8-7-2020

A Deep Learning approach to predict software bugs using micro patterns and software metrics

Marcus Brumfield

A Deep Learning approach to predict software bugs using micro patterns and software metrics

By

Marcus Brumfield

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

August 2020

Name: Marcus Brumfield

Date of Degree: August 7, 2020

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Stefano Iannucci

Title of Study: A Deep Learning approach to predict software bugs using micro patterns and software metrics

Pages of Study: 47

Candidate for Degree of Master of Science

Software bugs prediction is one of the most active research areas in the software engineering community. The process of testing and debugging code proves to be costly during the software development life cycle. Software metrics measure the quality of source code to identify software bugs and vulnerabilities. Traceable code patterns are able to describe code at a finer granularity level to measure quality. Micro patterns will be used in this research to mechanically describe java code at the class level. Machine learning has also been introduced for bug prediction to localize source code for testing and debugging. Deep Learning is a branch of Machine Learning that is relatively new. This research looks to improve the prediction of software bugs by utilizing micro patterns with deep learning techniques. Software bug prediction at a finer granularity level will enable developers to localize code to test and debug during the development process.

DEDICATION

To my parents Roderick and Adarianne Brumfield and my siblings Arielle and Arianna Brumfield.

ACKNOWLEDGEMENTS

I would like to thank Dr. Stefano Iannucci, Dr. Byron J. Williams, and Dr. Tanmay Bhowmik for guiding me throughout my matriculation at Mississippi State University. I also want to thank my family and friends for the continuous support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Identifying and predicting software bugs is an important research area in software engineering in terms of improving software quality. Early identification of files that contain software bugs will help improve the testing phase of the software development life cycle (SDLC). Bug prediction involves the use of static analysis and machine learning techniques to identify defects in software. Most software projects are developed with a tight deadline to push into production. Bug prediction enables developers to prioritize which files to debug and test, which leads to a faster development process. This technique also helps in ensuring software quality during the development process to significantly decrease the appearance of bugs in software. Even with the advances in this research field, there is a need for improving the results of bug prediction techniques [3].

1.1 Software Metrics

Software Metrics have been used in research to measure software quality. Complexity, volume, and object-oriented metrics have been utilized by researchers in correlation to software defects. Software metrics have been widely used in bug prediction models because they are useful, generalizable, easy to use, and widely-used. In terms of being useful, software metrics have been used in many companies in the world to predict defects. They are also generalizable in that many datasets and software projects in different programming languages can use them for measurement. Software metrics are easy to use because there are tools that can automatically process the results. They are also widely-used by researchers since around 1990 [18]. Existing research proves that these techniques aid developers in identifying software that has potential defects during development. The overall goal involves localizing the software bugs to ensure that only the affected files are flagged to test and debug. Researchers have also found correlations between metrics and software defects in software projects to build predictive models using static analysis and machine learning techniques. Although the software engineering research community have made strides towards measuring software quality, there is a need to address the limitations [5].

Existing software metrics are able to measure software quality, but with a high percentage of false negative rates. This limitation causes code that does not contain bugs to be tested and debugged, which increases the time to complete the testing phase of the SDLC. Also, software metrics do not give feedback to developers on how to ensure software quality.

There is not a consensus among the software engineering community for which metrics should be used for all software projects. Although software metrics are commonly used to evaluate software quality, there may be other undiscovered methods that can accurately identify defects. Another limitation of utilizing traditional software metrics involves the lack of localizing the areas of code that cause the bugs. There is still a need to identify metrics that can identify bugs in code at various levels of granularity [20].

1.2 Traceable Code Patterns

Traceable code patterns have been introduced to identify vulnerable code at a lower granularity level such as class and method level. The two types of traceable code patterns are class-level patterns called micro-patterns, and method-level patterns called nano-patterns [13, 26]. These patterns can be defined on many variants of modules written with the Java programming language. Gil and Maman [13] introduced the concept of traceable code patterns by presenting a catalog of 27 micro-patterns that can be mechanically recognized to identify the structure of a class. Singer et al. in [26] identified 17 nano-patterns that capture the properties of Java methods. Traceable code patterns enable code to be described at the function-level and the class-level. This will also help developers locate specific code locations to debug and test.

Utilizing traceable code patterns will enable developers to localize the classes and methods that are likely to contain bugs. This will improve the productivity of the testing phase by not only identifying the files that contain bugs, but also classes and functions that are affected [29]. Developers will also have a better understanding of which patterns are more likely to result in software bugs. Traceable code patterns such as micro patterns and nano-patterns have been identified for the Java programming language. Software tools have been developed to identify micro patterns and nano-patterns in Java programs. These tools are limited only to Java programs, but the concept of traceable code patterns could be applied to other programming languages. Locating bugs at a finer granularity level, class and method, significantly reduces the time and cost of testing and debugging before moving software to production [27].

1.3 Machine Learning Techniques

Traditional Machine Learning techniques such as Decision Trees, Support Vector Machine (SVM), and Naïve Bayes have been utilized to predict software bugs. Research studies have proven to predict software defects in terms of bugs and vulnerabilities. Techniques include text-based and metrics-based prediction. Traditional machine learning algorithms produce promising results in terms of accuracy, but false positive and false negative rates are relatively high. Also, most works involve conducting experiments on specific software projects [6].

Researchers have also investigated the use of Deep Learning for vulnerability prediction with software metrics and text classifier techniques [35]. Li et. al in [15] developed a Deep Learning convolutional neural network to predict software defects using information such as semantic and structural information for program files along with software metrics. Feedforward neural networks have shown promise in producing more accurate results in predicting vulnerable code versus traditional machine learning techniques [6]. Researchers have also conducted research with various techniques with neural networks such as text-based and metrics-based prediction [32]. The challenge in bug prediction involves fully utilizing Deep Learning methods at a larger scale. A dataset containing an order of millions of records would be considered to be sufficient for Deep Learning. Many researchers have utilized this technique at a smaller scale as a proof of concept.

Thus far, traceable code patterns have not been used in combination with Deep Learning techniques in current research studies. Sultana et al. in [31] used traditional machine learning algorithms to predict software vulnerabilities where micro patterns produced relatively high results for Recall compared to software metrics. Other measures such as False Negative rate, Precision, and F-measure were relatively low in terms of predicting vulnerable code. The open issue with software defect model involves giving developers a better sense of granularity. This will give developers a better technique for identifying the code location that is causing a software bug [28]. With the utilization of micro patterns, identifying software at the class level will be enabled.

1.4 Research Goal and Questions

The research goal is to compare the prediction results of software metrics and traceable pattern by using Deep Learning. In order to conduct this research, the following questions are proposed:

Research Question 1: *What is the performance of software bug prediction model using micro patterns with a multilayer perceptron?* The deep learning approach will be evaluated by using performance measures such as false positive rate, precision, recall, and f-measure. The results will be compared with traditional machine learning techniques that will include the algorithms Naïve Bayes, Random Forest, and Support Vector Machine.

Research Question 2: *What is the performance of the software bug prediction model when using class-level software metrics as features with a multilayer perceptron?* Similar to Research Question 1, the deep learning approach will be evaluated and compared to the results of the traditional machine learning techniques. Class-level software metrics will be used as features for predicting bugs.

Research Question 3: *What is the performance of the software bug prediction model as the training size increases?* The experiment related to this question evaluate the deep learning approach as the prediction model receives more training data. This also involves a comparison with the traditional machine learning algorithms. The goal of this research question is to show whether or not deep learning can outperform the traditional machine learning techniques as the training size increases.

1.5 Contribution

Identifying software bugs at the class-level will help software developers identify code constructs that contain bugs and also improve software quality. Identifying classes that contain software bugs will also help software developers in the testing phase of the Software Development Life Cycle (SDLC). So far, traceable code patterns have not been investigated along with Deep learning techniques to identify vulnerable classes and functions. To our knowledge, this research is the first to utilize micro patterns along with Deep Learning for software bug prediction. This research will provide the foundation for developing prediction models that utilize traceable code patterns for Deep Learning algorithms.

CHAPTER 2

RELATED WORK

This chapter discusses previous research that influenced this current study in bug prediction. The topics that will be discussed include software metrics used for bug prediction, traceable code patterns, and machine learning techniques. This will provide background information and serve as a literature review for software bug prediction.

2.1 Software Metrics

Metrics can be used to measure security in software. Research has shown that the number of code-level metrics is relatively lower compared to design level and system level metrics. Common software metrics include lines of code, cyclomatic complexity, nesting levels in functions, and interactions between functions [4, 5]. The studies show that these metrics were very common in predicting vulnerable code. There is a need for the development of new metrics that measure vulnerabilities in source code. Using metrics at the code-level increases the probability of identifying vulnerable code locations [28]. Chowdhury et al. in [5] utilized complexity, volume, and object-oriented metrics to predict vulnerabilities in source code. The metrics were used in a previous study for software defect prediction. Nagappan et al. in [22] included complexity metrics to predict defects and vulnerabilities.

Complexity is theorized to be the catalyst for insecure software. The number of lines of code, the number of independent paths, and the cyclomatic complexity in a program are examples of what complexity metrics evaluate. Complexity in software is regarded as being the biggest indicator of vulnerabilities. Software that is complex makes testing more difficult and also harder to understand for developers [21]. Research by Shin et al. in [24] proposed complexity metrics that are used to measure vulnerabilities in software. The measurements used for complexity involves file complexity, coupling, and comment metrics. The research hypothesized that vulnerable files have higher file complexity, higher coupling, and fewer amounts of comments. Higher complexity creates a higher chance for defects in code that lead to vulnerabilities.

Code churn is a metric used to evaluate the number of changes made to software. Making changes to software can bring up new vulnerabilities in software. The measurements used for this metric involves the number of times a file has been changed, the number of lines changed for a file, and the number of new lines added to a file. The higher number of times a file has been changed and the higher number of lines of code that have been changed are regarded to be indicators of vulnerabilities [24]. These metrics have been used in research conducted by Zimmerman et al. in [36] to predict vulnerabilities in the commercial project Windows Vista. It was found in this research that the results of using this metric yielded fewer false negatives but also higher false positives. This means that there was a high number of indicators of vulnerable code that were not actually vulnerable.

Object-oriented metrics are used to measure the object-oriented properties in source code. These types of metrics evaluate coupling between functions and methods. Coupling involves the passing of data and the interaction between other functions and methods. The theory is that highly coupled functions and methods are more likely to contain vulnerabilities. Higher coupling involves how modules in software interact with one another. Object-oriented software metrics have been utilized to measure the quality of classes and functions [6].

Software metrics are highly regarded as the standard way to measure the quality and security in source code. Previous studies have shown promising results in accurately predicting software bugs. Traditional software metrics do not indicate the code location that causes software bugs [32, 35]. This leads to research that utilizes other forms of metrics that can measure code at a finer granularity level. This research uses class-level software metrics to predict software bugs in Java classes. Table 2.1 lists the software metrics used in this research.

2.2 Traceable Code Patterns

Traceable code patterns have been introduced to identify vulnerable code at a finer granularity level such as class and method level. The two types of traceable code patterns are class-level patterns called micro-patterns, and method-level patterns called nano-patterns. These patterns are defined for Java classes and methods. Traceable code patterns are similar to design patterns, but they are able to describe source code at a lower level of abstraction. The use of these types of patterns will enable developers to locate the area of code that needs to be tested and debugged [13, 26].

Table 2.1: Class-level software metrics calculated in the research by [12].

Metrics	Description
LCOM5	Lack of Cohesion in Methods 5
NL	Nesting Level
NLE	Nesting Level Else-If
WMC	Weighted Methods per Class
CBO	Coupling Between Object classes
CBOI	Coupling Between Object classes Inverse
NII	Number of Incoming Invocations
NOI	Number of Outgoing Invocations
RFC	Response set For Class
AD	API Documentation
CD	Comment Density
CLOC	Comment Lines of Code
DLOC	Documentation Lines of Code
PDA	Public Documented API
PUA	Public Undocumented API
TCD	Total Comment Density
TCLOC	Total Comment Lines of Code
DIT	Depth of Inheritance Tree
NOA	Number of Ancestors
NOC	Number of Children
NOD	Number of Descendants
NOP	Number of Parents
LLOC	Logical Lines of Code
LOC	Lines of Code
NA	Number of Attributes
NG	Number of Getters
NLA	Number of Local Attributes
NLG	Number of Local Getters
NLM	Number of Local Methods
NLPA	Number of Local Public Attributes
NLPM	Number of Local Public Methods
NLS	Number of Local Setters
NM	Number of Methods
NOS	Number of Statements
NPA	Number of Public Attributes
NPM	Number of Public Methods
NS	Number of Setters
TLLOC	Total Logical Lines of Code
TLOC	Total Lines of Code
TNA	Total Number of Attributes
TNG	Total Number of Getters
TNLA	Total Number of Local Attributes
TNLG	Total Number of Local Getters
TNLM	Total Number of Local Methods
TNLPA	Total Number of Local Public Attributes
TNLPM	Total Number of Local Public Methods
TNLS	Total Number of Local Setters
TNM	Total Number of Methods
TNOS	Total Number of Statements
TNPA	Total Number of Public Attributes
TNPM	Total Number of Public Methods
TNS	Total Number of Setters

Gil et. al in [13] introduced micro-patterns by defining 27 patterns that can mechanically describe a class by expressing formal conditions. The catalog is shown in Table 2.2. The researchers also organized the patterns into eight different categories that are related to degenerate classes, containment, and inheritance. Their studies show that the majority of Java classes that were included in the experiments followed at least one of the micro-patterns defined in the catalog. The authors suggested that a combination of micro-patterns and nano-patterns will help aid developers during the development phase.

Table 2.2: The micro-patterns catalog defined by Gil and Maman [13].

Category	Pattern	Description
Degenerate Class	Designator	An interface with absolutely no members.
	Taxonomy	An empty interface extending another interface.
	Joiner	An empty interface joining two or more superinterfaces.
	Pool	A class which declares only static final fields, but no methods.
	Function Pointer	A class with a single public instance method, but with no fields.
	Function Object	A class with a single public instance method, and at least one instance field.
	Cobol Like	A class with a single static method, but no instance members.
	Stateless	A class with no fields, other than static final ones.
	Common State	A class in which all fields are static.
	Immutable	A class with several instance fields, which are assigned exactly once, during instance construction.
	Restricted Creation	A class with no public constructors, and at least one static field of the same type as the class.
	Sampler	A class with one or more public constructors, and at least one static field of the same type as the class.
Containment	Box	A class which has exactly one, mutable, instance field.
	Compound Box	A class with exactly one non primitive instance field.
	Canopy	A class with exactly one instance field that it assigned exactly once, during instance construction.
	Record	A class in which all fields are public, no declared methods.
	Data Manager	A class where all methods are either setters or getters.
	Sink	A class whose methods do not propagate calls to any other class.
Inheritance	Outline	A class where at least two methods invoke an abstract method on *this*.
	Trait	An abstract class which has no state.
	State Machine	An interface whose methods accept no parameters.
	Pure Type	A class with only abstract methods, and no static members, and no fields.
	Augmented Type	Only abstract methods and three or more static final fields of the same type.
	Pseudo Class	A class which can be rewritten as an interface: no concrete methods, only static fields.
	Implementor	A concrete class, where all the methods override inherited abstract methods.
	Overrider	A class in which all methods override inherited, non-abstract methods.
	Extender	A class which extends the inherited protocol, without overriding any methods.

Singer et al. in [26] presented a catalog of 17 nano-patterns that are used to characterize Java methods. The categories include patterns related to calling, object-orientation, control flow, and data flow. Table 2.3 lists the nano-patterns defined by the authors. This research demonstrated that nano-patterns enable learning-based techniques to be used to analyze Java applications. The authors also suggested that these patterns can be used by supervised learning techniques.

Table 2.3: Fundamental nano-patterns introduced by Singer et. al. in [26]

Category	Name	Description
Calling	NoParams	Takes no arguments.
	NoReturn	Returns void.
	Recursive	Calls itself recursively.
	SameName	Calls another method with the same name.
	Leaf	Does not issue any method calls.
Object-Orientation	ObjectCreator	Creates new objects.
	FieldReader	Reads (static or instance) field values from an object.
	FieldWriter	Writes values to (static or instance) field of an object.
	TypeManipulator	Uses type casts or instanceof operations.
Control Flow	StraightLine	No branches in method body.
	Looping	One or more control flow loops in method body.
	Exceptions	May throw an unhandled exception.
Data Flow	LocalReader	Reads values of local variables on stack frame.
	LocalWriter	Writes values of local variables on stack frame.
	ArrayCreator	Creates a new array.
	ArrayReader	Reads values from an array.
	ArrayWriter	Writes values to an array.

Research has been conducted to show how traceable code patterns can be used to identify software defects. Codabux et al. in [7] conducted a study to show that there is a correlation between traceable code patterns and code smells. The results indicate that developers can understand what traceable patterns are most likely to lead to code smells. The authors suggested that traceable code patterns can be used for prediction models. Sultana et al. in [30] utilized traditional machine learning algorithms to identify vulnerable methods with the use of nano-patterns. The results show that nano-patterns produced lower false positive rates and a higher recall compared to traditional software metrics. Sultana in [28]extended her research to develop a prediction model using traceable code patterns and software metrics. The result of the study indicated that pattern-based prediction models produce relatively better results in terms of recall and should continue to be used by newer machine learning techniques. This leads to the need for extended research that utilizes traceable code patterns and machine learning.

2.3 Machine Learning Techniques

Machine learning algorithms have been introduced in conjunction with software metrics for measuring quality, bugs, and vulnerabilities. Research conducted by Alenezi et al. in [1] used these metrics to evaluate machine learning algorithms that were used to predict vulnerabilities in PHP files. The study evaluated the performance of classification algorithms such as Naïve Bayes, Decision Tree, and Random Forests. The most influential metrics were extracted by utilizing the gain ration, which normalizes the measure of each feature to classification. The results indicate that software metrics can be used for vulnerability prediction. Research conducted by Alves et al. in [2] evaluated the machine learning algorithms Bayesian Network, Decision Tree, Naïve Bayes, Random Forest, and Logistic Regression. Machine learning algorithms give an unbiased prediction based on datasets and historical data. These algorithms were used on open source software such as Mozilla Firefox, Linux Kernel, Xen Hypervisor, Httpd and Glibc to predict vulnerabilities. The research yielded the best results from the Decision Tree algorithm for Precision and

Logistic Regression for Recall. Machine learning algorithms proved to be promising in regard to predicting vulnerabilities. Improving the use of datasets and metrics will give better results for machine learning algorithms.

Deep learning is a subset of machine learning that utilizes the concept of neural networks to enable automated feature extraction. Traditional Machine Learning algorithms require manual feature extraction that may affect classification results. Deep Learning makes use of neural networks by adding one or more hidden layers between the input and output. This enables incremental learning of the features that impact more accurate classification. The question of whether to use Deep learning over Machine learning comes down to a few aspects. Deep learning was introduced with the idea of working with large data sets. Also, deep learning would be suggested if there is a lack of understanding of the features to extract. This concept also proves to perform well with image classification, natural language processing, and speech recognition [16].

Researchers have also investigated the use of deep learning for vulnerability prediction with software metrics and text classifier techniques. Convolutional and feedforward neural networks have shown promise in producing more accurate results in predicting vulnerable code than traditional machine learning techniques [15, 35]. Russel et al. in [23] compared convolutional neural networks (CNN) and recurrent neural networks (RNN) to detect vulnerabilities from source code. Clemente et al. in [6] utilized a multilayer feedforward neural network with a combination of software metrics to predict security bugs in four C++ based applications. The study compared the results of the deep learning approach with traditional machine learning algorithms. The Deep Learning approach was able to outperform the traditional machine learning algorithms in terms of the evaluation metric Accuracy, which is a ration of number of correct predictions to the total number of predictions. To date, micro patterns have not been used in combination with deep learning techniques. Traditional machine learning algorithms produced relatively high results for Recall but other measures such as False Negative rate, Precision, and F-measure were relatively low in terms of predicting vulnerable code [30]. This research will utilize traceable code patterns with deep learning to compare software bug prediction with traditional machine learning algorithms.

CHAPTER 3

METHODOLOGY

This section will discuss the methodology that will be used to conduct the proposed research. This includes a description of our research goal and questions, experimental design, and the methods used to answer each research question.

3.1 Research Goal and Questions

Our research goal is to compare the prediction results of software metrics and traceable pattern by using Deep Learning. The following research questions have been proposed in order to accomplish this goal:

- Research Question 1: *What is the performance of software bug prediction model using micro patterns with a multilayer perceptron?* Performance measures such as false positive rate, precision, recall, and f-measure are used to evaluate the prediction model with micro patterns as features. The results are an evaluation of the multilayer perceptron algorithm compared to traditional machine learning algorithms using micro pattens as predictors of software bugs.
- Research Question 2: *What is the performance of software metrics when utilizing Deep Learning to predict software bugs?* This question will be answered by using a similar process as Research Question 1. In this case, class-level software metrics will be used as predictors for bugs. The performance measures include false positive rate, precision, recall, and f-measure are used to evaluate the prediction model with micro patterns as features.
- Research Question 3: *What is the performance of the software bug prediction model as the training size increases?* The results of these experiments will show how the multilayer perceptron algorithm and the traditional machine learning algorithms perform as the training size increases. The performance measures mentioned in Re-

search Question 1 and Research Question 2 will be used for evaluating each algorithm. This will include using micro patterns and class-level software metrics as features for two separate experiments.

3.2 Software Bug Dataset

Ferenc et al. in [12] produced an open source dataset called the Unified Bug Dataset. This dataset combined the datasets PROMISE, Eclipse Bug Dataset, Bugcatchers Bug Dataset, and GitHub Bug Dataset shown in Table 3.1. Information and source code of various open source Java systems are available for download. The Unified Bug Dataset provides software metrics that were calculated by the OpenStaticAnalyzer [1] static code analyzer on each of the four datasets. The number of reported bugs for each class are also provided in the download. The information for each of the datasets are provided in comma separated values (CSV) files.

Due to the Unified Bug Dataset containing multiple versions of the same project, we only considered the PROMISE and BugPrediction datasets. For the PROMISE dataset, the newest version of the system was only included to prevent duplicate classes. The BugPrediction dataset contained unique projects.

3.3 Micro Patterns Extraction

The micro patterns were extracted from the classes in Table 3.1 with a tool developed by Gil and Maman in [13]. The tool receives a JAR file as input and reports the presence of each micro pattern in Table 2.2 as 100% or 0% (100% if the micro pattern is present, 0% otherwise). A script was written to convert the class files into JAR files. Afterwards,

[1]https://github.com/sed-inf-u-szeged/OpenStaticAnalyzer

Table 3.1: The datasets used from the Public Unified Bug Dataset in [12].

Dataset	Number of Classes	System Name
PROMISE	125	Ant 1.3
	178	Ant 1.4
	293	Ant 1.5
	351	Ant 1.6
	745	Ant 1.7
	339	Camel 1.0
	608	Camel 1.2
	872	Camel 1.4
	965	Camel 1.6
	10	Ckjm 1.8
	6	Forest 0.6
	29	Forest 0.7
	32	Forest 0.8
	241	Ivy 1.4
	352	Ivy 2.0
	272	JEdit 3.2
	306	JEdit 4.0
	312	JEdit 4.1
	367	JEdit 4.2
	492	JEdit 4.3
	135	Log4J 1.0
	109	Log4J 1.1
	205	Log4J 1.2
	195	Lucene 2.0
	247	Lucene 2.2
	340	Lucene 2.4
	26	Pbeans 1
	51	Pbeans 2
	237	Poi 1.5
	314	Poi 2.0
	385	Poi 2.5
	442	Poi 3.0
	157	Synapse 1.0
	222	Synapse 1.1
	256	Synapse 1.2
	196	Velocity 1.4
	214	Velocity 1.5
	229	Velocity 1.6
	723	Xalan 2.4
	803	Xalan 2.5
	885	Xalan 2.6
	909	Xalan 2.7
	440	Xerces 1.2
	453	Xerces 1.3
	588	Xerces 1.4

Dataset	Number of Classes	System Name
Eclipse Bug Dataset	6,729	Eclipse 2.0
	7,888	Eclipse 2.1
	10,593	Eclipse 3.0
Bug Prediction Dataset	997	Eclipse JDT Core 3.4
	1,497	Eclipse PDE UI 3.4.1
	324	Equinox 3.4
	691	Lucene 2.4
	1,862	Mylyn 3.1
Bugcatchers Bug Dataset	191	Apache Commons
	1,582	ArgoUML 0.26 Beta
	560	Eclipse JDT Core 3.1
GitHub Bug Dataset	73	Android U.I.L. 1.7.0
	479	ANTLR v4 4.2
	5,908	Elasticsearch 0.90.11
	731	jUnit 4.9
	331	MapDB 0.9.6
	301	mcMMO 1.4.06
	1,887	MCT 1.7b1
	5,899	Neo4j 1.9.7
	1,143	Netty 3.6.3
	1,847	OrientDB 1.6.2
	533	Oryx
	1,468	Titan 0.5.1
	1,610	Eclipse p. for Ceylon 1.1.0
	3,412	Hazelcast 3.3
	1,593	Broadleaf C. 3.0.10
	63	Android U.I.L. 1.7.0
	411	ANTLR v4 4.2
	3,035	Elasticsearch 0.90.11
	308	jUnit 4.9
	137	MapDB 0.9.6
	267	mcMMO 1.4.06
	413	MCT 1.7b1
	3,278	Neo4j 1.9.7
	913	Netty 3.6.3
	1,503	OrientDB 1.6.2
	443	Oryx
	981	Titan 0.5.1
	699	Ceylon for Eclipse 1.1.0
	2,228	Hazelcast 3.3
	1,843	Broadleaf C. 3.0.10

a script was written to dump each JAR file into the micro patterns detection tool. The file name and the information for each micro pattern was stored in a separate CSV file for each dataset. The micro pattern instances were then converted into binary numbers (1 if the micro patten is present, 0 otherwise). After obtaining the micro pattern instances for the classes, a script was written to find the file name in the spreadsheets provided by the Unified Bug Dataset and include the number of reported bugs. A script was then written to convert the number of bugs into true/false values (true if the micro patten is present, false otherwise). The final result of the spreadsheets for the databases included the micro pattern instances and the true/false values for each class.

3.4 Software Bug Prediction

This research utilized two different sets of features, micro patterns and class-level metrics, from Java classes. The labeled data from the classes were collected and marked as true if the class contained reported bugs and false otherwise. The collected information was then fed into a prediction model and trained so that the selected algorithm can make predictions. Supervised learning was used due to the availability labeled data. In our case, the prediction model used a set of features (micro patterns or class-level software metrics) to identify patterns for bug occurrences in class files. Four algorithms were applied for supervised learning for bug prediction (Multilayer Perceptron, Naïve Bayes, Support Vector Machine, and Random Forest) because of their use in earlier studies in bug prediction [19]. The Multilayer Perceptron algorithm served as the Deep Learning approach where a neural network with one hidden layer is used for classification. The other algorithms (Naïve Bayes, Support Vector Machine, and Random Forest) were selected as the traditional machine learning algorithms, where they have been used in previous studies to predict software bugs [6, 27]. Although traditional machine learning algorithms have been used to predict software bugs with micro patterns, Deep Learning has not yet been used with these techniques [31].

The software tool used for this research is Waikato Environment for Knowledge Analysis (WEKA), a widely used data mining and machine learning tool written in Java. This software provides many algorithms for data analysis and predictive modeling. WEKA 3.8.4 [2] was used in our research. The parameters for the traditional machine learning algorithms (Naïve Bayes, Support Vector Machine, and Random Forest) were initialized with the default settings for WEKA. The Multilayer Perceptron algorithm was initialized to train for 10,000 episodes with a learning rate of 0.3 because these values produced the best results. The dataset used in our research was not balanced between the classes that contained bugs and the neutral classes. The ClassBalancer filter provided by WEKA 3.8.4 [3] was used to generate an even number of neutral classes with the classes that contain bugs. This is made possible by the re-weighting of the instances in the training data. For the experiments related to Research Question 1 and Research Question 2, 10-fold cross-validation was used for the prediction model. For Research Question 3, selected files were used for training data and test data. This process is based on the total number of software projects contained in the dataset. There is a counter that iterates until it reaches the number of projects contained in the dataset. During the current value of the counter, that number

[2]https://www.cs.waikato.ac.nz/ml/weka/

[3]https://weka.sourceforge.io/doc.dev/weka/filters/supervised/instance/ClassBalancer.html

of projects is selected randomly for the training data and the remaining are used as test data files. This process will show how well each algorithm performs as the training data increases.

3.5 Performance Measures

Performance measures are needed in this research to evaluate how well each algorithm predicts software bugs. The measures are as follows:

- **False Positive (FP) Rate:** The FP rate indicates the percentage of incorrect predictions in a class. This performance measure will show the rate at which an algorithm incorrectly predicts whether or not a software bug is present [6, 30].
- **Precision:** Precision is the number of correct predictions in a a class (true or false) over the total number of instances of each class. This will show how correct each algorithm performs with predicting software bugs [6, 30].
- **Recall:** Recall is the number of correct correct predictions in that class over the total number of instances in that class. This will indicate the likelihood that an algorithm will not incorrectly make a prediction for a class [6, 30].
- **F-Measure:** The F-Measure is a harmonic mean of the Precision and the Recall performance measures. This measure is important because it gives an equal importance between Precision and Recall [6, 30].

Weka provides each of these performance measures when classifying the provided data. The predictions are made for all instances from the provided data. In our case, predictions are made for whether a class contains a software bug or not. These measures are provided for each of these instances, in our case true and false. In other words, Weka provides the performance measures for predicting classes that contain software bugs and also classes that do not contain software bugs. The results that are shown in this research is a weighted average between the prediction of both instances.

CHAPTER 4

ANALYSIS AND RESULTS

This chapter explains the results from experiments related to the research questions stated in Chapter 1.4. The performance of the algorithms for each experiments are evaluated with the measures False Positive (FP), Precision, Recall, and F-Measure. The Bug Prediction Dataset and the PROMISE Dataset are used because all of the class-level software metrics are provided for those datasets.

4.1 Data Collection

The Micro Patterns Detector tool extracted the micro patterns from the PROMISE and Bug Prediction datasets provided by the Public Unified Bug Dataset. The datasets were selected due to the availability of all of the class-level software metrics to compare with the micro patterns results. For the PROMISE dataset, the newest version of the system was only included to prevent duplicate classes.

4.1.1 Research Question 1: What is the performance of the software bug prediction model using micro patterns as features with a multilayer perceptron?

The ClassBalancer filter provided by Weka was used to generate an even number of neutral classes with the classes that contain bugs. For each algorithm, 10-fold cross-validation was used for PROMISE, Bug Prediction, and the combination of the two datasets

Table 4.1: The datasets used from the Public Unified Bug Dataset in [12].

Dataset	Number of Classes	System Name
PROMISE	734	Ant 1.7
	32	Forest 0.8
	481	JEdit 4.3
	186	Log4J 1.2
	334	Lucene 2.4
	51	Pbeans 2
	308	Poi 2.5
	536	Xerces 1.4
Bug Prediction Dataset	996	Eclipse JDT Core 3.4
	993	Eclipse PDE UI 3.4.1
	243	Equinox 3.4
	697	Lucene 2.4
	834	Mylyn 3.1

shown in Table 4.1. Figure 4.1 show that the Multilayer Perceptron algorithm outperformed the traditional machine learning algorithms in terms of F-Measure, where this is a weighted harmonic mean of the Precision and Recall measures.

The Bug Prediction and PROMISE datasets were combined and evaluated in the same manner. The results shown in Figure 4.2 show that the Multilayer Perceptron algorithm performed the best in terms of F-Measure. The combination of the two datasets are consistent with the previous results in that the Multilayer Perceptron algorithm performs the best with micro patterns.

4.1.2 Research Question 2: What is the performance of the software bug prediction model when using class-level software metrics as features with a multilayer perceptron?

The experiment that is related to this research question has the same approach as Research Question 1. The ClassBalancer filter was used to generate an even number of neutral classes and classes that contain bugs. Also, 10-fold cross-validation was used for the

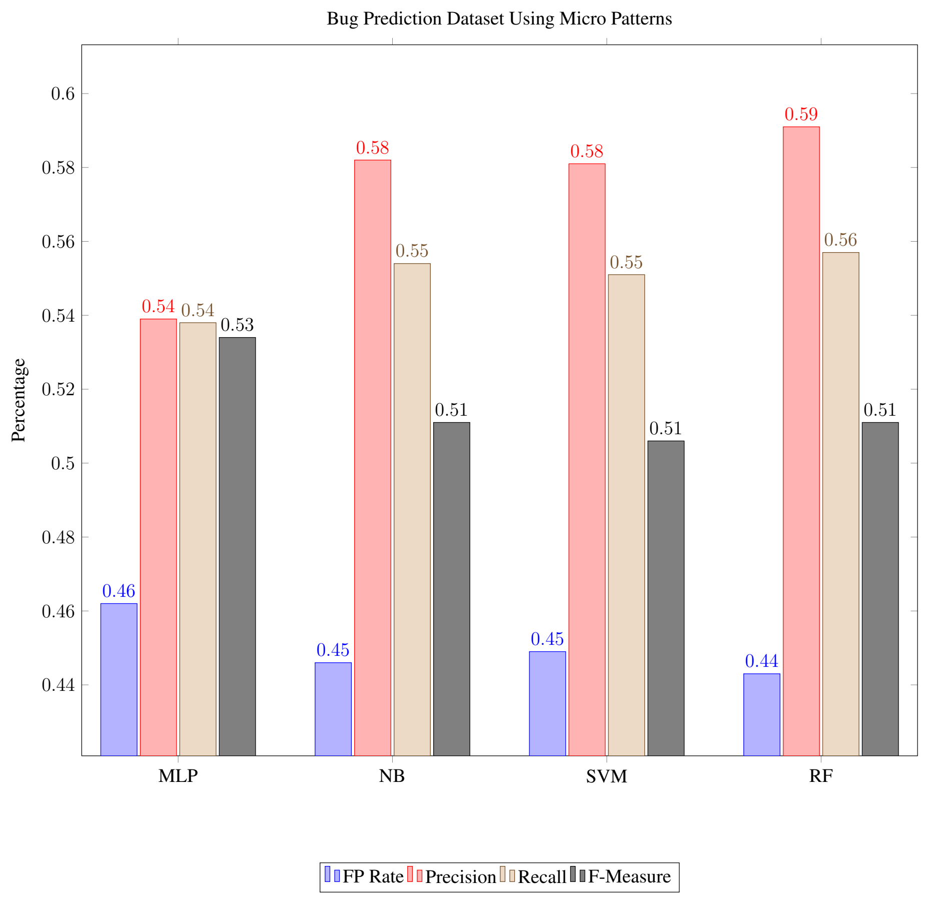

Figure 4.1: The results from the Bug Prediction dataset using micro patterns.

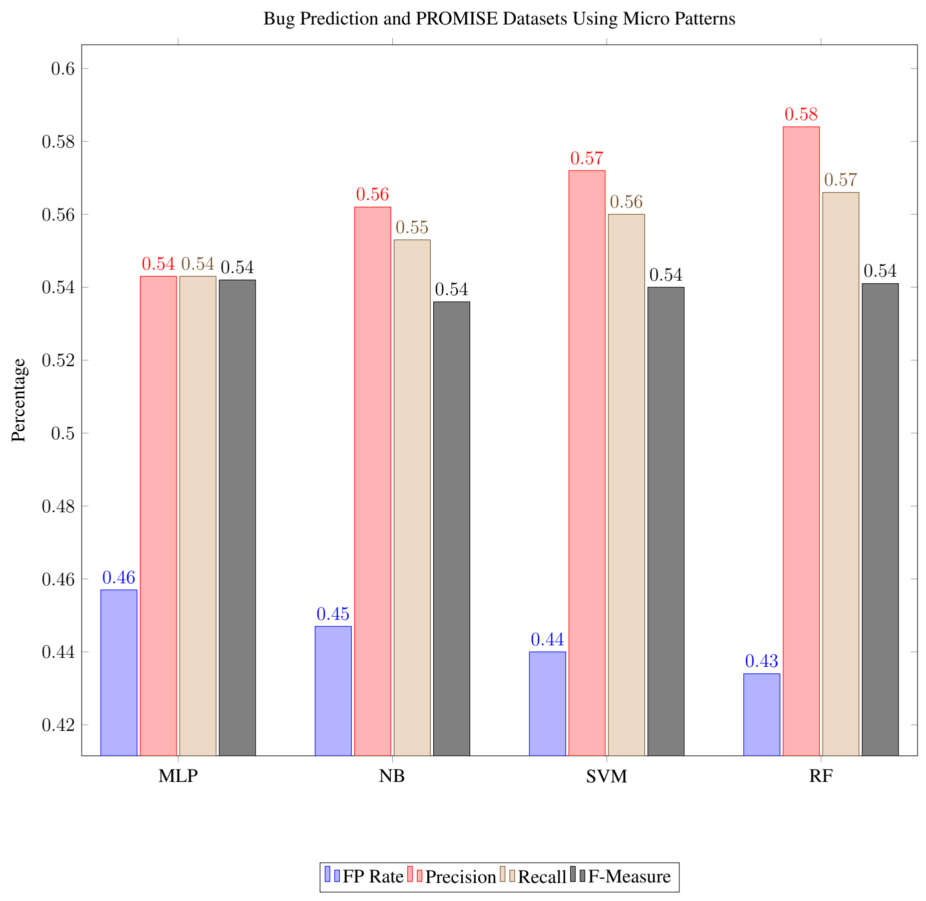

Figure 4.2: The results from the Bug Prediction and PROMISE datasets using micro patterns

Bug Prediction, PROMISE, and the combination of the two datasets shown in Table 4.1. Class-level software metrics are used as features in the model to predict software bugs. The results are shown in Figure 4.3 and Figure 4.4 for the FN, Precision, Recall, and F-Measure for each algorithm. The results from Figure 4.3 show that the Random Forest algorithm outperformed the other algorithms in all of the performance measures when using the Bug Prediction dataset.

There were similar results when the Bug Prediction and PROMISE datasets were combined with the class-level software metrics. Figure 4.4 show that the Random Forest algorithm also performed the best compared to the other algorithms used in the prediction model with the combination of the Bug Prediction and PROMISE datasets.

4.1.3 Research Question 3: What is the performance of the software bug prediction model as the training size increases?

In this experiment, the idea is to show how the algorithms perform as the training size increases. Training files of the individual projects are randomly selected for each iteration, where the averages at each stage are calculated. For example, the maximum number of iterations is one less than the number of projects in the dataset. The first iteration randomly selects a project as the training data, and uses the remaining projects as test data. The remaining iterations repeat this process, but increase the selected projects by one for each iteration until the maximum number of iterations is met. The average for the performance measures are calculated for each iteration. This process was repeated for the number of projects contained in the dataset.

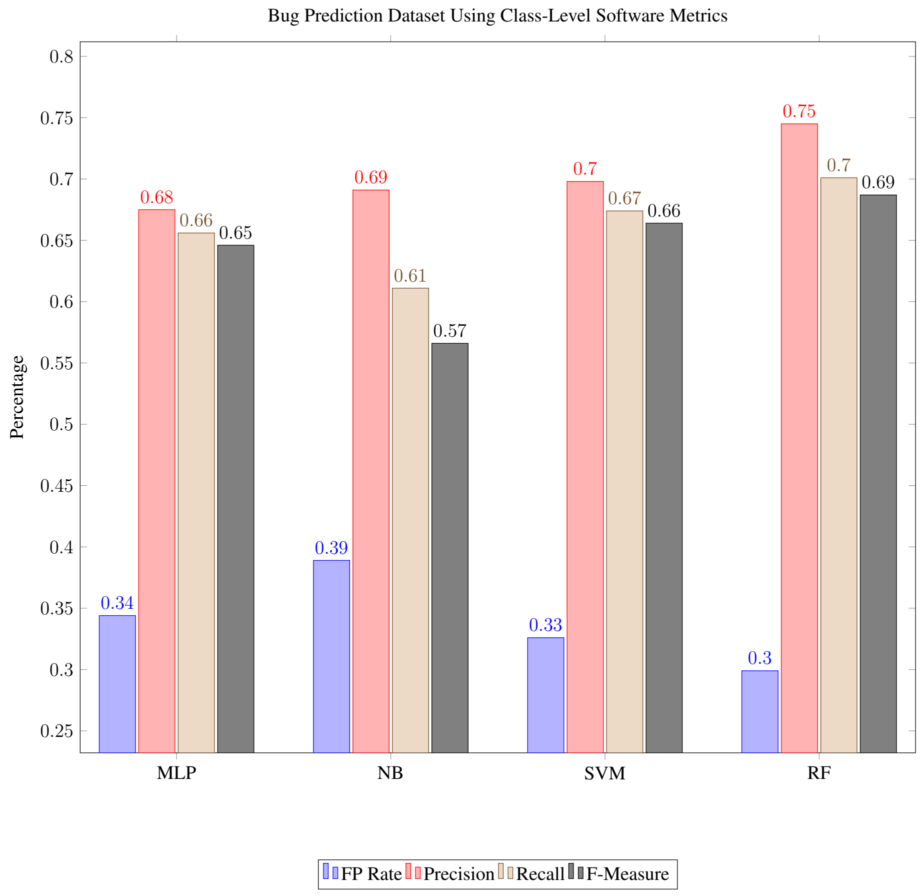

Figure 4.3: The results from the Bug Prediction datasets using software metrics.

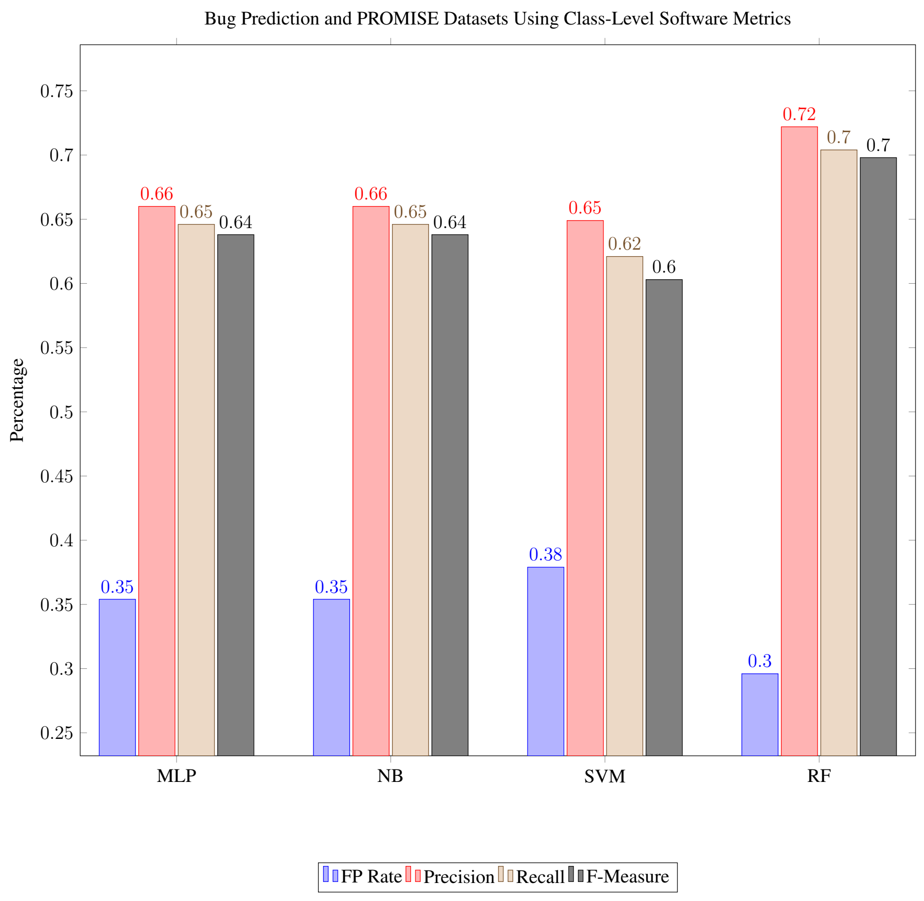

Figure 4.4: The results from the Bug Prediction and Promise datasets using software metrics.

The results of this experiment are shown in Figures 4.5, 4.6, 4.7, and 4.8 when micro patterns are used for bug predictions. Figures 4.5 and 4.6 show that the Naïve Bayes algorithm produced the lowest False Positive Rate and the highest Precision compared to the other algorithms. Figures 4.7 and 4.8 show that the Multilayer Perceptron outperforms the other algorithms in Recall and F-Measure as the training size increases, which is consistent with the previous findings for Research Question 1.

The same experimental method was used, but with software metrics. Figure 4.9 show that the Multilayer Perceptron algorithm had the highest FP Rate compared to the other algorithms as the training size increases. The results from Figures 4.10, 4.11, and 4.12 show that the Random Forest algorithm outperforms the other algorithms in Precision, Recall, and F-Measure. This is also consistent with the results from Research Question 2.

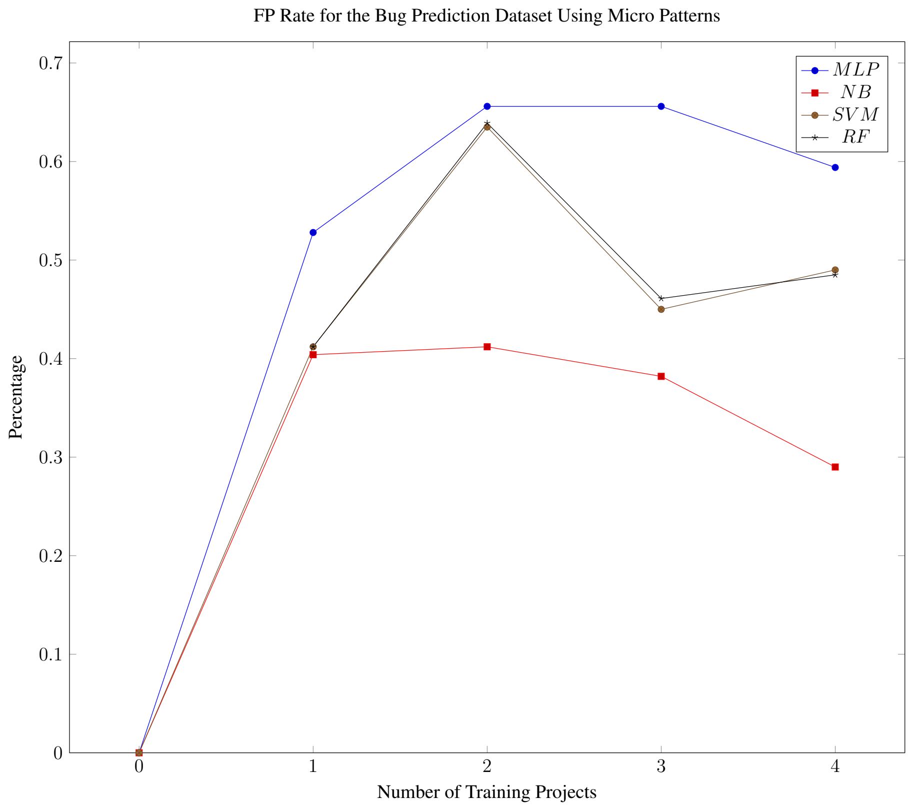

Figure 4.5: The False Positive Rate results from the Bug Prediction datasets using micro patterns as the training size increases.

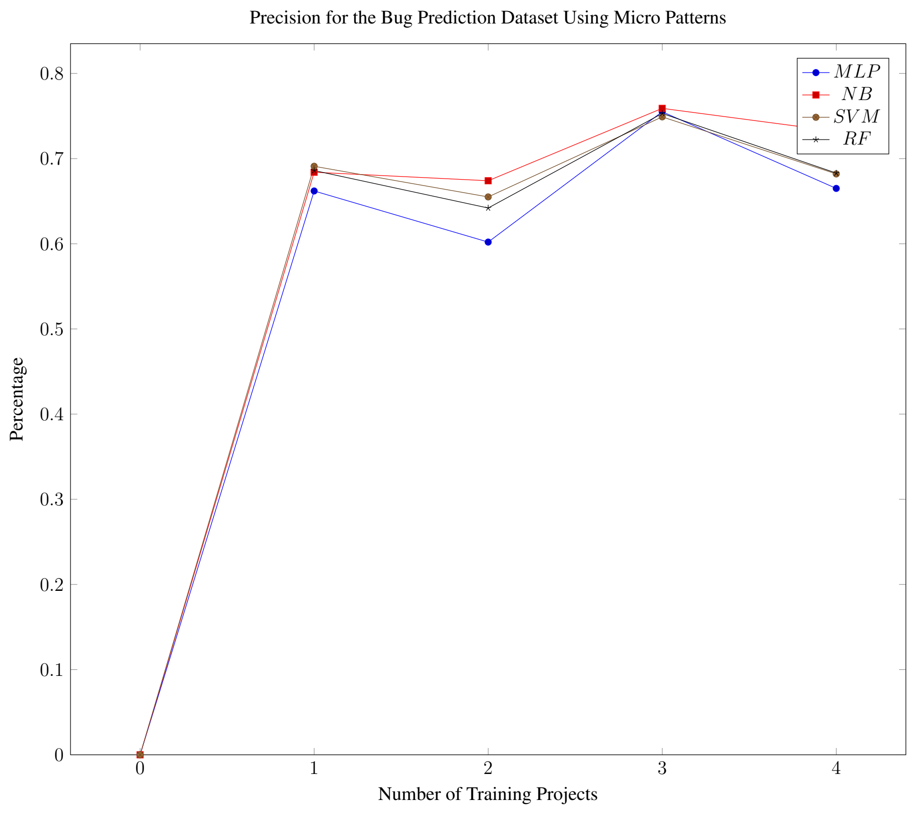

Figure 4.6: The Precision results from the Bug Prediction datasets using micro patterns as the training size increases.

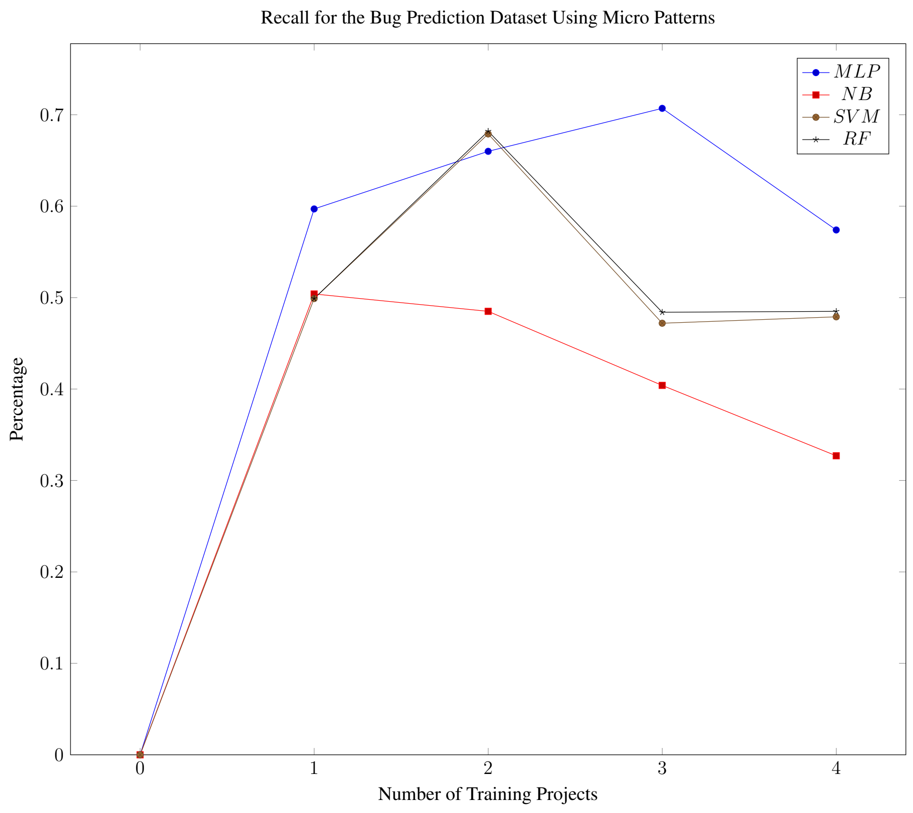

Figure 4.7: The Recall results from the Bug Prediction datasets using micro patterns as the training size increases.

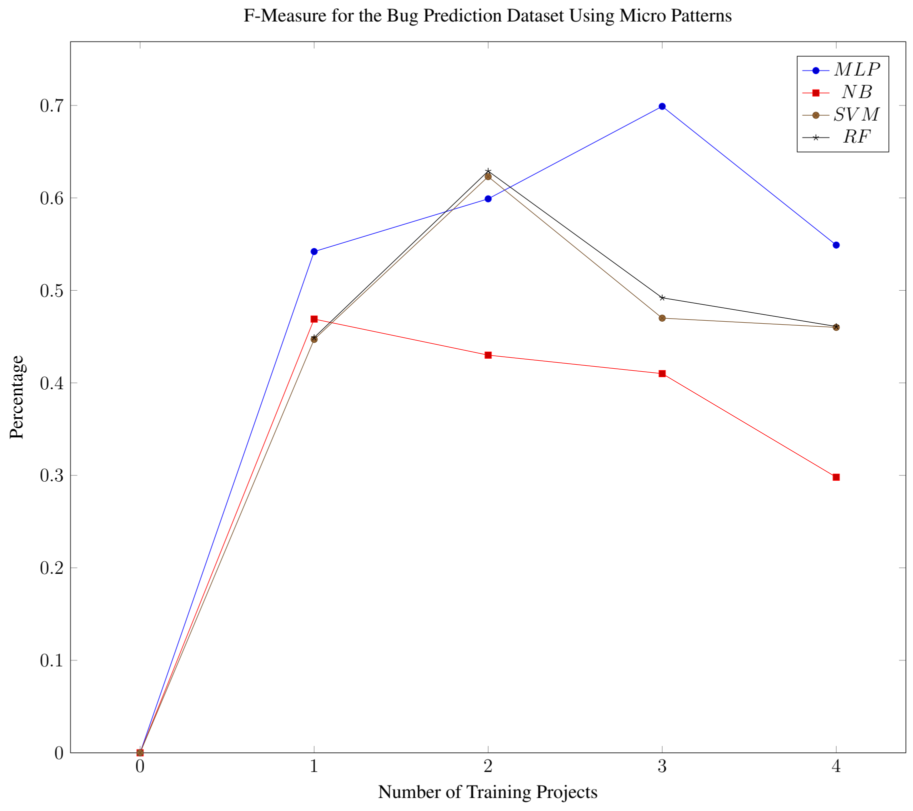

Figure 4.8: The F-Measure results from the Bug Prediction datasets using micro patterns as the training size increases.

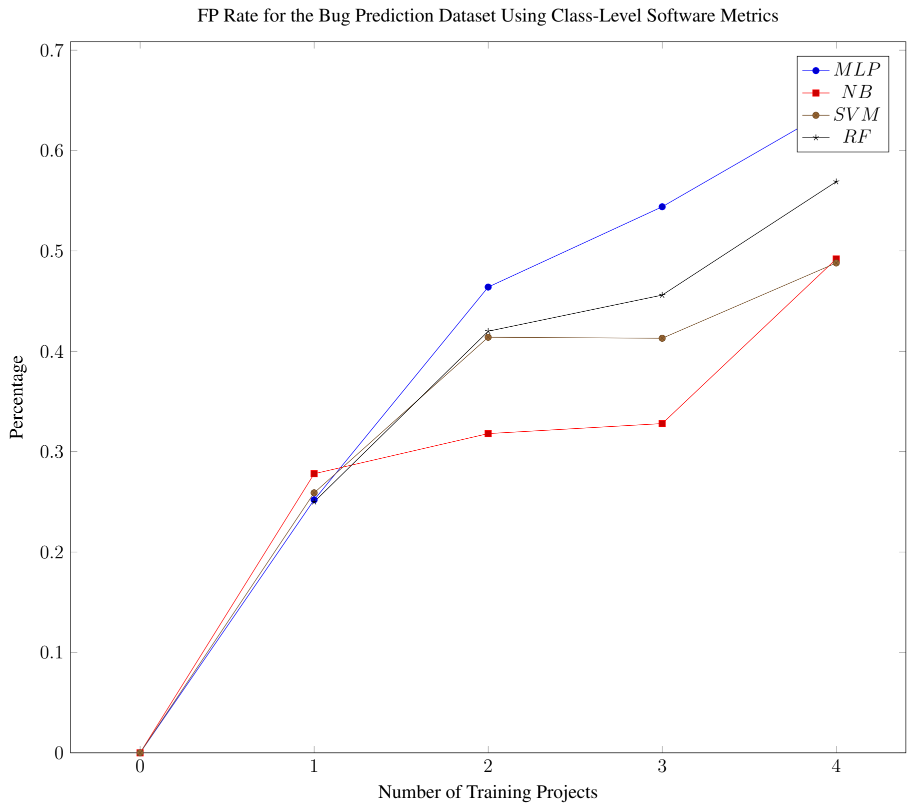

Figure 4.9: FP Rate for the Bug Prediction Dataset Using Class-Level Software Metrics.

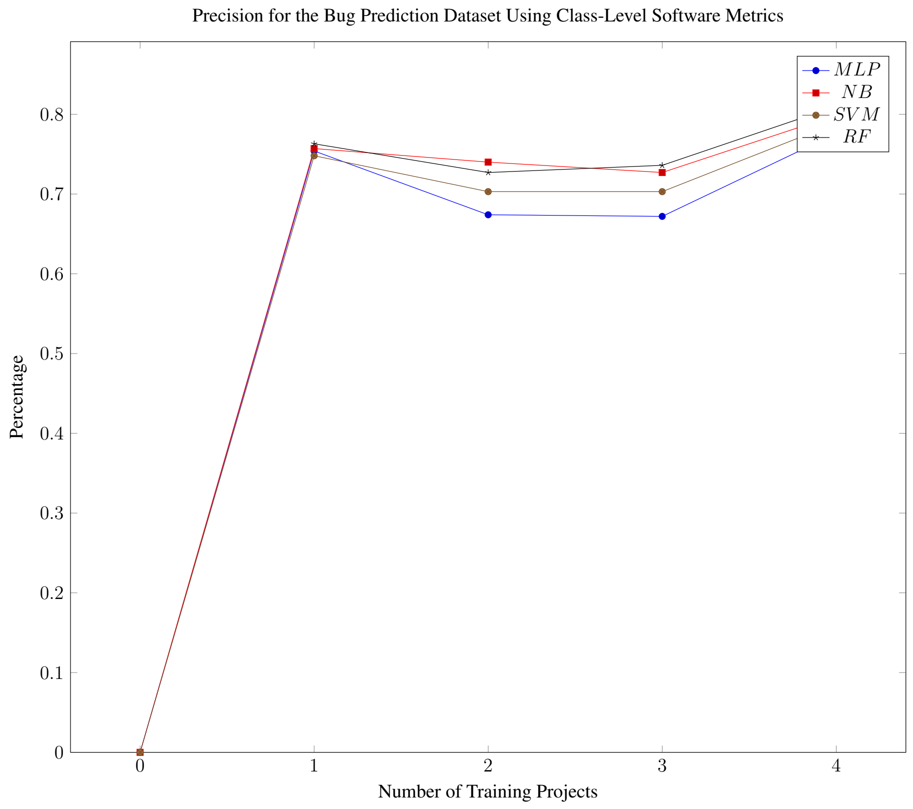

Figure 4.10: Precision for the Bug Prediction Dataset Using Class-Level Software Metrics.

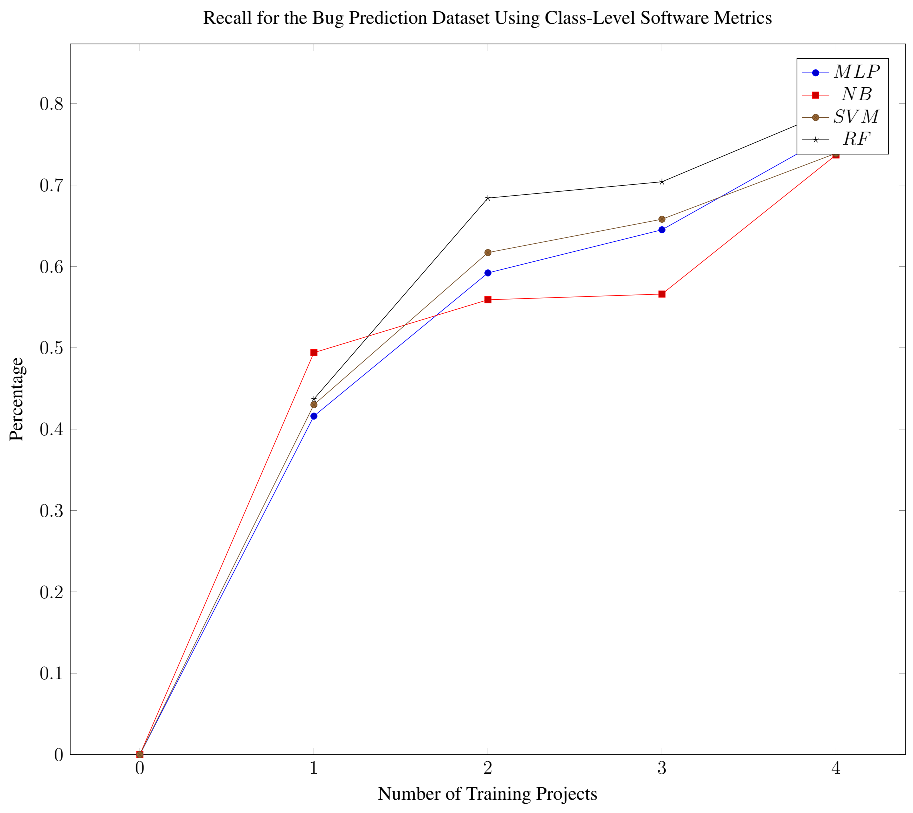

Figure 4.11: Recall for the Bug Prediction Dataset Using Class-Level Software Metrics.

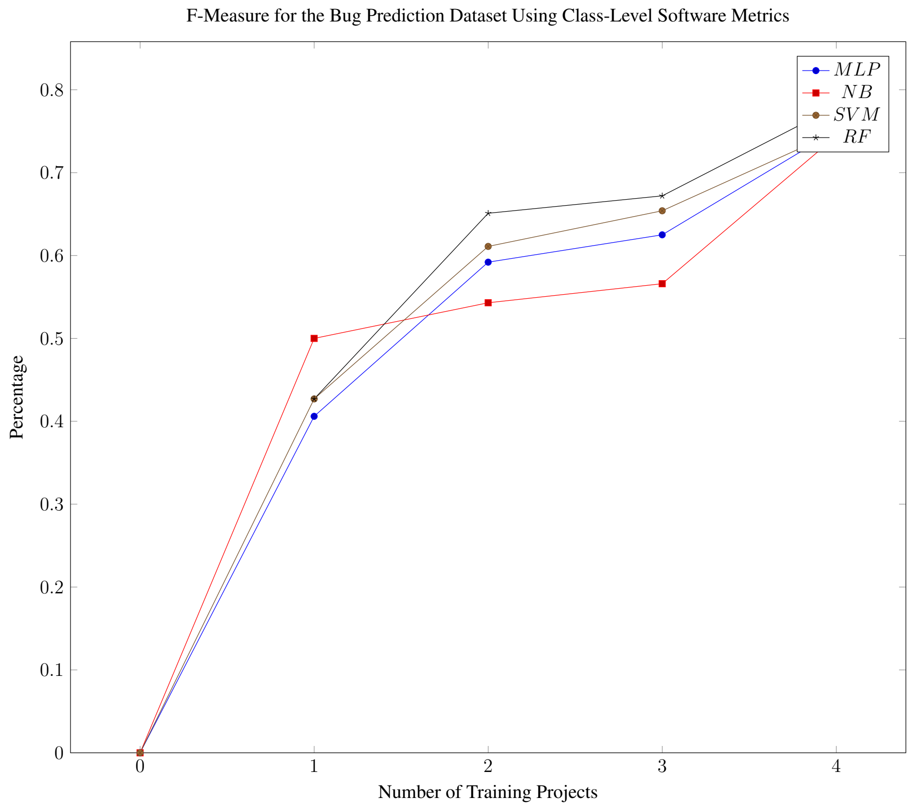

Figure 4.12: The F-Measure results from the Bug Prediction datasets using class-level software metrics as the training size increases.

CHAPTER 5

DISCUSSION

In this section, we interpret and explain the significance of our findings from the results from Chapter 4. The use of micro patterns gives software developers another technique to evaluate the quality of their code. This concept also mechanically describes the structure of programs written in Java at the class-level. Deep Learning has been used with software metrics to predict bugs. With this idea, micro patterns are used in this research to achieve the same goal.

The results in Figure 4.2 show that the Deep Learning approach by using the Multilayer Perception algorithm has a higher Recall and F-Measure than the traditional machine learning algorithms. This means that the algorithm has a higher harmonic mean between the Precision and Recall measures. Even with the higher value of False Positive Rate, the Multilayer Perceptron algorithm has a lower number of undetected classes with bugs because of the Recall value. The Bug Prediction Dataset can be predicted with a higher overall Recall and F-Measure when using the Multilayer Perceptron algorithm. The same can be concluded with the combination of the Bug Prediction Dataset and the PROMISE dataset in Figure 4.3.

In Figure 4.4, the same model was applied but with class-level software metrics instead. The Random Forest algorithm outperformed the other algorithms in terms of Precision, Recall, and F-Measure. Also, the False Positive Rate was the lowest compared the the other algorithms. It can be concluded that the Random Forest algorithm is able to predict software bugs for the Bug Prediction Dataset and the PROMISE dataset better than the other algorithms.

We conducted an experiment to show the performance of the algorithms as the training size increases. For the Bug Prediction Dataset, the Multilayer Perceptron algorithm had higher performances in terms of Recall and F-Measure compared to the other algorithms. This is consistent with the results in Figure 4.2, where the algorithm also outperformed the other algorithms. The results for the software metrics were also consistent with the results in Figure 4.3, where the Random Forest algorithm performed the best.

Existing literature with Deep Learning and software bug prediction shows promising results. Clemente et al. [6] utilized a feedforward neural network to predict software bugs by using software metrics. The Deep Learning method performed better than the traditional machine learning algorithms in terms of accuracy. In this study, more classes will be needed to fully utilize the Deep Learning approach. Studies by Sultana et al. [27] used micro patterns and nano-patterns to predict vulnerable code with traditional machine learning algorithms. Deep Learning and the use of traceable code patterns will provide another avenue to predict software bugs.

CHAPTER 6

THREATS TO VALIDITY

This section will discuss threats to validity in terms of construct, external, and internal. Construct validity involves what a tests claims to measure. External validity involves the ability that a test can generalize the results. Internal validity is any unwanted or unanticipated results.

Construct validity: Micro patterns are able to describe Java code at the byte code level, but they may not capture all characteristics of classes. This is another method that software developer can utilize to measure software quality. This study also utilizes the Public Unified Bug Dataset, where the neutral classes may have unreported software bugs.

External validity: Deep Learning works well with large amount of data. In this study, there is a limited amount of data to measure the performance of the Multilayer Perceptron algorithm. More classes are needed to fully utilize the performance of Deep Learning. Micro patterns are only used for class-level Java projects. It cannot be concluded that micro patterns perform better than class-level metrics since this method is only used for programs written in Java. Also, more research is needed for different Java frameworks.

Internal validity: This study evaluates the ability of predicting software bugs with micro patterns and class-level metrics. It cannot be concluded that micro patterns and metrics cause software bugs. The relation between software bugs and these methods are studied to provide software developers a way to measure software quality during the development process. The classes that are predicted for bugs need rigorous testing compared to the predicted neural classes.

CHAPTER 7

CONCLUSION AND FUTURE WORK

This research will help software developers identify vulnerable code at the class-level to pinpoint the specific code locations to perform rigorous tests. Traditional machine learning algorithms have proved to be able to predict vulnerable code, but the high rate of false negatives have been the biggest challenge. Deep learning techniques have recently been investigated to produce better results in predicting vulnerable code. So far, traceable code patterns have not been investigated along with Deep learning techniques to identify vulnerable classes and functions. Software developers would be able to better locate specific locations of vulnerable code if deep learning techniques preform better than traditional machine learning algorithms.

In the future, this work can be extended by venturing into other datasets to provide the prediction model with more data to improve the performance of Deep Learning approach. This work can also be extended by predicting software bugs into categories. The use of nano-patterns to predict software bugs would be another way to predict software bugs. We did not utilize nano-patterns because the dataset that we utilized contained class-level information rather than method-level. The use of feature selection techniques would determine the most relevant features of the prediction model to improve results and reduce complexity. For this research, we have provided a foundation of utilizing traceable code patterns to predict software bugs with Deep Learning.

REFERENCES

[1] M. Alenezi and I. Abunadi. Evaluating software metrics as predictors of software vulnerabilities. *International Journal of Security and Its Applications*, 9(10):231–240, 2015.

[2] H. Alves, B. Fonseca, and N. Antunes. Experimenting machine learning techniques to predict vulnerabilities. In *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*, pages 151–156, 2016.

[3] S. Broggi. Bug prediction with neural nets. B.S. Thesis, 2018.

[4] I. Chowdhury, B. Chan, and M. Zulkernine. Security metrics for source code structures. *Proceedings of the Fourth International Workshop on Software Engineering for Secure Systems*, 2008.

[5] I. Chowdhury and M. Zulkernine. Using complexity coupling and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, 2011.

[6] C. Clemente, F. Jaafar, and Y. Malik. Is predicting software security bugs using deep learning better than the traditional machine learning algorithms? *2018 IEEE International Conference on Software Quality Reliability and Security (QRS)*, July 2018.

[7] Z. Codabux, K. Sultana, and B. Williams. The relationship between code smells and traceable pattern - are they measuring the same thing. *International Journal of Software Engineering and Knowledge Engineering*, 27(9-10):1529–1547, 2017.

[8] M. D'Ambros, M. Lanza, and R. Robbes. An extensive comparison of bug prediction approaches. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 31–41. IEEE, 2010.

[9] K. Emam, W. Melo, and J. Machado. The prediction of faulty classes using object-oriented design metrics. *Journal of systems and software*, 56(1):63–75, 2001.

[10] M. Fagan. Design and code inspections to reduce errors in program development. In *Software pioneers*, pages 575–607. Springer, 2002.

[11] R. Ferenc, D. Bán, T. Grósz, and T. Gyimóthy. Deep learning in static, metric-based bug prediction. *Array*, page 100021, 2020.

[12] R. Ferenc, Z. Toth, G. Ladanyi, and I. Siket. A public unified bug dataset for java. *In Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2018.

[13] J. Gil and I. Maman. Micro patterns in java code. *OOPSLA '05: Proceedings of 20th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 97–116, 2005.

[14] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.

[15] J. Li, P. He, J. Zhu, and R. L. Michael. Software defect prediction via convolutional neural network. *QRS' 17: Proc. of the International Conference on Software Quality Reliability and Security*, pages 318–328, 2017.

[16] S. Mahapatra. Why deep learning over traditional machine learning? *https://towardsdatascience.com/why-deep-learning-is-needed-over-traditional-machine-learning-1b6a99177063*, March 21 2018.

[17] R. Malhorta, L. Bahl, S. Sehgal, and O. Priya. Empirical comparision of machine learning algorithms for bug prediction in open source software. *International Conference on Big Data Analytics and Computational Intelligence (ICBDAC)*, pages 40–45, 2017.

[18] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[19] P. Morrison, K. Herzig, B. Murphy, and L. Williams. Challenges with applying vulnerability prediction models. *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security. ACM-Association for Computing Machinery*, 2015.

[20] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. *Proc. ACM/IEEE 30th Int'l Conf. Software Eng.*, pages 181–190, 2008.

[21] S. Moshtari, A. Sami, and M. Azimi. Using complexity metrics to improve software security. *Computer Fraud and Security*, 2013(5):8–17, 2013.

[22] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. *Proc. 27th Int'l Conf. Software Eng.*, pages 284–292, 2005.

[23] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozemir, P. Ellingwood, and M. McConley. Automated vulnerability detection in source code using deep representation learning. *Proc. ICMLA*, pages 757–762, 2018.

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

[25] F. Shull, V. Basili, B. Boehm, A. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz. "what we have learned about fighting defects". *Proc. Eighth Int'l Software Metrics Symp.*, pages 249–258, 2002.

[26] J. Singer, G. Brown, M. Luján, A. Pocock, and P. Yiapanis. Fundamental nano-patterns to characterize and classify java methods. *Electronic Notes in Theoretical Computer Science*, 253(7):191–204, 2010.

[27] K. Sultana, B. Williams, and T. Bhowmik. A study examining relationships between micro patterns and security vulnerabilities. *Software Quality Journal*, 2017.

[28] K. Z. Sultana. Towards a software vulnerability prediction model using traceable code patterns and software metrics. *SE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 1022–1025, 2017.

[29] K. Z. Sultana, A. Deo, and B. J. Williams. Correlation analysis among java nano-patterns and software vulnerabilities. *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*, pages 69–76, 2017.

[30] K. Z. Sultana, B. Williams, and A. Bosu. A comparison of nano-patterns vs. software metrics in vulnerability prediction. *2018 25th Asia-Pacific Software Engineering Conference (APSEC),*, pages 355–364, 2018.

[31] K. Z. Sultana and B. J. Williams. Evaluating micro patterns and software metrics in vulnerability prediction. *2017 6th International Workshop on Software Mining (SoftwareMining)*, pages 40–47, 2017.

[32] J. Walden, J. Stuckman, and R. Scandariato. Predicting vulnerable components: Software metrics vs text mining. *2014 IEEE 25th International Symposium on Software Reliability Engineering, Naples*, pages 23–33, 2014.

[33] F. Wu, J. Wang, J. Liu, and W. Wang. Vulnerability detection with deep learning. *2017 3rd IEEE International Conference on Computer and Communications (ICCC)*, pages 1298–1302, 2017.

[34] J. Xu, D. Ho, and L. Capretz. An empirical validation of object-oriented design metrics for fault prediction. *Journal of Computer Science*, 4(7):571–577, 2008.

[35] Y. Zhang, D. Lo, X. Xia, B. Xu, J. Sun, and S. Li. Combining software metrics and text features for vulnerable file prediction. *2015 20th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 40–49, 2015.

[36] T. Zimmermann, N. Nagappan, and L. Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. *2010 Third International Conference on Software Testing, Verification and Validation*, pages 421–428, 2010.