_1	0.630	0.526	0.676	0.400	0.284
v_2	0.852	0.597	0.752	0.277	0.235

SLOC metrics are highly correlated with defectiveness (both fault content and fault proneness), they have defined two sets of variables so that the first set (v_1) includes these variables and the second set (v_2) includes these variables plus an additional variable LCOM. They use multiple linear regression to compare the predictive power of these two variable sets v_1 and v_2. Since results for fault content and fault proneness are similar, for simplicity we consider the results for fault proneness only. As it is seen in Table 4.2, the predictive power of v_2 is higher since v_2 achieves greater precision while having small false positive and false negative rates.

Assuming that adding LCOM to the variable set improves the performance of Bayesian network model, they use the Bayesian network model in Figure 4.2 that includes LCOM and observe that when a class is picked up randomly from the KC1 system, with 37.2 percent it has at least one or more defects. Although the findings found are related to the data set chosen to some extent, they claim that using Bayesian network modeling that relates product metrics to defectiveness is applicable across different software systems.

According to Zhang [33], Bayesian networks provide a very suitable and useful method for software defect prediction. He suggests to build a Bayesian network that reflects all software development activities like specification, design, implementation, testing and considers Bayesian network generation in three steps. His suggestion is very straightforward and explains the use of Bayesian network in defect prediction very well. Bayesian network generation steps defined by Zhang are:

1. Define Bayesian network variables. There are three different types of variables in the Bayesian network.
    - Information variables: These variables reflect the states that are observable

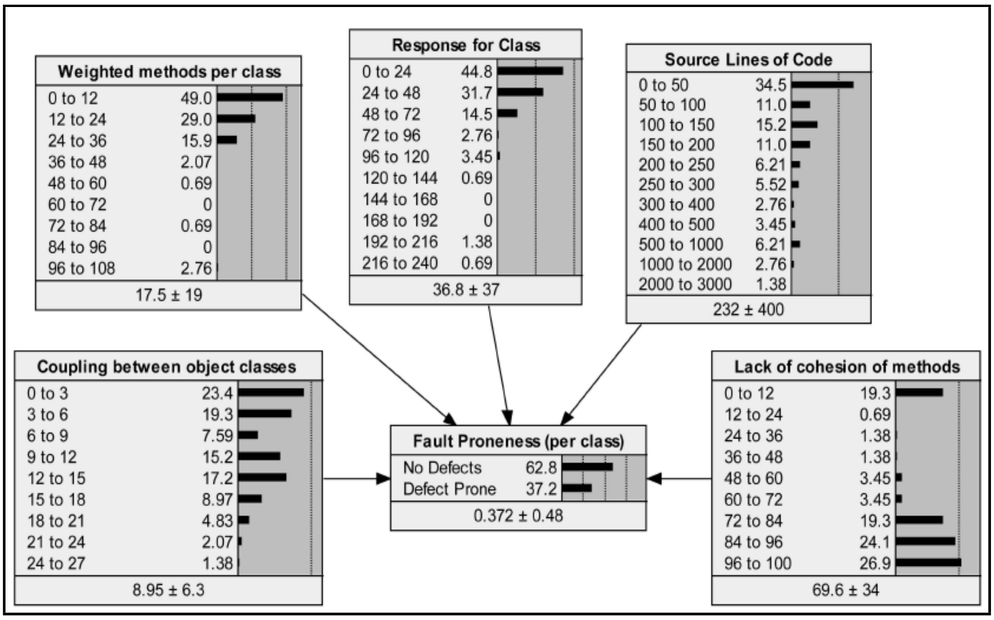


Figure 4.2. Bayesian network model for fault proneness analysis [1].

in the Bayesian network.

- Hypothesis variables: These are target variables for which we would like to calculate the probability distribution.
- Mediating variables: These are introduced for several purposes like reflecting the independence either between any information variables or information variables and the target variable.

2. Define the causal relationships among the network variables. These relationships are the arcs of the Bayesian network.
3. Generate the probability distribution of each variable in the network and calculate the joint probability distribution of the hypothesis variables.

Fenton *et al.* [32] suggest to use Bayesian network for defect, quality and risk prediction of software systems. They use the Bayesian network shown in Figure 4.3 to model the causal relationships among target variable “defects detected” (DD) and the information variables “test effectiveness” (TE) and “defects present” (DP). In this network, DP models the number of bugs/defects found during testing. TE gives the efficiency of testing activities and DD gives the number of defects delivered to the maintenance phase. For discretization, they assign two very simple states to each

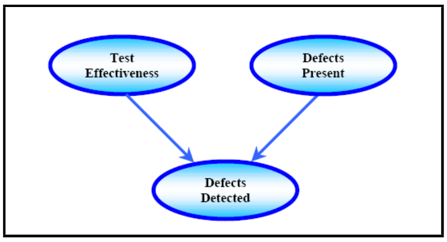


Figure 4.3. Bayesian network suggested by Fenton *et al.*

variable namely low and high. Using the Bayes' rule we have,

$$P(E|C) = \frac{P(C|E)P(E)}{P(C)} \tag{4.6}$$

where E represents the evident variable and C the class variable we are interested in.

To calculate $P(TE = High|DD = High)$, using Bayes' rule we have,

$$P(TE = High|DD = High) = \frac{P(DD = High|TE = High)P(TE = High)}{P(DD = High)} \tag{4.7}$$

From this equation it can be seen how a high DD rate affected by Test Effectiveness (TE) although not by a dramatic amount.

Using the Bayesian network model above, Fenton *et al.* show how Bayesian network is providing accurate results for software quality and risk management in a range of real world projects. They conclude that Bayesian networks can be used to model the causal influences in a software development project and the network model can be used to ask "What if?" questions under circumstances when some process underperforms [32].

In another study by Fenton and Neil, it is claimed that the proportion of defects introduced are influenced by many factors other than static code or design metrics [4]. Examples of such factors are:

- The complexity of the problem itself.
- The complexity of the software designed.

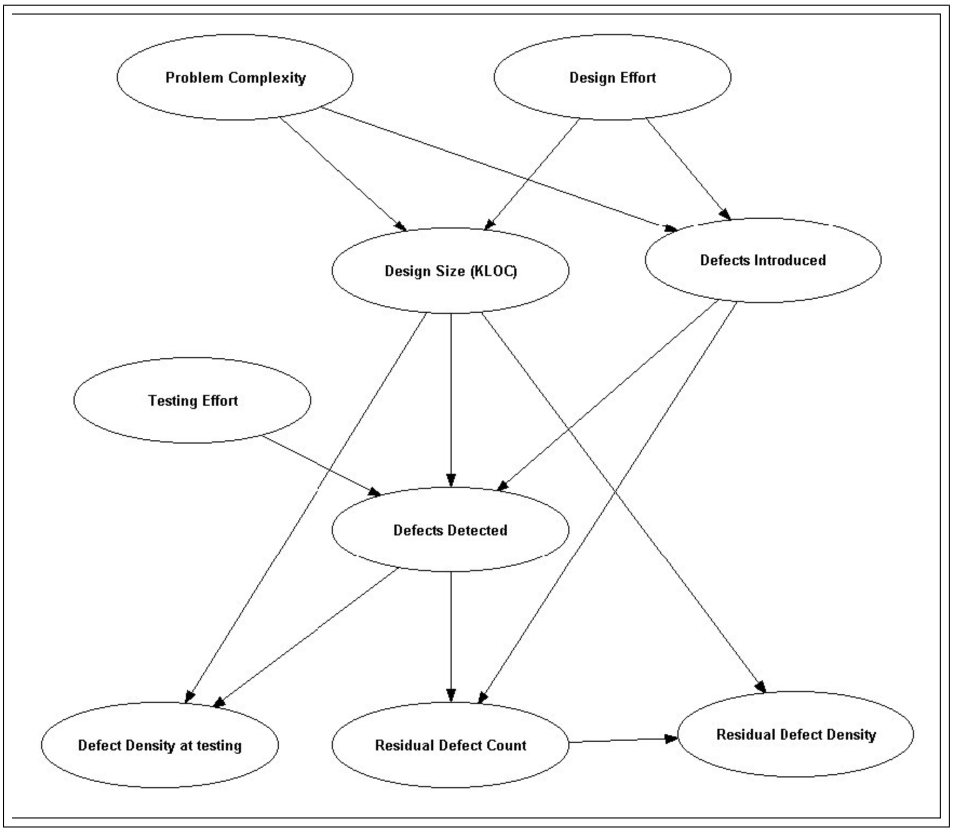


Figure 4.4. Bayesian network topology proposed by Fenton and Neil for defect prediction [4].

- The experience of the development team.
- Methods used in the software life cycle.

These factors are also effective on defect proneness and should be modeled. They propose the Bayesian networks shown in the Figure 4.4 to model the effect of these factors.

It is claimed that if the problem is complex, then more defects are introduced and that is why the size of the design is larger. At the same time, design effort affect the size of the design and the number of introduced defects. On the other hand, introduced defects are the sum of detected and residual defects. Moreover, testing effort affects the number of defects detected and detected defects influence the density of defects during testing [4].

On the other hand, Zhou and Leung use logistic regression and machine learning methods (naive Bayes network, random forest, and nearest neighbor with generalization) to determine the importance of object oriented metrics for determining fault severity [76]. They state a hypothesis for each object oriented metric and test them on open source NASA data set KC1 [48]. For ungraded severity, it was found that SLOC, CBO, WMC and RFC are significant in fault proneness prediction. Furthermore, LCOM is also significant but when tested with machine learning methods the usefulness of NOC is poor and the result is similar for DIT.

Bibi *et al.* use iterative Bayesian networks for effort estimation by modeling the sequence of the software development processes and their interactions. They state that Bayesian networks provide a highly visual interface to explain the relationships of the software processes and provide a probabilistic model to represent the uncertainty in their nature. They conclude that Bayesian networks could be used for software effort estimation effectively [77]. Furthermore, Minana and Gras use Bayesian networks to predict the level of fault injection during the phases of a software development process. They show that Bayesian networks provide a successful fault prediction model [78].

Amasaki *et al.* propose to use Bayesian networks to predict the quality of a software system. To generate a Bayesian network they use certain metrics collected during the software development phase like product size, effort, detected faults, test items, and residual faults. They conclude that the proposed model can estimate the residual faults that the software reliability growth model can not handle [79].

Fenton *et al.* review different techniques for software defect prediction and conclude that traditional statistical approaches like regression alone is not enough. Instead they believe that causal models are needed for more accurate predictions. They describe a Bayesian network, to model the relationship of different software life cycles and conclude that there is a good fit between predicted and actual defect counts [80]. In another study, Fenton *et al.* propose to use Bayesian networks to predict software defects and software reliability and conclude that using dynamic discretization algorithms while generating Bayesian networks leads to significantly improved accuracy for defects and reliability prediction [81].

In another research, Dejaeger *et al.* compare 15 different Bayesian network classifiers with famous defect prediction methods on 11 Data sets in terms of the AUC and H-measure. They observe that simple and comprehensible Bayesian networks can be constructed other than the simple Naive Bayes model and recommend to use augmented Bayesian network classifiers when the cost of not detecting a defective or non defective module is not higher than the additional testing effort [82]. Furthermore, as future work, they propose to discover the effects of different information sources with Bayesian networks which is something we consider by defining two extra metrics i.e. LOCQ and NOD and measuring their relationship with other metrics and defectiveness.

Regarding the effects of the number of developers on defect proneness there are contradictory findings in the literature. For example Norick *et al.* use 11 open source software projects, to determine if the number of committing developers affects the quality of a software system. As a result, they could not find significant evidence to claim that the number of committing developers affects the quality of software [83]. Furthermore, Pendharkar and Rodger investigate the impact of team size on the software development effort using over 200 software projects and conclude that when the size of the team increases, no significant software effort improvements are seen [84].

On the other hand, Nagappan *et al.* define a metric scheme that includes metrics like number of engineers, number of ex-engineers, edit frequency, depth of master ownership, percentage of organization contributing to development, level of organizational code ownership, overall organization ownership, and organization intersection factor to quantify organizational complexity. They use data from Windows Vista operating system and conclude that the organizational metrics predict failure-proneness with significant precision, recall, and sensitivity. Furthermore, they also show that organizational metrics are better predictors of failure-proneness than the traditional metrics used so far like code churn, code complexity, code coverage, code dependencies, and pre-release defect measures [85].

Furthermore Mockus *et al.* use two open source projects, the Apache web server and the Mozilla browser to define several hypotheses that are related to the developer count and the team size. They test and refine some of these based on an the analysis of Mozilla data set. They believe that when several people work on the same code, there are many potential dependencies among their work items. So, they suggest that regarding to the team size, around an upper limit of 10-15 people, coordination of the work for the team becomes inadequate [86].

Chapter 5

Kernel Machines

In general, defect prediction studies use so many metrics together to analyze the defect proneness of software system. Most of the time, it is not so easy to extract all metrics needed for the analysis. That is why it is very important to be able to achieve defect prediction with limited number of software metrics. At this point kernel methods can be used since they allow prediction of defects with limited number of metrics compared to other alternatives. This leads to spending less computing power and time which is very important for the purpose of defect prediction studies.

5.1. Background on Kernel Machines

In this section, we provide the technical background of support vector machines and kernel functions.

5.1.1. Support Vector Machines

Support vector machines generate the optimal hyperplane (or hypersphere, depending on the kernel) that can be used for classification or regression [87]. The optimal hyperplane is found by maximizing the margin which is defined as the distance between two nearest instances from either side of the hyperplane. As it is shown in the Figure 5.1, the hyperplane is $\frac{1}{\|\mathbf{w}\|}$ away from each class and it has a margin of $\frac{2}{\|\mathbf{w}\|}$.

Let's assume that we have a training set

$$S = \left\{(\mathbf{x}^i, y^i), \mathbf{x}^i \in R^t, y^i \in \{-1, 1\}\right\}_{i=1}^{n} \tag{5.1}$$

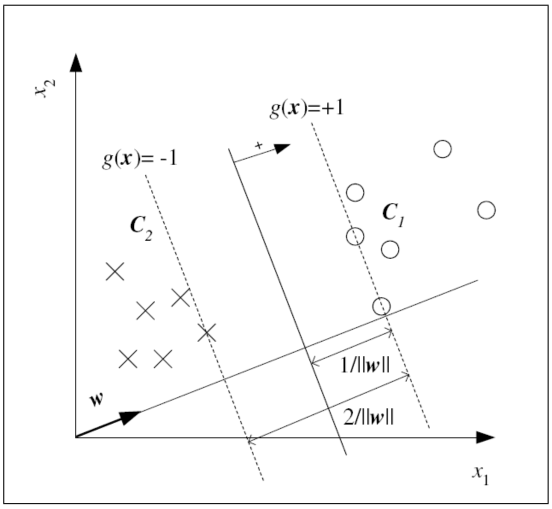


Figure 5.1. An optimal separating hyperplane [5].

where y^i's are either +1 (positive class) or -1 (negative class) and each $\mathbf{x}^i$ vector (with t entries) belongs to one of these classes. Based on this assumption, an hyperplane is defined as

$$g(\mathbf{x}) = \mathbf{w}^T\mathbf{x} - b \tag{5.2}$$

where $\mathbf{w}$ shows the vector normal to the optimal hyperplane and $b/\left\|\mathbf{w}\right\|$ is the offset of the hyperplane from the origin on the direction of $\mathbf{w}$. In order to calculate the margin of the optimum hyperplane, we need to find the boundaries of the two classes as hyperplanes first and then take the distance among these two hyperplanes. The boundary hyperplane of the positive class can be written as $g(\mathbf{x}) = 1$ whereas the boundary hyperplane of the negative class is $g(\mathbf{x}) = -1$.

Assuming that we have a linearly separable data set, the distance to the origin is $(b+1)/\left\|\mathbf{w}\right\|$ for the first hyperplane and $(b-1)/\left\|\mathbf{w}\right\|$ for the second one. The distance between these two hyperplanes is $2/\left\|\mathbf{w}\right\|$. The optimum hyperplane separating these two classes maximizes this margin or minimizes $\left\|\mathbf{w}\right\|$.

For positive instances we have

$$\mathbf{w}^T\mathbf{x}^i - b \geq 1 \tag{5.3}$$

and for negative instances we have

$$\mathbf{w}^T\mathbf{x}^i - b \leq -1 \tag{5.4}$$

These two constraints can be combined as $y^i(\mathbf{w}^T\mathbf{x}^i - b) \geq 1$. Now our problem becomes an optimization problem of

$$\begin{gathered}\text{Minimize } \|\mathbf{w}\|^2 \\ \text{s. t. } y^i(\mathbf{w}^T\mathbf{x}^i - b) \geq 1\end{gathered} \tag{5.5}$$

To control the sensitivity of SVM and tolerate possible outliers, slack variables (ξ) are introduced. Then the problem changes slightly and becomes an optimization problem of

$$\begin{gathered}\text{Minimize } \|\mathbf{w}\|^2 + C\sum \xi^i \\ \text{s. t. } y^i(\mathbf{w}^T\mathbf{x}^i - b) \geq 1 - \xi^i\end{gathered} \tag{5.6}$$

where the constant $C > 0$ determines the relative importance of maximizing the margin and minimizing the amount of slack.

Compared to the hard constraint defined in Equation 5.5, here some amount of slackness is allowed and it is represented by ξ^i's. In this new representation, we observe that if $0 < \xi^i \leq 1$, the data point lies between the margin and the correct side of the hyper plane. On the other hand if $\xi^i > 1$, then the data point is misclassified. Data points that lie on the margin are called support vectors and regarded as the most important data points in the training set. In Figure 5.2, support vectors are shown on a diagram where a hyperplane is separating two types of instances.

So far, we have assumed that the data points are linearly separable. What about the case when the data points are not linearly separable where we would not be able

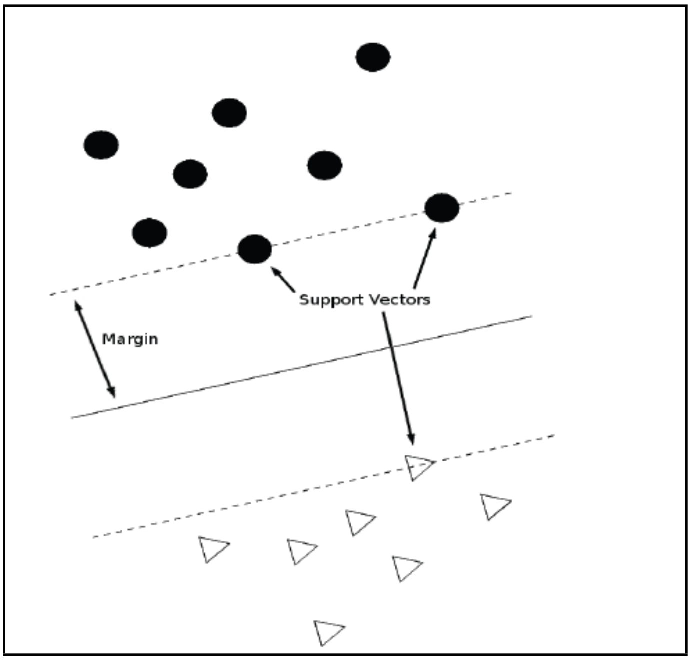


Figure 5.2. A binary classification toy problem: separate dots from triangles. The solid line shows the optimal hyperplane. The dashed lines parallel to the solid one show how much one can move the decision hyperplane without misclassification [6].

to define a hyperplane or hyperspehere to separate different classes? Consider the data points in Figure 5.3. In the first graph on the left hand side, the points are not linearly separable i.e. one can not draw a line or hyperplane to separate the classes given. But, when a transformation is applied and the graph on the right hand side is generated, then it is possible to achieve linear separation. The solution is to use a suitable transformation function to carry the data points to a high dimension where linear separation is possible. This is called kernel trick in the literature. The classification is based on the dot product of two vectors as $K(\mathbf{x}^i, \mathbf{x}^j) = (\mathbf{x}^i)^T \mathbf{x}^j$. Assuming that the transformation function θ is defined as $\theta : \mathbf{x} \rightarrow \Phi(\mathbf{x})$, our new dot product in the high dimensional space becomes $K(\mathbf{x}^i, \mathbf{x}^j) = \left\langle \Phi(\mathbf{x}^i)^T, \Phi(\mathbf{x}^j) \right\rangle$. So, the kernel function can be defined as a function that returns the inner product between the images of two inputs in a feature space where the image function is shown with Φ in our representation.

Assume that we have two dimensional data, i.e. $\mathbf{x}$ is in the form of $[x_1, x_2]$. Let's also assume that our kernel function is $K(\mathbf{x}^i, \mathbf{x}^j) = (1 + (\mathbf{x}^i)^T \mathbf{x}^j)^2$. We need to show that $K(\mathbf{x}^i, \mathbf{x}^j)$ is the inner product of some function $\Phi(\mathbf{x})$. Expanding $K(\mathbf{x}^i, \mathbf{x}^j) =$

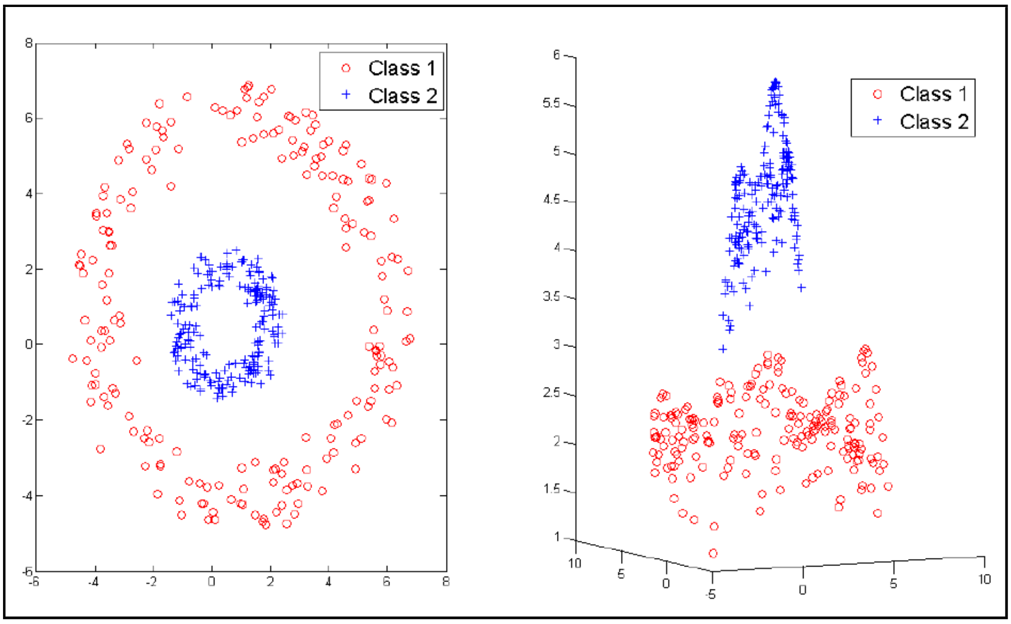


Figure 5.3. Transforming data points to higher dimensional space [7].

$(1 + (\mathbf{x}^i)^T \mathbf{x}^j)^2$ we have,

$$K(\mathbf{x}^i, \mathbf{x}^j) = (1 + ((x^i)_1(x^j)_1 + (x^i)_2(x^j)_2))^2$$
$$= 1 + 2(x^i)_1(x^j)_1 + 2(x^i)_2(x^j)_2 + (x^i)_1^2(x^j)_1^2 + 2(x^i)_1(x^j)_1(x^i)_2(x^j)_2 + (x^i)_2^2(x^j)_2^2$$

Now we can write $K(\mathbf{x}^i, \mathbf{x}^j)$ as the inner product of two vectors,

$$K(\mathbf{x}^i, \mathbf{x}^j) = \left[1, \sqrt{2}(x^i)_1, \sqrt{2}(x^i)_2, (x^i)_1^2, \sqrt{2}(x^i)_1(x^i)_2, (x^i)_2^2\right]$$
$$\left[1, \sqrt{2}(x^j)_1, \sqrt{2}(x^j)_2, (x^j)_1^2, \sqrt{2}(x^j)_1(x^j)_2, (x^j)_2^2\right]$$

Interpreting this result in the form of $K(\mathbf{x}^i, \mathbf{x}^j) = \Phi(\mathbf{x}^i)^T \Phi(\mathbf{x}^j)$ as we defined above, we have

$$\Phi(\mathbf{x}) = \left[1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, \sqrt{2}x_1x_2, x_2^2\right] \tag{5.7}$$

We have shown that there exist a kernel function $K(\mathbf{x}^i, \mathbf{x}^j)$ for a transformation from 1 to 2 dimensional space.

Checking whether any function constitutes a valid kernel, $K(\mathbf{x}^i, \mathbf{x}^j) = \left\langle \Phi(\mathbf{x}^i)^T, \Phi(\mathbf{x}^j) \right\rangle$ is a difficult task. Mercer's theorem states that every semi-positive definite symmetric function is a kernel. Checking semi-positive definiteness for a function corresponds to checking semi-positive definiteness of the matrix in the form of

$$\mathbf{K} = \begin{bmatrix} K(\mathbf{x}^1, \mathbf{x}^1) & K(\mathbf{x}^1, \mathbf{x}^2) & ... & K(\mathbf{x}^1, \mathbf{x}^N) \\ K(\mathbf{x}^2, \mathbf{x}^1) & K(\mathbf{x}^2, \mathbf{x}^2) & ... & K(\mathbf{x}^2, \mathbf{x}^N) \\ ... & ... & ... & ... \\ K(\mathbf{x}^N, \mathbf{x}^1) & K(\mathbf{x}^N, \mathbf{x}^2) & ... & K(\mathbf{x}^N, \mathbf{x}^N) \end{bmatrix} \tag{5.8}$$

The kernel matrices satisfy the semi-positive definiteness property, because a matrix K with real entries is said to be semi-positive definite if for any vector $\mathbf{v}$ with real components, the dot product of $\mathbf{Kv}$ and $\mathbf{v}$ is nonnegative, that is $\langle \mathbf{Kv}, \mathbf{v} \rangle \geq 0$.

Considering the definition of the kernel matrix, let $K^{ij} = K(\mathbf{x}^i, \mathbf{x}^j) = \langle \Phi(\mathbf{x}^i), \Phi(\mathbf{x}^j) \rangle$ for $i, j = 1, 2, ..., m$, where m is the size of the kernel matrix. For any vector $\mathbf{v}$ we can write

$$\begin{aligned} \mathbf{v}K\mathbf{v} &= \sum_{i,j=1}^{m} \mathbf{v^i v^j} K^{ij} \\ &= \sum_{i,j=1}^{m} \mathbf{v^i v^j} \left\langle \Phi(\mathbf{x}^i)^T, \Phi(\mathbf{x}^j) \right\rangle \\ &= \left\langle \sum_{i=1}^{m} \mathbf{v^i} \Phi(\mathbf{x}^i), \sum_{j=1}^{m} \mathbf{v^j} \Phi(\mathbf{x}^j) \right\rangle \\ &= \left\| \sum_{i=1}^{m} \mathbf{v^i} \Phi(\mathbf{x}^i) \right\|^2 \geq 0 \end{aligned} \tag{5.9}$$

Furthermore, if we apply the positive semi definiteness test and see that a matrix K is not a valid kernel matrix, then we can produce $K^2 = K^T K$ which is guaranteed to be a valid kernel, since for any vector $\mathbf{v}$

$$\begin{aligned} \mathbf{v}K^2\mathbf{v} &= \mathbf{v}KK\mathbf{v} \\ &= \|\mathbf{K}\mathbf{v}\|^2 \geq 0 \end{aligned} \tag{5.10}$$

There are other alternative ways of converting any symmetric but non semi-positive matrix to a valid kernel matrix. Some of these transformation methods are denoise, flip, diffusion, and shift and their effect on classification using kernel machines are given by Wu *et al.* [88].

5.1.2. Support Vector Machines for Regression

Let's assume that we have a training set where y^t's are output values corresponding to each $\mathbf{x}^t$ input vector.

In SVM regression proposed by Vapnik *et al.* [87] the goal is to find a function $g(\mathbf{x})$ that has at most ϵ deviation from the actually obtained target y^t values for all of the training data, and at the same time is as flat as possible. It means that the error e introduced by the function $g(\mathbf{x})$ must be less than ϵ for all possible $\mathbf{x}$ inputs. If the function $g(\mathbf{x})$ is defined as:

$$g(\mathbf{x}^t) = \mathbf{w}^T\mathbf{x}^t + b \tag{5.11}$$

the ϵ-sensitive error function can be defined as:

$$e_\epsilon(y^t, g(\mathbf{x}^t)) = \left\{ \begin{array}{cc} 0 & \text{if } |y^t - g(\mathbf{x}^t)| < \epsilon \\ |y^t - g(\mathbf{x}^t)| - \epsilon & \text{otherwise} \end{array} \right\} \tag{5.12}$$

where for the function to be flat, $\mathbf{w}$ should be small. To ensure a small $\mathbf{w}$, we can minimize $\|\mathbf{w}\|^2 = (\mathbf{w}^T\mathbf{w})$. Then, we can write this problem as a convex optimization problem of:

$$\text{Minimize } \frac{1}{2}\left\|\mathbf{w}\right\|^2$$
$$\text{s. t.} \begin{Bmatrix} y^t - (\mathbf{w}^T\mathbf{x}^t + b) \leq \epsilon \\ \\ \mathbf{w}^T\mathbf{x}^t + b - y^t \leq \epsilon \end{Bmatrix} \tag{5.13}$$

To control the sensitivity of SVM and tolerate possible outliers, slack variables (ξ_+, ξ_-) are introduced. Then the problem changes slightly and becomes an optimization problem of

$$\text{Minimize } \frac{1}{2}\left\|\mathbf{w}\right\|^2 + C\sum_t(\xi_+^t + \xi_-^t)$$
$$\text{s. t.} \begin{Bmatrix} y^t - (\mathbf{w}^T\mathbf{x}^t + b) \leq \epsilon + \xi_+^t \\ \\ \mathbf{w}^T\mathbf{x}^t + b - y^t \leq \epsilon + \xi_-^t \\ \\ \xi_+^t, \xi_-^t \geq 0 \end{Bmatrix} \tag{5.14}$$

where the constant $C > 0$ determines the trade-off between the flatness of $g(\mathbf{x})$ and the amount up to which deviations larger than ϵ are tolerated and two types of slack variables (ξ_+, ξ_-) are used, for positive and negative deviations, to keep them positive. This formulation also corresponds to the ϵ-sensitive error function given in Equation 5.12. The Lagrangian is:

$$\begin{aligned} L_p =& \frac{1}{2}\left\|\mathbf{w}\right\|^2 + C\sum_t(\xi_+^t + \xi_-^t) - \sum_t \alpha_+^t(\epsilon + \xi_+^t - y^t + \mathbf{w}^T\mathbf{x}^t + b) \\ & - \sum_t \alpha_-^t(\epsilon + \xi_-^t + y^t - \mathbf{w}^T\mathbf{x}^t - b) - \sum_t(\mu_+^t\xi_+^t + \mu_-^t\xi_-^t) \end{aligned} \tag{5.15}$$

Taking the partial derivatives of the Lagrangian and substituting the results we get the dual:

$$L_d = -\frac{1}{2}\sum_t \sum_s (\alpha_+^t - \alpha_-^t)(\alpha_+^s - \alpha_-^s)(\mathbf{x}^t)^T \mathbf{x}^s \\ -\epsilon \sum_t (\alpha_+^t + \alpha_-^t) - \sum_t y^t(\alpha_+^t - \alpha_-^t) \tag{5.16}$$

subject to

$$0 \leq \alpha_+^t \leq C, 0 \leq \alpha_-^t \leq C, \sum_t (\alpha_+^t - \alpha_-^t) = 0$$

When we solve this, we see that all instances that fall in the regression tube have $\alpha_+^t = \alpha_-^t = 0$ where these are instances that are fitted with enough precision. On the other hand, the support vectors are either $\alpha_+^t > 0$ or $\alpha_-^t > 0$ and are of two types. They can either be the instances on the boundary of the regression tube and can be used to calculate b in $g(\mathbf{x})$. Instances that fall outside of the regression tube are of the second type and we do not have a good fit for them ($\alpha_+^t = C$).

The dot product $(\mathbf{x}^t)^T\mathbf{x}^s$ in Equation 5.16 can be replaced with a suitable kernel function $K(\mathbf{x}^t, \mathbf{x}^s)$ to have a nonlinear fit. For example, when we use a polynomial kernel we can fit to a polynomial or when we use a Gaussian kernel we can have a nonparametric smoothing model.

5.1.3. Kernel Functions

Kernel methods are used to extract and model the similarities in a set of data points. This way, kernel methods make it possible to classify a set of data points that are not linearly related.

There are many types of kernel functions used in the literature. The function used is very much dependent on the type of the problem. For example a polynomial kernel allow us to model the feature conjunctions up to the order of the polynomial.

On the other hand, radial basis functions allow us to generate hyperspheres or circles whereas linear kernel generates hyperplanes or lines. Some kernel functions used in the literature are:

1. Linear kernel: Linear kernel is the simplest kernel function and it is the dot product of two vectors $\mathbf{x}^i$ and $\mathbf{x}^j$ plus a constant vector c. Compared to other kernel functions it is used less in defect prediction studies [39].

$$K(\mathbf{x}^i, \mathbf{x}^j) = (\mathbf{x}^i)^T \mathbf{x}^j + c. \tag{5.17}$$

2. Gaussian kernel: Gaussian kernel is a type of radial basis function and used in the literature for the purpose of defect prediction successfully [6]. It models the similarities among data points by exponentiating the square of the distance among vectors $\mathbf{x}^i$ and $\mathbf{x}^j$.

$$K(\mathbf{x}^i, \mathbf{x}^j) = \exp(-\gamma(\|\mathbf{x}^i - \mathbf{x}^j\|)^2) \tag{5.18}$$

where γ value must be tuned very carefully, not to cause any overestimation or underestimation. Furthermore Gaussian kernel has some advantages compared to other types of kernel functions, since it is suitable for nonlinear modeling, uses less parameters, and requires less computing power especially when compared with polynomial kernels [89].

3. Exponential kernel: It is also a radial basis function and also very much similar to the Gaussian kernel except that it uses just the distance of the vectors $\mathbf{x}^i$, $\mathbf{x}^j$ rather than its square.

$$K(\mathbf{x}^i, \mathbf{x}^j) = \exp(-\frac{\|\mathbf{x}^i - \mathbf{x}^j\|}{2\sigma^2}). \tag{5.19}$$

5.1.4. String Kernels

String kernels are used in areas like software plagiarism detection, document classification and filtering, information retrieval, and DNA analysis [90]. In all of these areas, the key point is to extract and measure similarities of text passages or strings. Below we list and briefly explain the most important and widely used string kernels in the literature.

1. Levenshtein distance (edit-distance) kernel: The edit distance kernel gives the distance between two strings s and t, by taking into account the number of insertions, deletions or substitutions needed to convert s to t. Assuming the string s has length m and t has length n, then $E(m+1, n+1)$ entry of the edit distance matrix gives the edit distance between strings s and t. Let f be a function where c_1 and c_2 are two characters.

   $$f(c_1, c_2) = \left\{ \begin{matrix} 0 & \text{if } c_1 = c_2 \\ 1 & \text{otherwise} \end{matrix} \right\}$$

   The edit distance matrix is filled in an iterative manner by first setting $E(i, 0) = i$ for $i = 1, 2, ..., m$ and $E(0, j) = j$ for $j = 1, 2, ..., n$. Then, each entry of the matrix is computed by

   $$E(i, j) = min \left\{ E(i-1, j) + 1; E(i, j-1) + 1; E(i-1, j-1) + f(s(i), t(j) \right\}$$

   where $s(i)$ is the i^{th} character of string s and $t(j)$ is the j^{th} character of string t.
2. Bag of words kernel: In document categorization or similarity extraction the collection of available documents is called a 'corpus' and the set of words in the dictionary are called 'terms'. A term is a sequence of letters and is used synonymously with word. A document is represented as a vector where each dimension is associated with a term in the dictionary. Since the order of the words in the document is not important in this representation, the model is

called 'Bag of Words'. The mapping θ is given by:

$$\theta : d \rightarrow \Phi(d) = (tf(t_1, d), tf(t_2, d), ...tf(t_N, d))$$

where $tf(t_i, d)$ is the frequency of the i^{th} term in the document d and N is the size of the dictionary [91]. Two documents d_1 and d_2 can be compared using the kernel:

$$K(d_1, d_2) = \langle \Phi(d_1), \Phi(d_2) \rangle = \sum_{j=1}^{N} tf(t_j, d_1) tf(t_j, d_2)$$

where documents d_1 and d_2 have N dimensions [91].

3. P-gram kernel: In this type of kernel, an alphabet is defined as a finite set symbols Σ and a string is defined as a finite sequence of symbols taken from alphabet Σ. Σ^n is the set of all strings with length n and Σ^* is the collection of all available strings where we can write:

$$\Sigma^* = \bigcup_{n=0}^{\infty} \Sigma^n$$

Furthermore, st represents the concatenation of strings s and t. If string s and its substrings u, v_1, v_2 are related as

$$s = v_1 u v_2$$

then, if v_1 is empty string then u is called a prefix of s, and if v_2 is empty then u is called a suffix of s. A feature map Φ can be defined as:

$$\Phi_u^n(s) = |\{(v_1, v_2) : s = v_1 u v_2|\}, u \in \Sigma^n$$

and the associated kernel matrix between strings s and t is defined as [91]:

$$K_n(s, t) = \langle \Phi^n(s), \Phi^n(t) \rangle = \sum_{u \in \Sigma^n} \Phi_u^n(s) \Phi_u^n(t)$$

.

5.2. Previous Work on Kernel Machines

5.2.1. Plagiarism Tools

According to Mozgovoy [92], plagiarism tools mainly use two methods namely fingerprint based techniques and content comparison techniques to detect plagiarism.

In fingerprint based methods, a fingerprint is generated for each file or class in the software system. During fingerprint generation, some statistical metrics (e.g. the number of operators or operands, the number of unique keywords) are used. A mathematical distance function is defined to compare the fingerprints of each file. If the fingerprints are similar (the mathematical distance among them are small) then the corresponding files are similar.

Content comparison techniques (structure metric systems) take into consideration the continuous matching strings in files inspected. Each file (or class) is converted into tokens and similar substrings are searched in these tokens. The percentage of similarity between the tokens shows the extent of plagiarism between their corresponding files (or classes).

5.2.2. Similarity Detection

Given a sentence s and a document d, Barrón-Cedeño and Rosso [93] try to answer the question 'is s plagiarized from d?' by using p-gram kernel. They show that as the size of the p-gram increases, the probability of finding common p-gram decreases and bigrams and trigrams are the best comparison sequences for automatic plagiarism detection based on p-grams.

Martins [94] review kernel methods like the bag-of-words kernel, the p-spectrum kernel and the weighted all-substrings kernel to measure string similarity for text authorship attribution, language detection, and cross-language document matching. He show that for the first two text categorization issues, all kernel methods yield accuracies close to 100 percent provided that the kernel parameters are properly fine-tuned.

While checking the similarity of source code, some code clone and plagiarism tools like JPlag uses greedy-string-tiling algorithm in comparing strings and generating similarity matrix. On the other hand, local alignment (Smith-Waterman) algorithm was developed to find similarities between nucleotide or protein sequences [95]. Local alignment finds similar contiguous subsequences of two strings and generates similarity matrix among subsequences. Ji *et al.* [96] use an adaptive version of local alignment where they took into consideration the frequencies of the subsequences while generating the kernel matrix. They show that the adaptive local alignment algorithm used is more robust compared to the greedy-string-tiling algorithm used in JPlag [97].

Roy *et al.* [98] classify the clone detection techniques based on their clone detection logic and compared their performance. They define four clone types i.e. type-1, type-2, type-3, and type-4 clones and classified clone detection tools and techniques according to which clone type they are able to detect. A type-1 clone is defined as identical code fragments where only layout, comments and white spaces might be changed. A type-2 clone is viewed as identical code fragments except identifiers, literals, white spaces or comments might be changed. In type-3 clones, besides literal or identifier variations, there might be some changes, additions or deletions in the statements. Lastly, type-4 clone implies functional similarity where two programs carry out the same task but with different source codes. The first three clone types (type-1 to type-3) consider syntactic similarities whereas type-4 clone considers functional or semantic similarities between code fragments. Roy *et al.* [98] show that graph based clone techniques (where functional similarity is detected) and metric based clone detection techniques are more successful compared to other text, tree or token based techniques.

5.2.3. Kernel Methods for Defect Prediction

Xing *et al.* state that support vector machines can model nonlinear relationships among the data points that are difficult to model with other techniques. With SVM, it is possible to predict the quality or defect proneness of a software system even when there are small number of software metrics [60].

The defect prediction models in the literature use many software metrics and that is why it is very difficult to generalize the results of a prediction model. It was shown that the results achieved in a study for one data set might not be valid for other data sets [38, 39].

Shin *et al.* use radial basis function with Gaussian kernel since its approximation capability is high compared to other kernels. Using Gaussian kernel with two software metrics (that is line of code (L) and cyclomatic complexity (V)), they show that defect proneness prediction is possible even with just these two metrics [8]. They use MDP data sets KC1 and KC2 for their study. They show that defectiveness is high when both L and V are high. When L and V are low, no defect proneness observed. This interesting observation for KC1 is shown in Figure 5.4. Note that class labels that are greater than zero (defect proneness) is observed when L is around 1000 and V is around 140.

Beside defect prediction, software risk prediction is another area that researchers focus on. A software system is regarded risky when it is defect prone. Hu *et al.* [35] compare ANN with SVM for software risk prediction. They find that, since SVM is capable of giving good results even with small training data, it is more successful than ANN in predicting software risks. The results show that ANN is capable of achieving prediction accuracy of 75 % whereas SVM is achieving 85 %.

Software defect prediction is very much related to the quality, risk and maintenance cost of a software system. A system with small number of defects or no defects at all, can be accepted as qualified and riskless (or low risk). Furthermore, the mainte-

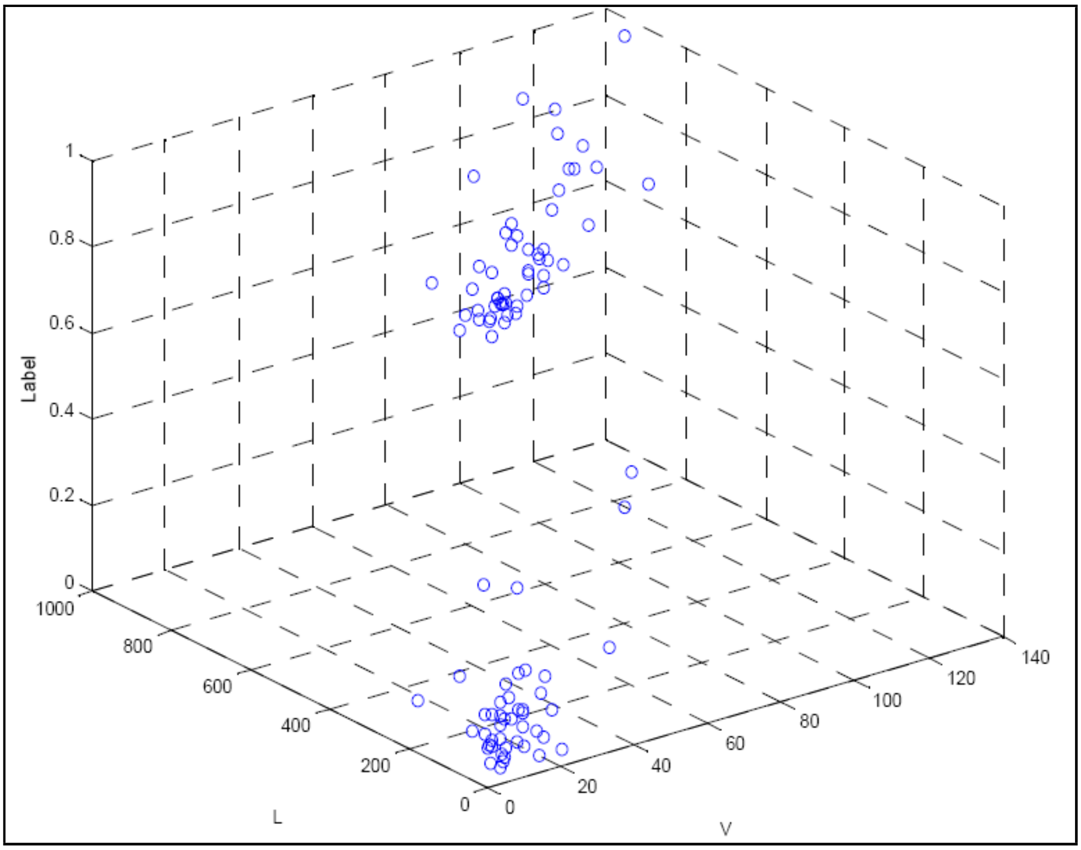


Figure 5.4. Class label versus (L, V) for KC1 [8].

Table 5.1. Object oriented metrics studied by Jin and Liu [2].

WMC	Weighted Methods per Class
RFC	Response for a Class
DAC	Data Abstraction Coupling
MPC	Message Passing Coupling
NOM	Number of Methods per Class

nance cost of such a system is low compared to systems with high number of defects. Jin and Liu [2] study the effort that will be spent in maintenance phase of a software system using 5 object oriented metrics shown in Table 5.1. The maintenance effort is defined by the number of lines changed per class where a line change could be an addition or a deletion. They conclude that support vector machines method using just 5 object oriented metrics is more successful to measure maintenance effort compared to unsupervised learning algorithms.

Dealing with so many software metrics is not easy both in terms of computing

Table 5.2. Average classifier performance of SVM and Naive Bayes observed by Shivaji *et al.* [3].

Technique	Features Percentage	Accuracy	Precision	Recall	F-measure
Bayes F-measure	6.83	0.91	0.96	0.67	0.79
SVM F-measure	7.49	0.81	0.82	0.54	0.62
Bayes accuracy	6.92	0.86	0.92	0.53	0.65
SVM accuracy	7.83	0.86	0.96	0.51	0.65

power and time. That is why, it is very crucial and desirable to work with less metrics if possible. Shivaji *et al.* [3] use feature selection to decrease the number of attributes while measuring defect proneness. They decrease the number of attributes 10 % each time and try to see at which stage the best results are obtained. They find that between 4.1% and 12.52 % of the total feature set, an optimal classification is achieved. As classification techniques, they focus on the use of SVM and naive Bayes methods and observe that SVM with feature selection performs considerably better than naive Bayes (See Table 5.2).

Chapter 6

Proposed Method

In this section we explain the novel techniques we propose for defect prediction using Bayesian networks and kernel methods. First, we explain how do we use Bayesian networks, in order to model the influential relationships of software metrics and measure their effect on defect proneness. Second, we explain the kernel method we propose that makes defect prediction possible without using any software metrics but just the available source codes.

6.1. Bayesian networks

Bayesian network is one of the most successful defect prediction models tried so far and can be used to model the inherent uncertainties in the nature of software development [34, 1, 33, 32, 4]. Bayesian networks allow to take into consideration a broad range of metrics at the same time like process, product, and developer metrics. Meanwhile they can model the influential relationships among different defect causes. In this section we explain our proposed method for defect prediction using Bayesian network model.

6.1.1. Bayesian network of Metrics and Defect Proneness

It is very important to model the associational relationships among the metrics and defect proneness. We first generate a Bayesian network among software metrics and defect proneness and then using this network, we calculate an overall marginal defectiveness probability of the software system. This network provides us two very important results:

Table 6.1. List of software metrics used to generate the Bayesian networks in Experiment I (During Experiment II, two more metrics LOCQ and NOD are added).

Metric	**Metric full name**
WMC	Weighted method per class [42]
DIT	Depth of inheritance tree [42]
NOC	Number of children [42]
CBO	Coupling between objects [42]
RFC	Response for class [42]
LCOM	Lack of cohesion of methods [42]
LCOM3	Lack of cohesion in methods [99]
LOC	Lines of code

- The dependencies among the metrics we choose. Which metrics are affected by other metrics and which ones are the most effective on defect proneness.
- The defect proneness probability of the software system itself. By learning from the data set, the Bayesian network tells us the marginal defectiveness probability of the whole system and one can interpret this as the probability of having at least one or more defects in a software module that is selected randomly.

In defect prediction studies, using static code metrics alone may ignore some very crucial causes of defects like poor requirement analysis or design, lack of quality of design or coding, unexperienced developers, bad documentation, managerial or financial problems. Although all of these factors could lead to an increase in defect proneness, static code metrics do not consider them effectively. Using Bayesian networks might be much more meaningful when additional data on causal, explanatory variables are available and included in the model. Unfortunately, it is not too easy to measure these causal and explanatory variables when there is no information regarding the software development processes. By inspecting project repositories of some data sets, we add two metrics i.e. LOCQ and NOD to our Bayesian model, in order to measure the effect of lack of coding quality and the number of developers.

Besides the object oriented metrics and line of code metric listed in Table 6.1, we introduce a new metric we call lack of coding quality (LOCQ) that measures the quality of the source code. We run PMD source code analyzer plugin in Netbeans, to generate the LOCQ values for each class of the open source Apache projects listed in Table 7.4. PMD inspects the given source code and looks for potential problems like possible bugs, dead code, suboptimal code, overcomplicated expressions, and duplicate code. It counts the number of detected problems for each class and package in the software system. We believe that this measurement gives an idea about the quality of the source code and has a relationship with defectiveness. That is why, we include the LOCQ metric in our experiments and try to understand how it is related with the defect proneness and other well known static code metrics in the literature.

We ask if for a specific class or file, the number of developers is positively related with the extent of defectiveness or not? Receiving inspiration from famous idiom “too many cooks spoil the broth” we wonder if a higher number of developers for a certain class or file, leads to a more defective or messed up source code? For some of the data sets listed in Table 7.4 that have developer information in the source code files, we generate the number of developers (NOD) metric, which shows the number of distinct developers per each class in the software system. Then we learn a Bayesian network from each of these data sets and extract the relationship of the NOD metric with defectiveness.

One problem to reach a clear conclusion on this issue is the conclusion instability problem. We think that conclusion instability comes from an inherent property of software engineering data sets; i.e. real world data is noisy. To remove noise from the data sets and draw more accurate conclusions, we cross check our results on 10 subsets of each data set. While generating the subsets, we stratify the data with different seeds and include 67 percent of the data each time. For the remaining data sets like Ant, Lucene, and Synapse, there was no developer information in the source code repositories, so we could not generate the NOD metric for them. Since the developer information is not available for all data sets but just for a subset of them, we present the experiments carried out with the NOD metric separately in Section 7.2.5.

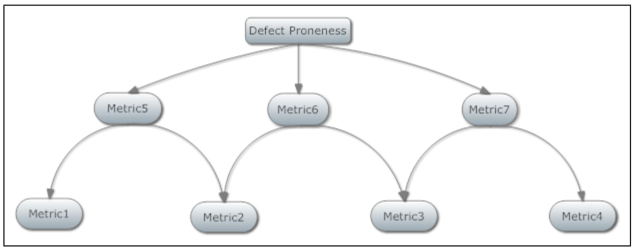


Figure 6.1. Proposed Bayesian network to model the relationships among metrics.

While we model the influential relationships among different product and process metrics, we learn the Bayesian network from the data set. Figure 6.1 shows the general form of our model. In this Bayesian network, we see the interactions among different product, process or developer metrics. We may see that a metric is not affected by any other metric whereas some metrics may be affected by one or more product metrics (like $Metric_5$). According to this Bayesian network $Metric_5$, $Metric_6$, and $Metric_7$ are the most important metrics since they affect defectiveness directly. On the other hand, $Metric_1$, $Metric_2$, $Metric_3$, and $Metric_4$ are less important since they are indirectly related with defectiveness.

As a summary, our Bayesian network is a graph G of E edges and V vertices where each V_i represents a metric and each E_j represents the dependency between two metrics or between a metric and defectiveness. If an edge E is present from metric m_2 towards metric m_1, then this would mean metric m_1 is effective on metric m_2. Similarly, if there is an edge from defectiveness to metric m_1, then it would mean that metric m_1 is effective on defectiveness. This way, we determine the metrics that affect defectiveness directly or indirectly.

6.1.2. Ordering Metrics for Bayesian Network Construction

In order to learn a Bayesian network with K2 algorithm, it is necessary to specify the order of the nodes. That is why, we decide to order the software metrics considering their effect on defectiveness, prior to the generation of Bayesian networks.

We believe that as the size of a software system gets larger, the probability of having fault prone classes increases, since more effort would be needed to ensure a

Table 6.2. The order of software metrics used during Bayesian network construction.

Metric Groups	**Order (left to right)**		
$Group_1$	LOC	CBO	LOCQ
$Group_2$	WMC	RFC	
$Group_3$	LCOM - LCOM3	DIT	NOC

defect free software. We also believe that besides size, complexity of the software is also very important because as the design gets more complex, it would be more difficult for developers to ensure non-defectiveness. That is why, for the initial ordering the metrics, we decide to give LOC and CBO as the first metrics since LOC is the best indicator of software size and CBO shows how much complex a software system is by counting the number of couples for a certain class where coupling means using methods or instance variables of other classes. As one would easily accept, as coupling increases, the complexity of the software system would also increase. Furthermore, as everybody can accept, when the quality of the source code increase, the probability of having a defective software decrease. So, we introduce the LOCQ metric as the third metric in the first group after LOC and CBO.

Although, RFC may explain complexity to some extent, it may not be the case if a class is using internal methods or instance variables only. That is why, RFC together with WMC are entitled as the second group of metrics. On the other hand, NOC indicates the number of children of a class and is not a good indicator for both size and complexity, since the parent-child relationship does not contribute to the complexity if there is no caller-callee relationship between them which is the case for most of the time. Due to similar reasons DIT also does not explain size or complexity alone. So, we decide to give DIT and NOC as the last metrics in the initial ordering.

Following our reasoning, we generate three group of metrics where LOC, CBO, and LOCQ are in $Group_1$, WMC and RFC are in $Group_2$ and LCOM, DIT, and NOC are in $Group_3$. $Group_1$ metrics are more important than $Group_2$ metrics and $Group_2$ metrics are more important than $Group_3$ metrics in terms of their effect on defectiveness (See Table 6.2).

In defect prediction, it is very crucial to model the dependencies among different modules in a software system. If say a software module is dependent on three highly defect prone modules, then we would expect a high defectiveness rate for this module. That is why, rather than calculating defectiveness of software modules locally, it is a good idea to model the effect of parent nodes (modules that call this module) on these nodes. The first step is to extract the dependency graph of a software system. It is a graph G of V vertices and E edges, where each vertex represents different software modules in the system and an edge represents the dependency between vertices. For instance, there will be an edge e_1 from vertice v_1 to v_2, if the module that corresponds to v_1 is calling a sub routine from the module that represents v_2. This way we can generate a dependency graph of the software system using Bayesian network model and then use this Bayesian network model to calculate the influential relationships among different modules by learning the conditional probability tables of the network. Using this Bayesian network, we see the modules that are affected most by other modules and also are effective on the defectiveness of the whole system. We deduce that modules with higher marginal defectiveness probability are more critical and should be tested more carefully. We plan to explore the inter module influential relationships with Bayesian networks as a future work.

6.2. Kernel Methods to Predict Defectiveness

Kernel methods make it possible to model the relationships among data points that do not have a linear correlation. When a set of data points is given as input to a kernel method, it is possible to classify these data points and say whether any two data points are in the same group or not. Furthermore, the relationships among different data points (metrics and defectiveness in this case) are not known and finding these relations is one of the goals of defect prediction. As it is explained in the literature review above, the importance of kernel methods have been noticed by researchers and many articles have been published to prove the efficiency and superiority of this technique [60, 8, 35, 2, 3].

The main idea of our research is that the extent of similarity between two source codes is related with the extent of similarity in their defectiveness. We believe that similarity must be measured in terms of both syntactic and semantic features. There are many clone detection techniques in the literature most of which detects only lexical similarity and we believe they are not good means for source code similarity extraction. On the other hand, some plagiarism tools like MOSS and JPlag consider structural similarity together with fingerprint based approaches and we believe that they are more suitable to measure the similarity we want. From the output of the plagiarism tools, we generate a kernel matrix and use it to learn the relationship of source code similarity and defectiveness.

We use SVM with a precomputed kernel matrix composed of similarity measurements among the files in a software system. We give the kernel matrix with the class information of files (defective or not) to the SVM classifier and make the prediction process easier compared to traditional methods.

Assume that the software system is composed of one or more versions, in total of N files (modules) where each m_i represents a software file (module). Then the kernel matrix is defined as

$$K = \begin{bmatrix} K(m_1,m_1) & K(m_1,m_2) & ... & K(m_1,m_N) \\ K(m_2,m_1) & K(m_2,m_2) & ... & K(m_2,m_N) \\ ... & ... & ... & ... \\ K(m_N,m_1) & K(m_N,m_2) & ... & K(m_N,m_N) \end{bmatrix} \tag{6.1}$$

where $K(m_i, m_j)$ represents the similarity between files m_i and m_j. Similarity can be measured by calculating the number or percentage of similar code patterns.

The similarity matrix is generated by using plagiarism software. The plagiarism software tells us how much parallelism exists among files (or classes) of a given software. Assuming there are N files in the software system, the output generated by the plagiarism software is used to create an N by N kernel matrix that is given as input to the SVM classifier.

Table 6.3. An example kernel matrix showing similarity between files of a software system composed of $File_1$, $File_2$, $File_3$, and $File_4$.

	$File_1$	$File_2$	$File_3$	$File_4$
$File_1$	99	25	0	0
$File_2$	25	99	15	12
$File_3$	0	15	99	0
$File_4$	0	12	0	99

An example kernel matrix for an hypothetical software system that is composed of $File_1$, $File_2$, $File_3$ and $File_4$ is shown in Table 6.3. The values in this matrix show the percentage of similarity among the files of the system. For instance $File_1$ and $File_2$ are 25 percent similar. A value of zero means no similarity exits which is the case for files $File_1$ and $File_3$. Furthermore, the diagonal similarity scores in the kernel matrix are high since a file is compared with itself.

Although it is not a necessity that the similarity based matrices we generate are semi-definite, the kernel matrices computed for the data sets in our experiments satisfy the positive semi-definiteness attribute and are therefore valid kernels.

We normalized the precomputed kernel matrices to the unit norm, as this may effect their generalization ability and also result in a smaller range for C. The normalized matrix K_n is defined as

$$K_n(x, z) = \frac{K(x, z)}{\sqrt{K(x, x)K(z, z)}} \tag{6.2}$$

for all examples x, z.

6.2.1. Selecting Plagiarism Tools and Tuning Their Input Parameters

Plagiarism tools measure structural similarity using fingerprint based and content comparison techniques. JPlag [97] and MOSS [100] are two of the most famous and widely used plagiarism tools worldwide that measure structural similarity in source codes.

JPlag [97], is an open source plagiarism tool that can be used through a web service. JPlag is capable of measuring similarity in terms of the number of similar tokens. JPlag has some user defined parameters such as minimum match length and maximal number of matches and these parameters must be tuned very carefully not to affect the results negatively. To adjust these parameters, -especially the minimum match length- the user has to know the underlying string matching algorithm. If the minimum match length is chosen to be very small, then some coincidental similarities may be included in the results and the results might be less meaningful. Furthermore, as it is confirmed in some previous studies [101], we also confirm that JPlag is not very successful in detecting all similarities and the kernel matrix generated from the JPlag is more sparse compared to the MOSS. That is why, during our experiments, we select MOSS to extract similarities among classes or files of software systems and to generate the precomputed kernel matrix for the SVM classifier.

MOSS (Measure Of Software Similarity) is a tool developed by Alex Aiken and hosted by Stanford University [100]. MOSS has a web interface and the users are capable of submitting source code to detect plagiarism. For the purpose of plagiarism, two parameters i.e. m and n need to be tuned very carefully. m represents the maximum number of times a given passage may appear before it is ignored. If a certain substring appears more than m times, then it is not regarded as plagiarism but may be accepted as a library function that is used frequently in many files. For our purpose, m needs to be high, because our aim is to measure similarity rather than plagiarism, so all types of similarities are meaningful for us (We set m 1,000,000 in our experiments). n represents the maximum number of matching files to include in the results and is set to 250 by default. Since we need to extract as much similarity as

Table 6.4. List of software metrics used for SVM with linear and RBF kernels.

Metric	**Metric full name**	**Source**
wmc	Weighted method per class	[42]
dit	Depth of inheritance tree	[42]
noc	Number of children	[42]
cbo	Coupling between Objects	[42]
rfc	Response for class	[42]
lcom	Lack of cohesion of methods	[42]
ca	Afferent couplings	[102]
ce	Efferent coupling	[102]
npm	Number of public methods	[103]
lcom3	Lack of cohesion in methods	[99]
loc	Lines of code	
dam	Data access metric	[103]
moa	Measure of aggregation	[103]
mfa	Measure of functional abstraction	[103]
cam	Cohesion among methods of class	[103]
ic	Inheritance coupling	[104]
cbm	Coupling between methods	[104]
amc	Average method complexity	[104]
max_cc	max. (McCabe's cyclomatic complexity)	[36]
avg_cc	avg. (McCabe's cyclomatic complexity)	[36]

possible, we set this value high (around 5,000) depending on the size of the attribute set we tried to learn.

We believe that there is no need to tune m and n in the experiments. First, a high m value is needed in order not to miss any similarity information. It should be higher than the maximum number of times a token or passage can appear in a software system and our setting (1,000,000) always satisfies this criteria for the data sets we chose. Second, n should be large enough to include all similarities among the files in a software system (even similarity scores less than 5 percent), and the setting of 5,000 was appropriate to list all similarity information in our data sets.

6.2.2. Data Set Selection

First, we looked at the data sets that are large enough to perform 5×2 cross validation. That is why, we eliminated some of the data sets that have less than 100 entries and are small in terms of size. Second, since we use the source code to extract similarity kernel matrix, we need the source code corresponding to exactly the same version specified in the defect data. So, we preferred data sets whose source code is available in the open source project repositories. For instance the Log4j data set has defect data for versions 1.0, 1.1, and 1.2 in the Promise data repository, but the sources corresponding to these versions are not present in Apache repositories. Although the sources are available for many sub versions of 1.0, 1.1, and 1.2, there is no source code marked with version 1.0, 1.1, or 1.2 exactly. That is why, we eliminated some more data sets whose sources are not available or open for the versions specified in the Promise data repository. Based on these criteria, we used open source projects Camel, Tomcat, Poi, Xalan, JEdit, Velocity, Ant, Lucene, Synapse, and Ivy shown in Table 6.5 from Promise data repository [49]. Furthermore, the object oriented and other type of software metrics included in these datasets are listed in Table 6.4. In experiments for SVM with linear or RBF kernels, these metrics are used to explore the relationship between metrics and defect proneness.

6.2.3. Kernel Matrix Generation

For our experiments both defect data and source code of the projects are used together. Defect data is used to get the class information whereas the source code is used to extract similarities across files and generate the kernel matrix. The value at (i, j) index of the kernel matrix shows how much structural parallelism exists between the i 'th and j 'th files of the software system. The parallelism was measured by the average of the percentage of similarity between corresponding files. If there are no shared substrings then the (i, j) term of the kernel matrix is set to zero.

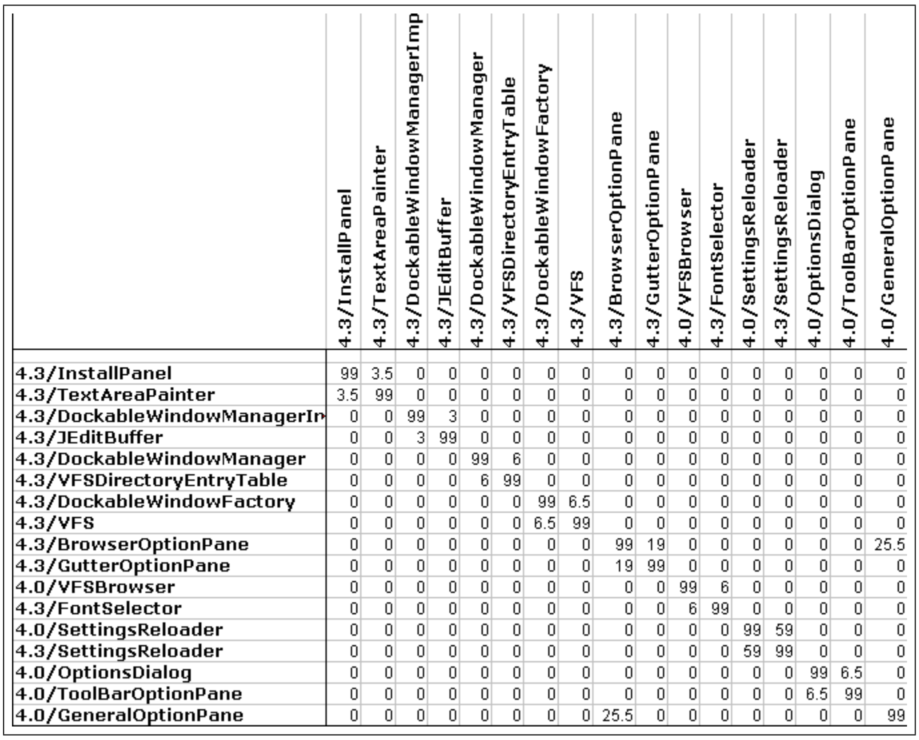
Table 6.5. The 10 data sets used in the experiments to predict defectiveness and the number of defects with Kernel methods (Experiment III and IV).

Data Set	**Version**	**Num. of Instances**
Camel	1.0	339
Tomcat	6.0	858
Poi	3.0	442
Xalan	2.5	803
JEdit	4.0	306
Velocity	1.5	214
Ant	1.7	745
Lucene	2.4	340
Synapse	1.2	256
Ivy	2.0	352

Table 6.6. A sample part of MOSS output that shows similarity scores of files in a software system that is composed of 4.0 and 4.3 versions.

File 1	**File 2**	**Similarity (%)**
4.3/InstallPanel.java	4.3/TextAreaPainter.java	3.5
4.3/DockableWindowImpl.java	4.3/JEditBuffer.java	3
4.3/DockableWindow.java	4.3/VFSDirectoryEntryTable.java	6
4.3/DockableWindowFactory.java	4.3/VFS.java	6.5
4.3/BrowserOptionPane.java	4.3/GutterOptionPane.java	19
4.0/VFSBrowser.java	4.3/FontSelector.java	6
4.0/SettingsReloader.java	4.3/SettingsReloader.java	59
4.0/OptionsDialog.java	4.0/ToolBarOptionPane.java	6.5
4.0/GeneralOptionPane.java	4.3/BrowserOptionPane.java	25.5

We use MOSS in CygWin environment on Windows. MOSS outputs a table giving file based similarities (An example MOSS output is shown in Table 6.6). The third column of the table (Similarity) shows the percentage of similarity detected for the files in the first and second columns.

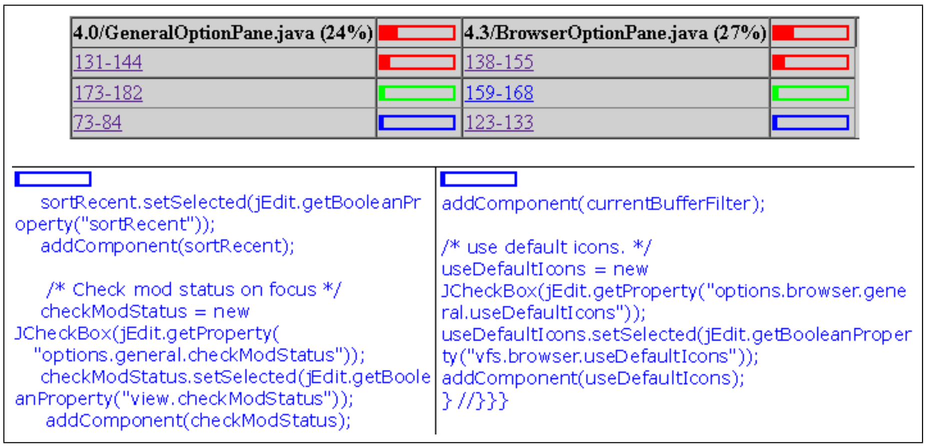


	4.3/InstallPanel	4.3/TextAreaPainter	4.3/DockableWindowManagerImp	4.3/JEditBuffer	4.3/DockableWindowManager	4.3/VFSDirectoryEntryTable	4.3/DockableWindowFactory	4.3/VFS	4.3/BrowserOptionPane	4.3/GutterOptionPane	4.0/VFSBrowser	4.3/FontSelector	4.0/SettingsReloader	4.3/SettingsReloader	4.0/OptionsDialog	4.0/ToolBarOptionPane	4.0/GeneralOptionPane
4.3/InstallPanel	99	3.5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4.3/TextAreaPainter	3.5	99	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4.3/DockableWindowManagerIr	0	0	99	3	0	0	0	0	0	0	0	0	0	0	0	0	0
4.3/JEditBuffer	0	0	3	99	0	0	0	0	0	0	0	0	0	0	0	0	0
4.3/DockableWindowManager	0	0	0	0	99	6	0	0	0	0	0	0	0	0	0	0	0
4.3/VFSDirectoryEntryTable	0	0	0	0	6	99	0	0	0	0	0	0	0	0	0	0	0
4.3/DockableWindowFactory	0	0	0	0	0	0	99	6.5	0	0	0	0	0	0	0	0	0
4.3/VFS	0	0	0	0	0	0	6.5	99	0	0	0	0	0	0	0	0	0
4.3/BrowserOptionPane	0	0	0	0	0	0	0	0	99	19	0	0	0	0	0	0	25.5
4.3/GutterOptionPane	0	0	0	0	0	0	0	0	19	99	0	0	0	0	0	0	0
4.0/VFSBrowser	0	0	0	0	0	0	0	0	0	0	99	6	0	0	0	0	0
4.3/FontSelector	0	0	0	0	0	0	0	0	0	0	6	99	0	0	0	0	0
4.0/SettingsReloader	0	0	0	0	0	0	0	0	0	0	0	0	99	59	0	0	0
4.3/SettingsReloader	0	0	0	0	0	0	0	0	0	0	0	0	59	99	0	0	0
4.0/OptionsDialog	0	0	0	0	0	0	0	0	0	0	0	0	0	0	99	6.5	0
4.0/ToolBarOptionPane	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6.5	99	0
4.0/GeneralOptionPane	0	0	0	0	0	0	0	0	25.5	0	0	0	0	0	0	0	99

Figure 6.2. A sample kernel matrix generated from the MOSS output shown in Table 6.6.

An example kernel matrix generated from the MOSS output given in Table 6.6 is shown in Figure 6.2. We see that files that have high structural similarity have high percentage score in the kernel matrix. For example, the files GeneralOptionPane.java from version 4.0 and BrowserOptionPane.java from version 4.3 are 25.5 percent similar. This is something expected, since both of the classes in these files are derived from the same parent. Similarly, SettingsReloader.java in version 4.0 is 59 percent similar to SettingsReloader.java in version 4.3, which means the file was not changed too much in version 4.3. Moreover, the diagonal terms are very high since the file is compared with itself.

MOSS provides more detailed information about the parts of the source code where similarities have been found (See Figure 6.3). First of all, a master table is presented showing the sections of files that are similar. Each row of the master table shows the extent of the similarity found between specified sections of the files where the amount of the similarity is represented with colorful bars. Secondly, when a row is clicked then the similar parts of files are shown. As it is seen in the similar sections



4.0/GeneralOptionPane.java (24%)	🟥	4.3/BrowserOptionPane.java (27%)	🟥
131-144	🟥	138-155	🟥
173-182	🟩	159-168	🟩
73-84	🟦	123-133	🟦

```
   sortRecent.setSelected(jEdit.getBooleanPr
operty("sortRecent"));
   addComponent(sortRecent);

    /* Check mod status on focus */
   checkModStatus = new
JCheckBox(jEdit.getProperty(
  "options.general.checkModStatus"));
   checkModStatus.setSelected(jEdit.getBoole
anProperty("view.checkModStatus"));
    addComponent(checkModStatus);
```

```
addComponent(currentBufferFilter);

/* use default icons. */
useDefaultIcons = new
JCheckBox(jEdit.getProperty("options.browser.gene
ral.useDefaultIcons"));
useDefaultIcons.setSelected(jEdit.getBooleanProper
ty("vfs.browser.useDefaultIcons"));
addComponent(useDefaultIcons);
} //}}}
```

Figure 6.3. A detailed MOSS output about the similarities between files GeneralOptionPane and BrowserOptionPane from Table 6.6.

of the files GeneralOptionPane.java (lines 73-84) and BrowserOptionPane.java (lines 123-133) in Figure 6.3, structural parallelism is captured while syntactic similarities are ignored.

6.3. Kernel Methods to Predict the Number of Defects

We propose to use MOSS [100] in extracting class or file based similarities and generate a precomputed kernel matrix to predict the number of defects. One can submit batches of source code to MOSS server and get a link to the HTML page showing the similar or plagiarized source code parts for the submitted source code. MOSS uses robust winnowing algorithm, which is more efficient and scalable since it selects fewer finger prints for the same quality of results than previous algorithms tried [105].

Imagine a software system that has N files and a file is represented as f_i. Then the precomputed kernel matrix we generate from MOSS is represented as:

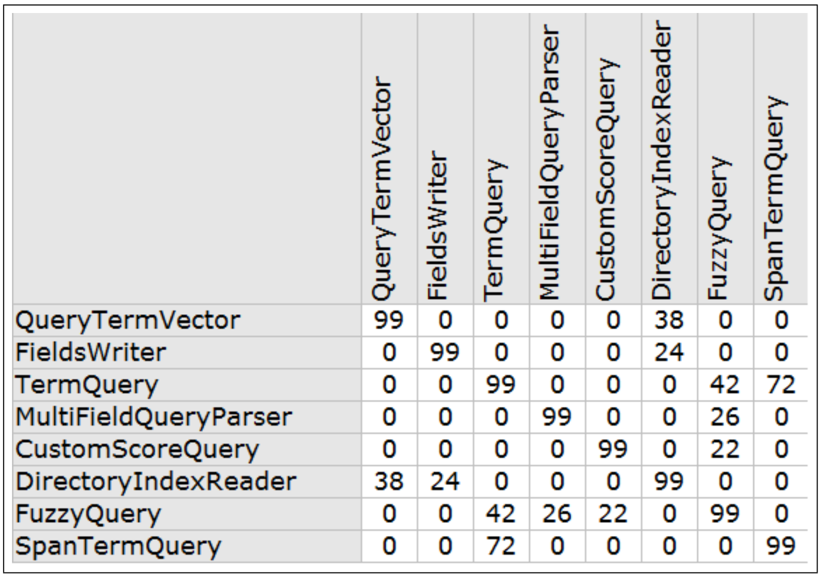
$$K = \begin{bmatrix} K(f_1,f_1) & K(f_1,f_2) & ... & K(f_1,f_N) \\ K(f_2,f_1) & K(f_2,f_2) & ... & K(f_2,f_N) \\ ... & ... & ... & ... \\ K(f_N,f_1) & K(f_N,f_2) & ... & K(f_N,f_N) \end{bmatrix} \tag{6.3}$$

where $K(f_i, f_j)$ shows the similarity between files f_i and f_j. This kernel matrix is

Table 6.7. Part of the MOSS output for Lucene data set.

Class1	**Class2**	**Similarity (%)**
QueryTermVector	DirectoryIndexReader	38
FieldsWriter	DirectoryIndexReader	24
TermQuery	FuzzyQuery	42
TermQuery	QueryTermVector	38
TermQuery	SpanTermQuery	72
MultiFieldQueryParser	FuzzyQuery	26
CustomScoreQuery	FuzzyQuery	22

generated from the output of MOSS and is given as a precomputed kernel to SMOReg method, to predict the number of defects.



	QueryTermVector	FieldsWriter	TermQuery	MultiFieldQueryParser	CustomScoreQuery	DirectoryIndexReader	FuzzyQuery	SpanTermQuery
QueryTermVector	99	0	0	0	0	38	0	0
FieldsWriter	0	99	0	0	0	24	0	0
TermQuery	0	0	99	0	0	0	42	72
MultiFieldQueryParser	0	0	0	99	0	0	26	0
CustomScoreQuery	0	0	0	0	99	0	22	0
DirectoryIndexReader	38	24	0	0	0	99	0	0
FuzzyQuery	0	0	42	26	22	0	99	0
SpanTermQuery	0	0	72	0	0	0	0	99

Figure 6.4. A sample kernel matrix generated from the MOSS output shown in Table 6.7.

To illustrate the precomputed kernel matrix generation with a simple example, a small part of the MOSS output for Lucene data set is shown in Table 6.7. Similarity between each class file pair is shown in terms of percentages. For instance, TermQuery and SpanTermQuery are 72 % similar which is something expected since both are descendants of the base Query class. Furthermore, the classes TermQuery and FuzzyQuery are found to be 42 percent similar, which can also be justified by looking at the similar code pieces and inheritance relationship in each class. For instance, the class TermQuery is extended from the base Query class, whereas the FuzzyQuery is extended from an abstract class named MultiTermQuery which is an extension of the base Query class.

A kernel matrix generated from the output in Table 6.7 is shown in Figure 6.4. Assigning the class files of the software system as rows and columns of the kernel matrix, each cell is filled using the similarity output generated by the plagiarism tool. We also observe that the diagonal terms in the kernel matrix are filled with very high percentage values since a class is compared with itself. Furthermore, when there is no similarity detected between any two files, the value in the corresponding cell of the kernel matrix is zero.

We use Weka [106] to compare the performance of the proposed kernel method with linear, RBF kernels and other defect prediction methods LinR and IBK. In each experiment, 5×2 fold cross-validation is used and all the necessary parameters of SVM (C, ϵ, and γ) are tuned. Furthermore, we use the Root Mean Square Error (RMSE) metric, to compare the novel kernel method with other kernels and algorithms (LinR and IBK). We select 10 data sets Camel, Tomcat, Poi, Xalan, JEdit, Velocity, Ant, Lucene, Synapse, and Ivy shown in Table 6.5 from the Promise data repository [49] that are open source and have enough entries in their defect files to be able to apply 5×2 cross validation.

Chapter 7

Experiments and Results

7.1. Experiment I: Determine Influential Relationships Among Metrics and Defectiveness Using Bayesian Networks

7.1.1. Experiment Design

In our experiments, we determine the influential or associational relationships among the software metrics and defectiveness and identify the most effective metrics by giving them scores considering their effect on defect proneness. We use public data sets Xerces, Ant, Tomcat, Xalan, Jedit, Velocity, Synapse, Poi, Lucene, Log4j, Arc, Camel shown in Table 7.1 from Promise data repository [49] and use Weka [106] for Bayesian network structure learning. We learn the network structure from the data sets and use SimpleEstimator while constructing Bayesian networks for defect proneness. Furthermore, we select K2 as the search algorithm and use predefined ordering of nodes of LOC, CBO, WMC, RFC, LCOM, LCOM3, DIT, and NOC. We use AUC while comparing the performance of the Bayesian networks in our experiments.

7.1.2. Results

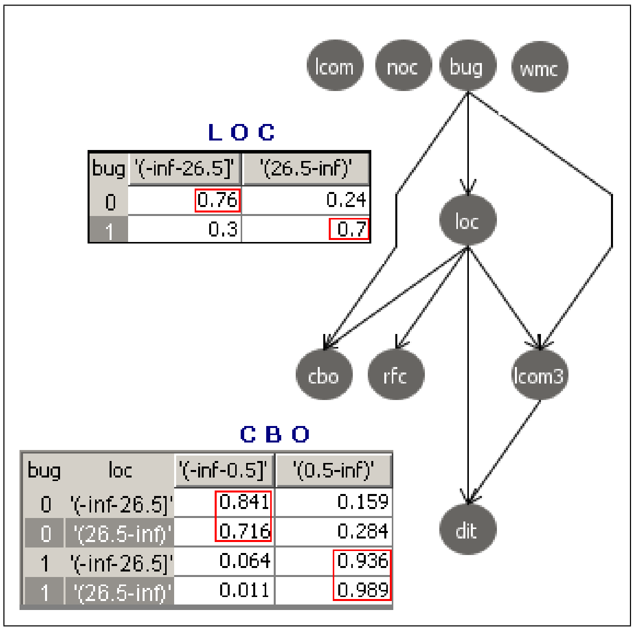
The AUC values found for each network are shown in Table 7.2. The highest AUC value is found for the Xerces data set whereas the smallest AUC value is observed for the Log4j data set. A high AUC value implies that the data set used is able to define the structure of the Bayesian network better. The AUC values found in our experiments are relatively high and we believe that this situation makes our results more important and reliable.

Table 7.1. Brief details of 12 data sets used in Experiment I.

Data Set	**Version**	**No. of Instances**	**% Defective Instances**
Xerces	1.4	588	74.32
Ant	1.7	745	22.28
Tomcat	6.0	858	8.97
Xalan	2.7	909	98.79
Jedit	4.3	492	2.24
Velocity	1.6	229	34.06
Synapse	1.2	256	33.59
Poi	3	442	63.57
Lucene	2.4	340	59.71
Log4j	1.2	205	92.2
Arc	1.0	234	11.54
Camel	1.6	965	19.48

Table 7.2. The AUC values of the Bayesian networks in Experiment I.

Data Sets	**AUC**
Xerces	0.922
Ant	0.792
Tomcat	0.758
Xalan	0.746
Jedit	0.816
Velocity	0.632
Synapse	0.665
Poi	0.838
Lucene	0.658
Log4j	0.576
Arc	0.745
Camel	0.577



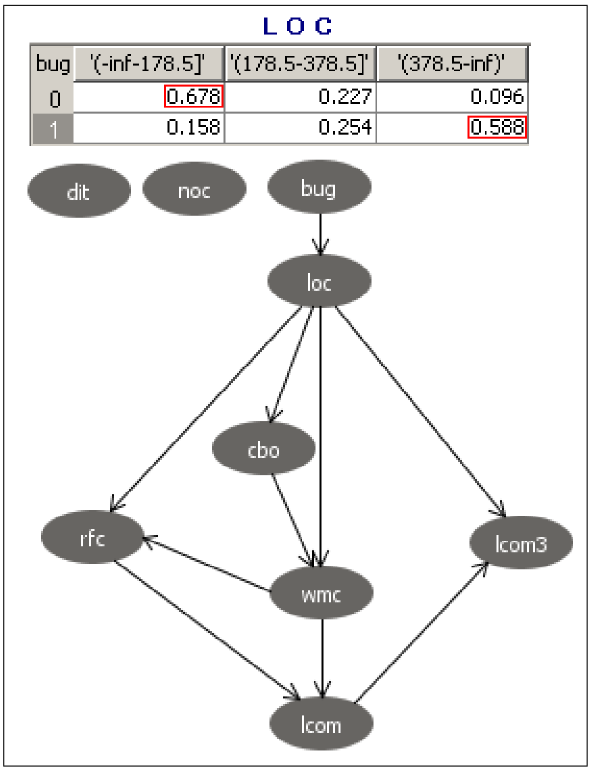
LOC

bug	'(-inf-26.5]'	'(26.5-inf)'
0	0.76	0.24
1	0.3	0.7

CBO

bug	loc	'(-inf-0.5]'	'(0.5-inf)'
0	'(-inf-26.5]'	0.841	0.159
0	'(26.5-inf)'	0.716	0.284
1	'(-inf-26.5]'	0.064	0.936
1	'(26.5-inf)'	0.011	0.989

Figure 7.1. Bayesian network showing the relationships among software metrics in Xerces version 1.4.

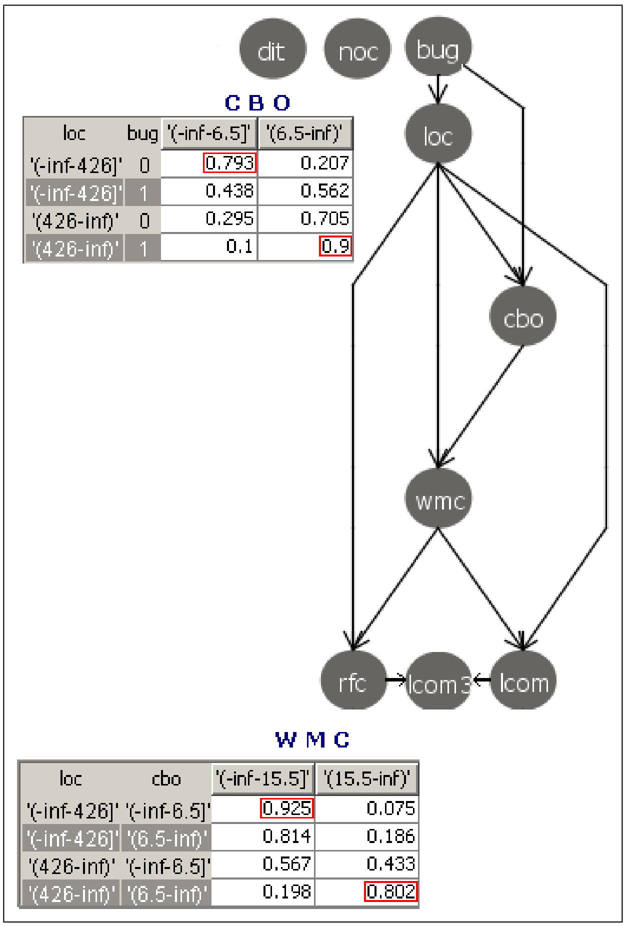


LOC

bug	'(-inf-178.5]'	'(178.5-378.5]'	'(378.5-inf)'
0	0.678	0.227	0.096
1	0.158	0.254	0.588

Figure 7.2. Bayesian network showing the relationships among software metrics in Ant version 1.7.

The Bayesian network for Xerces data set is shown in Figure 7.1. First of all, LOC, CBO, and LCOM3 are the most effective metrics since they are directly connected with defectiveness. RFC and DIT are indirectly effective on fault proneness.



C B O

loc	bug	'(-inf-6.5]'	'(6.5-inf)'
'(-inf-426]'	0	0.793	0.207
'(-inf-426]'	1	0.438	0.562
'(426-inf)'	0	0.295	0.705
'(426-inf)'	1	0.1	0.9

W M C

loc	cbo	'(-inf-15.5]'	'(15.5-inf)'
'(-inf-426]'	'(-inf-6.5]'	0.925	0.075
'(-inf-426]'	'(6.5-inf)'	0.814	0.186
'(426-inf)'	'(-inf-6.5]'	0.567	0.433
'(426-inf)'	'(6.5-inf)'	0.198	0.802

Figure 7.3. Bayesian network showing the relationships among software metrics in Tomcat version 6.0.

However, WMC, LCOM, and NOC are not effective at all. On the other hand, from the conditional probability table of LOC, we see that there is a positive correlation between LOC and defect proneness. When LOC value is high, the software is more defect prone; when LOC value is low, the software is less defect prone. Similarly, from the conditional probability table of CBO, we see that there is also a positive correlation between CBO and defect proneness. Furthermore, we observe that when both LOC and CBO are high, the defect proneness probability is the highest (0.989). This shows that the size and the complexity metrics together affect defectiveness more, compared to the effect of either size or complexity alone.

We obtain similar but slightly different results for Ant data set as it is shown in Figure 7.2. Similar to Xerces data set, LOC is the most effective metric again, but now CBO is indirectly effective compared to LOC. RFC, WMC, and LCOM3 are at the same level in terms of their effect on defectiveness. However, DIT and NOC are not effective on fault proneness. Moreover, similar to the Xerces data set, conditional probability table of LOC implies that classes with higher LOC are more defect prone. Furthermore, we also observe that a higher LOC implies higher CBO and a higher CBO leads to higher LOC, as one would expect by looking at the definitions of LOC and CBO. The Bayesian network for Tomcat data set shows that LOC and CBO are directly effective on defect proneness, whereas WMC is indirectly effective. But DIT and NOC metrics are not effective in determining the defect prone classes (See Figure 7.3). Furthermore, similar to our previous findings for Xerces and Ant, when both LOC and CBO are high, the defectiveness probability is higher compared to the case when either one of them is small. Additionally, when both LOC and CBO are high, the probability of having a high WMC is higher.

For the remaining 9 data sets, Bayesian networks are shown in Figure 7.4. For Xalan data set, metrics LOC and DIT are directly effective on defect proneness whereas other metrics are conditionally independent of defectiveness. Furthermore, WMC, RFC, LCOM, and LCOM3 form a disjoint Bayesian network where they have no connection with defect proneness. For JEdit data set, LOC, CBO, and WMC are directly effective on defectiveness whereas RFC, LCOM, and LCOM3 are effective indirectly. Moreover, DIT and NOC are found to be independent from defect proneness. Similar to the previous findings, for Velocity data set, DIT, NOC, and LCOM3 are independent of defect proneness where LOC, CBO and LCOM are directly effective on defectiveness. Confirming the results found for Velocity data set, DIT and NOC metrics are not effective on defect proneness for Synapse data set also. But, LOC and WMC are more effective compared to other metrics. Furthermore, there are influential relationships among the metrics other than DIT and NOC and these relationships are expressive considering the meaning of each metric. For instance, LOC is directly influential on RFC, CBO, and WMC as one may expect. For Poi data set, LOC, CBO, WMC, and LCOM are directly effective on defect proneness whereas NOC is not effective at all. The Bayesian network learned from Lucene data set, confirms some of the findings of the previous experiments as DIT, NOC, and LCOM3 are disconnected i.e. they are conditionally independent from defect proneness. In parallel with other experiments,

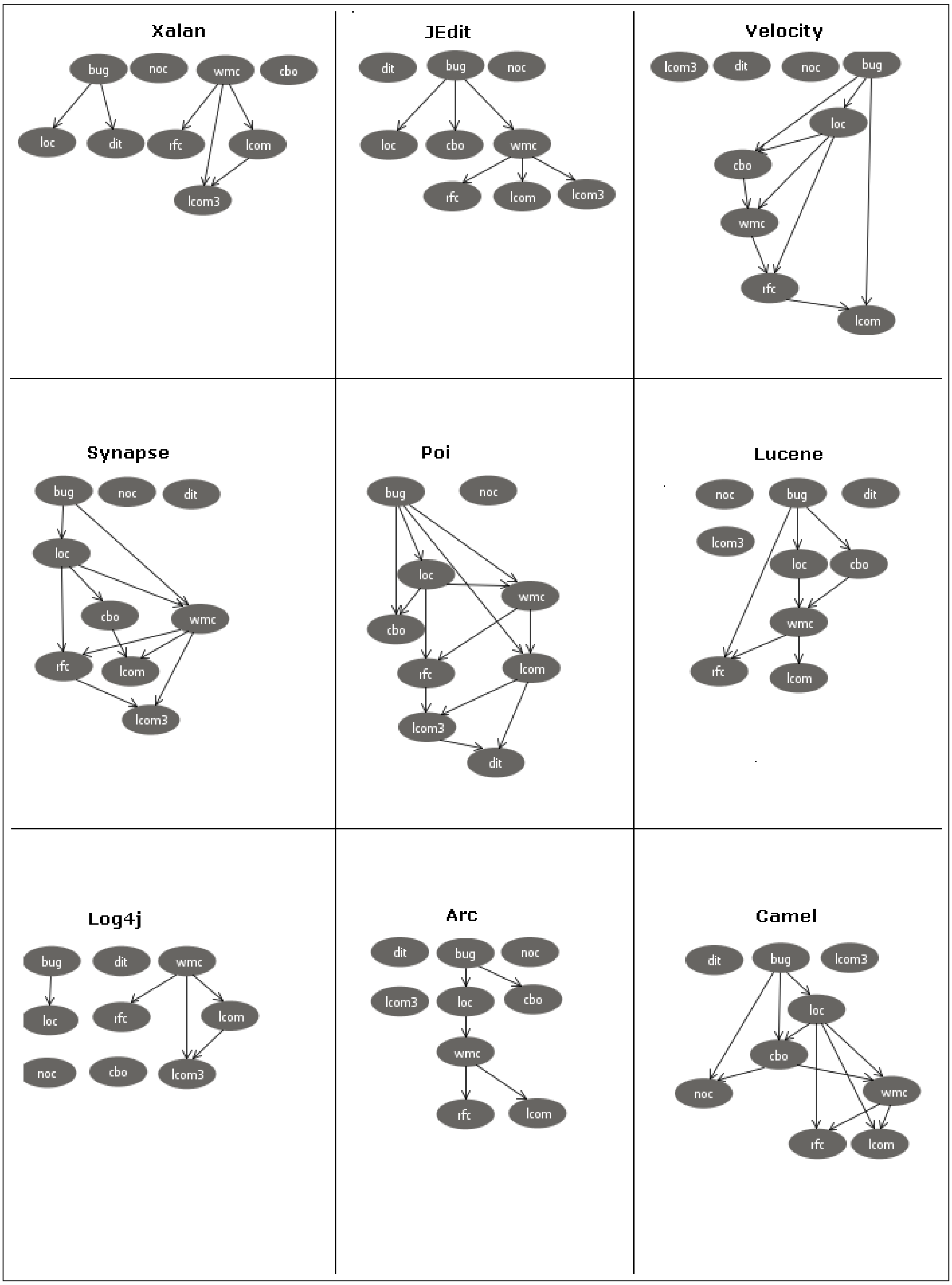


Figure 7.4. Bayesian networks for Xalan, Jedit, Velocity, Synapse, Poi, Lucene, Log4j, Arc, and Camel data sets, showing the relationships among software metrics and defect proneness (bug).

Table 7.3. The scores of metrics obtained in Experiment I.

Experiments	**LOC**	**CBO**	**WMC**	**RFC**	**LCOM**	**LCOM3**	**DIT**	**NOC**
Xerces	1	1	0	1	0	1	1	0
Ant	1	1	1	1	1	1	0	0
Tomcat	1	1	1	1	1	1	0	0
Xalan	1	0	0	0	0	0	1	0
Jedit	1	1	1	1	1	1	0	0
Velocity	1	1	1	1	1	0	0	0
Synapse	1	1	1	1	1	1	0	0
Poi	1	1	1	1	1	1	1	0
Lucene	1	1	1	1	1	0	0	0
Log4j	1	0	0	0	0	0	0	0
Arc	1	1	1	1	1	0	0	0
Camel	1	1	1	1	1	0	0	1
Total Score	**12**	**10**	**9**	**10**	**9**	**6**	**3**	**1**

Bayesian networks learned from data sets Log4j, Arc, and Camel shown in Figure 7.4 clarify that LOC is directly effective on defect proneness whereas DIT and NOC are not effective most of the time.

To measure the effect of metrics quantitatively, we give scores to the metrics in each experiment. If a metric affects defectiveness, we assign it a score of 1, if it has no relationship with defectiveness it is assigned a zero score. Table 7.3 shows the scores of metrics assigned in each experiment. As it is seen in the total score row of the table, the score of LOC is the highest and it is the most effective metric. whereas DIT and NOC are the least effective metrics. Furthermore, DIT and NOC are untrustworthy since their effectiveness is not consistent in all experiments. For instance, DIT is effective in Xerces and Poi experiments whereas it has no importance in other 10 experiments and there is a similar situation for NOC also.

7.2. Experiment II: Determine The Role Of Coding Quality And Number Of Developers On Defectiveness Using Bayesian Networks

7.2.1. Experiment Design

In a separate experiment setup we define two more metrics i.e. LOCQ and NOD. Lack of coding quality (LOCQ) measures the quality of the source code and the number of developers (NOD) metric shows the number of distinct developers for each class. We add these two metrics to the previous set of metrics and learn their relationship with defectiveness.

We determine the influential or associational relationships among the software metrics and defectiveness and identify the most effective metrics by giving them scores considering their effect on defect proneness. While choosing the data sets from Promise data repository, first we look at the data sets that are large enough to perform cross validation. So, we eliminate some of the data sets in the repository that are small in terms of size. Second, to extract additional metrics LOCQ and NOD, we need the source repositories of the data sets. That is why, we prefer the data sets in Promise data repository whose source code is available in the open source project repositories. For instance the Log4j data set has defect data for versions 1.0, 1.1, and 1.2 in the Promise data repository, but the sources corresponding to these exact versions are not present in the Apache repository. We eliminate some data sets whose sources are not available or are skewed which could affect the learning performance of the Bayes net classifier. Based on these criteria we select public data sets Ant, Tomcat, Jedit, Velocity, Synapse, Poi, Lucene, Xalan, and Ivy from Promise data repository [49] (See Table 7.4 for the details of the datasets).

We use Weka [106] for Bayesian network structure learning where we learn the network structure from the data sets and use SimpleEstimator while constructing Bayesian networks for defect proneness. Furthermore, we select K2 as the search algorithm and use predefined ordering of nodes of LOC, CBO, LOCQ, WMC, RFC, LCOM, LCOM3, DIT, and NOC.

Table 7.4. Brief details of data sets used in Experiment II where Lack of Coding Quality metric (LOCQ) is considered together with LOC, CBO, WMC, RFC, LCOM, LCOM3, DIT, and NOC metrics.

Data Set	**Version**	**No. of Instances**	**% Defective Instances**
Ant	1.7	745	22.28
Tomcat	6.0	858	8.97
Jedit	4.3	492	2.24
Velocity	1.6	229	34.06
Synapse	1.2	256	33.59
Poi	3	442	63.57
Lucene	2.4	340	59.71
Xalan	2.5	741	48.19
Ivy	2.0	352	11.36

Table 7.5 shows the AUC values for each Bayesian network. The results show that Ivy dataset has the highest AUC value whereas Xalan data set has the smallest AUC value. A high AUC value implies that the data set used is able to define the structure of the Bayesian network better. The AUC values found in our experiments are relatively high and we believe that this makes our results more important and reliable.

We obtain the Bayesian network shown in Figure 7.5 for Ant data set. The metrics LOC and LOCQ are the most effective whereas CBO, WMC, RFC, LCOM, and LCOM3 are indirectly and less effective on defectiveness. However, DIT and NOC are not effective on the bug attribute. When we look at the conditional probability table of LOC, we observe that there is a positive correlation between LOC and defect proneness. For instance the non-defectiveness probability is 0.678 for small LOC, whereas it is 0.096 for high LOC which means that as the LOC increase the probability of defectiveness increases too. We observe a similar relationship between LOCQ and defectiveness also where the defect proneness probability is high for high LOCQ values. Furthermore, we observe that when the LOC is high, the probability of having a high CBO is also high

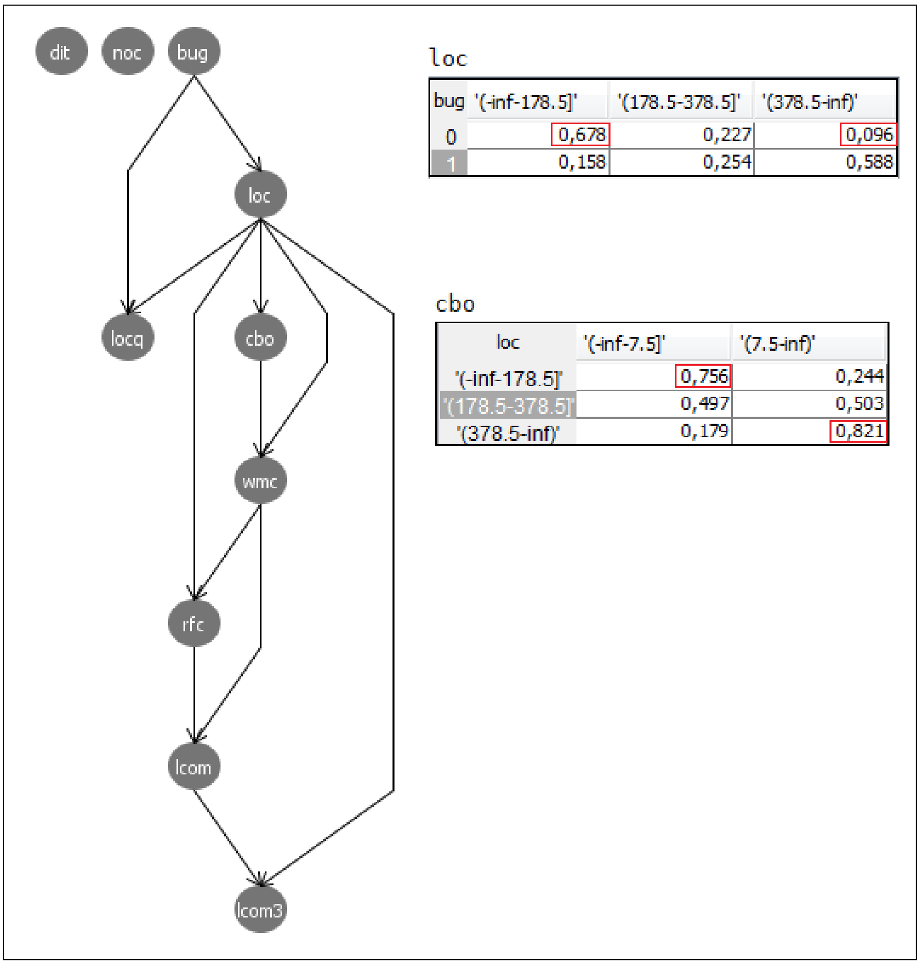
Table 7.5. The AUC values of the Bayesian networks in Experiment II.

Data Sets	**AUC**
Ant	0.820
Tomcat	0.766
Poi	0.845
Jedit	0.658
Velocity	0.678
Synapse	0.660
Lucene	0.633
Xalan	0.624
Ivy	0.846

which means that there is a positive correlation between LOC and CBO too.

The Bayesian network for Tomcat data set shows that LOC and CBO are directly effective on defect proneness, whereas WMC, LOCQ, RFC, LCOM, and LCOM3 are indirectly effective. But DIT and NOC metrics are not effective in determining the defect prone classes (See Figure 7.6). Furthermore, when both LOC and CBO are high, defectiveness probability is higher (0.9) compared to the case when either one of them is small. Similarly, when both are low, the non defectiveness probability is higher (0.793) compared to cases when either one of them is high.

The Bayesian network obtained for Poi data set is shown in Figure 7.7. First of all, CBO, LOC, LOCQ, WMC, and LCOM are the most effective metrics since they are directly connected with defectiveness. RFC, DIT, and LCOM3 are indirectly effective on fault proneness. On the other hand, NOC is not effective at all. Similarly, from the conditional probability table of CBO, we see that there is also a positive correlation between CBO and defect proneness. Furthermore, we observe that when both LOC and CBO are high, the defect proneness probability is the highest (0.904). This shows that the size and the complexity metrics together affects defectiveness more compared to the effect of either size or complexity alone.



loc

bug	'(-inf-178.5]'	'(178.5-378.5]'	'(378.5-inf)'
0	0,678	0,227	0,096
1	0,158	0,254	0,588

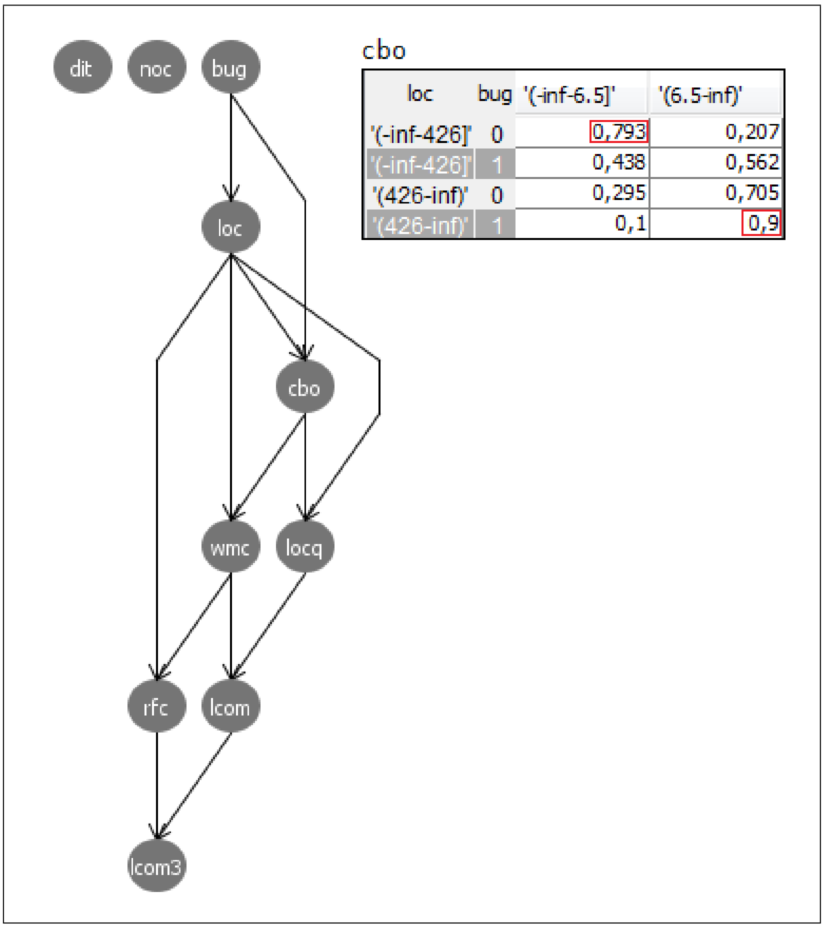
cbo

loc	'(-inf-7.5]'	'(7.5-inf)'
'(-inf-178.5]'	0,756	0,244
'(178.5-378.5]'	0,497	0,503
'(378.5-inf)'	0,179	0,821

Figure 7.5. Bayesian network showing the relationships among software metrics in Ant version 1.7.

For the remaining 6 data sets, Bayesian networks are shown in Figure 7.8. For Synapse data set, DIT, NOC, and LCOM3 metrics are not effective on defect proneness. On the other hand, LOC is the most effective metric and LOCQ, CBO, WMC, RFC, and LCOM are indirectly effective on defect proneness. The probability of having a defect free software is 0.839 for small LOC, whereas it is 0.161 for higher LOC values. We also observe that there is a positive correlation between LOC and LOCQ metrics. For instance, when the LOC is small, LOCQ is also small with a probability of 0.942. Similarly, the probability of having both LOC and LOCQ high is 0.877.

We observe a similar result for Lucene data set also where LCOM3, DIT, and LOC are not effective on defect proneness whereas CBO and LOC are the most effective metrics. For higher CBO values, the defect proneness probability is 0.743 whereas it is



cbo

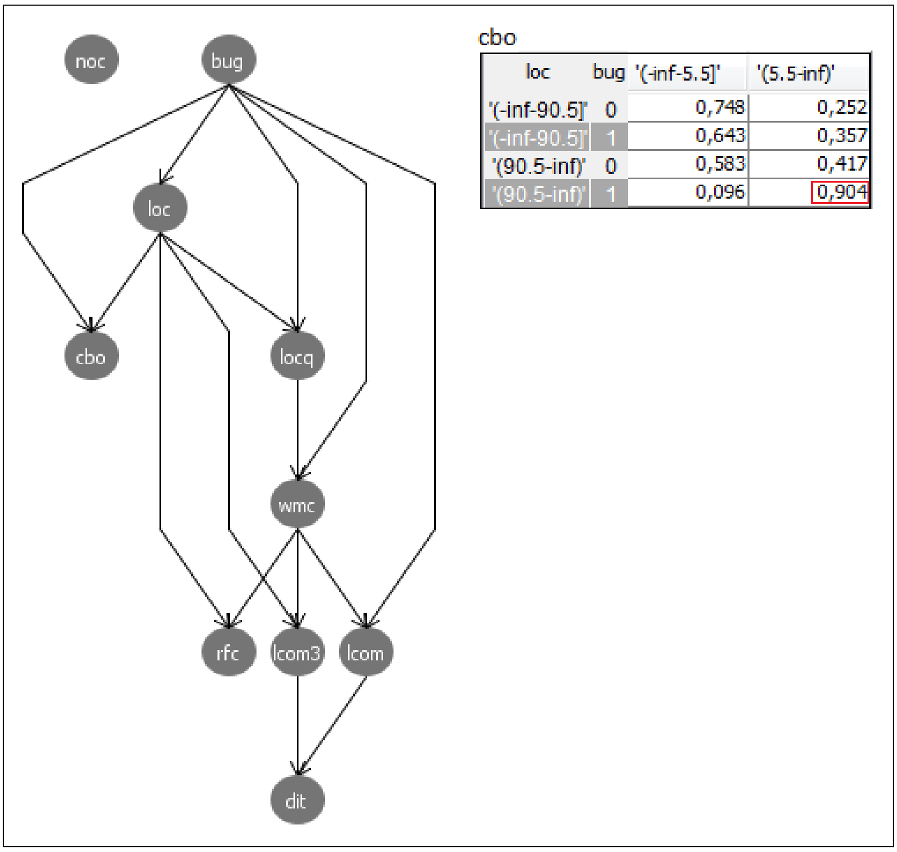
loc	bug	'(-inf-6.5]'	'(6.5-inf)'
'(-inf-426]'	0	0,793	0,207
'(-inf-426]'	1	0,438	0,562
'(426-inf)'	0	0,295	0,705
'(426-inf)'	1	0,1	0,9

Figure 7.6. Bayesian network showing the relationships among software metrics in Tomcat version 6.0.

only 0.257 for smaller CBO. That means as the coupling between objects increase, the probability of defectiveness increases also. On the other hand, LOCQ, WMC, RFC, and LCOM are indirectly effective on defect proneness.

Similar to the previous findings, for Velocity data set, DIT, NOC, WMC, and LCOM found to be independent of defect proneness where LOC is directly effective on defectiveness. On the other hand, CBO, LOCQ, RFC, and LCOM3 are indirectly and less effective compared to LOC. When we look at the conditional probability table for LOC, we observe that the defectiveness probability is 0.64 for higher LOC whereas it is 0.36 for smaller LOC values.

For JEdit data set, metrics DIT and NOC are independent from defect proneness whereas LOC, CBO, WMC, and LCOM3 are effective on the bug attribute. On the



cbo

loc	bug	'(-inf-5.5]'	'(5.5-inf)'
'(-inf-90.5]'	0	0,748	0,252
'(-inf-90.5]'	1	0,643	0,357
'(90.5-inf)'	0	0,583	0,417
'(90.5-inf)'	1	0,096	0,904

Figure 7.7. Bayesian network showing the relationships among software metrics in Poi version 3.0.

other hand, LOCQ, RFC, and LCOM are indirectly and less effective compared to LOC, CBO, WMC, and LCOM3 metrics.

We observe that LOC and LCOM3 metrics are directly effective on defect proneness whereas DIT, NOC, CBO, and LCOM are not effective for Xalan data set. On the other hand, LOCQ, WMC, and RFC are indirectly effective on defectiveness. Furthermore, for a lower LOC, the probability of having a lower LOCQ is 0.967 whereas the probability of a higher LOCQ is 0.033. Similarly, for a higher LOC, the probability of having a higher LOCQ is 0.568, whereas the probability of a lower LOCQ is 0.432.

For Ivy data set, LOC, CBO, and LOCQ are the most important metrics, whereas DIT, NOC, and LCOM3 are not effective on fault proneness. Furthermore, WMC, RFC, and LCOM are less effective on defectiveness compared to LOC, CBO, and LOCQ. Similar to the previous findings, when both LOC and CBO are high, the

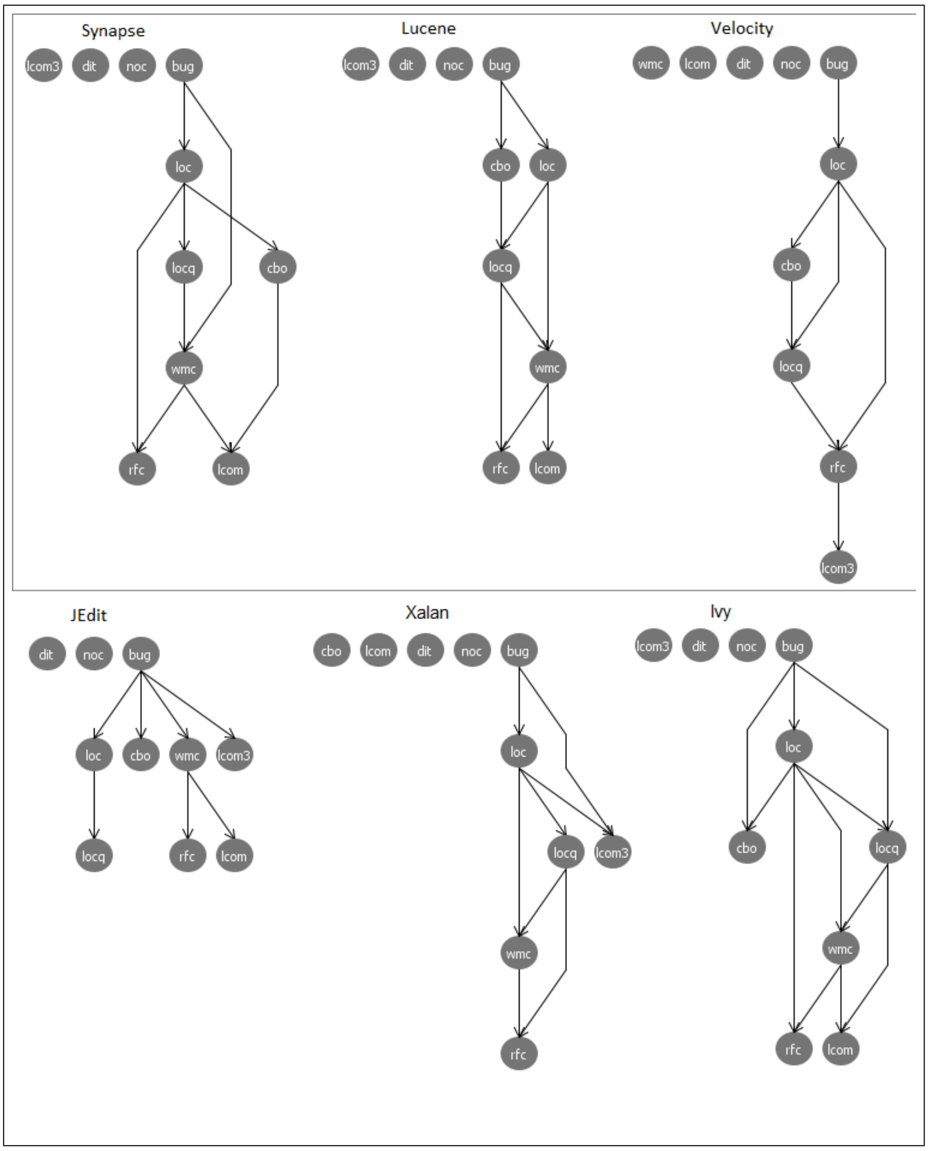


Figure 7.8. Bayesian networks for Synapse, Lucene, Velocity, Jedit, Xalan, and Ivy data sets, showing the relationships among software metrics and defect proneness (bug).

defectiveness probability is the highest (0.983).

To measure the effect of metrics quantitatively, we give scores to the metrics in each experiment. If a metric is affecting defectiveness (directly or indirectly) we assign it a score of 1, if it has no relationship with defectiveness it is assigned a zero score. Table 7.6 shows the scores of metrics assigned in each experiment. According to the average scores, LOC, CBO, LOCQ, WMC, and RFC are the most effective metrics, whereas DIT and NOC are the least effective ones. Furthermore, DIT and

Table 7.6. The scores of metrics obtained in Experiment II.

Data Sets	LOC	CBO	LOCQ	WMC	RFC	LCOM	LCOM3	DIT	NOC
Ant	1	1	1	1	1	1	1	0	0
Tomcat	1	1	1	1	1	1	1	0	0
Poi	1	1	1	1	1	1	1	1	0
Jedit	1	1	1	1	1	1	1	0	0
Velocity	1	1	1	0	1	0	1	0	0
Synapse	1	1	1	1	1	1	0	0	0
Lucene	1	1	1	1	1	1	0	0	0
Xalan	1	0	1	1	1	0	1	0	0
Ivy	1	1	1	1	1	1	0	0	0
Average	**1**	**0.89**	**1**	**0.89**	**1**	**0.78**	**0.67**	**0.11**	**0**

NOC are untrustworthy since their effectiveness is not consistent in all experiments. For instance, DIT is effective in Poi whereas it has no importance in other data sets. Similarly, NOC is independent from defect proneness in all experiments. Moreover, we observe that LCOM and LCOM3 are more effective compared to DIT and NOC and less effective compared to others.

7.2.2. Conclusion Instability Test

Some times the results found for a data set, might not be valid for its subsets due to some uncommon local attributes [68]. So, we check if the results shown in Table 7.6 are valid for the subsets of the data sets too. Therefore, we make 20 experiments with different 2/3rd subsets of each data set and calculate the average score for each metric based on these 20 experiments. While generating the subsets, we stratify the data and use a different seed to ensure each subset is different from the previously generated ones. For each data set, the average scores we find at the end of 20 experiments are listed in Table 7.7. When we look at the average scores of the metrics on all data sets, we observe that although the results are slightly different from the results presented in Table 7.6 in terms of ordering, there are very strong similarities. For instance, still LOC, CBO, LOCQ, WMC, and RFC are the most important metrics. Furthermore

Table 7.7. The average scores of the metrics obtained for 20 different subsets of Ant, Tomcat, Poi, Jedit, Velocity, Synapse, Lucene, Xalan, and Ivy.

Data Sets	LOC	CBO	LOCQ	WMC	RFC	LCOM	LCOM3	DIT	NOC
Ant	1.00	1.00	1.00	1.00	1.00	0.90	0.70	0.10	0.00
Tomcat	1.00	1.00	0.90	0.90	1.00	1.00	0.50	0.10	0.10
Poi	1.00	0.70	1.00	0.90	1.00	0.80	0.80	0.40	0.00
Jedit	0.60	0.50	0.30	0.60	0.50	0.40	0.00	0.00	0.00
Velocity	0.90	0.50	0.50	0.40	0.90	0.10	0.30	0.00	0.00
Synapse	0.70	0,80	0.60	0.60	0.70	0.60	0.00	0.00	0.00
Lucene	0.30	0.70	0.60	0.80	0.90	0.60	0.20	0.00	0.00
Xalan	1.00	0.10	0.95	0.55	0.95	0.65	0.50	0.05	0.10
Ivy	1.00	0.90	1.00	1.00	0.95	0.70	0.50	0.00	0.00
Average	**0.83**	**0.69**	**0.76**	**0.75**	**0.88**	**0.64**	**0.39**	**0.07**	**0.02**

DIT and NOC are the least effective and untrustworthy metrics. Similar to the results shown in Table 7.6, LCOM and LCOM3 are more effective compared to DIT and NOC and less effective compared to other metrics.

7.2.3. Effectiveness of Metric Pairs

We look at the 180 Bayesian networks (generated for 20 subsets of 9 data sets), in terms of which metric pairs are the most effective on defectiveness. For a specific Bayesian network, if both of the metrics in the pair have a relationship with defectiveness we assign a score of 1 to the metric pair. If either or neither of the metrics is related with defect proneness we assign a zero score for the metric pair. The sum of the scores of the metric pairs calculated for all subsets of the data sets are shown in Table 7.8 (Only the most effective ten metric pairs are included in the list). We observe that metric pairs LOC-RFC, RFC-LOCQ, RFC-WMC are the most effective pairs and their scores are 145, 136, and 133 respectively. We see that the metric pairs that have the highest scores are composed of metrics that got the highest score in the previous evaluation where each metric is considered alone (See Table 7.6). For instance the metric pair LOC-RFC got the highest score and we see that metrics LOC and RFC

Table 7.8. The scores of metric pairs obtained for the 20 subsets of Ant, Tomcat, Poi, Jedit, Velocity, Synapse, Lucene, Xalan, and Ivy data sets.

Metric Pairs	**Total Score**
LOC-RFC	145
RFC-LOCQ	136
RFC-WMC	133
LOC-LOCQ	131
LOC-WMC	121
RFC-CBO	120
LOCQ-WMC	119
WMC-CBO	109
LOC-CBO	108
LOCQ-CBO	106

alone are among the metrics that got the highest scores in Table 7.6.

7.2.4. Feature Selection Tests

Using Bayesian network model, it is possible to make probabilistic causal or diagnostic inferences about the effectiveness of a metric on another metric or on the defectiveness. At the end of learning a Bayesian network, we not only determine the set of most important metrics but also find the relationship among them and the probability of their effect on defect proneness. Therefore, with Bayesian networks we are able to model the uncertainties better, compared to other machine learning methods. Although they do not give the extent of influential relationships among metrics and defectiveness, using Feature selection methods, we can determine the most important metrics and make a cross check with the results of Bayesian network model.

At the end of our experiments, we observe that considering all data sets we use in our experiments, LOC, CBO, LOCQ, WMC, and RFC are the most effective metrics and DIT and NOC are the least effective ones (See Tables 7.6 and 7.7). We run two

Table 7.9. The results of feature selection tests with CfsSubsetEval and BestFirst search method where a metric is assigned a score of 1 if it is selected and is assigned zero score otherwise.

Data Sets	LOC	CBO	LOCQ	WMC	RFC	LCOM	LCOM3	DIT	NOC
Ant	1	1	1	0	1	1	0	0	0
Tomcat	1	1	1	0	1	0	0	0	0
Poi	0	1	1	0	1	1	1	0	0
Jedit	0	0	0	0	1	0	0	0	0
Velocity	1	1	0	0	1	0	1	0	0
Synapse	1	1	0	1	1	0	0	0	0
Lucene	0	1	1	0	1	1	0	0	1
Xalan	1	0	1	0	1	0	1	0	0
Ivy	1	1	1	0	1	0	1	0	0
Average	**0.67**	**0.78**	**0.67**	**0.11**	**1**	**0.33**	**0.44**	**0**	**0.11**

feature selection algorithms CFS (CfsSubsetEval attribute evaluator with BestFirst search method) and Relief (ReliefFAttributeEval with Ranker search method) to see which metrics are selected as the most important attributes and whether the results of the feature selection experiments are different from the results shown in Table 7.3. For CFS tests, for each data set, each metric is assigned a score of 1 if it is among the selected metrics and it is assigned 0 otherwise. Table 7.9 shows the results of feature selection tests with CFS. When we look at the average scores of the metrics, we observe that LOC, CBO, LOCQ, and RFC are among the most important features and DIT and NOC are the least important ones. So, although the ordering found at the end of feature selection is slightly different from the ordering shown in Table 7.6, except WMC, we can say that there is a coherence between the feature selection test and our experiments in terms of the most and least effective attributes.

We repeat the feature selection test with Relief where Ranker is used as the search method. Relief gives the rankings of the attributes for each data set. When we take the average of the rankings for each metric on all data sets, we observe that except for DIT metric, the results of the feature selection tests with Relief are coherent with our results shown in Table 7.6. For instance, still LCOM, LCOM3, and NOC are less effective compared to other metrics. Similar to the results shown in Table 7.6, metrics RFC and CBO are the among most effective metrics. Although the average rankings found for LOC, LOCQ, and WMC are not so good, they are still more important compared to LCOM, LCOM3, and NOC (See Table 7.10).

Table 7.10. The results of feature selection tests with ReliefFAttributeEval and Ranker search method where the rankings of the metrics are shown for each data set (If the average ranking of a metric is smaller, then it means the metric is more important).

Data Sets	LOC	CBO	LOCQ	WMC	RFC	LCOM	LCOM3	DIT	NOC
Ant	5	6	3	4	2	9	8	1	7
Tomcat	5	2	4	6	3	9	8	1	7
Poi	6	5	3	1	2	7	9	4	8
Jedit	5	3	6	4	2	7	8	1	9
Velocity	5	2	6	4	1	7	9	3	8
Synapse	2	3	6	5	1	7	8	4	9
Lucene	7	2	5	4	3	8	1	9	6
Xalan	1	3	5	2	4	7	6	9	8
Ivy	5	2	4	6	3	9	8	1	7
Average	**4.56**	**3.11**	**4.67**	**4.00**	**2.33**	**7.78**	**7.22**	**3.67**	**7.67**

7.2.5. Effectiveness of the Number of Developers (NOD)

Among the data sets listed in Table 7.4, we use Poi, Tomcat, and Xalan to extract the number of developers since developer names could be retrieved from their source code repositories. We count the number of distinct developers (NOD) for each class of each data set. We use the NOD metric together with the metrics listed in Table 6.1 and LOCQ, to learn a Bayesian network for each data set and to extract its relationship with other metrics and the extent of defect proneness. Furthermore, we select K2 as the search algorithm and use predefined ordering of nodes of LOC, NOD, CBO, LOCQ, WMC, RFC, LCOM, LCOM3, DIT, and NOC. To see the relationship of NOD and the level of defectiveness better, we define three states for defect proneness. All class

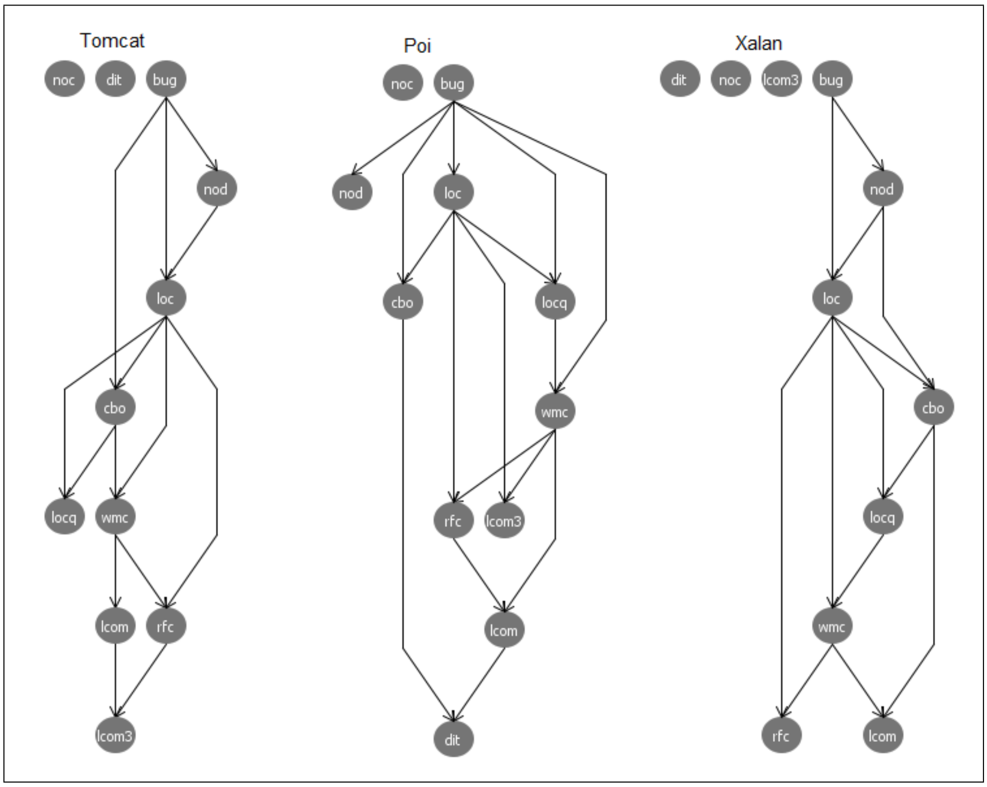


Figure 7.9. Bayesian networks showing the relationship of the number of developers with the level of defectiveness in Poi, Tomcat, and Xalan data sets.

instances where bug is zero are accepted as defect free classes. The classes that have 1 or 2 bugs, are marked as less defective, and the classes that have more than 2 bugs are accepted as more defective. As a result, we simply define three defect proneness states which are, defect free, less defective, and more defective. There is nothing special for the threshold values we use to define the level of defectiveness, someone else might use different thresholds or define more levels for defect proneness. The Bayesian networks we obtain at the end of our experiments are shown in Figure 7.9.

For all Bayesian networks learned, NOD is directly effective on defect proneness and we observe a positive correlation between NOD and the level of defectiveness. For instance for Poi data set, we see that as the number of developers increases, the defectiveness increases too. If the number of developers is less than 3, the non defectiveness probability is 0.997, but it is 0.003 if there are more than 3 developers per class. For Tomcat data set, we observe that if the number of developers is more than 1, the probability of a defect free class is 0.167. But the probabilities of having a less

Table 7.11. The average scores of metrics obtained at the end of runs on 10 subsets of Poi, Tomcat, and Xalan data sets (If a metric affects defectiveness we assign it a score of 1, if it has no relationship with defectiveness it is assigned a zero score).

	Data sets			
Metrics	**Poi**	**Tomcat**	**Xalan**	**Average**
NOD	0.7	1	0.7	**0.80**
LOC	1	1	1	**1.00**
CBO	1	1	0.9	**0.97**
LOCQ	1	1	0.9	**0.97**
WMC	1	0.9	0.9	**0.93**
RFC	1	1	1	**1.00**
LCOM	1	0.9	1	**0.97**
LCOM3	1	0.6	0.6	**0.73**
DIT	0.7	0	0	**0.23**
NOC	0	0	0	**0.00**

or more defective class are 0.514 and 0.812 respectively. For Xalan, we observe a similar relationship between NOD and the level of defectiveness, where if the number of developers is less than 2 then the non defectiveness probability is 0.705. If NOD is 2 or 3 then the non defectiveness probability is 0.276 and if NOD is greater than 3 then it is only 0.019. Apparently, as the number of developers increases, the non defectiveness probability decreases or the level of defectiveness increases.

To be sure that our results do not suffer from conclusion instability, and our observations are valid for the subsets of the data sets too, we repeat our experiments with the 10 subsets of each data set. Each data set is stratified and 67 percent of its data is included in the subsets. Furthermore, for each stratification a different seed is used. For the Bayesian network obtained in each experiment, a metric is assigned 1, if it has a relationship with defectiveness and assigned zero otherwise. The average scores of the metrics for Poi, Tomcat, and Xalan data sets are shown in Table 7.11. In 7 experiments for Poi, in all experiments for Tomcat, and in 7 experiments for Xalan,

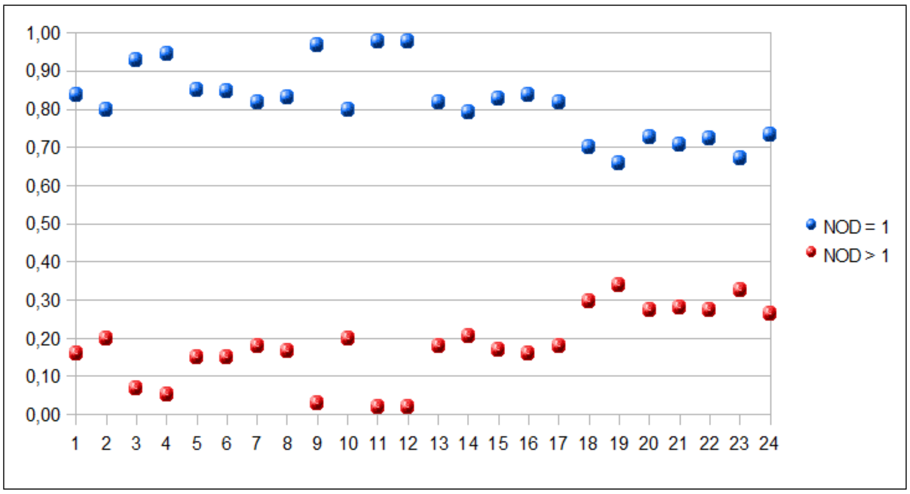


Figure 7.10. The non defectiveness probabilities of two cases i.e. when the number of developers is 1 ($NOD = 1$) and when it is greater than 1 ($NOD > 1$).

NOD is directly effective on defectiveness and there is a positive correlation with the developer count and defectiveness. Furthermore, supporting the results observed in the previous experiments (Tables 7.6 and 7.7), LOC and RFC are the most effective metrics whereas DIT and NOC are the least effective ones.

We compare the non defectiveness probabilities of two cases i.e. when the number of developers is 1 ($NOD = 1$) and when it is greater than 1 ($NOD > 1$). Figure 7.10 shows the non defectiveness probabilities of these two cases, for 30 experiments carried out on 10 stratified subsets of Poi, Tomcat, and Xalan (3 experiments for Poi and 3 experiments for Xalan data sets where NOD is not related with defectiveness are not included). To check for the statistical significance of the results, we apply a t-test (in 95 % confidence interval) to the non defectiveness probabilities of the two cases and show that the non defectiveness probability when NOD = 1 is better with a p value of zero.

We conclude that as the number of developers increases for a specific class, the class tends to be more defective and show that the common idiom “too many cookers spoil the broth” is valid for Software Engineering. At the end of our experiments, we observe that too many developers make a class more defect prone. This is due to the fact that when too many developers work on the same piece of code, the number of potential dependencies among their work items increases. We recommend project managers to explore the cost benefit curve of NOD versus defectiveness level where the value of adding more developers should be controlled with respect to the introduced number of defects. We must emphasize that this conclusion is based on the experiments on the data sets used and other researchers shall make more experiments on more data sets to justify our findings.

7.3. Experiment III: Defect Proneness Prediction Using Kernel Methods

7.3.1. Experiment Design

The major steps we follow in each experiment are:

1. We choose a data set from Promise data repository. The data set must have enough entries to be able to apply cross validation and its source code must be public, to extract a similarity based kernel matrix.
2. We choose one or more versions of this data set and download corresponding sources from open source data repositories.
3. We edit the defect data and change all bug info that is greater than 1 with 1. Since we perform classification rather than regression, the bug feature must be zero for non defective files and 1 for defective ones.
4. We create a class feature file from the changed defect data file that includes only the file names and the bug feature, in order to give the class feature to SVM together with the precomputed kernel matrix.
5. We input the source code to MOSS and generate an output that shows how much similarity exists among the source files of the data set we chose.
6. We convert the similarity output to a similarity kernel matrix using the software developed during this research. The software processes each row from the MOSS output and generates an n by n matrix assuming there are n files in the software system chosen. Each cell of this kernel matrix shows how much similarity exists, between the files in the row and column of the matrix in terms of percentage. Then the kernel matrix is normalized and saved in a file. An example kernel

Table 7.12. Average and standard deviations of the Area Under ROC Curve for P-SVM, L-SVM, RBF-SVM, LR, and J48 in Experiment III.

Datasets	**P-SVM**	**L-SVM**	**RBF-SVM**	**LR**	**J48**
Camel	0.62 ± 0.16	0.50 ± 0.00	0.50 ± 0.0	0.74 ± 0.04	0.50 ± 0.01
Tomcat	0.61 ± 0.04	0.53 ± 0.05	0.50 ± 0.0	0.53 ± 0.06	0.51 ± 0.02
Poi	0.73 ± 0.03	0.78 ± 0.03	0.78 ± 0.041	0.85 ± 0.04	0.67 ± 0.12
Xalan	0.65 ± 0.01	0.66 ± 0.02	0.69 ± 0.02	0.72 ± 0.02	0.65 ± 0.02
JEdit	0.67 ± 0.03	0.67 ± 0.05	0.50 ± 0.005	0.69 ± 0.07	0.59 ± 0.07
Velocity	0.63 ± 0.04	0.71 ± 0.07	0.55 ± 0.05	0.76 ± 0.06	0.67 ± 0.08
Ant	0.63 ± 0.02	0.62 ± 0.04	0.50 ± 0.005	0.60 ± 0.11	0.56 ± 0.04
Lucene	0.65 ± 0.03	0.68 ± 0.05	0.50 ± 0.005	0.65 ± 0.05	0.60 ± 0.06
Synapse	0.65 ± 0.03	0.64 ± 0.03	0.50 ± 0.00	0.66 ± 0.08	0.57 ± 0.04
Ivy	0.60 ± 0.04	0.50 ± 0.01	0.50 ± 0.00	0.55 ± 0.07	0.59 ± 0.08

matrix generated from a sample output in Table 6.6 is shown in Figure 6.2.

7. We give the class feature file and the computed kernel matrix as input to SVM and learn the relationship between defectiveness and similarity. The classification is run on Weka experimenter with 5×2 cross validation.

7.3.2. Results

We use Weka [106] to compare the performance of the precomputed kernel matrix with linear and RBF kernels. For each data set, 5×2 fold cross-validation is used and C parameter of SVM is tuned.

Usually the defect prediction data sets are skewed, that is the percentage of defective files is much lower than the percentage of non-defective ones. In this case, the accuracy is not a good metric for making comparisons. Instead, we choose the area under the ROC curve (AUC) to compare the classification algorithms.

Table 7.12 shows average and standard deviation of AUC for SVM with precal-

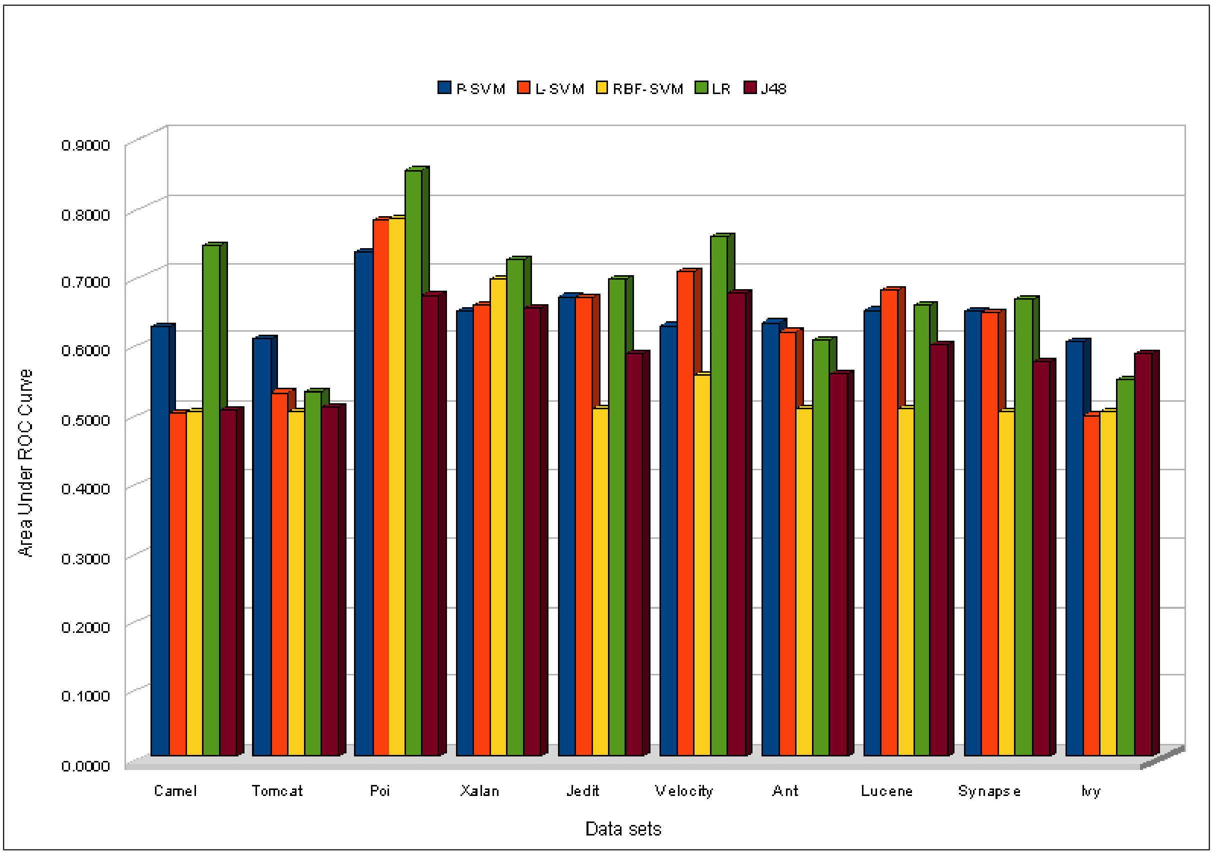


Figure 7.11. AUC of P-SVM, L-SVM, RBF-SVM, LR and J48 for Camel, Tomcat, Poi, Xalan, JEdit, Velocity, Ant, Lucene, Synapse, and Ivy data sets in Experiment III.

culated kernel matrix (P-SVM), linear kernel (L-SVM), and RBF kernel (RBF-SVM). Furthermore, it also lists the average AUC values for linear logistic regression (LR) and J48 algorithms to have an idea about how good our novel method is compared to some existing methods. First, the results show that the performance of the P-SVM is better than the performance of the L-SVM and RBF-SVM. Because in 6 of 10 experiments, its AUC is higher than L-SVM and in 8 of 10 experiments its AUC is higher than RBF-SVM numerically. Second, P-SVM achieves better results than J48 and is more consistent when compared to LR. Because, although the average AUC value achieved by P-SVM is lower than LR in 6 experiments, the worst AUC for P-SVM is 0.60 in these experiments. On the other hand, in 4 experiments P-SVM is better than LR, but the lowest AUC for LR is 0.53 in these experiments. So, we believe that this makes P-SVM more trust worthy since its results are good enough irregardless of the data set chosen (See Figure 7.11).

Table 7.13. Comparison of P-SVM with L-SVM, RBF-SVM, LR and J48 algorithms in Experiment III. The first, second, and third values in each cell shows the number of significant cases where P-SVM performs better or worse than the algorithm in the column or the number of cases where there is no difference respectively.

	L-SVM	**RBF-SVM**	**LR**	**J48**
P-SVM	3, 2, 5	8, 2, 0	2, 4, 4	6, 0, 4

To check for the statistical significance of the results, we applied unpaired *t*-test (in 95 % confidence interval) to the results of P-SVM, L-SVM, RBF-SVM, LR, and J48. Table 7.13 shows statistical comparison of the P-SVM with L-SVM, RBF-SVM, LR and J48 algorithms. The first, second, and third values in each cell shows the number of statistically significant cases where P-SVM performs better or worse than the algorithm in the column or the number of the cases where there is no difference. For instance, the table shows that in 3 out of 10 experiments P-SVM is better and in 2 out of 10 experiments L-SVM performs better, whereas in 5 experiments there is no significant difference between P-SVM and L-SVM. We also see that P-SVM is statistically better than RBF-SVM and J48. Moreover, when the number of statistically significant cases are considered, LR performs better than P-SVM, since it achieves better AUC in 4 out of 10 experiments.

We also observe that the performance P-SVM is more consistent than both L-SVM and RBF-SVM because in all experiments the AUC for P-SVM is above 0.60 including the ones where L-SVM or RBF-SVM is better (like experiments with Poi or Xalan data sets for example). On the other hand, although for some experiments a high AUC is observed for L-SVM, it has a very low AUC for Camel, Tomcat and Ivy data sets which are relatively skewed. Similarly, in 2 out of 10 experiments, for Poi and Xalan data sets, RBF-SVM is significantly better than P-SVM for balanced data sets, but in other 8 experiments its AUC value is very low around 0.50. As a summary, based on these observations, one can argue that although its AUC is smaller than L-SVM and RBF-SVM for some data sets, the precalculated kernel we propose is more trust worthy compared to both L-SVM and RBF-SVM since its performance is good for both skewed and balanced data sets.

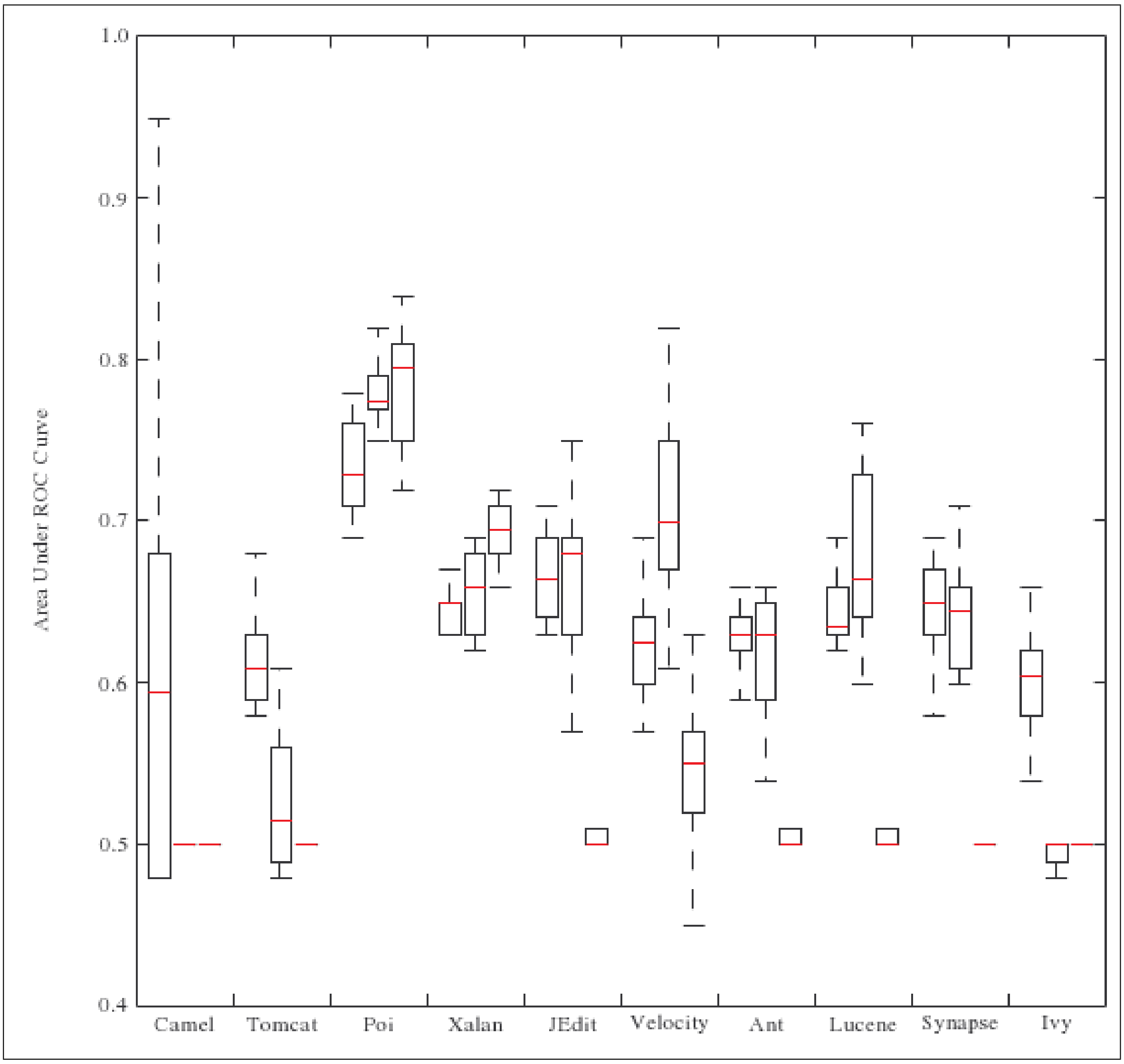


Figure 7.12. Box plots of AUC of P-SVM, L-SVM, and RBF-SVM for Camel, Tomcat, Poi, Xalan, JEdit, Velocity, Ant, Lucene, Synapse, and Ivy data sets.

The box plots of the AUC values observed are shown in Figure 7.12. For each data set, the first, the second and the third box plots shows P-SVM, L-SVM, and RBF-SVM respectively. P-SVM is significantly better than L-SVM on Camel, Tomcat, and Ivy data sets, whereas L-SVM is significantly better than P-SVM on Poi and Velocity data sets. On the other hand, P-SVM is significantly better than RBF-SVM on Camel, Tomcat, JEdit, Velocity, Ant, Lucene, Synapse, and Ivy data sets whereas RBF-SVM performs significantly better than P-SVM on Poi and Xalan data sets. As a summary, the precalculated kernel we propose (P-SVM), achieves better AUC values when compared with both L-SVM and RBF-SVM.

Table 7.14. The AUC found for precalculated kernel matrix (P-SVM) and the average similarity (AS) score in Experiment III.

	P-SVM	**AS**
Camel	0.62	1.8
Tomcat	0.61	0.07
Poi	0.73	3.17
Xalan	0.65	1.4
JEdit	0.67	1.43
Velocity	0.63	0.6
Ant	0.63	0.43
Lucene	0.65	1.38
Synapse	0.65	1.37
Ivy	0.6	1.43

The box plots of the AUC values observed are shown in Figure 7.13 where the first, the second and the third box plots shows P-SVM, LR, and J48 respectively. P-SVM is significantly better than LR on Tomcat and Ivy data sets, whereas LR is significantly better than P-SVM on Camel, Poi, Xalan, and Velocity data sets. On the other hand, P-SVM is significantly better than J48 on Camel, Tomcat, JEdit, Ant, Lucene, and Synapse data sets whereas there is no data set for which J48 performs significantly better than P-SVM. As a summary, considering 10 data sets, the performance of P-SVM is better than J48. Moreover, although P-SVM is more consistent than LR, since it is able to learn better considering all experiments i.e. its lowest AUC is 0.60, its number of win cases are lower when compared to LR.

We observe that when the amount of similarity among the files of a software system is high, then the AUC achieved by SVM with precalculated kernel matrix is also high. Higher similarity means a less sparse kernel matrix or a matrix that is filled with larger percentage values. When the similarity is high, there are more relationships among the files of the software system and this helps SVM to achieve better AUC while predicting defects.

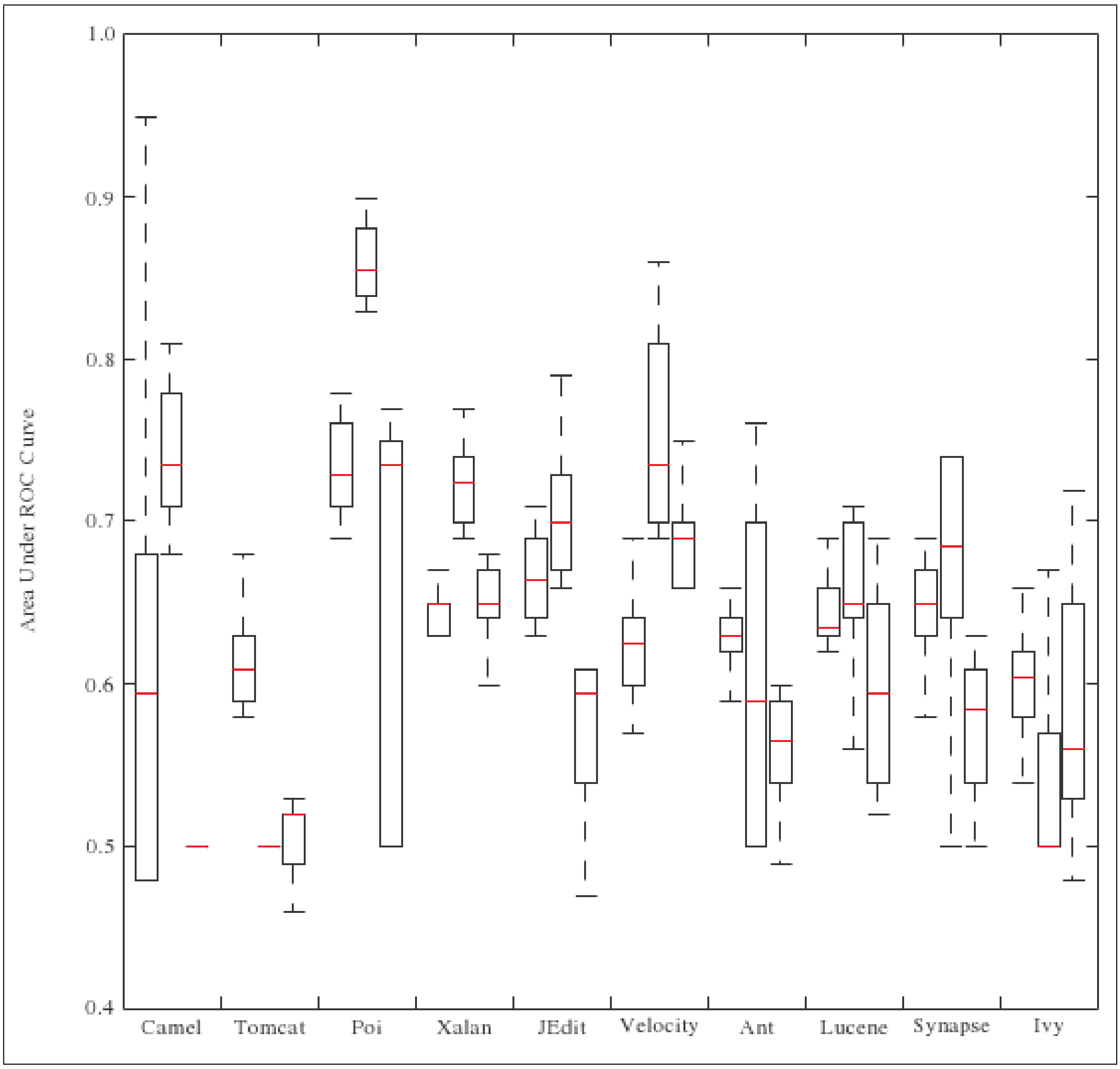


Figure 7.13. Box plots of AUC of P-SVM, LR, and J48 for Camel, Tomcat, Poi, Xalan, JEdit, Velocity, Ant, Lucene, Synapse, and Ivy data sets.

For each data set, the amount of similarity is calculated by the average of the similarity scores in the kernel matrix. In our experiments, the size of the kernel matrices are measured by the number of cells in the matrix. For example if a kernel matrix has 500 rows and 500 columns its size is counted as 250,000. Then the average similarity (AS) is calculated by dividing the sum of similarity scores in the kernel matrix by the size of the matrix for each data set. The average similarity values calculated is shown in Table 7.14. The highest average similarity values observed are 3.17 for Poi, 1.43 for JEdit, and 1.4 for Xalan.

We see a relationship between the AUC achieved and the average similarity scores

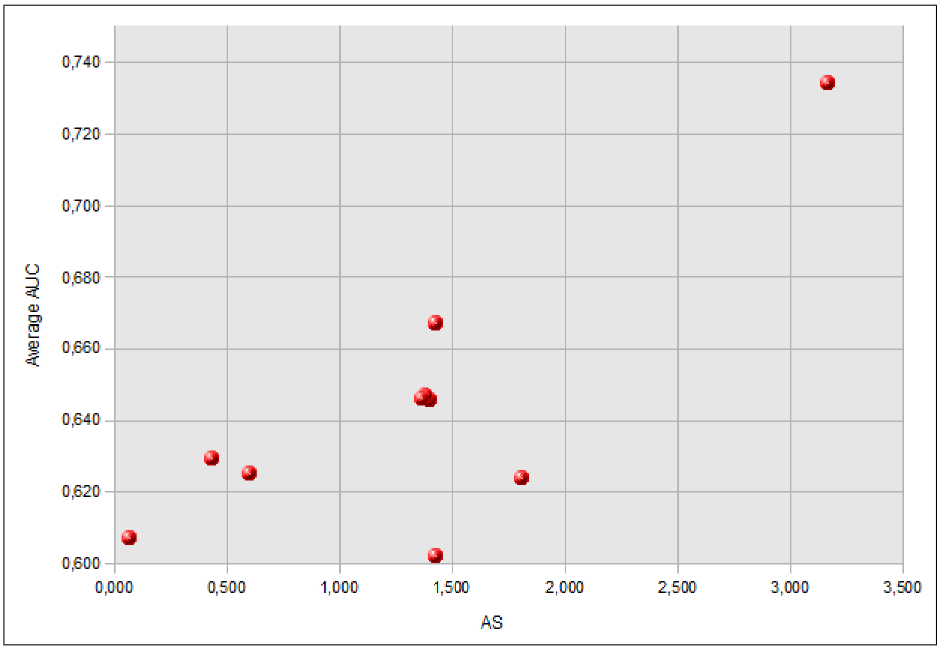


Figure 7.14. The relationship of AUC and average similarity (AS) in Experiment III.

found. For instance, the highest AS value (3.17) is observed for Poi data set for which the highest AUC value (0.73) is found. Similarly, except for Ivy data set, a low AUC is observed for data sets like Tomcat, Velocity and Ant, where the average similarity value found is also low. (See Figure 7.14).

This observation justifies our belief that structurally similar code patterns (either good or bad) are good hints to predict defectiveness. If we are able to detect more similarities among the files of a software system, and/or consider semantic similarity besides structural similarity, SVM with precomputed kernel matrix could perform much better and consistent in defect prediction.

7.4. Experiment IV: Prediction of the Number of Defects with Kernel Methods

7.4.1. Experiment Design

We define the major steps followed for each experiment below:

1. First we choose a data set from Promise data repository. While choosing the

data set, two conditions must be satisfied: First the data set must have enough entries for learning with cross validation and second it must have public source code to extract a source code similarity based precomputed kernel matrix (See Table 6.5).

2. We download the source code for the corresponding data sets from open source data repositories (like Apache.org).
3. We give the source code to MOSS as input and generate an output that shows how much similarity exists among the classes of the data set we chose (A sample MOSS output is shown in Table 6.7).
4. We convert the generated similarity output in the previous step to a precomputed kernel matrix. Each cell of this kernel matrix, shows how much similar the classes (or files) in the row and the column are (The kernel matrix is later normalized when processed in Weka). An example kernel matrix generated from a sample output in Table 6.7 is illustrated in Figure 6.4.
5. We create a new arff file (defect file) from the Promise data defect file that includes only the file names and the number of bugs.
6. To measure the performance of the proposed method (P-SVM), the defect file generated in step 5 and the precomputed kernel found in step 4 are given as inputs to SVM regression (SMOReg) in Weka with 5×2 cross validation.
7. To measure the performance of other kernels (L-SVM and RBF-SVM) and existing defect prediction methods (Linear regression and IBK), the original defect data file from Promise data repository is used.
8. The results of all methods are compared using the Root Mean Square Error (RMSE) where a smaller error interpreted as a better fit for regression.

7.4.2. Results

Table 7.15 shows average and standard deviation of RMSE for SVM with precalculated kernel matrix (P-SVM), linear kernel (L-SVM), and RBF kernel (RBF-SVM). Furthermore, we also compare the proposed kernel technique (P-SVM) with existing defect prediction methods linear regression (LinR) and IBK algorithms. We see that P-SVM achieves a smaller RMSE compared to L-SVM in 6 of 10 experiments, whereas

Table 7.15. Average and standard deviations of the Root Mean Square error for P-SVM, L-SVM, RBF-SVM, LinR, and IBK in Experiment IV.

Datasets	**P-SVM**	**L-SVM**	**RBF-SVM**	**LinR**	**IBK**
Camel	0.21 ± 0.06	0.21 ± 0.05	0.21 ± 0.05	0.56 ± 0.02	0.26 ± 0.17
Tomcat	0.61 ± 0.13	0.51 ± 0.12	0.51 ± 0.121	0.69 ± 0.07	0.6 ± 0.13
Poi	1.88 ± 0.53	1.76 ± 0.54	1.81 ± 0.75	1.89 ± 0.67	1.83 ± 0.55
Xalan	1.01 ± 0.13	1.09 ± 0.17	1.12 ± 0.18	0.98 ± 0.12	1.13 ± 0.15
JEdit	2.18 ± 1.08	2.27 ± 1.02	2.27 ± 1.01	2.19 ± 0.98	1.79 ± 0.56
Velocity	1.84 ± 0.33	1.97 ± 0.36	1.97 ± 0.36	1.9 ± 0.33	2.05 ± 0.38
Ant	1.12 ± 0.20	1.2 ± 0.19	1.21 ± 0.20	1.15 ± 0.16	1.2 ± 0.13
Lucene	2.78 ± 1.16	2.91 ± 1.03	2.94 ± 1.00	2.86 ± 0.90	2.7 ± 0.85
Synapse	1.08 ± 0.30	1.15 ± 0.37	1.15 ± 0.38	1.08 ± 0.33	1.2 ± 0.28
Ivy	0.51 ± 0.12	0.51 ± 0.12	0.51 ± 0.12	0.66 ± 0.08	0.52 ± 0.10

Table 7.16. A comparison of P-SVM with L-SVM, RBF-SVM, LinR, and IBK algorithms in Experiment IV. The first values in each cell show the number of cases where P-SVM is better than the algorithm in the column. The second values in each cell give the number of cases where P-SVM is worse than the algorithm in the column. The third values in each cell, represent the number of cases where there is a tie.

	L-SVM	**RBF-SVM**	**LinR**	**IBK**
P-SVM	6, 3, 1	6, 2, 2	8, 2, 0	6, 4, 0

it is worse in 3 experiments and in 1 experiment there is no difference. Similarly, it generates better RMSE in 6 experiments, whereas it is worse in only 2 experiments when compared to RBF-SVM. When we compare the results with existing techniques, we observe that P-SVM is better in 6 experiments but worse in 4 experiments when compared with IBK. Similarly, for LinR, it is better numerically in 8 experiments and it is worse in only 2 experiments (See Table 7.16).

We apply the Nemenyi post hoc test (in 95 % confidence interval) to detect if the performance of P-SVM differs significantly from other kernels and defect prediction techniques [107]. In this test, for all pairs of classifiers, the null hypothesis that

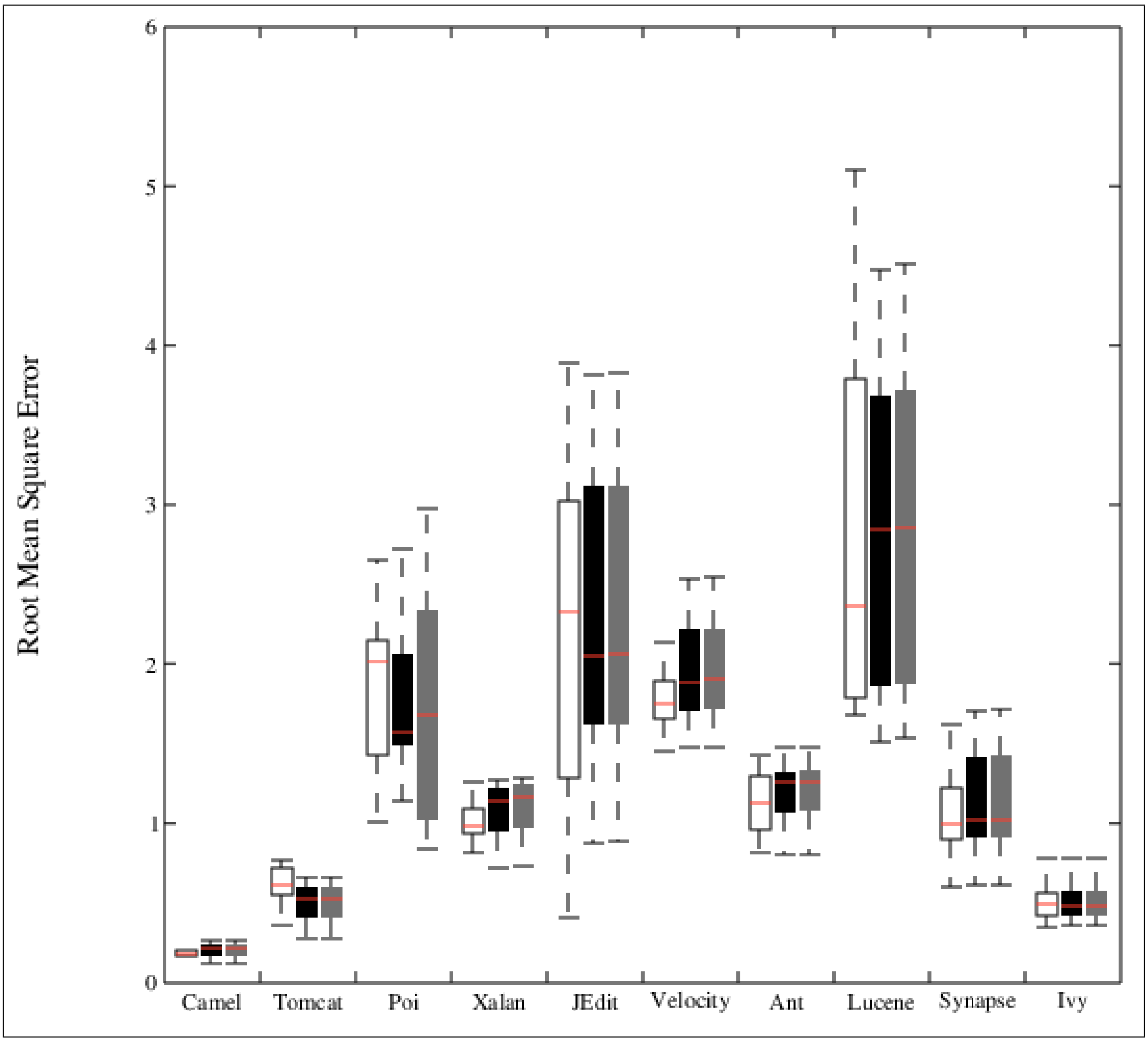


Figure 7.15. Box plots of RMSE of P-SVM, L-SVM, and RBF-SVM for each data set Camel, Tomcat, Poi, Xalan, JEdit, Velocity, Ant, Lucene, Synapse, and Ivy data sets.

their respective mean ranks are equal is tested. The null hypothesis is rejected if the difference between their mean ranks exceeds the critical difference (CD) defined as:

$$CD = q_\alpha \sqrt{\frac{k(k+1)}{6N}} \tag{7.1}$$

The value of q_α is based on the Studentized range statistic divided by $\sqrt{2}$ and is listed in statistical textbooks [107]. k represents the number of methods tested and N shows the number of observations for each method. The result of the test is shown in Figure 7.17. We observe that although P-SVM is the best algorithm, it is still on the same block with other algorithms and there is no significant difference among the mean ranks of the tested algorithms. However, since it is on the border of the critical difference

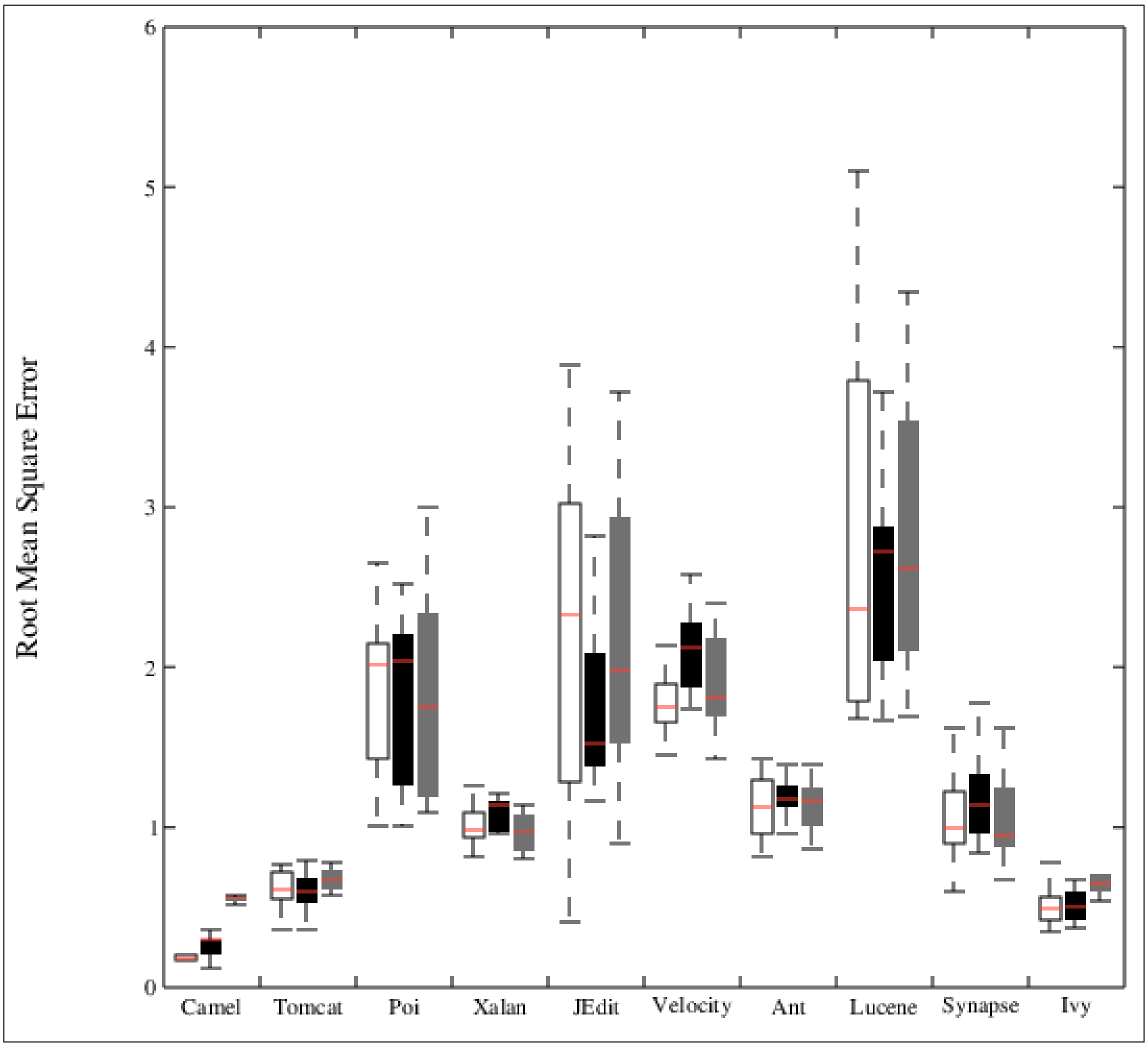


Figure 7.16. Box plots of RMSE of P-SVM, IBK, and LinR for each data set Camel, Tomcat, Poi, Xalan, JEdit, Velocity, Ant, Lucene, Synapse, and Ivy data sets.

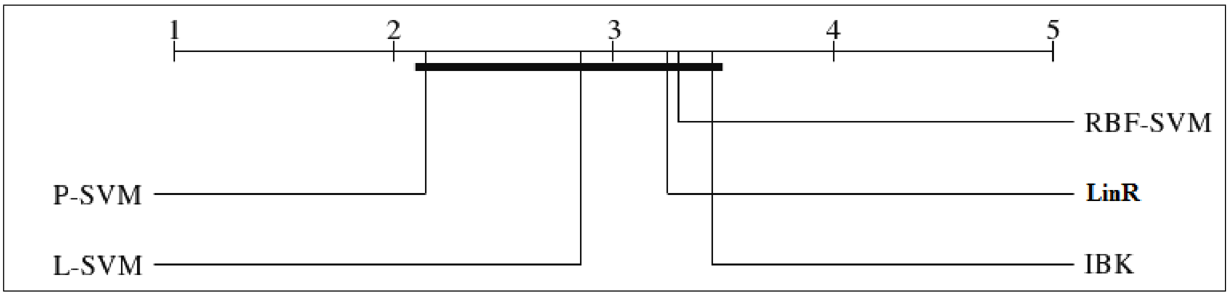


Figure 7.17. The mean ranks of the P-SVM, L-SVM, RBF-SVM, IBK, and LinR in Experiment IV.

block, if more algorithms are tested on the same data sets, it could be possible to visualize the difference and superiority of our proposed precomputed kernel method better.

We also check the statistical significance of the results, by applying unpaired t-test (in 95 % confidence interval) to the results obtained for P-SVM, L-SVM, RBF-SVM, LinR, and IBK methods. We found that, although the mean ranks of the techniques are not different statistically, we observe significant differences for some data sets. For example, the performance of P-SVM is better than LinR, for Camel and Ivy data sets with a p-value of 0.0 and 0.004 respectively (See Figure 7.16).

As a conclusion, we see that our proposed precomputed kernel (P-SVM) to predict the number of defects is comparable not only with existing linear and RBF kernels, but also with existing defect prediction techniques in the literature i.e. LinR and IBK.

7.5. Threats to Validity

According to Perry *et al.* there are three types of validity threats that should be considered in research studies. We briefly explain the methodology we follow to alleviate these threats [108].

An internal validity threat might arise if a cause effect relationship could not be established between the independent variables and the results. We address this issue by cross checking our results on different subsets of the data sets. During our experiments, not only we use 10 fold cross validation, but also we replicate the experiments on different subsets of all the data sets when necessary.

Construct validity threats might be observed when there are errors in the measurements. To mitigate this threat, first we automatize the metric extraction and preparation process and minimize the manual interventions, second we cross check the extracted metrics and try to find if any abnormal values exist.

External validity threats might arise if the results observed for one data set are not valid for other data sets. To mitigate external validity, we test our proposed method on several data sets and replicate the experiments on their subsets whenever necessary. Although our results are promising and our work is unique in the sense that metric effectiveness investigated on more than one data set, further research with other data sets and other search algorithms could be helpful to justify our findings. We should emphasize that the results we find are valid in the domain we choose, and our observations can not be generalized.

Chapter 8

Conclusion

8.1. Summary of Results

8.1.1. Bayesian Networks

We propose a novel method using Bayesian networks to define relationships among software metrics and defect proneness. We use data sets from Promise data repository and show that RFC, LOC, and LOCQ are the most useful metrics in defect proneness prediction. On the other hand, the effect of NOC and DIT on defectiveness is limited and untrustworthy.

8.1.2. Kernel Methods to Predict Defectiveness

We propose a new defect prediction model based on SVM with a precalculated kernel matrix. The precalculated kernel matrix shows the extent of the similarity among the files of the software system and is generated from the outputs of the plagiarism tool MOSS. We compare our approach with the linear and RBF kernels and logistic regression and J48 algorithms on 10 open source projects from Promise data repository and show that our precomputed kernel matrix can achieve better results than the linear and RBF kernels. On the other hand, when the defect prediction performance is considered, the precalculated kernel method we propose is clearly better than J48 algorithm, and is comparable with LR method.

We also show that when the amount of similarity among the files or classes of a software system is high, in other words, if the calculated kernel matrix is less sparse or contains larger similarity measurements, then the AUC achieved by the precalculated kernel matrix is also high.

8.1.3. Kernel Methods to Predict the Number of Defects

We propose a new kernel method to predict the number of defects in the software modules (classes or files). The proposed method is based on a precomputed kernel matrix which is based on the similarities among the modules of the software system. We compare our novel kernel method with existing kernels in the literature (linear and RBF kernels) and show that it achieves comparable results with them. Furthermore, the proposed defect prediction method is also comparable with some existing famous defect prediction methods in the literature i.e. linear regression and IBK.

8.2. Contributions

8.2.1. Bayesian Networks

The main contributions of our findings regarding Bayesian networks:

- We use Bayesian networks to model the relationships among metrics and defect proneness on multiple data sets. For instance Gyimothy *et al.* [43] used Mozilla data set whereas Zhou *et al.* and Pai and Dugan used KC1 data set from Nasa repository [76, 1]. The results obtained using one data set might be misleading since a metric might perform well on one data set but poor on another one. As Menzies *et al.* suggest, it is not adequate to assess defect learning methods using only one data set and only one learner, since the merits of the proposed techniques shall be evaluated via extensive experimentation [38]. Our work is a good contribution to the literature, since we determine the probabilistic causal or influential relationships among metrics and defect proneness, considering 9 data sets at the same time.
- We introduce a new metric we call Lack of Coding Quality (LOCQ) that can be used to predict defectiveness and is as effective as the famous object oriented metrics like CBO and WMC.

- We extract the Number of Developers (NOD) metric for data sets whose source code include developer information and show that there is a positive correlation between the number of developers and the extent of defect proneness. So, we suggest project managers not to assign too many developers to one class or file.
- It was found that in most experiments NOC and DIT are not effective on defectiveness.
- Furthermore, since LOC achieves one of the best scores in our experiments, we believe that it could be used for a quick defect prediction since it can be measured more easily compared to other metrics.
- LCOM3 and LCOM are less effective on defect proneness compared to LOC, CBO, RFC, LOCQ, and WMC.

8.2.2. Kernel Methods to Predict Defectiveness

As a summary, the main contributions of our findings are:

- We propose a novel kernel for defect prediction that could be used as an alternative for linear or RBF kernels.
- Extracting the metrics of a software system prior to defect prediction is necessary since most defect prediction techniques are based on the software metrics. But unfortunately, for some software projects, metrics may not be ready for defect proneness prediction. Our method is making it possible to predict defectiveness without the need to extract the software metrics, since SVM with the precalculated kernel we propose is based on the source code similarity.
- We prove that as the similarity among the files of a software system increases, SVM with source code similarity based kernel (P-SVM) achieves better AUC.

8.2.3. Kernel Methods to Predict the Number of Defects

- We propose a novel kernel method to predict the number of defects in the software modules.
- Prior to test phase or maintenance, developers can use our proposed method to

easily predict the most defective modules in the software system and focus on them primarily rather than testing each and every module in the system. This can decrease the testing effort and the total project cost automatically.

8.3. Future Work

As a future work, we plan to refine our research to include semantic similarity while calculating the kernel matrix. Graph based clone detection techniques are able to detect functional similarity and are more successful compared to the token based techniques in extracting semantic similarities [98]. We believe that when semantic similarity is considered besides structural similarity, the accuracy of defect prediction using a precalculated kernel matrix would be much higher.

On the other hand, we plan to add other software and process metrics to our Bayesian network model, to reveal the relationships among these metrics and defect proneness. We believe that rather than dealing with a large set of software metrics, focusing on the most important ones will improve the success rate and effectiveness of defect prediction studies.

References

1. G. Pai and J. Dugan, "Empirical analysis of software fault content and fault proneness using bayesian methods," *IEEE Transactions on Software Engineering*, vol. 33, no. 10, pp. 675–686, 2007.

2. C. Jin and J.-A. Liu, "Applications of support vector machine and unsupervised learning for predicting maintainability using object-oriented metrics," in *2010 Second International Conference on Multimedia and Information Technology (MMIT)*, vol. 1, pp. 24 –27, 2010.

3. S. Shivaji, E. Whitehead, R. Akella, and S. Kim, "Reducing features to improve bug prediction," in *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, pp. 600 –604, 2009.

4. N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 675–689, 1999.

5. E. Alpaydın, *Introduction to Machine Learning.* The MIT Press, 2004.

6. D. Gray, D. Bowes, N. Davey, Y. Sun, and B. Christianson, "Using the support vector machine as a classification method for software defect prediction with static code metrics," in *Engineering Applications of Neural Networks*, Springer Berlin Heidelberg, 2009.

7. T. Fletcher, "Support vector machines explained," 2008.

8. M. Shin, S. Ratanothayanon, A. L. Goel, and R. A. Paul, "Parsimonious classifiers for software quality assessment," *IEEE International Symposium on High-Assurance Systems Engineering*, pp. 411–412, 2007.

9. W. Humphrey, “Why big software projects fail: The 12 key questions.,” *The Journal of Defense Software Engineering*, vol. 18, pp. 25–29, 2005.

10. The Standish Group, “Chaos report,” 1995. http://www.cs.nmt.edu/cs328/reading/Standish.pdf, last visited 15th of June, 2008.

11. C. Jones, “Patterns of large software systems: Failure and success,” *Computer*, vol. 28, no. 3, pp. 86–87, 1995.

12. J. Brooks and P. Frederick, *The Mythical Man-Month.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

13. J. Verner, J. Sampson, and N. Cerpa, “What factors lead to software project failure?,” in *Research Challenges in Information Science, 2008. RCIS 2008. Second International Conference on*, pp. 71 –80, june 2008.

14. K. R. Linberg, “Software developer perceptions about software project failure: A case study,” *Journal of Systems and Software*, vol. 49, no. 23, pp. 177 – 192, 1999.

15. J. D. Procaccino, J. M. Verner, S. P. Overmyer, and M. E. Darter, “Case study: Factors for early prediction of software development success,” *Information & Software Technology*, vol. 44, no. 1, pp. 53–62, 2002.

16. N. Cerpa and J. M. Verner, “Why did your project fail?,” *Communication ACM*, vol. 52, pp. 130–134, Dec. 2009.

17. S. P. Masticola, “A simple estimate of the cost of software project failures and the breakeven effectiveness of project risk management,” *Economics of Software and Computation, International Workshop on*, vol. 0, p. 6, 2007.

18. K. Ewusi-Mensah, *Software Development Failures: Anatomy of Abandoned Projects.* Mit Press, 2003.

19. M. K. Daskalantonakis, “A practical view of software measurement and implementation experiences within motorola,” *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 998–1010, 1992.

20. M. Suffian and M. Abdullah, “Establishing a defect prediction model using a combination of product metrics as predictors via six sigma methodology,” in *Information Technology (ITSim), 2010 International Symposium in*, pp. 1087 –1092, 2010.

21. J. Ekanayake, J. Tappolet, H. Gall, and A. Bernstein, “Tracking concept drift of software projects using defect prediction quality,” in *Mining Software Repositories, 2009. MSR ’09. 6th IEEE International Working Conference on*, pp. 51 –60, 2009.

22. M. Shepperd and G. Kadoda, “Comparing software prediction techniques using simulation,” *IEEE Transactions on Software Engineering*, vol. 27, November 2001.

23. Q. Song, M. Shepperd, M. Cartwright, and C. Mair, “Software defect association mining and defect correction effort prediction,” *Software Engineering, IEEE Transactions on*, vol. 32, pp. 69 – 82, feb. 2006.

24. T. Khoshgoftaar, K. Ganesan, E. Allen, F. Ross, R. Munikoti, N. Goel, and A. Nandi, “Predicting fault-prone modules with case-based reasoning,” in *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*, pp. 27 –35, 2-5 1997.

25. T. Khoshgoftaar, E. Allen, and J. Busboom, “Modeling software quality: The software measurement analysis and reliability toolkit,” in *Tools with Artificial Intelligence, 2000. ICTAI 2000. Proceedings. 12th IEEE International Conference on*, pp. 54 –61, 2000.

26. T. M. Khoshgoftaar, A. S. Pandya, and D. L. Lanning, “Application of neural networks for predicting program faults.,” *Annals of Software Engineering*, vol. 1,

pp. 141–154, 1995.

27. M. M. T. Thwin and T.-S. Quah, "Application of neural network for predicting software development faults using object oriented design metrics," in *Neural Information Processing, 2002. ICONIP '02. Proceedings of the 9th International Conference on*, vol. 5, pp. 2312 – 2316 vol.5, nov. 2002.

28. A. Kaur, P. Sandhu, and A. Bra, "Early software fault prediction using real time defect data," in *Machine Vision, 2009. ICMV '09. Second International Conference on*, pp. 242 –245, dec. 2009.

29. J. C. Munson and T. M. Khoshgoftaar, "The detection of fault-prone programs," *IEEE Transactions on Software Engineering*, vol. 18, pp. 423–433, May 1992.

30. G. D. Boetticher, "Nearest neighbor sampling for better defect prediction," in *Proceedings of the 2005 workshop on Predictor models in software engineering*, PROMISE '05, 2005.

31. A. G. Koru and H. Liu, "An investigation of the effect of module size on defect prediction using static measures," *SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–5, 2005.

32. N. Fenton, P. Krause, and M. Neil, "Software measurement: Uncertainty and causal modeling," *IEEE Software*, vol. 19, pp. 116–122, 2002.

33. D. Zhang, "Applying machine learning algorithms in software development," in *Proceedings of the 2000 Monterey Workshop on Modeling Software System Structures in a Fastly Moving Scenario*, pp. 275–291, 2000.

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

35. Y. Hu, X. Zhang, X. Sun, M. Liu, and J. Du, "An intelligent model for software

project risk prediction," in *International Conference on Information Management, Innovation Management and Industrial Engineering, 2009*, vol. 1, pp. 629 –632, 2009.

36. T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, pp. 308–320, 1976.

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

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

39. T. Zimmermann and N. Nagappan, "Predicting defects with program dependencies," *International Symposium on Empirical Software Engineering and Measurement*, pp. 435–438, 2009.

40. A. H. Watson, T. J. Mccabe, and D. R. Wallace, "Special publication 500-235, structured testing: A software testing methodology using the cyclomatic complexity metric," in *U.S. Department of Commerce/National Institute of Standards and Technology*, 1996.

41. T. J. McCabe and C. W. Butler, "Design complexity measurement and testing," *Communications of the ACM*, vol. 32, no. 12, pp. 1415–1425, 1989.

42. S. R. Chidamber and C. F. Kemerer, "Towards a metrics suite for object oriented design," *SIGPLAN*, vol. 26, no. 11, pp. 197–211, 1991.

43. T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897–910, 2005.

44. K. E. Emam, W. Melo, and J. C. Machado, "The prediction of faulty classes using

object oriented design metrics," *Journal of Systems and Software*, vol. 56, no. 1, pp. 63–75, 2001.

45. S. Matsumoto, Y. Kamei, A. Monden, K.-i. Matsumoto, and M. Nakamura, "An analysis of developer metrics for fault prediction," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, (New York, NY, USA), pp. 18:1–18:9, ACM, 2010.

46. E. Arisholm, L. C. Briand, and E. B. Johannessen, "A Systematic and Comprehensive Investigation of Methods to Build and Evaluate Fault Prediction Models," *Journal of Systems and Software*, vol. 83, no. 1, 2009.

47. E. Mills, C.-M. U. S. E. Institute, and U. S. D. of Defense, *Software Metrics*. Technical report (Carnegie-Mellon University. Software Engineering Institute), Software Engineering Institute, 1988.

48. "Nasa/wvu iv and v facility, metrics data program available from http://mdp.ivv.nasa.gov/; internet accessed 2010."

49. G. Boetticher, T. Menzies, and T. Ostrand, "Promise repository of empirical software engineering data http://promisedata.org/ repository, west virginia university, department of computer science," 2007.

50. A. Tosun, A. Bener, and R. Kale, "Ai-based software defect predictors: Applications and benefits in a case study," *Proceedings of the Twenty-Second Innovative Applications of Artificial Intelligence Conference*, 2010.

51. T. Menzies, J. DiStefano, A. Orrego, and R. M. Chapman, "Assessing predictors of software defects," *Workshop on Predictive Software Models*, 2004.

52. L. Pickard, B. Kitchenham, and S. Linkman, "An investigation of analysis techniques for software datasets," *IEEE International Symposium on Software Metrics*, p. 130, 1999.

53. W. P. Giancarlo Succi, Milorad Stefanovic, “Advanced statistical models for software data,” Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB, Canada, 2001.

54. N. F. Schneidewind, “Investigation of logistic regression as a discriminant of software quality,” *IEEE International Symposium on Software Metrics*, p. 328, 2001.

55. V. R. Basili, L. Briand, and W. L. Melo, “A validation of object-oriented design metrics as quality indicators,” *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751–761, 1995.

56. D. Wolpert and W. Macready, “No free lunch theorems for optimization,” *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, 1997.

57. I. Myrtveit, E. Stensrud, and M. Shepperd, “Reliability and validity in comparative studies of software prediction models,” *IEEE Transactions on Software Engineering*, vol. 31, pp. 380–391, May 2005.

58. L. Guo, Y. Ma, B. Cukic, and H. Singh, “Robust prediction of fault-proneness by random forests,” in *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, 2004.

59. A. Kaur and R. Malhotra, “Application of random forest in predicting fault-prone classes,” in *Advanced Computer Theory and Engineering, 2008. ICACTE ’08. International Conference on*, 2008.

60. F. Xing, P. Guo, and M. Lyu, “A novel method for early software quality prediction based on support vector machine,” in *International Symposium on Software Reliability Engineering*, 2005.

61. L. Hatton, “Reexamining the fault density-component size connection,” *IEEE Software*, vol. 14, no. 2, pp. 89–97, 1997.

62. J. E. Gaffney, “Estimating the number of faults in code,” *IEEE Transactions on*

Software Engineering, vol. 10, no. 4, pp. 459 –464, 1984.

63. V. R. Basili and B. T. Perricone, “Software errors and complexity: An empirical investigation,” *Communications of the ACM*, vol. 27, no. 1, pp. 42–52, 1984.

64. K.-H. Moller and D. Paulish, “An empirical investigation of software fault distribution,” in *Proceedings of the First International Software Metrics Symposium*, pp. 82–90, 1993.

65. M. Neil, “Multivariate assessment of software products,” *Software Testing, Verification and Reliability*, vol. 1, no. 4, pp. 17–37, 1992.

66. N. Fenton and N. Ohlsson, “Quantitative analysis of faults and failures in a complex software system,” *IEEE Transactions on Software Engineering*, vol. 26, no. 8, pp. 797–814, 2000.

67. T. Menzies and M. Shepperd, “Special issue on repeatable results in software engineering prediction,” *Empirical Software Engineering*, vol. 17, no. 1-2, pp. 1–17, 2012.

68. T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. Cok, “Local vs global models for effort estimation and defect prediction,” in *Proceedings of the 26st IEEE/ACM International Conference on Automated Software Engineering*, (Lawrence, Kansas, USA), November 2011.

69. D. Posnett, V. Filkov, and P. T. Devanbu, “Ecological inference in empirical software engineering.,” pp. 362–371, IEEE, 2011.

70. Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu, “A general software defect-proneness prediction framework,” *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 356 –370, 2011.

71. T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, “A systematic review of fault prediction performance in software engineering,” *IEEE Transactions on*

Software Engineering, vol. 99, no. PrePrints, 2011.

72. I. Gondra, “Applying machine learning to software fault-proneness prediction.,” *Journal of Systems and Software*, vol. 81, no. 2, pp. 186–195, 2008.

73. K. O. Elish and M. O. Elish, “Predicting defect-prone software modules using support vector machines,” *Journal of Systems and Software*, vol. 81, no. 5, pp. 649–660, 2008.

74. E. Arisholm, L. C. Briand, and M. Fuglerud, “Data mining techniques for building fault-proneness models in telecom java software,” in *Software Reliability, 2007. ISSRE ’07. The 18th IEEE International Symposium on*, 2007.

75. G. F. Cooper and E. Herskovits, “A bayesian method for the induction of probabilistic networks from data,” *Machine Learning*, vol. 9, no. 4, pp. 309–347, 1992.

76. Y. Zhou and H. Leung, “Empirical analysis of object-oriented design metrics for predicting high and low severity faults,” *IEEE Transactions on Software Engineering*, vol. 32, pp. 771–789, 2006.

77. S. Bibi and I. Stamelos, “Software process modeling with bayesian belief networks,” *Proceedings of 10th International Software Metrics Symposium*, 2004.

78. E. Pérez-Miñana and J.-J. Gras, “Improving fault prediction using bayesian networks for the development of embedded software applications: Research articles,” *Software Testing, Verification and Reliability*, vol. 16, pp. 157–174, Sept. 2006.

79. S. Amasaki, Y. Takagi, O. Mizuno, and T. Kikuno, “A bayesian belief network for assessing the likelihood of fault content,” in *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE ’03, (Washington, DC, USA), IEEE Computer Society, 2003.

80. N. Fenton, M. Neil, W. Marsh, P. Hearty, D. Marquez, P. Krause, and R. Mishra, “Predicting software defects in varying development life cycles using bayesian

nets," *Inf. Softw. Technol.*, vol. 49, pp. 32–43, Jan. 2007.

81. N. Fenton, M. Neil, and D. Marquez, "Using bayesian networks to predict software defects and reliability," *Proceedings of the Institution of Mechanical Engineers, Part O, Journal of Risk and Reliability*, 2008.

82. K. Dejaeger, T. Verbraken, and B. Baesens, "Towards comprehensible software fault prediction models using bayesian network classifiers," *IEEE Transactions on Software Engineering*, vol. 99, no. PrePrints, 2012.

83. B. Norick, J. Krohn, E. Howard, B. Welna, and C. Izurieta, "Effects of the number of developers on code quality in open source software: A case study.," in *ESEM* (G. Succi, M. Morisio, and N. Nagappan, eds.), ACM, 2010.

84. P. C. Pendharkar and J. A. Rodger, "An empirical study of the impact of team size on software development effort," *Information Technology and Management*, vol. 8, no. 4, pp. 253–262, 2007.

85. N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality: An empirical case study," in *Proceedings of the 30th international conference on Software engineering*, ICSE '08, (New York, NY, USA), pp. 521–530, ACM, 2008.

86. A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 3, pp. 309–346, 2002.

87. C. Cortes and V. Vapnik, "Support vector networks," *Machine Learning*, vol. 20, no. 3, pp. 273–297, 1995. 10.1007/BF00994018.

88. G. Wu, E. Y. Chang, and Z. Zhang, "An analysis of transformation on non-positive semidefinite similarity matrix for kernel machines," in *Proceedings of the 22nd International Conference on Machine Learning*, 2005.

89. C.-W. Hsu, C.-C. Chang, and C.-J. Lin, “A practical guide to support vector classification,” tech. rep., Department of Computer Science, National Taiwan University, 2003.

90. T. Joachims, “Text categorization with support vector machines: Learning with many relevant features,” *Machine Learning ECML98*, vol. 1398, no. 23, 1998.

91. J. Shawe-Taylor and N. Cristianini, *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004.

92. M. Mozgovoy, “Desktop tools for offline plagiarism detection in computer programs,” *Informatics in education*, vol. 5, pp. 97–112, January 2006.

93. A. Barrón-Cedeño and P. Rosso, “On automatic plagiarism detection based on n-grams comparison,” in *Proceedings of the 31th European Conference on IR Research on Advances in Information Retrieval*, ECIR ’09, (Berlin, Heidelberg), pp. 696–700, Springer-Verlag, 2009.

94. A. Martins, “String kernels and similarity measures for information retrieval,” tech. rep., 2006.

95. T. Smith and M. Waterman, “Identification of common molecular subsequences,” *Journal of Molecular Biology*, vol. 147, no. 1, pp. 195–197, 1981.

96. J.-H. Ji, S.-H. Park, G. Woo, and H.-G. Cho, “Source code similarity detection using adaptive local alignment of keywords,” in *Proceedings of the Eighth International Conference on Parallel and Distributed Computing, Applications and Technologies*, PDCAT ’07, (Washington, DC, USA), pp. 179–180, IEEE Computer Society, 2007.

97. L. Prechelt, G. Malpohl, and M. Philippsen, “Jplag: Finding plagiarisms among a set of programs,” technical report, University of Karlsruhe, Department of Informatics, 2000.

98. C. K. Roy, J. R. Cordy, and R. Koschke, “Comparison and evaluation of code clone detection techniques and tools: A qualitative approach,” *Science of Computer Programming*, 2009.

99. B. Henderson-Sellers, *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

100. A. Aiken, “Moss (measure of software similarity).” http://cs.stanford.edu/ aiken/moss/, 1997.

101. G. Cosma, *An Approach to Source-code Plagiarism Detection Investigation Using Latent Semantic Analysis.* PhD thesis, University of Warwick, Coventry CV4 7AL, UK, 2008.

102. T. Schanz and C. Izurieta, “Object oriented design pattern decay: A taxonomy,” in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM ’10, (New York, NY, USA), ACM, 2010.

103. J. Bansiya and C. Davis, “A hierarchical model for object-oriented design quality assessment,” *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4–17, 2002.

104. M.-H. Tang, M.-H. Kao, and M.-H. Chen, “An empirical study on object-oriented metrics,” in *Proc. International Software Metrics Symposium*, pp. 242–249, IEEE, 1999.

105. S. Schleimer, D. S. Wilkerson, and A. Aiken, “Winnowing: Local algorithms for document fingerprinting,” in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD ’03, (New York, NY, USA), ACM, 2003.

106. M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten,

"The WEKA Data Mining Software: An Update," *SIGKDD Explorations*, vol. 11, no. 1, pp. 10–18, 2009.

107. J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *Journal of Machine Learning Research*, vol. 7, pp. 1–30, 2006.

108. D. E. Perry, A. A. Porter, and L. G. Votta, "Empirical studies of software engineering: A road map," in *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, (New York, NY, USA), pp. 345–355, ACM, 2000.

Curriculum Vitae