CHAPTER 1
INTRODUCTION

INTRODUCTION

1.1 PROBLEM STATEMENT

Software Bug Prediction (SBP) is a critical process in software development and maintenance, that is concerned with the overall success of software. This is because predicting software faults earlier in the development process improves software quality, reliability, efficiency, and reduces software costs. However, creating a robust bug prediction model is a difficult task, and numerous techniques have been proposed in the literature. This paper presents a machine learning (ML)-based software bug prediction model. Based on historical data, three supervised ML algorithms were used to predict future software faults. These classifiers include Nave Bayes (NB), Decision Trees (DT), and Artificial Neural Networks (ANN) (ANNs). The evaluation process demonstrated that ML algorithms can be used effectively and accurately.

1.2 RESEARCH OBJECTIVE

Identifying which Machine Learning (ML) algorithms that have been used in previous studies of SDP is highly important for this master thesis to make sure that it contributes to the research community. [The purpose of this master thesis is to investigate if the prediction performance of metric sets are increased if test metrics are included in the set. By using algorithms used in other studies, [the effects of incorporating test metrics can be directly studied]. The prediction performance of the produced tool can also be made state of the art as lessons from other studies regarding prediction performance can be incorporated.

It is of major importance to recognize what kind of source code metrics that previously have been studied. This makes sure that the results of the study can be compared to those of other studies, and in turn making sure that the thesis contributes to the research community.

1.3 Project Scope and Limitations

Their one-phase model uses only previously fixed files as labels in the training process, and therefore cannot be used to recommend files that have not been fixed before when being presented with a new bug report.

Existing methods require runtime executions.

CHAPTER 2
BACKGROUND WORK

2.1 Feature identification: A novel approach and a case study

In [1] by G. Antoniol and Y. -G. Gueheneuc described:

2.1.1 Introduction

Feature identification is a well-known technique to identify subsets of a program source code activated when exercising a functionality. Several approaches have been proposed to identify features. We present an approach to feature identification and comparison for large object-oriented multi-threaded programs using both static and dynamic data. We use processor emulation, knowledge filtering, and probabilistic ranking to overcome the difficulties of collecting dynamic data, i.e., imprecision and noise. We use model transformations to compare and to visualize identified features. We compare our approach with a naive approach and a concept analysis-based approach using a case study on a real-life large object-oriented multi-threaded program, Mozilla, to show the advantages of our approach. We also use the case study to compare processor emulation with statistical profiling.

In this paper, we describe our approach of building micro-architectures, subsets of a program architecture, from feature identification using static and dynamic data. We assume that the program source code is available and that a compiled version of the program can be exercised under different scenarios. We use data collected from the realisation of a functionality, under different scenarios, to filter static data modelled as class diagrams, thus relating classes with features and with scenarios, and highlighting differences among features. Maintainers can use our approach to build and to compare micro-architectures to locate responsibilities and feature differences precisely. Our approach decomposes in a process and a set of tools to support the process. The process makes use of processor emulation, knowledge filtering, probabilistic ranking, and model transformations to support the analysis of large multi-threaded object-oriented programs and to deal with imprecision and noise.

2.1.2 Literature Survey

Work relates to static and dynamic program analysis, to feature analysis, meta-modelling and model transformation techniques. The following sub-sections describe related work in each category.

Static and Dynamic Analysis: Program instrumentation via source-to-source translation is a common technique to collect dynamic data. For example, Ernst et al., with DAIKON, focus on dynamic techniques for discovering invariants in traces. However, DAIKON does not offer front end for C++. Jeffery and Auguston present the UFO dynamic analysis framework. This framework combines a language and a monitor architecture to ease building new dynamic analysis tools. The main limitations of the UFO framework in the context of our work are the lack of support for C++ and possible implementation and optimization issues. Finally, Ernst discusses synergies and dualities between static and dynamic analyses. Reiss and Renieris present an approach to encode dynamic data, to explore program traces, and languages for dynamic instrumentation respectively. The problem of handling large amounts of data when performing dynamic analysis is discussed in several work, for example Hamou-Lhadj et al., where the authors present an algorithm for identifying patterns in traces of procedure calls.

Feature Identification: Several works propose feature identification techniques. However, few works attempt to compare features with one another. In their precursor work, Wilde and Scully propose a technique to identify features by analyzing execution traces of test cases. They use two sets of test cases to build two execution traces: An execution trace where a functionality is exercised; An execution trace where the functionality is not. Then, they compare execution traces to identify the feature associated with the functionality in the program. In their work, the authors only use dynamic data to identify features, no static analysis of the program is performed.

2.2 Debugadvisor: A recommender system for debugging

In [2] B. Ashok, J. Joy, H. Liang et. al described:

2.2.1 Introduction

Debugging large software is difficult. Large systems have several tens of millions of lines of code. No single person understands all the code in such systems, and often times new hires, who have little or no context about the code, are given the job of debugging failures. Our recent study with Microsoft's Windows Serviceability group revealed that developers and testers spend significant time during diagnosis looking for similar issues that have been resolved in the past. They search through bug databases, articles from the on-line Microsoft Developer Network (known as MSDN), email threads, and talk to colleagues to find this information. For example, the same bug or a very similar bug may have been encountered and fixed in another code branch, and the programmer would greatly benefit from knowing this information. Consequently, we decided to build a recommender system to improve productivity of debugging by automating the search for similar issues from the past. Prior work in mining software repositories such as Hipikat , eRose , and Fran provides us inspiration and very useful ideas. However, there are important challenges in building a recommender system for debugging.

2.2.2 Literature Survey

Mining software repositories and using the computed information to help improve the productivity of programming tasks has been an active area of research for the past several years. Kim, Pan and Whitehead have used historical data from fixed bugs in version control to detect project specific bugs that still exist in the code. The eROSE tool mines software version histories and uses mined information to suggest other places that need to be changed when the programmer attempts to change a method or a class. The Vulture tool correlates information mined from source code modification history, vulnerability reports, and software component structure to provide risk assessments of areas of code that are more likely to be sources of future vulnerabilities. Our work mines software repositories to help programmers and testers during debugging.

The presence of duplicate bugs in software repositories has received much attention. Runesan et al propose using natural language processing techniques, and Jalbert and Weimer propose using textual similarity and clustering techniques to detect duplicate bug reports. Wang et al. propose combining similarity information from both natural language text and execution traces to detect duplicate bugs. Bettenburg et al. propose detecting and merging duplicate bug reports so as to give developers maximum information from various sources to diagnose and fix bugs. Bettenburg et al. present a study on what constitutes a good bug report by conducting surveys with open-source developers and bug reporters.

Interestingly, their paper states that the presence of a stack trace is one of the most important desiderata for a developer to resolve the bug quickly. The mixture of structured and unstructured information in bug reports has been noticed before. Bettenburg et al. have proposed a tool infoZilla to extract structural information from bug reports. The first phase of DebugAdvisor is the next step in this line of research —we allow domain experts to identify structure in bug reports, and propose a transformation that allows such domain specific notions of similarity to be implemented using existing search engines that are structure agnostic. The distinguishing features of our work are twofold:

- We allow our users to query our system using a "fat query" which is a textual concatenation of all the information they have about the issue in hand, including text from emails, text from logs of debug sessions, viewing core dumps in the debugger, etc.

- Our tool architecture and the notion of typed documents enables a clean and systematic way to incorporate domain-specific knowledge into an information retrieval system for handling fat queries. This enables leveraging the domain expert's knowledge about feature similarity while still reusing existing robust and scalable infrastructures for full-text search.

2.3 Expectations, outcomes, and challenges of modern code review

In [3] A. Bacchelli and C. Bird described:

2.3.1 Introduction

Peer code review, a manual inspection of source code by developers other than the author, is recognized as a valuable tool for reducing software defects and improving the quality of software projects. In 1976, Fagan formalized a highly structured process for code reviewing, based on line-by-line group reviews, done in extended meetings—code inspections. Over the years, researchers provided evidence on code inspection's benefits, especially in terms of defect finding, but the cumbersome, time-consuming, and synchronous nature of this approach hinders its universal adoption in practice. Nowadays many organizations are adopting more lightweight code review practices to limit the inefficiencies of inspections. In particular, there is a clear trend toward the usage of tools developed to support code review. In the context of this paper, we define Modern Code Review, as review that is informal (in contrast to Fagan-style), tool-based, and that occurs regularly in practice nowadays, for example at companies such as Microsoft, Google, Facebook, and in other organizations and open-source software (OSS) projects.

Developers and other software project stakeholders can use empirical evidence about expectations and outcomes to make informed decisions about when to use code review and how it should fit into their development process. Researchers can focus their attention on the challenges faced by practitioners to make code review more effective.

2.3.2 Literature Survey

Previous studies have examined the practices of code inspection and code review. Stein et al. conducted a study focusing specifically on distributed, asynchronous code inspections. The study included evaluation of a tool that allowed for identification and sharing of code faults or defects. Participants at separated locations can then discuss faults via the tool. Laitenburger conducted a survey of code inspection methods, and presented a taxonomy of code inspection techniques. Johnson conducted an investigation into code review practices in OSS development and their effect on choices made by software project managers.

Porter et al. reported on a review of studies on code inspection in 1995 that examined the effects of factors such as team size, type of review, and number of sessions on code inspections. They also assessed costs and benefits across a number of studies. These studies differ from ours in that they were not tool-based and were the majority involved planned meetings to discuss the code.

However, prior research also sheds light on why review today is more often tool-based, informal, and often asynchronous. The current state of code review might be due to the time required for more formal inspections. Votta found that 20% of the interval in a “traditional inspection” is wasted due to scheduling. The ICICLE tool, or “Intelligent Code Inspection in a C Language Environment,” was developed after researchers at Bellcore observed how much time and work was expended before and during formal code inspections. Many of todays review tools are based on ideas that originated in ICICLE. Other similar tools have been developed in an effort to reduce time for inspection and allow asynchronous work on reviews. Examples include CAIS and Scrutiny.

CHAPTER 3

RESULTS AND DISCUSSION

3.1 Comparison and Discussion

There are many studies about software bug prediction using machine learning techniques. For example, the study in [6] proposed a linear Auto-Regression (AR) approach to predict the faulty modules. The study predicts the software future faults depending on the historical data of the software accumulated faults. The study also evaluated and compared the AR model and with the Known power model (POWM) used Root Mean Square Error (RMSE) measure. In addition to, the study used three datasets for evaluation and the results were promising.

The studies in [7], [8] analyzed the applicability of various ML methods for fault prediction. Sharma and Chandra [8] added to their study the most important previous researches about each ML techniques and the current trends in software bug prediction using machine learning. This study can be used as ground or step to prepare for future work in software bug prediction. R. Malhotra in [9] presented a good systematic review for software bug prediction techniques, which using Machine Learning (ML). The paper included a review of all the studies between the period of 1991 and 2013, analyzed the ML techniques for software bug prediction models, and assessed their performance, compared between ML and statistic techniques, compared between different ML techniques and summarized the strength and the weakness of the ML techniques.

In [10], the paper provided a benchmark to allow for common and useful comparison between different bug prediction approaches. The study presented a comprehensive comparison between a well-known bug prediction approach, also introduced new approach and evaluated its performance by building a good comparison with other approaches using the presented benchmark. The study in [9] assessed various object-oriented metrics by used machine learning techniques (decision tree and neural networks) and statistical techniques (logical and linear regression). The results of the study showed that the Coupling Between Object (CBO) metric is the best metric to predict the bugs in the class and the Line Of Code (LOC) is fairly well, but the Depth of Inheritance Tree (DIT) and Number Of Children (NOC) are untrusted metrics.

Singh and Chug [9] discussed five popular ML algorithms used for software defect prediction i.e, Artificial Neural Networks (ANNs), Particle Swarm Optimization (PSO), Decision Tree (DT), Naïve Bayes (NB) and Linear Classifiers (LC). The study presented important results including that the ANN has lowest error rate followed by DT, but the linear classifier is better than other algorithms in term of defect prediction accuracy, the most popular methods used in software defect prediction are: DT, BL, ANN, SVM, RBL and EA, and the common metrics used in software defect prediction studies are: Line Of Code (LOC) metrics, object oriented metrics such as cohesion, coupling and inheritance, also other metrics called hybrid metrics which used both object oriented and procedural metrics, furthermore the results showed that most software defect prediction studied used NASA dataset and PROMISE dataset.

Moreover, the studies in [08], [9] discussed various ML techniques and provided the ML capabilities in software defect prediction. The studies assisted the developer to use useful software metrics and suitable data mining technique in order to enhance the software quality. The study in determined the most effective metrics which are useful in defect prediction such as Response for class (ROC), Line of code (LOC) and Lack of Coding Quality (LOCQ). Bavisi et al. presented the most popular data mining technique (k-Nearest Neighbors, Naïve Bayes, C-4.5 and Decision trees). The study analyzed and compared four algorithms and discussed the advantages and disadvantages of each algorithm. The results of the study showed that there were different factors affecting the accuracy of each technique; such as the nature of the problem, the used dataset and its performance matrix.

3.2 Data Collection and Performance Metrics

The used datasets in this study are three different datasets, namely DS1, DS2 and DS3. All datasets are consisting of two measures; the number of faults (Fi) and the number of test workers (Ti) for each day (Di) in a part of software projects lifetime. The DS1 dataset has 46 measurements that involved in the testing process presented in [1]. DS2, also taken from [1], which measured a system fault during 109 successive days of testing the software system that consists of 200 modules with each having one kilo line of code of Fortran. DS2 has 111 measurements. DS3 is developed in [2], which contains real measured data for a test/debug program of a real-time control application presented in . Tables I to III present DS1, DS2 and DS3, respectively.

The datasets were preprocessed by a proposed clustering technique. The proposed clustering technique marks the data with class labels. These labels are set to classify the number of faults into five different classes; A, B, C, D, and E. Table IV shows the value of each class and number of instances that belong to it in each dataset.

In order to evaluate the performance of using ML algorithms in software bug prediction, we used a set of well-known measures [19] based on the generated confusion matrixes. The following subsections describe the confusion matrix and the used evaluation measures.

TABLE II. DS1 - THE FIRST SOFTWARE FAULTS DATASET

D_i	F_i	T_i	D_i	F_i	T_i
1	2	75	24	2	8
2	0	31	25	1	15
3	30	63	26	7	31
4	13	128	27	0	1
5	13	122	28	22	57
6	3	27	29	2	27
7	17	136	30	5	35
8	2	49	31	12	26
9	2	26	32	14	36
10	20	102	33	5	28
11	13	53	34	2	22
12	3	26	35	0	4
13	3	78	36	7	8
14	4	48	37	3	5
15	4	75	38	0	27
1`6	0	14	39	0	6
17	0	4	40	0	6
18	0	14	41	0	4
19	0	22	42	5	0
20	0	5	43	2	6
21	0	9	44	3	5
22	30	33	45	0	8
23	15	118	46	0	2

A. Confusion Matrix:

The confusion matrix is a specific table that is used to measure the performance of ML algorithms. Table V shows an example of a generic confusion matrix. Each row of the matrix represents the instances in an actual class, while each column represents the instance in a predicted class or vice versa. Confusion matrix summarizes the results of the testing algorithm and provides a report of the number of True Positive (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN).

B. Accuracy:

Accuracy (ACC) is the proportion of true results (both TP and TN) among the total number of examined instances. The best accuracy is 1, whereas the worst accuracy is 0. ACC can be computed by using the following formula:

$$ACC = (TP + TN) / (TP + TN+ FP + FN)$$

C. Precision (Positive Predictive Value):

Precision is calculated as the number of correct positive predictions divided by the total number of positive predictions. The best precision is 1, whereas the worst is 0 and it can be calculated as:

$$Precision = TP / (TP + FP)$$

D. Recall (True Positive or Sensitivity):

Recall is calculated as the number of positive predictions divided by the total number of positives. The best recall is 1, whereas the worst is 0. Generally, Recall is calculated by the following formula:

$$Recall = TP / (TP + FN)$$

E. F-Measure:

F-measure is defined as the weighted harmonic mean of precision and recall. Usually, it is used to combine the Recall and Precision measures in one measure in order to compare different ML algorithms with each other. F-measure formula is given by:

F- measure= (2* Recall * Precision)/ (Recall + Precision)

F. Root Mean Square Error:

RMSE is a measure for evaluating the performance of a prediction model. The idea herein is to measure the difference between the predicted and the actual values. If the actual value is X and the predicted value is XP then RMSE is calculated as follows:

$$RMSE = \sqrt{\frac{1}{n} * \sum_{i=1}^{n} (Xi - XPi)^2}$$

Datasets	NB	DT	ANNs
DS1	1	1	1
DS2	0.905	1	0.990
DS3	0.972	1	0.981
Average	**0.959**	**1**	**0.990**

Table 3.2.2: RECALL MEASURE FOR THE THREE ML ALGORITHMS OVER DATASETS

The third evaluation measure is the recall measure. Table VIII shows the recall values for the three classifiers on the three datasets. Also, herein the ML algorithms achieved a good recall value. The best recall value was achieved by DT classifier, which is 100% in all datasets. On the other hand, the average recall values for ANNs and NB algorithms are 99% and 96%, respectively.

In order to compare the three classifiers with respect to recall and precision measures, we used the F-measure value. Fig. 1 shows the F-measure values for the used ML algorithms in the three datasets. As shown the figure, DT has the highest F-measure value in all datasets followed by ANNs, then NB classifiers.

Finally, to evaluate the ML algorithms with other approaches, we calculated the RMSE value. The work in [2] proposed a linear Auto Regression (AR) model to predict the accumulative number of software faults using historical measured faults. They evaluated their approach with the POWM model based on the RMSE measure. The evaluation process was done on the same datasets we are using in this study.

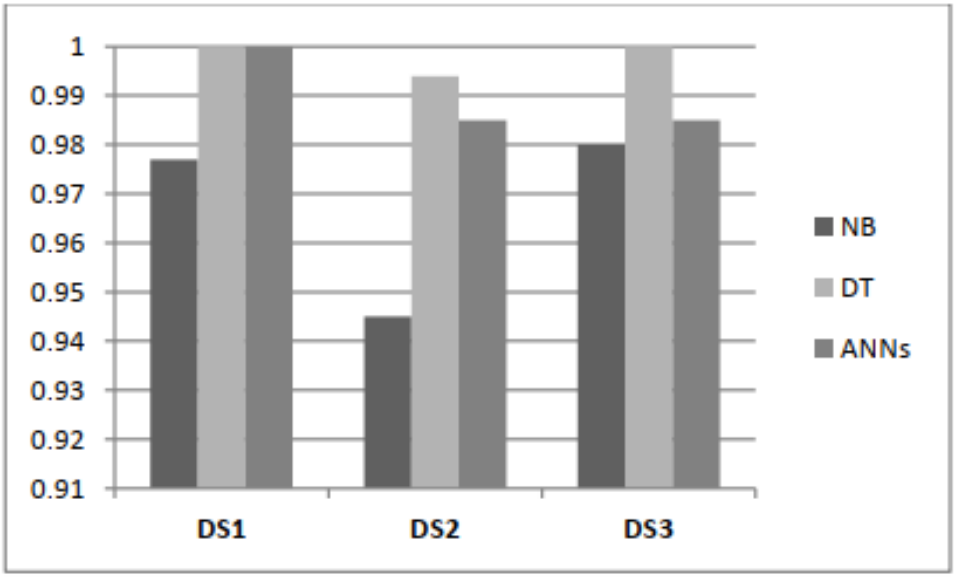

Table 3.2.3: F-measure values for the used ML algorithms in the three datasets

CHAPTER 4

CONCLUSION

4.1 CONCLUSION

To locate a bug, developers use not only the content of the bug report but also domain knowledge relevant to the software project. We introduced a learning-to-rank approach that emulates the bug finding process employed by developers. The ranking model characterizes useful relationships between a bug report and source code files by leveraging domain knowledge, such as API specifications, the syntactic structure of code, or issue tracking data. Experimental evaluations on six Java projects show that our approach can locate the relevant files within the top 10 recommendations for over 70 percent of the bug reports in Eclipse Platform and Tomcat. Furthermore, the proposed ranking model outperforms three recent state-of-the-art approaches. Feature evaluation experiments employing greedy backward feature elimination demonstrate that all features are useful. When coupled with runtime analysis, the feature evaluation results can be utilized to select a subset of features in order to achieve a target trade-off between system accuracy and runtime complexity. In future work, we will leverage additional types of domain knowledge, such as the stack traces submitted with bug reports and the file change history, as well as features previously used in defect prediction systems. We also plan to use the ranking SVM with nonlinear kernels and further evaluate the approach on projects in other programming languages.

CHAPTER 5
REFERENCES

REFERENCES

[1] G. Antoniol and Y.-G. Gueheneuc, “Feature identification: A novel approach and a case study,” in Proc. 21st IEEE Int. Conf. Softw. Maintenance,Washington, DC, USA, 2005, pp. 357–366.

[2] B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala, “Debugadvisor: A recommender system for debugging,” in Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng., New York, NY, USA, 2009, pp. 373–382.

[3] A. Bacchelli and C. Bird, “Expectations, outcomes, and challenges of modern code review,” in Proc. Int. Conf. Softw. Eng., Piscataway, NJ, USA, 2013, pp. 712–721.

[4] J. Zhou, H. Zhang, and D. Lo, “Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports,” in Proc. Int. Conf. Softw. Eng., Piscataway, NJ, USA, 2012 pp. 14–24.

[5] A. Sheta and D. Rine, “Modeling Incremental Faults of Software Testing Process Using AR Models”, the Proceeding of 4th International Multi Conferences on Computer Science and Information Technology (CSIT 2006), Amman, Jordan. Vol. 3. 2006.

[6] D. Sharma and P. Chandra, "Software Fault Prediction Using Machine Learning Techniques," Smart Computing and Informatics. Springer, Singapore, 2018. 541-549.

[7] R. Malhotra, "Comparative analysis of statistical and machine learning methods for predicting faulty modules," Applied Soft Computing 21, (2014): 286-297

[8] Malhotra, Ruchika. "A systematic review of machine learning techniques for software fault prediction." Applied Soft Computing 27 (2015): 504-518.

[9] D'Ambros, Marco, Michele Lanza, and Romain Robbes. "An extensive comparison of bug prediction approaches." Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on. IEEE, 2010.