6-4-2021 11:00 AM

A Technique for Evaluating the Health Status of a Software Module Using Process Metrics

. Ria, *The University of Western Ontario*

Supervisor: Kontogiannis, Konstantinos, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Master of Science degree in Computer Science

Recommended Citation

Ria, ., "A Technique for Evaluating the Health Status of a Software Module Using Process Metrics" (2021). *Electronic Thesis and Dissertation Repository*. 7855.
https://ir.lib.uwo.ca/etd/7855

Abstract

Identifying error-prone files in large software systems has been an area where significant attention has been paid over the years. In this thesis, we propose a process-metrics based method for predicting the health status of a file based on its commit profile in its GitHub repository. Precisely, for each file and each bug fixing commit a file participates, we compute a dependency score of the committed file with its other co-committed files. The calculated score is appropriately decayed if the file does not participate in the new bug-fixing commits in the system. By examining the trend of the dependency score for each file as time elapses, we try to deduce whether this file will be participating in a bug fixing commit in the immediately following commits. This approach has been evaluated in 21 mediums to large open-source systems by correlating the dependency metric trend against the known outcome obtained from a data set we use as a gold standard.

Keywords

Software Repository Mining, System Analysis, Software Maintenance, Software Error-Proneness, Process Metrics, Software Analytics

Summary for Lay Audience

In most approaches to date, classifying a file as error prone or not is primarily based on metrics and other structural information extracted from the source code, and to a lesser extent on information related to process metrics extracted from the commit history of a file. This thesis proposes a process metrics-based method for predicting the error proneness of a file based on its commit profile in its GitHub repository. Our approach is based on the calculation of a per-file strength metric which indicates the level of dependency and commit frequency a file has with other files in a commit. The dependency score, i.e., strength metric is appropriately decayed if the file does not participate in the new bug-fixing commits in the system. By examining the trend of this strength value of a file over a period of commits, we aim to predict the error proneness of the file in immediately following commits.

Acknowledgements

I wish to express my deep sense of gratitude and indebtedness to my elite guide and supervisor, Dr. Kostas Kontogiannis, Professor, Department of Computer Science, Western University, London Ontario, for their constant support throughout my dissertation. I am thankful to them for their persistent interest, constant encouragement, vigilant supervision, and critical evaluation. Their encouraging attitude has always been a source of inspiration for me. Their helping nature, invaluable suggestions, and academic guidance have culminated in the present work. I would also like to thank Prof. Miriam Capretz, Prof. Nazim Madhavji, and Prof. Anwar Haque for their valuable comments and for serving as examiners for this thesis.

I want to thank Mr. Marios Stavros Grigoriou, Research Scholars, Department of Computer Science, Western University, for their extensive help and valuable suggestions during most of my dissertation work.

Besides, I am thankful to my dear husband, Mr. Shubham, for his endless support and motivation. He is the one whose constant emotional support provides comfort in a challenging environment. Finally, I want to thank my parents for each and everything they have done for me throughout my academic journey.

I am also thankful to all my friends, faculty members, and other members of the Department for their help, support, and valuable discussion throughout this dissertation work. I thank all members who helped me directly or indirectly in bringing this report to this present form.

(Ria)

Table of Contents

List of Tables

List of Figures

Chapter 1 – Introduction

1.1 Preface

Each industrial software system follows a Software Development Life Cycle (SDLC) before delivering it to the client. Once the software is designed and developed, it goes into different testing cycles that include validating the software against functional and non-functional business requirements received during the SDLC requirement gathering phase. The successful completion of testing ensures that the software system meets its release criteria. However, industrial systems have strict release deadlines and therefore developers and testers need to carefully consider and prioritize the testing depending upon the associated risk of failure (error-proneness) for each file or module being tested. Software companies cannot overwhelm the testing process, and therefore early detection of error-prone or risk modules is of essence. In this respect, we need to devise techniques that can help developers and testers to be notified early with respect to the error-proneness of a file. Utilizing such techniques, the developers and testers can assess the risk early on and prioritize their test efforts on files with high risk and delay testing on files with high likelihood been healthy. This technique is referred by the software engineering community as "shift-left", as it aims to spread the evaluation of the health of a file across the life cycle and not only in the testing phase of the SDLC.

The primary objective of this thesis is to provide a strategy to model the process-metrics based technique in order to calculate a dependency score between a file and its co-committed files, and consequently use this score to predict the error proneness of the file. More specifically, for each file and each bug fixing commit a file participates in, we compute a dependency score this file has with its other co-committed files. We refer to this dependency score as the file's *Overall Strength*. This score is appropriately decayed as new bug-fixing commits appear, which do not include this file. By examining the trend of the strength score for each file as time elapses in a group of consequent commits we refer to as a *segment*, we try to deduce whether this file will be participating in a bug fixing commit or a non-bug fixing commit in the immediately following commits (i.e. the next segment). This information is beneficial in developing, testing, and maintaining a software product so that testers and developers can apprehend the risk associated with the file and diligently prioritize developing and testing the high-risk file.

The thesis first, aims to collect the process metrics and quantitative information from the GitHub repositories second, analyze the file-level commit behavior to be modeled in the form of commit and file change trends and third, use past trend behavior to predict the future error-proneness or health of a file. More specifically, for each file, we collect information about the files it is co-committed with, information about its code churn, number of calls between files, and information about its commit frequency. All this information is used to compute the dependency score of one file with its co-committed files. We refer to the *file-to-file* association strength as the *Binary Strength* of the file. This metric can then be used to compute the *Overall Strength* value of the file by aggregating all the file-to-file *Binary Strength* values of the file with all its co-committed files, in a given GitHub commit record. By examining how this *Overall Strength* metric behaves over time and taking into account the appropriate decay in case of the file's commit-inactivity, we aim to predict how the file will behave in the immediate future, that is predicting whether the file will be identified as buggy or not in the near future commits (i.e. the commits which constitute the next *segment*).

In this thesis, we present a tractable system that uses the process metrics, like the number of times two files are committed together, the number of commits has elapsed since the two files have participated in the commit, etc. obtained from GitHub Repository and Bugzilla to forecast the health status of a file. According to IEEE terminology [1] we use the term error or bug in this thesis to indicate a mistake in the computer program that causes deviation from its specified observable and expected behavior.

The approach has evaluated data collected from 21 open-source systems by comparing the obtained results against a gold standard we have compiled by reconciling GitHub and Bugzilla data. This result is applied to some known error prone files in a system and files whose process metrics trends fluctuate in localized periods.

1.2 Problem Description and Motivation

The problem of using process metrics at the file level over a period of a system's operational life in order to assess the health status of a file in the immediate future can be denoted by the following statement:

"Given a system S composed of files F_1, F_2, ... F_k, *which participate in commits* C^1_1 , C^1_2, C^1_3 , ... C^1_m , ... C^2_1 , ... C^2_p , ... C^k_1 C^k_w , *and where each file* F_i *in commit* C^i_j *is denoted by a process metrics vector* M^i_j , *devise a tractable technique which uses the metrics vectors in commits* C^i_{j-k} , $C^i_{j-(k-1)}$,... C^i_{j-1} *to predict the behavior of the file* F_i *in commit* C^i_j *".*

The rationale behind the motivation of addressing this problem lies first, on the increasing complexity of software systems and second, on the need to release software systems using shorter release cycles and applying a continuous engineering approach on the DevOps cycle. More specifically, over the past few years, we experience a paradigm shift from classic DevOps processes to a more agile continuous integration, deployment, and delivery model aiming to shorten the software release cycles. It is also very recently that the focus has also shifted towards the Ops side of DevOps, emphasizing the use of AI and intelligent data analytics to assist software engineers in identifying error-prone and risky modules, while at the same time, they maintain and evolve large software systems. This area is heralded as AIOps and so far, has gained significant traction in the software engineering community. In this respect, AIOps aims to analyze data from the field as the system operates or evolves, and to a lesser extent, from static source code properties of the system.

In this research, we aim to bridge the gap between software developers and testers by providing them with a list of risky files that could be buggy in future. This will provide a heads up to both developers and testers to prioritize their test cases and helps to mitigate the associated risk of a file in the production. Also, minimizing the bugs that may help to save both resources and time. Moreover, we provide a system that allows using process metrics and related quantitative information collected from GitHub to predict the health status of a file in the system. With the help of such a system, developers and testers can potentially reduce the time consumed in testing and prioritize the test cases based on fault-prone areas of the project.

1.3 Thesis Contributions and Scope

Machine Learning kept evolving in recent years, and significant research has already been conducted to predict the health status of a file using these technologies. However, most approaches utilize machine learning and metrics extracted from the source code and to a lesser extent on the data related to process metrics to detect error-prone modules. Researchers and practitioners are still experimenting with various machine learning algorithms, artificial intelligence, source code metrics, process metrics, and repository features to develop a framework or a model to predict the behavior of a file over time.

There are certainly many approaches to predict the fault-proneness of a software module. However, two widely used one's are Machine Learning and Software Metrics. We have covered literature review of these approaches in Chapter 2. The certain grey areas, constantly appearing in the research, that indeed require attention are:

- Results indicate there is no single set of metrics that applies to all systems to predict the bugginess of a file.
- Results indicate that there is no specific learning technique that performs the best for all the data sets (i.e. systems).

Thus, we need to devise a framework where we can plug and play with the different strategies of choosing software metrics. Additionally, contrary to other software domains where a single model can correctly classify all the related instances, this has not been proven in the case of Software Engineering Bug Prediction. In particular, if a classifier is trained to classify flowers, as is the well-known IRIS example, it can classify other irises of the same species too. In contrast, a model trained to perform bug prediction in one project, cannot be used to perform the same task in other projects even if the same team is working, or they do the same thing. Thus, we have a quest of developing a framework that can not only more robustly provide an understanding of the healthy and non-healthy systems, but also clearly define the implementation and minute details of how things are working internally that can help developers to clear up the smokes of the Black Box Approach of Machine Learning. Thus, we have solely focus on process metrics to evaluate our Hypothesis.

This thesis lies in the area of Experimental Software Engineering. The research objective is to predict the health status of a file using process metrics, i.e., the file commit history, the total number of lines modified in a file, how long it has been since the file is committed, etc. As part of this thesis, we have analyzed 21 open-source systems to perform the experiments. The smallest size of commit data of a system that we have studied is 9.923 KLOC, whereas the most extensive commit data of a system is of size 1453.70 KLOC, and the rest of the size of commit data of a system falls between the above size range. We have classified the systems we have experimented with into three categories: small size systems, large size systems, and medium-size systems, as shown in the Table II.

Overall, the contributions of this thesis are:

1. Propose a programming language agnostic technique in order to model and analyze process related metrics extracted from software DevOps repositories, for fault-proneness prediction at file level. More specifically, the thesis proposes a novel technique whereby a) sequences of commits are grouped in so called segments; b) fault-proneness prediction related process metrics are calculated per commit and per segment; c) process metrics trends (upward, downward) within a segment are computed and; c) analysis of whether process metrics trends in sequences of segments can serve as a predictor of fault proneness.
2. Propose a dynamic framework and a set of novel metrics whereby a) *file-to-file* intra-commit dependency metrics and *overall strength* metrics for a file can be computed and serve as fault-proneness indicators for a given file and; b) these file-to-file dependency and overall strength metrics can be decayed over time so that the fault-proneness identification system can exhibit a dynamic realistic behavior over time.
3. Propose an efficient hash table-based data structure to model and tractably analyze in dynamic memory complex commit dependencies and process metrics extracted from very large software systems involving tens of thousands of commits.
4. Develop a modular architecture for the aforementioned system whereby different metrics, and analysis strategies can be considered as *plugins*, providing thus the opportunity for different techniques to be evaluated using the proposed architecture. In

this respect the architecture can serve as framework and a tool-bench for developing new fault-proneness prediction systems.

5. Provide a comprehensive set of results from experiments conducted with different metrics and analysis strategies applied on various large open source systems in order to evaluate the effectiveness and performance of the proposed technique.

The system encompasses the following architectural components:

- A module to extract process related information from a GitHub repositories.
- A module to reconcile information extracted for GitHub and Bugzilla repositories, in order to establish a gold standard for evaluation purposes.
- A module to compute *Binary Strength* (i.e. file-to-file) and *Overall Strength* (i.e. file-to-all-files) scores between a file and its co-committed files.
- A module to efficiently store and analyze the extracted information and calculate trends and strengths of files over time.
- A module to analyze the *overall strength* score trend patterns of system files by leveraging different correlation strategies and scenarios in order to assess the health status of these files.
- A module to generate result reports in the form of csv files and performance reports.

For this thesis, we have computed the results on the data collected from GitHub repositories of 21 medium to large open source software systems. However, we have not implemented the same on the real-time systems or enterprise-level software. Thus, it is difficult to conclude how the software will behave for real-time applications.

1.4 Thesis Organization

The rest of the thesis organized as follows:

In Chapter 2, we explore related work or fault and error prediction.

In Chapter 3, we discuss the data extraction process, the calculation of the process metrics, and the design of the modeling framework used for calculating the *Binary* and the *Overall* strength scores along with their trends (i.e. slopes) in groups of consequent commits (i.e. *segments*).

In Chapter 4, we report the results obtained, and we discuss these results using different strategies and scenarios.

Finally, in Chapter 5 we summarize the results, and present pointers for future research in this area.

Chapter 2 – Background and Related Work

This chapter will present related work in the domain of software fault prediction, the significance of software metrics in fault prediction, and how machine learning and software analytics are currently being used for fault prediction.

2.1 Background

2.1.1 Software Fault Prediction (SFP)

SFP is a standardized way of assuring and improving the quality of a software system by leveraging past information to foresee possible bugs or faults that are likely to occur in the software in the future. Software is jeopardized by faults that can impede the performance of a software product; moreover, it directly impacts the quality of the system [2], may make it diverge from its specified requirements, fall out of compliance or even violate laws [2].

Software is designed to provide quality services to users. Gradually, the requirements of the stakeholders start increasing and so does the complexity of the software. The introduction of increasingly more complex functionality into software products is a potential reason for system breakdown. It is indispensable to determine the error-prone modules before release to ensure the quality of the delivered component. It is observed from many studies that a fault in one software component can completely shut down the entire application, something that would make this a critical fault. Thus, extensive research should be performed to predict the error-prone modules to confirm software reliability [5]. The more accurate the prediction is, the less likely the software is to exhibit buggy behavior [4].

Thus, the main challenge for researchers and software scientists is to select an appropriate prediction model or a technique that can assist them in predicting defects and to help them in conducting result validation [5]. However, there is no direct methodology or strategy for fault prediction. Thus, computer scientists utilize different metrics, i.e., source, process, and a combination of metrics for different estimation strategies. Using the appropriate metrics contributes to achieving the desired estimation.

Once a set of metrics for fault prediction has been selected, it is essential to select the appropriate prediction techniques. The prediction can be achieved in two ways, regression and classification. The primary purpose of regression techniques is to estimate the number of defects in software components. In contrast, classification techniques aim to categorize a software module as faulty or clean [78]. It has been shown that classification models can be trained from defect data on earlier versions of the system [16].

In the next sub-section, 2.2 and 2.3, we will take a deep dive into different software quality metrics and Machine Learning techniques, respectively, for predicting fault in software modules.

2.1.2 Software Metrics

The only thing that comes to mind when predicting faults in software, is software metrics since they are used to measure the software quality. There is a further classification of software metrics, i.e., in-process [6] and end-process metrics. As the name suggests, in-process metrics has the main objective to improve the development process. In contrast, end-process metrics pay closer attention to the final product.

The two major classes of software metrics based on software life cycle phases are static and dynamic code metrics. Static metrics can be achieved at the early stages of Software Development Life Cycle (SDLC) and deals with structural features of software [7]. It quantifies the effort required to develop and maintain a software product. In comparison, dynamic metrics are retrieved at the later cycle of SDLC. However, in contrast with static, dynamic code metrics are tough to obtain from the source code. It determines the behavior of the system, along with its maintainability, reliability vectors.

The next section, we will briefly introduce software code metrics, process metrics, and combinational metrics approaches.

Software Code Metrics

Deciding whether a component has a high likelihood to be defective or not has been proved to have a strong correlation with several software metrics. Identifying and measuring those software metrics is vital for an array of reasons, including estimating program execution, measuring the effectiveness of software processes, estimating required efforts for processes, estimating the number of defects during software development, and monitoring and controlling software project processes [8] [9]. Various software metrics have been commonly used for defect prediction, including lines of code (LOC) metrics, McCabe metrics, Halstead metrics, and object-oriented software metrics. Hence, the automated prediction of defective components from extracted software metrics evolved as a very active research area. [10]. In [11], Nagappan aims to find the best code metrics to predict bugs. This work concludes that complexity metrics can successfully predict post-release defects, but there is no single set of metrics applicable to all systems. Hassan et al. have investigated the impact of different aspects of the modeling process on the results and the interpretation of the models [12] [13] [14] [15]. In defect prediction models, classifiers in classification techniques help to identify the defect proneness of a system. The classifier has a configurable parameter that usually controls the characteristics of the classification technique [12]. According to Hassan et al. [12], the default configurable parameters underperform compared to the automated parameterization. Thus, the author has carried out his study on 18 datasets and outlined that AUC's performance improves by 40 percentage points. Moreover, Hassan et al. [13] evaluate the influence of class rebalancing techniques on performance and interpretation of defect models. After evaluating 101 datasets it is evident that SMOTE and under-sampling technique are beneficial as it increases AUC and Recall. Moreover, Hassan [14] addresses the multicollinearity problem in defect prediction models and identifies the importance of feature selections and various reduction techniques beneficial to improve the performance of the defect prediction model. In the paper [15], author Chakkrit Tantithamthavorn evaluates the impact of correlated metrics on the performance of defect models.

Shakhovska et al. investigate the use of unsupervised learning for performing Defect Prediction utilizing SOMs (Self Organizing Maps) and Hierarchical Clustering on data from the promise software [18] engineering research repository [17].

Software Process Metrics

If there is any change made in the system during SDLC it is quantifiable in terms of process metrics. The software changes concerning each commit are collected during the lifecycle of a project with multiple software releases [19]. It is calculated using different sources like the experience of the developer [5], version history of software [20] [21] [22], change frequency, and more. Significant research has already been done to identify various process metrics that can help to predict the fault proneness of a file. NML, NR, NDA, and NDVP are the most widely used process metrics.

1) Number of Modified Lines (NML) – It measures the difference between the two software builds in terms of Lines of Change [20][21][22][23] These metrics calculate the total number of added and deleted lines from the source code of a file compared with the history of a file. Nagappan and Ball [77] identify that the Number of Modified lines in a file had a good defect density prediction performance.
2) Number of Revision (NR)- The consequent deployment of the file and its frequently changing revisions during development can also predict the fault proneness of a file. Version Control Systems, like GitHub helps to keep track of file history on the main branch. The NR metrics has already been used as a different name by several authors [71] [72][73][74].
3) Number of Distinct Authors/Committers (NDA)- The NDA metrics defined as number of distinct authors who have participated in the deployment of a file into the production. In many researches, it is quite evident there is a high correlation between NDA metrics with pre and post-deployment failures [71] [75].
4) Number of Defects Appeared in Previous Version (NDPV)- The NDVP metrics evaluate the correlation of fault proneness of a file based on the history of the defects appeared in the previous revision of a file [76].

Let us assume that a few lines are added to a file, then delta value will be non-zero. However, if we add and delete the same number of lines in a file, the delta value is going to show that by remaining equal to zero. Code Churn Metrics (CCM)helps to calculate the change in the code. The significance of adhering to the process metrics is that it contains more illustrative information about which part of the code is defective. In [24], Majumder et. al performed a large-scale comparison between Source Code Metrics and Process Metrics utilizing four different statistical models for prediction over a collection of 700

GitHub projects comprising 722,471 commits. Their results indicated that when models were trained and tested on source code metrics and process metrics, the process metrics outperformed the source code metrics.

2.1.3 Machine Learning

Machine Learning (ML) is a subset of Artificial Intelligence, it is a computer algorithm that enables a system or a machine to evolves automatically, learn from data and experience, build prediction models. [25]. ML builds a model based on a training data set that can generate predictions without the prediction having been explicitly programmed [26]. ML classification depends on the type of data input, the algorithm, and the output expected from the algorithm. Based on the above parameters, ML approaches are broadly classified as 1) Supervised Machine Learning Algorithm 2) Unsupervised Machine Learning Algorithm. In SLA, the model is trained with training data that contain a set of input and its corresponding output. In contrast, in ULA, the model is provided with the data containing only the input variables and attempts to identify common properties of the data structure in the data set.

There are different types of supervised learning algorithms, which include classification, regression, and active learnings [27]. It has been shown that classification models can be trained from defect data on earlier versions of the system being analyzed. Some of the most commonly used supervised learning techniques for defect prediction are outlined below:

Logistic Regression (LR): Logistic regression is a supervised classification algorithm whereby the target variable O (i.e., output), can take on values in the interval [0, 1] representing the probability for a given set of input features I to belong to class 1 or 0.

Random Forest (RF): RF is an ensemble type of learning method used for both classification and regression problems. The key idea behind RF is the construction of several decision trees at training time and outputting the mode/mean prediction of the individual trees.

Support Vector Machine (SVM): SVM is a discriminative classifier formally defined by a separating hyperplane. In SVMs, given a labeled training data set whereby each data item is marked as belonging to one or the other of two categories, the algorithm outputs an optimal hyperplane, which classifies new unseen data in one of these two categories.

k-Nearest Neighbors (k-NN): k-NN is a non-parametric method that can be used for both classification and regression problems. In both cases, the input consists of the k closest training examples in a feature space. The output depends on whether k-NN is used for classification or regression. In classification, the result is to categorize an input to one of the equivalence classes. In regression, the output is to assign a value to the input, usually the average of the values of its closest k-neighbors.

Neural Networks (NN): Neural Networks are nonlinear predictive structures that consists of interconnected processing elements called neurons that work together in parallel within a network to produce output, often simulating an unknown function or phenomenon.

Multi-layer Perceptron (MLP): MLPs refer to a class of feedforward artificial neural network (ANN). An MLP comprised of a directed graph of multiple layers of nodes, which are fully connected to the nodes of the next layer. For training purposes, MLP utilizes a supervised learning technique defined as backpropagation. Apart from supervised learning, unsupervised techniques have also been used in a number of research papers. Such unsupervised learning techniques are outlined below.

Self-Organizing Maps (SOM): SOM is a type of artificial neural network (ANN) that is trained using unsupervised learning to produce a low-dimensional, discretized representation of the input space of the training samples, called a map, and is therefore a method to do dimensionality reduction.

K-Means Clustering (KMeans): K-means clustering is a method of vector quantization, that aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster.

Both these techniques can be used to partition data into two classes without having to use labeled data. Such techniques have been investigated [29] by Yang et. al to compare their efficacy at performing Just-in-Time Defect Prediction.

2.1.4 Evaluation of Classification Models

Once the type of classification model is decided and created, its effectiveness need to be evaluated. This evaluation uses a testing data set that collects data instances that have not already been used for training. Specifically, of particular interest are the accuracy and precision of a model. There are different ways through which we can assess the effectiveness of classification models.

Accuracy: It is the initial step to evaluate the performance of a classification model. It is the frequency of correctly identified instances. However, it is not self-sufficient to evaluate the model prediction. Thus, computer scientists recommended using recall, precision, and F1 score for model evaluation. The formula for calculating accuracy is given below:

$$Accuracy \ = \ \frac{True\ Positive + True\ Negative}{True\ (Positive\ + Negative) + False\ (Positive\ + Negative)}$$

Precision: It is a way to identify the number of files that are classified as buggy files to the total number of files. Total number of files contain both buggy and non-buggy files. This is used along with the accuracy to know how model performs to identify the bugginess of a file. The formula for calculating precision is given below:

$$Precision \ = \frac{True\ Positive}{(True\ Positives + False\ \ Positives)}$$

Recall: It is the proportion of files that are classified as buggy to the total number of files that are actually buggy. It is also called sensitivity. The formula for calculating recall is given below:

$$Recall = \frac{True\ Positive}{(True\ Positives + False\ \ Negatives)}$$

F1-Score: It measures the validity of the classification model using precision and recall. In other words, it is the harmonic mean of recall and precision. The F1 score values vary between 1 and 0. If the F1 score is 1, it is considered as the best fit model for the given data set. In contrast, if it 0, then the model doesn't fit in the provided data set.

$$F1 - Score = 2 * \frac{Precision\ * Recall}{(Precision\ + Recall)}$$

Confusion Matrix: It is well known as an error matrix and helps in providing a tabular visualization of evaluating the performance of machine learning algorithms [29]. In the table, instances of predicted class are represented in rows whereas instances of the actual class define in the column.

Table I: Confusion Matrix

	Actual		
Predicted		**Positive**	**Negative**
	Positive	True Positive (TP)	False Positive (FP)
	Negative	False Negative (FN)	True Negative (TN)

2.1.5 Technical Debt

Ward Cunningham introduces the concept of technical debt in 1992 positing that the introduction of debt helps in quickly achieving software development goals by postponing specific code development and the satisfaction of certain test cases to ensure the rapid delivery of the product. It seems a lucrative approach. However, fixing these issues in the long term can require copious amounts of human effort, time, and costs. Inflexible design of the software, poor formatting, lack of documentation, improper test cases, and absence of test reports are few examples of technical debt. With the understanding of TD, we can assess the risk associated with the software components. The below section defines various types of technical debt in software development [30].

Testing Technical Debt: It is one of the most commonly known technical debt types. It mainly occurs due to the limited number of test cases, improperly documented test results, and the absence of automated test cases.

Design Technical Debt: When the hasty developer does not follow the design principles and OOPS terminology during development, the client has to accumulate inflexible design debt. Poorly followed design principles, ignorance of object-oriented class, highly coupled instances, low cohesion, and the existence of God classes are a few examples of design debt.

Code Technical Debt: The inexperienced developer who does not follow the desired coding practices can add code debt to the software product. It can take a toll on rewriting the complete module or refactoring it because the written code may be redundant, the performance of the system is deteriorating, and not following the coding standards makes it difficult to understand.

Architectural Technical Debt (ATD): It is defined as the hasty decision taken by software architects or developers to satisfy short-term requirements. However, it impedes long-term goals [31]. It is stated by many architects that ATD is incurred in software systems due to lack of documentation. Usually, in the software industry, design discussions with the stakeholders are happened either via chat or e-mails. As a result, developers and architects are unable to track it in one consolidated document [31].

Build Technical Debt (BTD): BTD is generally incurred in the software system when the developer has unbuildable targets, unnecessary command-line flags, and unneeded or redundant dependent jars [32]. These files take a toll on computational resources while building a software application.

Requirement Technical Debt (RTD): Software Developers often set the order of the requirements based on the priority of the client or user needs. Thus, the long-term gain in exchanged for short-term profit [37]. The main issue that can cause the RTD is expensive requirements that are hard to achieve, unnecessary requirements, and requirement gap.

Documentation Technical Debt (DTD): DTD can identify missing, inadequate, or incomplete software components [34]. Insufficient information on requirement definitions and requirement unpredictability can add to the DTD of the software system. Also, a lack of non-functional requirements and dependency between the requirements can impact the DTD [35] [36].

2.1.6 Bugzilla

Bugzilla is an open-source defect tracking tool developed by Mozilla on August 26, 1998. It is developed on Linux, Apache, using MySQL, and the PHP language. It provides a consolidated place for tracking defects in a system that can help the manager or teams track the faulty components and frequently evaluate the risk associated with the components. Moreover, we can assess individual developer performance and give them feedback from time to time. Bugzilla can track multiple projects at the same time [38]. Besides, it helps to submit and review patches and manage quality assurance. One of the best features of Bugzilla is a web interface along with its programmatically accessible API which is kept up to date through the years while incorporating newer technologies.

2.1.7 GitHub

GitHub is a version control website built around the central concept of a repository and code change (commit). It has greatly simplified the open-source software development, accommodating many web applications and supporting versioning very straightforwardly and accessible to all. Besides, it offers access control to manage the repository as a private, public, and shared repository. GitHub hosts 40 million users and almost 190 million repositories, according to collected data in January 2020 [39] [40]. We can even clone a repository to our local machine. With the help of GitHub, one can track the development of any component. Every commit in GitHub is associated with commit_id, commit_comments, file_id, file_path, user, commit_date, commit_time, and milestone. With the help of these attributes, one can identify when who, and which file (or files) is committed in the repository. On the other note, during development, it is advisable to create a new branch using the main branch and allow merges once the child branch testing is successful, which along with other less significant details constitute a set of good practices for developing using GitHub.

2.2 Related Work

2.2.1 Defect Prediction using Machine Learning

A variety of machine learning methods have been proposed and assessed for addressing the software bug prediction problem. These methods include decision trees [41], neural networks [42], [43], Naive Bayes [44], [45], support vector machines [46], Bayesian networks [47], and Random Forests [48].

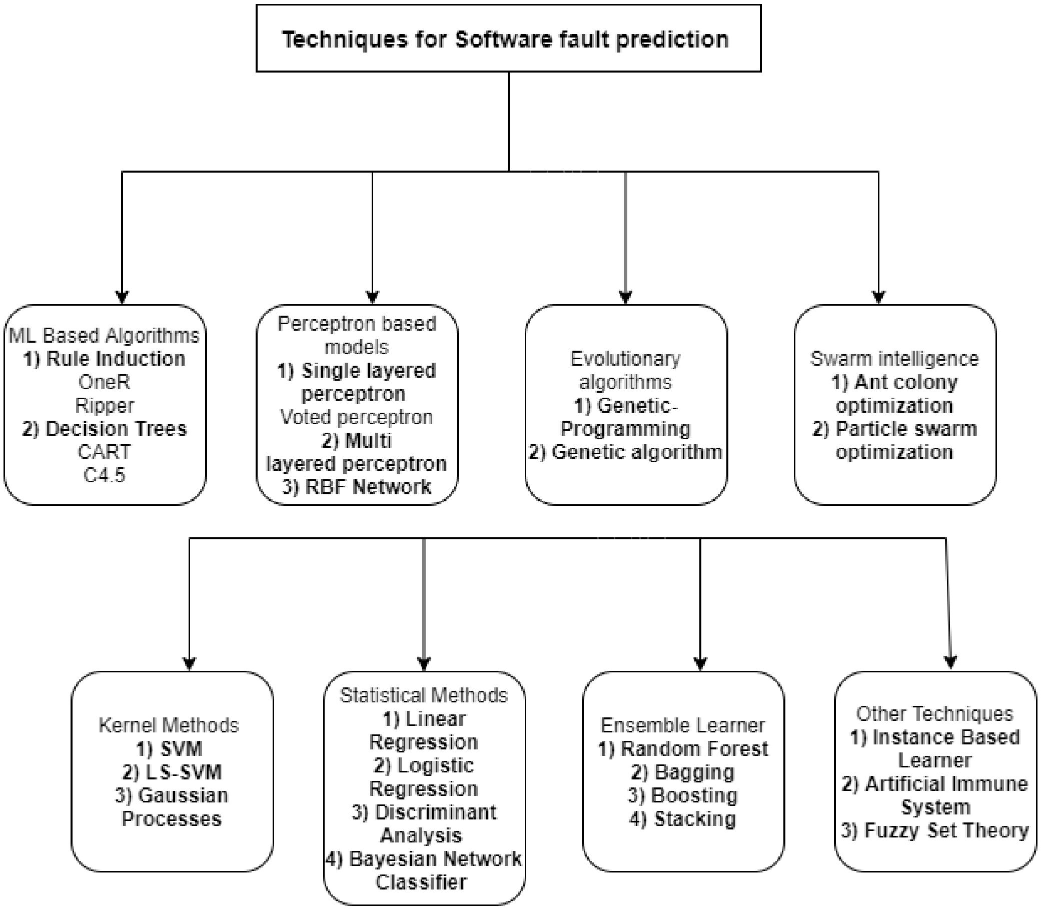


Fig. 1 Supervised Classification for Software Default Prediction [2.9.4]

The work by Cagatay Catal [49] had investigated how the size of a dataset, metrics sets, and feature selection technique can affect the results of software fault prediction. The study identified random forest offers the best prediction performance for the large data set, while the Naïve Bayesian Network algorithm provides the best prediction for small data set. This study has selected 13 metrics to evaluate and compare the performance of algorithms.

The study performed by Ezgi Erturk [50] uses McCabe metrics to evaluate the performance predictive models. Artificial Neural Network and Support Vector Machine are used to measure the performance of an Artificial Neural Network Inference System. The results obtained using McCabe Metrics are 0.7795, 0.8685, and 0.8573 for Support Vector

Machine, Artificial Neural Network, and Artificial Neural Network Inference System respectively [50].

Dejaeger [51] performed a study on an open-source data set from the Eclipse Foundation and NASA IV&V facility dataset to evaluate the performance of fifteen Bayesian Network (BN) classifiers using Halstead, Lines of Code, and McCabe complexity metrics. The paper concludes that Augmented Naïve Bayes Classifier can produce comparable or better performance than the Naïve Bayes Classifier [51].

According to the study by author Shanthini [51], 21 method-level metrics proposed by Halstead, McCabe and ten class-level oriented metrics to evaluate the performance of different Naïve Byes, SVM, K-Stars, and Random Forest Algorithms. The study concluded that the Support Vector Machine (SVM) performance is better than all other algorithms.

In [53] the author C. W. Yohannese proposed a framework for software default prediction and predicts the performance of the model using four scenarios. Learning from normal data sets, feature selection data sets, balanced feature selection data subsets, and noise filtered and balanced feature selection subset. The author concludes that if we combine feature selection, data balance, and noise filtering approaches for data preprocessing to perform SDP the performance is better than the one that does not [53].

In a study, by Xin Xia [54] deep-learning techniques are proposed to predict fault proneness in the file. The designed framework consists of two phases, i.e., the model-building phase and the prediction phase. In the model-building phase, they aim to build a statistical model from past changes. While in the prediction phase, they aim to predict if new changes are buggy or clean. In the proposed framework, 14 features have been used, and then data preprocessing has been performed, including two steps, i.e., data normalization and resampling. The results are validated against six large open-source projects using F1-score and cost-effectiveness metrics. The results of the above two metrics are 0.69 of recall, and 0.45 of F1-Score is observed. In cost-effectiveness, 20% of lines of code help, the framework can predict 50% of defect-related changes.

In [55], Guisheng Fan has proposed a model called defect prediction via an attention-based recurrent neural network (DP-ARNN) [55]. The process is divided into steps: Firstly, DP-ARNN parses the abstract syntax tree of programs and extracts as vectors [55]. Secondly, it encrypts vectors as inputs to the DP-ARNN. It allows DP-ARNN to learn syntactic and semantic features automatically. Additionally, it provides a mechanism to produce features for accurate defect prediction. The author has opted for F1-measure and area under the curve as evaluation criteria for seven open-source Java projects to evaluate the model. The F1-measure values of DP-ARNN, RNN, and CNN are 0.515, 0.506, and 0.473, respectively, whereas RF + RBM and RF are 0.310 and 0.396. In the performance comparison, DP-ARNN, RNN, and CNN beat conventional methods.

2.2.2 Defect Prediction using Software Metrics

Deciding whether a component has a high likelihood to be defective or not has been proved to have a strong correlation with several software metrics. Identifying and measuring software metrics is vital for various reasons, including estimating program execution, measuring the effectiveness of software processes, estimating required efforts for processes, estimating the number of defects during software development as well as monitoring and controlling software project processes [8] [9]. Various software metrics have been commonly used for defect prediction, including lines of code (LOC) metrics, McCabe metrics, Halstead metrics, and object-oriented software metrics. Hence, the automated prediction of defective components from extracted software metrics evolved as a very active research area [10].

In [11], Nagappan et. al. aimed to find the best code metric to predict bugs. The authors performed their research on five Microsoft projects. Firstly, they collected post-release failure data and then they mapped the post-release failures to defects in the entities. In the next step, they computed standard complexity metrics for the entities and finally, using principal component analysis, they determined the various combinations of metrics to find out which metrics best fitted in predicting the failure of the new entity. At last, they concluded that complexity metrics can successfully predict post-release defects, but there is no single set of metrics that is applicable to all systems.

In [80], the author investigated mapping between the classes to the number of defects that were observed or reported in the first six months before and after release. The authors calculated the correlation metrics between pre-and post-release failures. In their research, they identified that the number of changes in the pre-release failure had highest correlation coefficient as compared to the pre-release failure between different developers.

In [81], the author investigated several historical characteristics of files, i.e., Defect History, Release History, hotfix, post-Release, pre-release, last-minute, moderation and their change history. The main objective of this research is to identify the relationship between the different historical characteristics of a file and its defect count. They had used an empirical approach that uses statistical techniques and visual representations of the data to determine indicators for a file's defect count. In their study, the independent variables were historical characteristics and independent variables were defect count of a file that occurred between two consecutive releases during its history. They applied their approach to nine open-source Java Projects. At last, they concluded that defect count does not increase with the number of revisions of a file. There is stronger Statistical evidence of a relationship between the number of changes performed just after the file's release and defect count. Overall, they said that the software's history is a good indicator of its quality.

In [82], the author investigated various developer metrics, like, number of code churns made by each developer, number of commits made by each developer, and number of developers involved in each module to predict the fault proneness of a module or software. Also, they investigated the effectiveness of developer metrics for performance improvement in fault prediction models. At the first step, the author gathered the developer information from the version control system. In the next step, they collected the information about the bug from the bug tracking system. After collecting the information, they measured the proposed developer’s metrics defined in [18]. They concluded that those software modules that are touched by more developers are prone to more errors. Also, it is a good predictor for software bug prediction.

In [71], Graves et al. researched 1.5 million lines of the system, where they aim to predict the fault in 80 modules of a system written in C language. The modelling was performed using simple generalized linear models (GMS) that extend the idea of linear regression. In their analyses, if they predict the number of faults that appear in the module using the log of the number of lines of code in the module, then the number of faults in a module is a Poisson Distribution with a mean equal to a constant multiple of some power to the number of lines. The author also discussed the stable model that helps to predict the number of future faults using numbers of past faults. At last, they concluded the number of lines of code in the modules and the number of different developers who had worked on a module is not a good predictor in identifying the bug.

One widely used metric but whose definitions vary from author to author, i.e., age of the file. It is considered an important process metric in many studies. Illes-Seifer et al. [81] defined three categories: Newborn, Young, Old; Ostrand et al. [72, 74] has categorized file into New File and Old File. It is known as File Age, i.e. number of months a given file has existed. Different studies indicate different views and result about this indicator. Many authors confirmed in their findings that a new file is more prone to error as compared to an old file. Ostrand et al. and Bell et al. concluded that fault proneness decreases with the age of the file. Also, Graves et al. [71] found that age metrics when combined with delta have improved the performance of the model.

Hassan et al. investigated the impact of different aspects of the modelling process on the end results and the interpretation of the models [12] [13] [14] [15]. In [15], Hassan et al. examined the effect of correlated metrics on the interpretations of defect models and advancement of the performance of defect models when correlated metrics removed. Firstly, they removed the correlated metrics to obtain data sets (mitigated data sets), then they built the model using the mitigated data sets and un-mitigated data sets. Secondly, they analyzed the designed model in a two-step process. In the first step, they calculated important score metrics and in the second step, they identified the ranking of metrics. Finally, they analyzed the performance of the model and concluded that after eliminating all the top-ranked correlated metrics, the consistency of techniques was improved by 15%-64%. Also, they advised that researchers and scientists should be cautious while removing the correlated metrics.

In [57], Venkata et al. compared different machine learning predictor models for finding faulty software components. The study was performed on the real-time defect datasets obtained from NASA. The authors used 70% of training data sets and 30% of test data sets, in which the input data to the model was continuous values while the output data could take either continuous or discrete values based on the classifier. For the Decision Trees, Naïve-Bayes, Logistic Regressions, 1-Rule, and Nearest Neighbor, the input could be either continuous or discrete while the output must be discrete. Neural Networks accepted both continuous and discrete values for input and output. Their results showed that the combination of 1R and Instance-based Learning along with the Consistency-based Subset Evaluation techniques provided comparatively better results [57]. Also, for the performance of the model, they concluded that size and complexity metrics are not sufficient for correctly predicting software defects.

In [58], Wang and Yao investigated different types of class imbalance learning methods that could help software defect prediction with the objective of finding better solutions. In this process, they found that imbalanced distribution between classes in bug prediction was the root cause of its learning difficulty. In their research, they selected the data sets that were having high imbalance rates, data sizes, and programming languages. The data sets used in the research came from the practical projects. Firstly, they sorted the datasets in the order of imbalance rate. Each dataset was comprised of an attribute, i.e., module/method and a label that identify whether the module contains a defect or not. In the next step, they examined five class imbalancing learning methods and covered three types, i.e., Boosting-based Ensembles, threshold-moving, and under-sampling. After analyzing each method, the balanced random under-sampling had a better defect detection rate than the other class imbalance learning methods, but it is still not as good as Naive Bayes.

Similarly, in [59], Zimmermann et al. proposed an approach to predict bugs on cross-language systems. The work examined many such systems and concluded that only 3.4% of the systems had precision and recall prediction levels above 75%. The authors also tested the influence of several factors on cross-language prediction success and concluded that there was no single factor that led to such successful predictions. The authors used decision trees to train the model and to estimate precision, recall, and accuracy before attempting a prediction across systems.

In [60], Hassan discussed how frequent source code "commits" in the repository negatively affect the quality of the software system, meaning that the more changes incurred to a file, the higher the chance that the file will contain critical errors. Furthermore, the author in [60] presents a model that can quantify the overall system complexity using historical code change data instead of plain source code features.

In [61], Majumder et al. checked how specific conclusions, generated from analytics in-the-small, and use those conclusions using analytics in-the-large. The authors performed a large-scale comparison between Source Code Metrics and Process Metrics utilizing four different statistical models for prediction over a collection of 700 GitHub projects comprising 722,471 commits. The data was collected in a three-step process. The first step was to collect the data for each file in each commit by extracting the commit history of the project then further analyzing each commit for their metrics. While assessing each commit, they created an object every time a new file was encountered and kept a detailed record of process metrics. Along with process metrics, authors kept a track of the files that were modified together to calculate the commit-based process metrics. In the next step, they identified the bug-fixing commits using simple search algorithms. In the third step, they used GitHub release API to assemble the release information for each project. Finally, they mined the product metrics using Understand by SciTools. Their results specified that process metrics generated better predictors than process metrics for defects. Also, learning methods like logistic regression that worked well in-the-small perform comparatively much worse when applied in-the-large.

2.2.3 Defect Prediction using Technical Debt

Technical Debts start piling up in software when the developers trade of short-term goals with long-term goals Zazworka, Nico [62] has proposed four different approaches to detect Technical Debts using multiple tools to know their similarities and dissimilarities to assess their relationship with the interest indicators. The author has selected four Technical Debt Identification techniques, evaluated them, and applied them to thirteen versions of the

Apache Hadoop open-source project [62]. Violation of Modularity, accumulated grime, code smells, and automatic static analysis issues are the four TD techniques. The output shows that different problems in the source code with different approaches. Each technique identifies the various area of concern in the source code, i.e., areas of concern vary with the selected method. Hence, these techniques have a minimal overlap among themselves. To calculate the association between the approaches a five steps process is proposed:

1) Calculate how indicators are related to each other to evaluate the Technical Debt.

2) Filter out the strongly related indicators using statistical functions.

3) Combine the association measures.

4) Combine the thirteen Hadoop versions into one cumulative measure.

5) Apply visualization.

The study concludes the following:

- There is an increase in Technical Debt as the size of Hadoop increases.
- Modularity Debt also increased exponentially with the Hadoop releases. Also, modularity violation does not occur with code smells.
- There is a strong coupling between the find bugs issue (high) and a code smell.
- Finally, change-prone classes tend to be defect-prone classes and vice versa.

In the paper [63], the accumulated grime (non-pattern-related code) can create an issue of test debt. The main reason for the testing debt is the evolution of a system with time because of added functionalities. The paper has conducted a study that indicated that design pattern grime and organizational grime decrease the system's testability. Hence, testing requirements increase as grime accumulated.

In this paper [64], the author proposes four types of bugs, i.e., tag bugs, debt-prone bugs, duplicate bugs, and re-opened bugs. The author has performed a study on Mozilla to evaluate the effect of debt-prone bugs in software systems. For this, the author has classified the debt prone bugs into three features: the time required to fix the bug, the rate of occurrence of debt-proneness, and the number of bugs. A thorough analysis of these debts concludes that it can affect the average time cost of fixing bugs in the product. Secondly, ML algorithms are applied to train prediction models on attributes and the average time cost of fixing bugs using the product's history. The algorithm results will provide a ballpark number to predict the average time taken to fix the bug and help the developer monitor the software quality.

In the discussed literature review we have observed three different approaches to predict the bugs in software modules. These are Machine Learning, Software Metrics, and Technical Debt. Machine Learning and Software Metrics are two widely used approaches in the research area of Software Engineering Bug Prediction. These two approaches have their challenges like there is no single set of metrics that applies to all systems to predict the bugginess of a file. Also, there is no specific machine learning model that performs the best for all the data sets or systems. Thus, we need to devise a framework or system which can not only more robustly provide an understanding of the healthy and non-healthy systems, but also clearly define the implementation and minute details of how things are working internally that can help developers to clear up the smokes of the Black Box Approach of Machine Learning. Thus, for our research, we have used process metrics to evaluate our Hypothesis. In the following chapter, we have defined the data extraction techniques and data model of the designed system to predict faults in software modules.

Chapter 3 – Data Extraction and Modeling

In this chapter, we will first demonstrate how to select appropriate systems and process metrics to predict the fault proneness of a file. Second, we will define the data extraction techniques utilized to perform the data reconciliation in order to identify the bug fixing commits. Third, we will describe the data model of the designed system and the concrete data structure used throughout the system. Finally, we will present a case study conducted on projects collected from a git source code repository and the KDE.bugzilla bug report repository.

3.1 Overall Framework

The outline of the overall framework capturing the complete process is depicted in Fig 2. The first step is to identify and select the appropriate systems based on the selection criteria defined in section 3.2. The second step is to create a program able to retrieve the appropriate information from GitHub and Bugzilla. The complete extraction process is described in section 3.3. The third step entails the reconciliation of the data collected, from GitHub and Bugzilla, not only to enable the evaluation of results but also to create cleaner datasets with more precise data points. The reconciliation procedure is explained in section 3.3.3. In the fourth and final step, the collected information is processed, and a dependency score is calculated for each file with respect to its co-committed files. The calculated score is decayed as new commits appear, which do not include the particular file. By examining the trend of the dependency score for each file as time elapses, we try to reason on whether this file will be participating in a bug fixing commit in the immediately following commits.

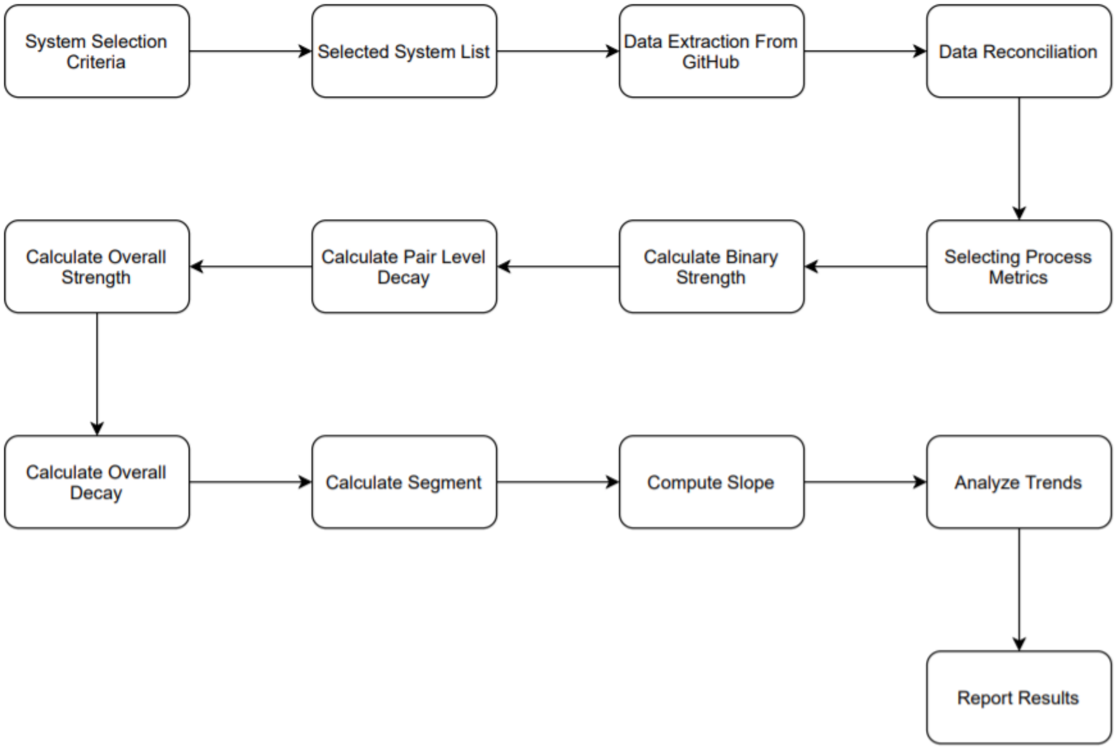

System Selection Criteria
Selected System List
Data Extraction From GitHub
Data Reconciliation
Selecting Process Metrics
Calculate Binary Strength
Calculate Pair Level Decay
Calculate Overall Strength
Calculate Overall Decay
Calculate Segment
Compute Slope
Analyze Trends
Report Results

Fig. 2 Outline of fault proneness prediction process

3.2 System Selection

In [65] Kalliamvakou et al. have documented thirteen risks associated with project selection on GitHub. Considering these threats, we have selected our projects based on the criteria below.

C1: Many repositories in GitHub refer to individual projects and do not relate to major software development: As GitHub is an open-source tool, it is evident that a lot of the publicly available repositories pertain to individual software development. If these projects are considered, then this may skew the results. Thus, as a part of this thesis, we investigated only those open source projects that are extensively used in the software application. However, we can separate those repositories by checking the number of distinct committers in the repository.

C2: Identifying the active or inactive projects: According to Kalliamvakou et al., many projects are dormant or exhibit minimal activity over a period of time. To counter this threat, we have selected only those projects that are active in software development and testing.

C3: *Repository is not a Software Project:* In GitHub, a repository can contain multiple projects or be a part of a network of repositories. Thus, we judiciously identified such projects that are not part of a network of repositories.

C4: *GitHub does not expose all data*: The GitHub API does not expose the entire GitHub database, but rather a subset of the events or entities.

C5: *Many active projects do not use the GitHub completely and very few projects use the pull request feature of GitHub:* As part of the current study, we try to find all the projects that have shown high GitHub activity. However, we are unable to verify if only GitHub is used in these selected projects of other means of version control are also employed.

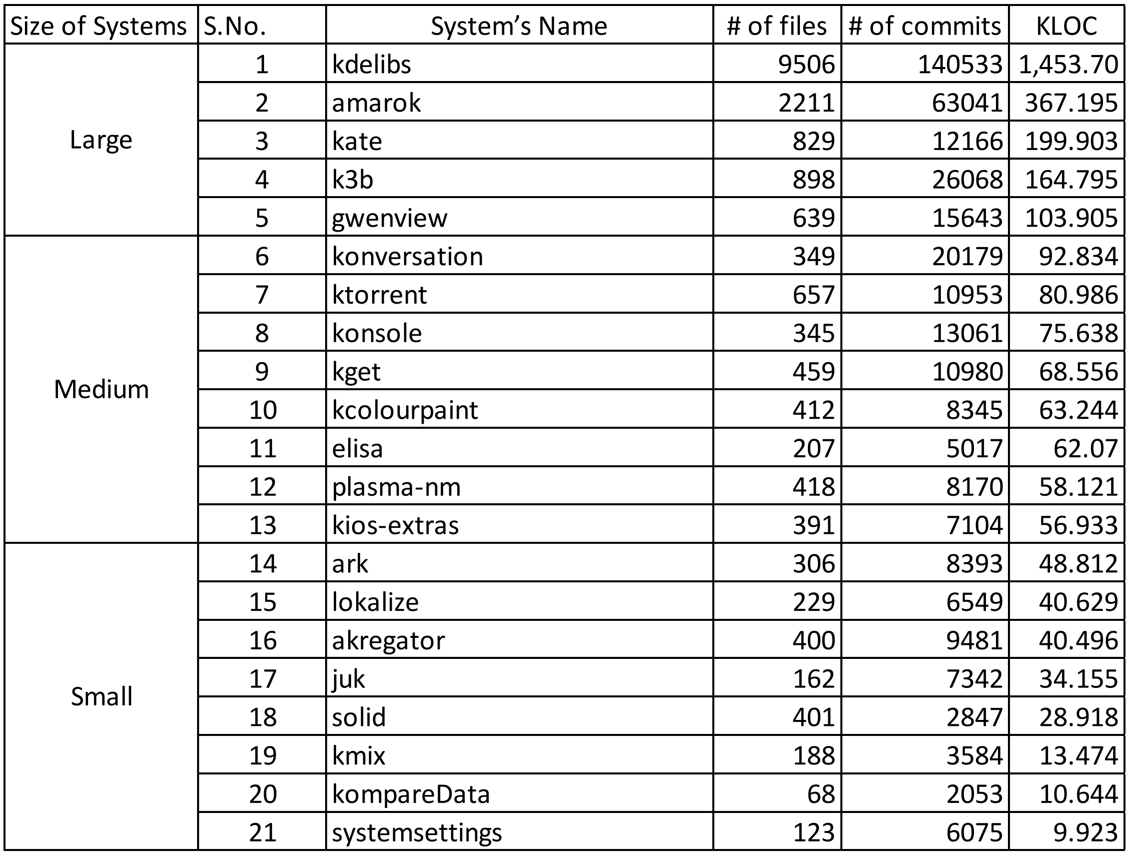
C6: Only *Successful Merges are considered:* Working with a team on the same project requires frequent merges on a main development branch as teammates often work on a child/cloned branch to introduce their changes. Thus, commit data on child/clone branches are not examined.

Along with these selection criteria, we also considered the size of the candidate projects. In order for any results stemming from our hypothesis to be generalizable, and to acquire more accurate results, we have considered small, medium, and large projects. Table II presents the projects that are used to validate the hypothesis.

3.3 Data Extraction

As part of this research, we have collected the process-related metrics from 21 Open Source Software Systems of various sizes and complexities.

Table II: System Examined



Size of Systems	S.No.	System’s Name	# of files	# of commits	KLOC
Large	1	kdelibs	9506	140533	1,453.70
	2	amarok	2211	63041	367.195
	3	kate	829	12166	199.903
	4	k3b	898	26068	164.795
	5	gwenview	639	15643	103.905
Medium	6	konversation	349	20179	92.834
	7	ktorrent	657	10953	80.986
	8	konsole	345	13061	75.638
	9	kget	459	10980	68.556
	10	kcolourpaint	412	8345	63.244
	11	elisa	207	5017	62.07
	12	plasma-nm	418	8170	58.121
	13	kios-extras	391	7104	56.933
Small	14	ark	306	8393	48.812
	15	lokalize	229	6549	40.629
	16	akregator	400	9481	40.496
	17	juk	162	7342	34.155
	18	solid	401	2847	28.918
	19	kmix	188	3584	13.474
	20	kompareData	68	2053	10.644
	21	systemsettings	123	6075	9.923

3.3.1 GitHub Data

The data acquisition process from GitHub is divided into two steps. The first step is to utilize a custom-made client-side extractor tool to connect to and extract data from GitHub. The second step of the raw data acquisition process is to fuse the information extracted by each repository record into one repository, which conforms to the raw data schema depicted in Fig. 3. The extractor application and its data fusion module are implemented using Python 3. The data model is populated by initially downloading a complete repository from the corresponding GitHub site and then moving through each commit on the master branch, adding the relative data iteratively, thus maintaining the initial structure. Once the model is populated, it undergoes several steps of preprocessing that are defined below:

- The first task is to remove all files that cannot contribute to a defect, such as any non-compilable and non-configuration-related files.
- The next task is to use a simple heuristic to clean up the extracted commits so that only actual code-changing commits remain. This entails removing all commits that are clearly annotated as a refactoring commit and also retroactively removing all files which have eventually been removed from the system.
- Finally, all merge commits are also removed since they contain change information pertaining to different branches and will therefore introduce large amount of noisy data points to the dataset. Also, since this study is not focusing on defect introducing software changes, the removal of refactoring and merge commits from the dataset will not impact its ability to discern between faulty and healthy files.

After the cleanup stage is completed, the most important remaining task is that of assigning the class label for each commit and file.

This task is accomplished by parsing the commit message for terms that may indicate that it is a bug-fixing commit as opposed to a clean one, linking commits to issue tracker entries labeled as faults, and optionally applying the same parsing as above to the issue tracker messages [66]. However, not all files in a bug-fixing commit may be defective [67], that means that we also need to reconcile the information from a GitHub bug-fixing commit cid with Bugzilla entries bzid of the same time period in order to identify with higher accuracy which was the particular defective file or files fixed in a given commit cid.

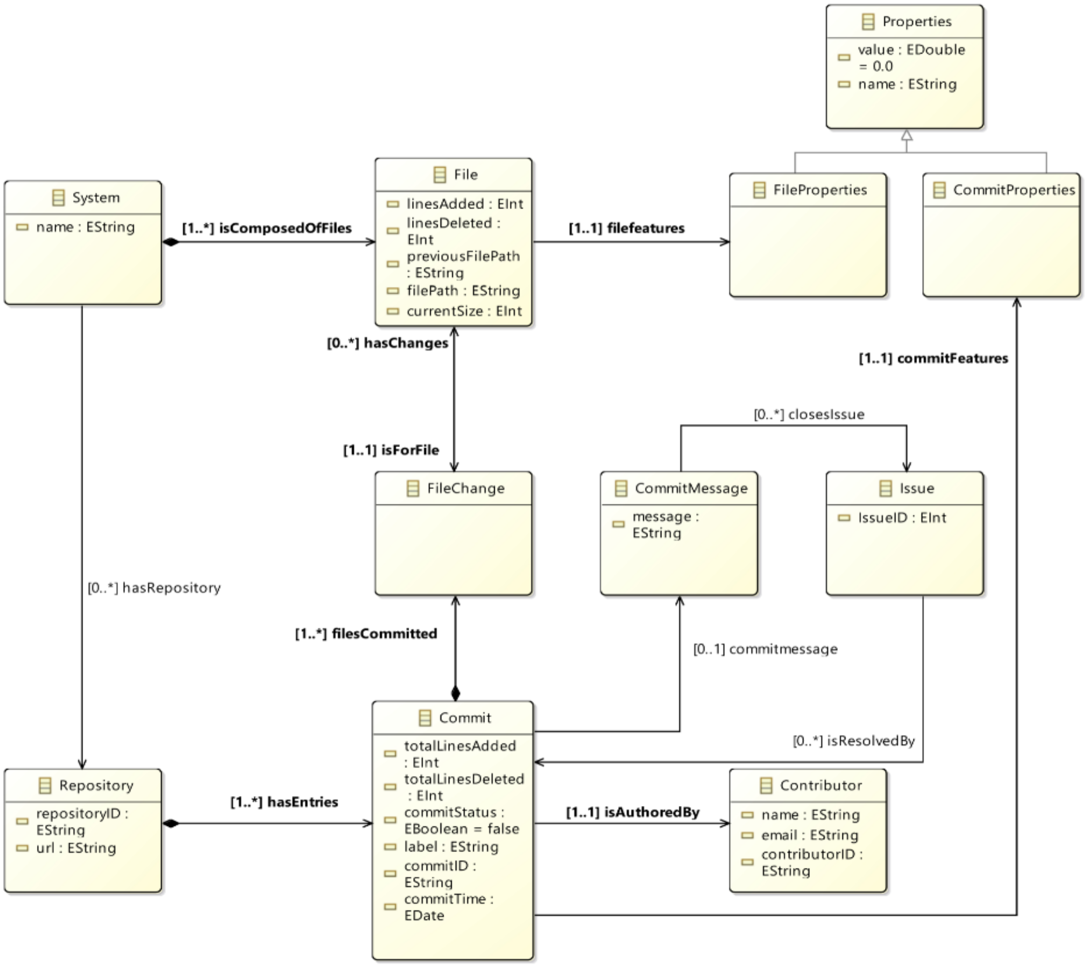


Fig. 3 Data Model for Raw Repository Data

The GitHub/Bugzilla reconciliation process is discussed in more detail in the below section.

3.3.2 Bugzilla Data

The Bugzilla data acquisition process is also based on a custom-made extractor we have built for this purpose. The data collected from Bugzilla repositories adhere to the Bugzilla data schema, which can be found on [68]. The primary entity within a repository is the bug report, which has the following fields bug report ID, product, description, date of submission, date of resolution, author name, bug status, and comments that contain a list of all the submitted comments for the particular bug. Each comment may also include an attachment that may contain additional information about the reported bug or precise modifications that will lead to its resolution. The date and comments can be extracted from the bug report data object for all such reports having a bug status equal to resolved, fixed. From these comments, the precise files modified to resolve the bug can be identified either as plain text submissions or as attachments containing change logs originating from the versioning system used for a given project.

3.3.3 GitHub Bugzilla Data

The data collected from GitHub and KDE.bugzilla repositories are reconciled to identify the faulty file or files in a GitHub commit. It is an essential step for evaluating the obtained results (see Chapter 4 for Results). The reconciliation process was performed after the extraction of all data was completed and both the GitHub and Bugzilla data models were populated.

Firstly, all resolutions for a particular project were ordered chronologically and grouped by bug ID. Secondly, all of the commits available from the GitHub extraction process were iterated until one of the dates available in the Bugzilla resolution data was reached. At that point, the search space was limited to a window around the date of resolution. The next step was calculating the maximal intersection between files committed in each of the commits and all modified files of the current resolution. Finally, the commit exhibiting the largest intersection while remaining the closest to the Bugzilla resolution date had these particular files annotated as faulty. This process was then repeated with the remaining files in the Bugzilla resolution set until either the set was exhaustively matched to corresponding commits or the search space was depleted.

More formally, let B_k be a Bugzilla report with timestamp t_k that references a set of files E^k and resolves file F_k^j. Next, we identify the GitHub commits cid_1, cid_2, ... cid_m, which are within the timestamp window $[t_k – x, t_k + x]$ where x is set to one month. Let also $S^i = \{F_1^i, F_2^i, …., F_p^i\}$ be the set of files appearing in commit cid_i. We select $cid_n \in \{cid_1, cid_2, …., cid_m\}$ for which the intersection of E^k and S^n is maximal, and its timestamp is closest to t_k. The concept is depicted in Fig. 4. After cid_n and the corresponding files F_i are successfully matched they are then removed from the collection of commits $\{cid_1, cid_2, …., cid_m\}$ and E^k respectively and the process is repeated until no further matches can be performed within the timeframe.

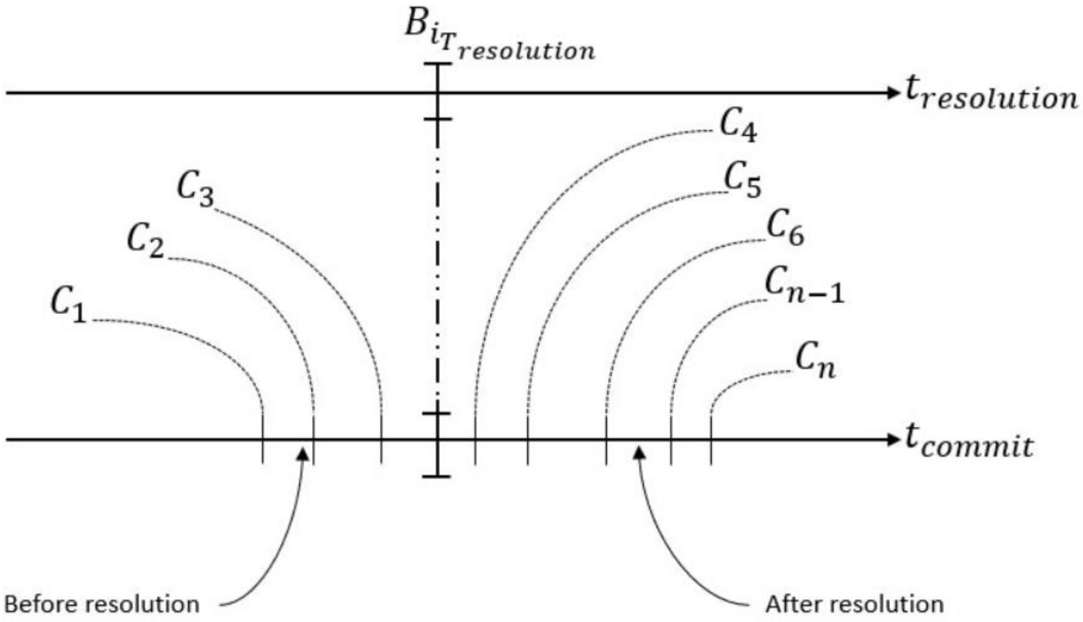


Fig. 4 Timeline for reconciled data between GitHub and Bugzilla

3.4 Data Modelling

The data is stored for internal use in. json format and are shared for use by other systems in .csv format. Each data set entry is comprised of the following attributes: commit_id, branch, message, parents_id, author, authored_at, committed_at, committer, commit_additions, commit_deletions, file_path, changed_files, is_bug_link, is_fix_related, is_bug_fixing, previous_file_path, file_additions, file_deletions, and file_id.

However, in the current system, we have used only specific attributes from the data, namely file_additions, file_deletions, committed_at, commit_deletions, commit_id, is_bug_fixing, and commit_additions. The data is stored internally in guava tables [67].

3.4.1 Guava Table

A Guava Table [67] is a data structure used in the system for organizing the commit information of a file. It helps in systematically organizing the complex information and facilitates the retrieval of particular portions of it with less of a computation overhead. It is a unique structured map where two keys, i.e., row and column, are combined to refer to a single value. The syntax of the guava table is Table<rowKey, columnKey, value>. We have enhanced the existing table structure and organized the information in a Guava Table with the following structure:

Table<String, String, Map<String, List <Object>>>

Guava Tables offer a wide range of methods that contribute to faster operations at a lower computational cost. The most popular operations used are:

1. *boolean contains (Object rowKey, Object columnKey):* It returns a boolean value if the combination of specified rowKey and columnKey is present in the table.
2. *boolean contains Column (Object columnKey):* It returns a boolean value if the specified columnKey is present in the table.
3. *V get (Object rowKey, Object columnKey):* It returns the corresponding value associated with specified rowKey and columnKey.
4. *V put(R rowKey, C columnKey, V value):* It puts the respective value at the specified location, i.e., rowKey and columnKey.
5. *Boolean isEmpty():* It returns true if the table contains no mapping else, it will return false.

This example of a Guava Table corresponds to Fig. 5 : *(27c1b0a5, d1b0) = {abc = [0, 0, 0, 0, 0, 0.0, 25.0, 25.0, 2015-10-28 04:30:37+00:00, 0, 51, false]}, {def = [7, 8, 0, 7, 8, 7.5, 25.0, 75.0, 2015-12-05 15:36:23+00:00, 0, 30, false]}, (27c1b0a5,c1b0) = {ghi = [1,2, 2, 2, 2, 2,0, 25.0, 2016-12-05 15:36:23+00:00, 0, 20, true]}*

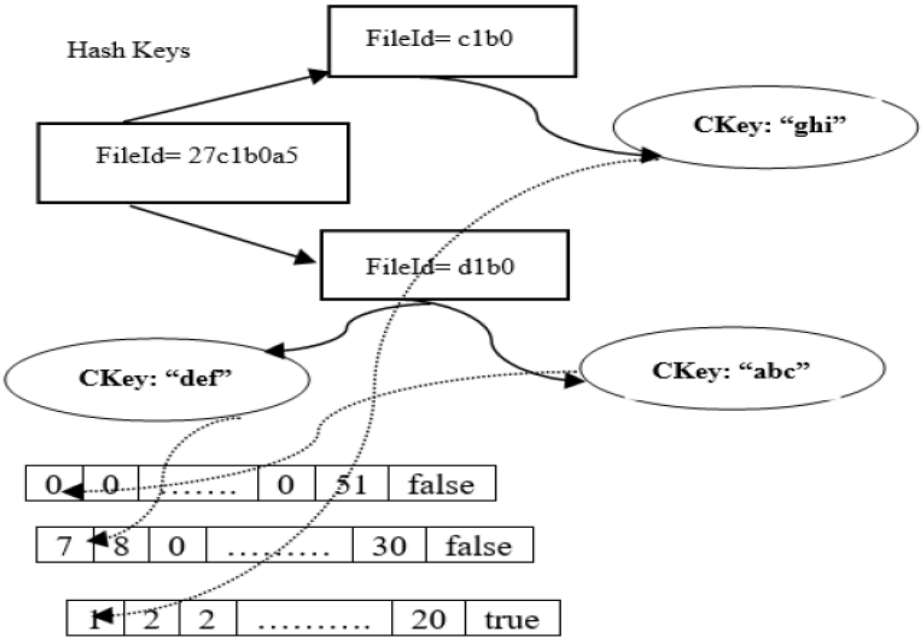


Fig. 5 An instance of the Guava Table Interface defining a file's association with its attributes

Where, List<Object> in the instance of guava table, i.e., Table < String, String, Map < String, List<Object>>> comprises of following attributes:

1. How many times the source file appeared in a bug fixing commit
2. How many times the source file appeared in a non-bug fixing commit
3. Number of lines changed in the source
4. Number of lines changed in the destination
5. Avg Number of lines changed in the commit
6. Min Number of lines changed in the commit
7. Max Number of lines changed in the commit
8. Date of the commit
9. Number of lines in a commit that has been modified.
10. Number of lines in a commit that have been deleted.
11. Is it a bug fixing commit or a clean commit?

3.4.2 Process Metrics

Our approach is based on calculating a per-file strength metric that indicates the level of dependency and co-commit frequency a file has with other files in a commit. By examining the trend of this strength over groups (henceforth called segments) of commits, we endeavor to predict the error proneness of the file in the commits of the next segment.

The hypothesis is that if the trend is upward, then this is an indication that the file will be error-prone in the next segment, while if the trend is downward, then this is an indication that the file will not be error-prone.

The strength value of a file is influenced by many factors, namely, how often a file has been committed alone or with another file, the total number of modified, added or deleted lines in a file, the time elapsed between consecutive commits of a file, number of distinct authors who have committed the file, and number of time file participated in bug fixing commit. For calculating file strength, the system requires a data structure to organize the information extracted from the GitHub repositories efficiently. This is where the Guava Tables data structure was leveraged to retrieve the values from the table with less of a computation overhead. Fig 6. depicts the data model for the evaluation of fault proneness of a file.

This section presents a detailed discussion of strength calculation, strength decay, and strength value trend (slope) calculations.

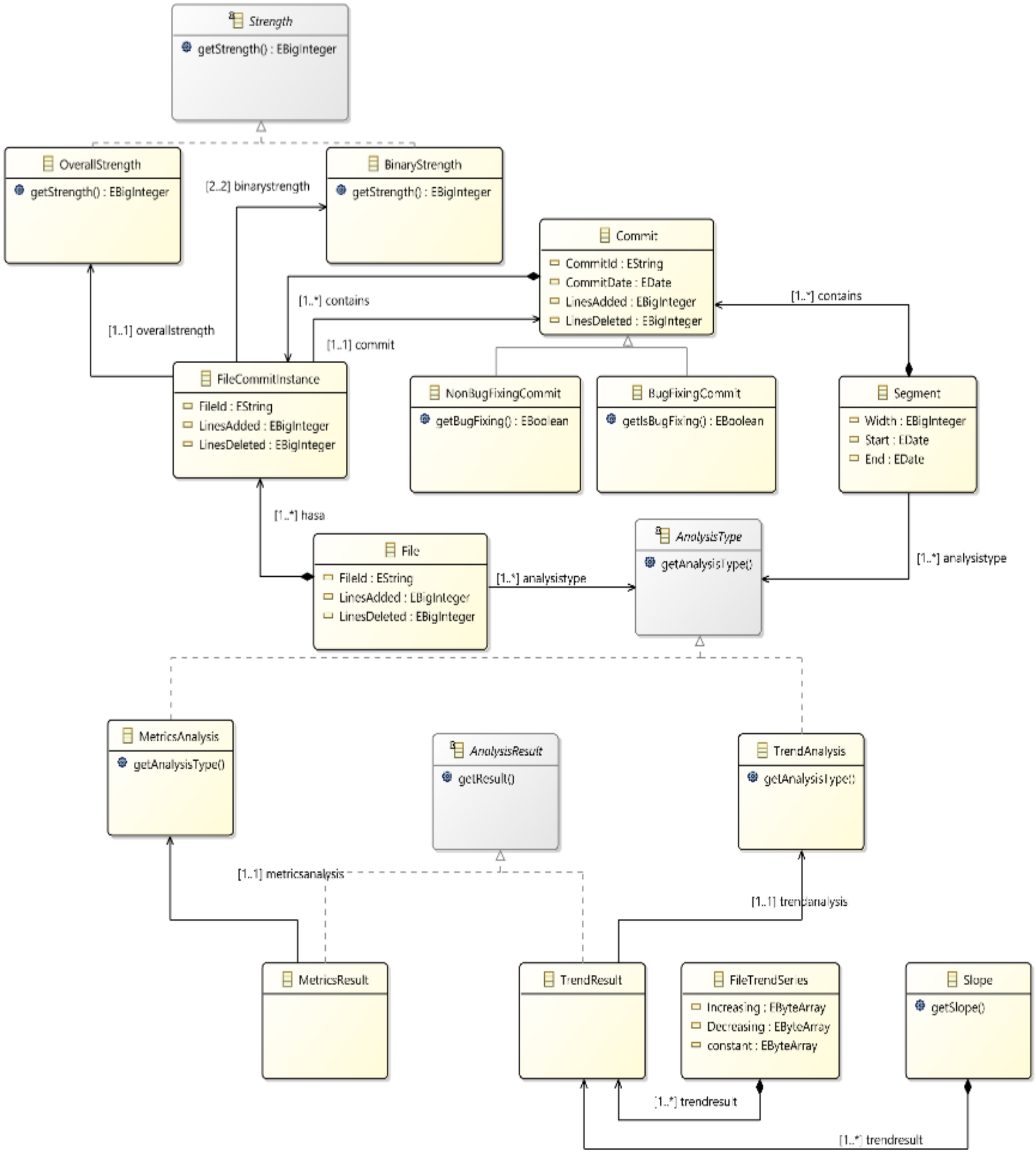


Fig. 6 Data Model for fault proneness prediction

Binary Strength (File-to-File Strength)

As described in the section 3.4.2 the strength of a file is impacted by many factors. Thus, we have selected different strategies to compare and evaluate results. This section defines different criteria used to calculate the binary strength of a file.

Strategy 1

The file-to-file strength of two files A and B in a commit cid denoted as Bin_Strength(A, B, cid) is defined as the summation of the following process-related features of files A and B:

$$Bin\_Strength\ (A, B, cid) = CT\ (A, B) + LR\ (A, cid) + LR\ (B, cid) + OC\ (A, B) \qquad (1)$$

where:

- CT (A, B) The number of times files A and B have been co-committed divided by the sum of the number of times File A and File B have been committed throughout the history of the project.
- LR (A, cid) is the number of modified lines of file A in commit cid divided by the total lines modified in the commit cid, not counting the number of modified lines in files A and B.
- LR (B, cid) is the number of modified lines of file B in commit cid divided by the total lines modified in the commit cid, not counting the number of modified lines in files A and B.
- OC (A, B) is the total number of commits where file A is committed with other files (except B/ without B), between two subsequent co-commits of files A and B divided by the number of times File A is committed so far.

Strategy 2

With careful observation of the previous Eq. (1), we found out that OC(A, B) should get subtracted from the Bin_Strength (A, B, cid) as we are calculating the Binary Strength of File A and File B, whereas OC(A, B) evaluates the value when File A is committed without File B. Thus, the new equation for calculating binary strength is:

$$Bin\_Strength\ (A, B, cid) = CT\ (A, B) + LR\ (A, cid) + LR\ (B, cid)\ -\ OC\ (A, B) \qquad (2)$$

where: CT (A, B), LR (A, cid), LR (B,cid), OC(A,B) same as defined in Eq. (1).

Strategy-3

In this strategy, we improved the Binary Strength value by including the coupling between two files by looking at its source code metrics. The coupling between two files defined as the number of methods calls between two files. Although, we cannot fetch the value of those method calls for all 21 projects mentioned in Table II, we have enough data for an initial comparison of the results. The new binary strength is thus modified to:

$$Bin\_Strength\ (A, B, cid) = CT\ (A, B) + LR\ (A, cid) + LR\ (B, cid) + CC\ (A, B, cid) - OC\ (A, B) \qquad (3)$$

where: CT (A, B) , LR (A, cid), LR (B, cid), OC (A, B), and where CC (A, B, cid) is defined below:

- *CC (A, B, cid)* is the number of calls between A to B divided by the average number of calls from A to all other co-committed files. Also, we have ignored the self-calls value.

Whenever two files A and B are co-committed in a commit cid, their individual binary (file-to-file) strength is recalculated based on the selected strategy. However, for two files A, and B, which were co-committed in a previous commit cid_p and are not co-committed in a subsequent commit $cid_{f,}$ then for each such commit $cid_{f,}$ the binary strength Bin_Strength (A, B, cid_p) is decayed to yield a new Bin_Strength (A, B, cid_f). The pair-wise decay is defined in Eq. (4) and Eq. (5) below.

Binary Strength Decay

At each commit where file A is co-committed with file B, their binary strength is recalculated using a given strategy. However, suppose file C, which has previously co-committed with file A in a commit cid_p, is not co-committed in the current commit cid. In that case, the binary strength of file A with file C Bin_Strength (A, C, cid_p) is decayed to yield a new decayed Bin_Strength(A, B, cid) value. The decay is computed as per Eq. (4) and (5).

$$\mathit{Bin\_Strength}\ (A, B, cid) = \mathit{Bin\_Strength}\ (A, B, cid_p) * \mathit{Pair\_Decay}\ (A, B, cid) \qquad (4)$$

where cid_p in Eq. (4) is the binary strength of file A with file B in the commit preceding cid, as calculated in Eq. (1) and,

$$\mathit{Pair\_Decay}\ (A, B, cid) = e^{-T\ (A,\ B,\ cid)*0.5} \qquad (5)$$

where at time of commit cid, T (A, B, cid) is the total number of commits elapsed since File A, and File B have been co-committed.

Overall Strength

We define the overall strength of a file A in a commit cid denoted as Strength(A, cid) to be the summation of the binary strengths Bin_Strength(A, B_i, cid) for all files B_1, B_2, ..., B_k which are co-committed with file A in commit cid, plus all decayed binary strengths of the file A with files C_1, C_2, ..., C_n which has been previously co-committed with file A in a previous commits cid_p and not co-committed with it in the current commit cid (see equation (6)).

$$\mathit{Strength}\ (A, cid) = \sum_{i=1}^{k} \mathit{Bin\_Strength}(A, B_i, cid) + \sum_{j=1}^{n} \mathit{Bin\_Strength}\left(A, C_j, cid_{p,j}\right) \qquad (6)$$

Overall Strength Decay

It may be the case, that a file A does not participate in subsequent commits cid_f. In this case, its overall strength should also decay. While file A does not participate in subsequent commits cid_f, we define the new decayed overall strength of file A in the current commit cid in which A does not participate as:

$$Strength\ (A, cid) = Decay\ (A, cid) * Strength\ (A, cid_p) \tag{7}$$

where cid_p is the commit file A was last committed, and

$$Decay\ (A, cid) = e^{-(NF(A)/NT(A)*Tdiff\,(A,\ cid))} \tag{8}$$

where NF(A) is the total number of commits of file A, NT(A) is the total number of commits seen until the current commit *cid*, and T_{diff} is the number of commits that have passed from current *cid* back to the time the file A was last seen committed, that is in commit cid_p.

Working Example

Let us assume the following example represented in Table III. In this example, file A is co-committed with four files B, C, D, and E, at commit X. In commit Y, file A is co-committed with two files B, D, and in commit Z, file A is co-committed with file E only. While in the commit K, file A does not participate.

Table III: Example Commits

Commit_ID	Source File	Co-Committed Files
X	A	B, C, D, E
Y	A	B, D
Z	A	E
K	B	E

Let us assume that at commit X, the initial binary strengths of file A with files B, C, D, and E are 2.0, 2.5, 1.5, and 3, respectively. Using equation 7, the overall strength of file A at commit X is 9. The decayed results for binary and overall strengths for file A are depicted in Table IV.

At commit Y, file A is co-committed with file B and D, and the binary strengths of file A with file B and file D are recalculated using equation 1. However, the binary strengths of file A with files C and E are decayed as files C and E are not co-committed with file A in commit Y. The decayed binary strength of file A with file C and E are 1.5 and 1.8, respectively, using Eq. (4) and Eq. (5), where the value of T(A, B, Y) is 1 as just one commit elapsed since the co-commit of file A with files C and E.

At commit Z, file A is co-committed with file E, and the binary strength of file A with file E is re-calculated using equation 1. However, the binary strength of file A is decayed with all the other files except for file E, as E is co-committed with A in commit Z. As per equation 3, the value of T (A, B, cid) T (A, D, Z) for files B and D are 1. The value T (A, C, Z) for file C is 2, the number of commits elapsed since files A and C were co-committed is 2.

At commit K, file A does not participate in the commit. Thus, the overall decay is applied to the value of file A in commit K using equation 5, where the value T_{diff}(A, K) is 1 as the total number of commits elapsed since file A was committed is 1. If we assume the segment width for file A is 4, then in segment [X, K] the slope is negative (↓) i.e. from Strength (A, X) = 9 to Strength (A, K) = 2.6).

Table IV: Decayed Value of an Example Commits

Commit_ID	BS(A,B,cid)	BS(A,C,cid)	BS(A,D,cid)	BS(A,E,cid)	OS(A,cid)
X	2	2.5	1.5	3	9
Y	2.1	1.5	1.7	1.8	7.1
Z	1.26	0.9	1.02	2.4	5.58
K	NA	NA	NA	NA	2.6

3.4.3 Error Proneness Identification

Segments: When we consider a system's commit timeline as a sequence of commits L = [cid_1, cid_2, …., cid_n] we can partition this sequence into an ordered list S of consecutive segments S_1, S_2, …., S_k, that is S = [S_1, S_2, …., S_k]. Each segment S_i is composed of consecutive commits cid_m, cid_{m+1}, …., $cid_{m+(p-1)}$. In this case, we say that the width of the segment is p commits long. Fig. 7 is a visual representation of a segment width of file A.

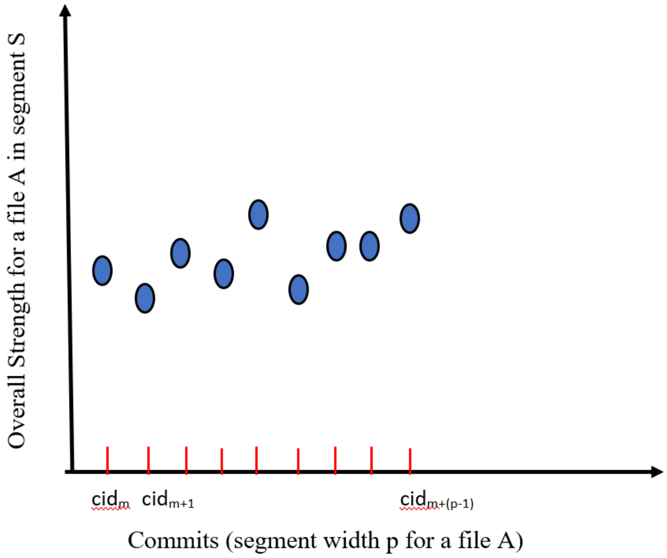


Fig. 7 Schematic representation of a Segment

In this respect, one approach would be to set the width of all segments constant and consider one width for all segments of all files. However, as different files in the system are committed at different rates throughout the system's operational life, the width of segments with respect to a given file should vary. Otherwise, files exhibiting high commit frequencies will be over represented in a segment of fixed-width p, while infrequently committed files may not have any commits in a segment of fixed width p.

In the proposed framework, we opt to vary the width of the segment per individual file. Therefore, one file, say file A, will be analyzed considering segments of width p_A, while another file, say B will be analyzed considering segments of width p_B. The computation of the segment width to be used for a given file is discussed below.

Segment Width: Let us assume for file A its n-many commits $cid_{A,1}$, $cid_{A,2}$, ..., $cid_{A,n}$, over the system's operational life so far. If we consider the time difference in hours ($cid_{A,i}$, $cid_{A,i+1}$) of commits elapsed between one commit cid_{Ai} where file A is committed and the next commit $cid_{A,i+1}$ where A is again committed, we can obtain a sequence of commit segments $cseg_{A,1,2}$ between commit $cid_{A,1}$ and $cid_{A,2,}$ $cseg_{A,2,3}$ between $cid_{A,2}$ and $cid_{A,3}$, all the way up to $cseg_{A,n-1,n}$ for commits $cid_{A,n-1}$ and $cid_{A,n}$. Once we calculated the time elapsed between two commits, we need to normalize the value by taking the modulus, i.e., the remainder of the value when divided by 24. The value 24 is selected because one day has 24 hours.

For each such sequence of values $cseg_{A,1,2}$, $cseg_{A,2,3}$, …., $cseg_{A,n-1,n}$ for file A, we compute the mean value to be the value of segment width for file A. We define the segment width $Swidth_A$ for file A as:

$$Swidth_A = mean\ (cseg_{A,1,2}, cseg_{A,2,3}, ..., cseg_{A,n-1,n}) \qquad (9)$$

That is, $Swidth_A$ is the average number of appearances of file A per commit for all files throughout the operational life of the system. As files are committed with different frequencies, the value $Swidth_A$ varies for each file A.

Segment Modelling: Based on the above framework, we consider the vector $V_A = [S_{A,1}, S_{A,2}, ..., S_{A,k}]$, of segments for file A. Internally, each segment $S_{A,i}$ is represented by the structure:

$$< Fd_i, Fcid_i, Strength\ (A, cid_j), ..., Strength\ (A, cid_{j+(k-1)}), Ld_i, Lcid_i, EP\ (A, S_{i+1}) > \qquad (10)$$

where Fd_i, corresponds to the first commit date of the segment $S_{A,i}$, $Fcid_i$ is the first commit_id of the segment $S_{A,i}$, followed by the sequence of the overall strength of file A for each of its k-many commits in segment $S_{A,i}$. The number of commits of file A in the segment and the number of occurrences of overall strength values in the segment depend entirely on the segment's width. The sequence of strength values is followed by a file followed by Ld_i, and $Lcid_i$, denoting the last commit_id and last commit date in the segment $S_{A,i}$. Finally, $EP\ (A, S_{i+1})$ is a Boolean value that contains information about whether file A has been found buggy in its subsequent segment $S_{A,i+1}$.

Segment Slope: Let us assume that for file A we have calculated its segment width as $Swidth_A$, and that in this segment we have k-many commits $cid_{A,m}$, $cid_{A,m+1}$, $cid_{A,m+(k-1)}$ of the file A, as depicted in Fig. 8. Since for each such commit $cid_{A,j}$, $j \in \{m, m + 1, ..., m+(k-1)\}$ we have computed the value Strength(A, $cid_{A,j}$), we can form a sequence of k-many values Strength(A, $cid_{A,m}$), Strength(A, $cid_{A,m+1}$), …., Strength(A, $cid_{A,m+(k-1)}$).

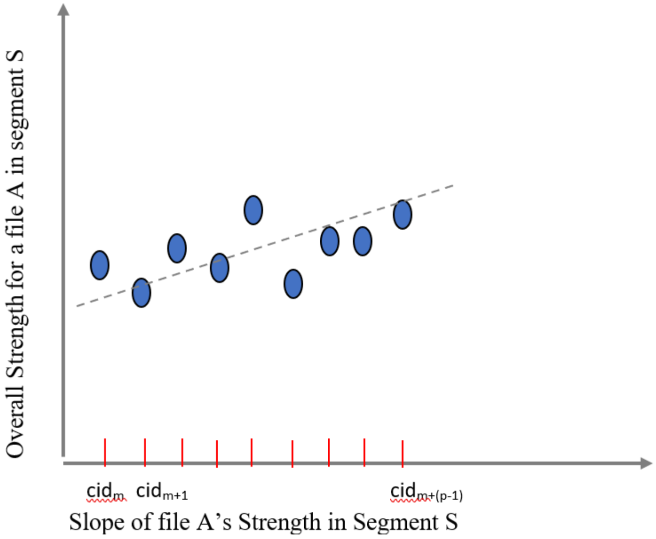


Fig. 8 Schematic representation of a Slope

We define the slope of a segment Si with respect to file A to be the slope of the line $L_{A,Si}$ that linearly interpolates all the points ($cid_{A,m}$, Strength(A, $cid_{A,m}$)), ($cid_{A,m+1}$, Strength(A, $cid_{A,m+1}$)), …., ($cid_{A,m+(k-1)}$, Strength(A, $cid_{A,m+(k-1)}$))

$$SegSlope\ (S_i, A) = Slope\ (L_A, S_i) \qquad (11)$$

Error Prediction: For our system, we have considered three different scenarios. In the first scenario, we have looked at the slope of two consecutive segments and tried to predict whether the file was buggy or not in the third segment. In the second scenario, we have looked at the slope of three consecutive segments and tried to predict whether the file was buggy or not in the fourth segment. Finally, in the third scenario, we have looked at the slope of four consecutive segments and tried to predict whether the file was buggy or not in the fifth segment. We consider as gold standard the reconciled GitHub and Bugzilla data. Fig. 9 schematic representation of an error prediction of a file A.

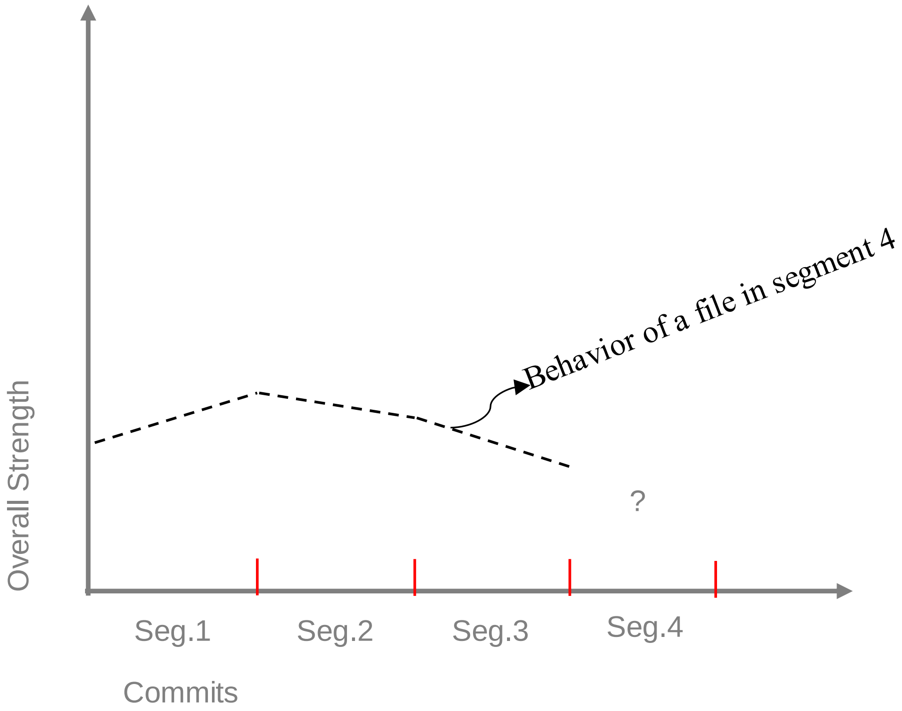


Fig. 9 Schematic representation of an error prediction

The logic by which we predict whether a file will be buggy in the next segment given its slopes for the previous segments is depicted in Table V.

Table V: Segment Combination Prediction Scenarios

2 Segments Slope	Predict 3rd	3 Segments Slope	Predict 4th	4 Segments Slope	Predict 5th
any of the two ↑	Buggy	three ↑	Buggy	four ↑	Buggy
any of the two →	Buggy	any two ↑	Buggy	any three ↑	Buggy
two ↓	Non-Buggy	any two →	Buggy	any two ↑	Buggy
NA NA	NA	any ↑ any →	Buggy	any three →	Buggy
NA NA	NA	any two ↓	Non-Buggy	any two →	Buggy
NA NA	NA	three ↓	Non-Buggy	any three ↓	Non-Buggy
NA NA	NA	NA	NA	four ↓	Non-Buggy

3.4.4 The Case Study of a File

This section outlines the case study performed at files collected from the GitHub repository and KDE.bugzilla.

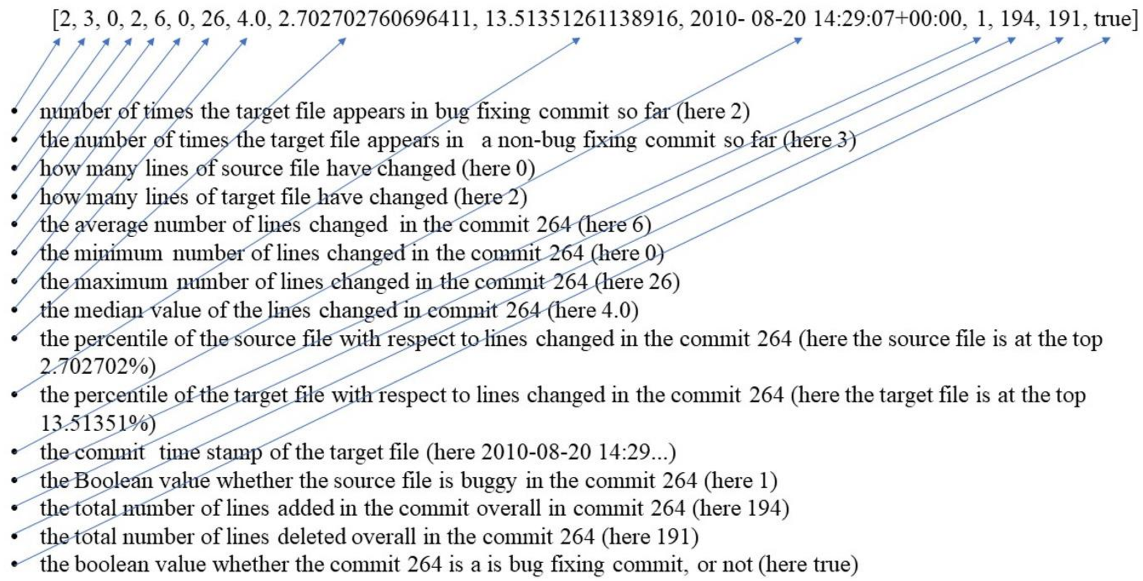
Parsing Strategy: The modeling framework uses the univocity parser [34] to parse the csv file produced by the repository data extractor. It directly maps the rows to beans using java annotations, limiting the rows to read, ignoring white spaces, and concurrent reading with optimized memory cache. Hence, it is suitable for reading large data sets as it reduces the computation load. Parsed data is organized into a HashMap so that with the help of two keys, we can easily refer to the value in the table. The value associated with the keys includes the list of all the files (objects) that belong to the same commit. The commit object contains the commit_id, and the file object contains the following attributes: file_id, whether the file is buggy or not, the number of lines added, deleted, and changed in the file. The structure depicting the files and their attributes for a given commit (here the commit ’c78ec24c74d6443ee8de768b9e5c855d7001c812’) is listed below.

CommitDetails {commit_id= ‘c78ec24c74d6443ee8de768b9e5c855d7001c812’ {= [File_Details{file_id=’27c1d7d6-3e41-11eab1b8482ae32cf5b4’,commitId=’c78ec24c74 d6443ee8d e768b9e5c855d7001c82’, addition =’86’, deletion =’81’, bugFixing=false, date = ‘2019-04-08 17:53:59+00:00’, caddition = ‘257’, cdeletion =’530’}], [FileDetails {fileId = ‘27c1d7f2-3e41-11ea-ac36482ae32cf5b4’, commitId = ’c78ec24c7 4d6443ee8de768b9e5c855d7001c812’, addition=’2’, deletion=’0’, bugFixing=false, date=’2019-040817:53:59+00:00’, addition=’257’, cdeletion = ’530’}]} (12)

In Eq. (12) above, File Id’s ‘27c1d7f2-3e41-11ea-ac36482ae32cf5b4’ and ‘a27c1d7d6-3e41-11eab1b8482ae32cf5b4’ are committed together in commit_id ‘c78ec24c74d6443ee8de-768b9e5c855d7001c812’. The lines added and deleted per file and for the commit as a whole, are depicted in attributes ‘addition’, ‘deletion’, ‘caddition’ and ‘cdeletion’, respectively. The attribute bugFixing indicates whether the commit is a bug fixing commit or not. The HashMap presented above in Eq. (12) is then further transformed to better depict important file information for any co-committed files. The resulting HashMap complies with the Guava table depicted in Fig. 5.

c49ebaac-cdcf-11ea-bed7-482ae32cf5b4={c49ebab1-cdcf-11ea-a075-482ae32cf5b4= {264=[2, 3, 0, 2, 6, 0, 26, 4.0, 2.702702760696411, 13.51351261138916, 2010-08-20 14:29:07+00:00, 1, 194, 191, true], 267=[3, 3, 0, 2, 6, 0, 26, 4.0, 2.702702760696411, 13.51351261138916, 2010-09-01 21:37:58+00:00, 1, 194, 191, true], 579=[6, 8, 21, 3, 34, 3, 132, 17.0, 63.6363639831543, 9.090909004211426, 2011-07-21 23:14:54+00:00, 0, 223, 192, false]}, c49f2fe8-cdcf-11ea-85b8-482ae32cf5b4={267=[3, 3, 0, 8, 6, 0, 26, 4.0, 2.702702760696411, 67.56756591796875, 2010-09-01 21:37:58+00:00, 1, 194, 191, true], 579=[7, 8, 21, 4, 34, 3, 132, 17.0, 63.6363639831543, 18.18181800842285, 2011-07-21 23:14:54+00:00, 0, 223, 192, false], 372=[4, 6, 4, 14, 78, 1, 520, 10.0, 27.27272, 63.63636398, 2011-08-16 22:40:54+00:00 0, 459, 473, false] . . . ..} *(13)*

The instance of the resulting Hash Table is shown in Eq. (13), where 'c49ebaac-cdcf-11ea-bed7482ae32cf5b4' is the source file_id (i.e., the file for which the binary strength is calculated), and the rest of the structure contains the details of the target files (i.e., the files with which the source file is associated with, and which are co-committed with the source file), in this case, the files with file_ids 'c49ebab1-cdcf-11ea-a075-482ae32cf5b4', and' c49f2fe8-cdcf-11ea-85b8-482ae32cf5b4'. It also contains the details of all commits where these files are committed together. For example in Eq. (13) the source file ’c49ebaac-cdcf-11ea-bed7-482ae32cf5b4’ is co-committed with target file ’c49ebab1-cdcf-11ea-a075-482ae32cf5b4’ in commits 264, 267, and 579, while the source file is co-committed also with the target file ’c49f2fe8-cdcf-11ea-85b8-482ae32cf5b4’ in commits 267 and 579.



[2, 3, 0, 2, 6, 0, 26, 4.0, 2.702702760696411, 13.51351261138916, 2010- 08-20 14:29:07+00:00, 1, 194, 191, true]

- number of times the target file appears in bug fixing commit so far (here 2)
- the number of times the target file appears in a non-bug fixing commit so far (here 3)
- how many lines of source file have changed (here 0)
- how many lines of target file have changed (here 2)
- the average number of lines changed in the commit 264 (here 6)
- the minimum number of lines changed in the commit 264 (here 0)
- the maximum number of lines changed in the commit 264 (here 26)
- the median value of the lines changed in commit 264 (here 4.0)
- the percentile of the source file with respect to lines changed in the commit 264 (here the source file is at the top 2.702702%)
- the percentile of the target file with respect to lines changed in the commit 264 (here the target file is at the top 13.51351%)
- the commit time stamp of the target file (here 2010-08-20 14:29...)
- the Boolean value whether the source file is buggy in the commit 264 (here 1)
- the total number of lines added in the commit overall in commit 264 (here 194)
- the total number of lines deleted overall in the commit 264 (here 191)
- the boolean value whether the commit 264 is a is bug fixing commit, or not (here true)

Fig. 10 Visual representation of the attributes

The values in the sequence [2, 3, 0, 2, 6, 0, 26, 4.0, 2.702702760696411, 13.51351261138916, 2010- 08-20 14:29:07+00:00, 1, 194, 191, true] indicate in order, the number of times the target file appears in bug fixing commit so far (here 2), the number of times the target file appears in a non-bug fixing commit so far (here 3), how many lines of source file have changed (here 0), how many lines of target file have changed (here 2), the average number of lines changed in the commit 264 (here 6), the minimum number of lines changed in the commit 264 (here 0), the maximum number of lines changed in the commit 264 (here 26), the median value of the lines changed in commit 264 (here 4.0), the percentile of the source file with respect to lines changed in the commit 264 (here the source file is at the top 2.702702%), the percentile of the target file with respect to lines changed in the commit 264 (here the target file is at the top 13.51351%), the commit time stamp of the target file (here 2010-08-20 14:29...), the Boolean value whether the source file is buggy in the commit 264 (here 1), the total number of lines added in the commit overall in commit 264 (here 194), the total number of lines deleted overall in the commit 264 (here 191), and the boolean value whether the commit 264 is a is bug fixing commit, or not (here true). The same structure repeats for commit 579, and later on for the second target file ’c49f2fe8-cdcf-11ea-85b8- 482ae32cf5b4’.

Calculation of Strength Scores: The hash table in Eq. (13) is processed to calculate the binary strength value between two files. The higher the strength value, the higher the association between the two files. After processing, a file's binary strength is stored in a HashMap to provide fast access for retrieval. As discussed above, the file's binary strength is decayed if it does not participate in the commit (see Eq. (4) and equation Eq. (5)).

The overall file strength of a file is computed as the sum of binary strength values of the file with all the other files committed together in the commit plus the decayed binary strength values of the file with other files which has been co-committed in the past. For example, suppose file A is co-committed with file B, file C, and file D in a commit X and with file E in a previous commit. In that case, the overall strength of file A is the sum of the binary strength of all the other files co-committed with file A in commit X, plus the decayed binary strength of file A with file E (see Eq. (6)).

The introduction of decay of the strength value of a file helps to analyze the behavior of a file over time as a file may not be committed in every single commit. In this respect, the decay of the overall strength covers the concept that a file may be dormant in the sense that it has been committed sometime in the past and not seen since then again on a commit. In this case, we should not keep considering the file as highly related to other files. Hence, we introduce the decay of the overall strength of a file (see equation 7 and equation 8).

The listing in Eq. (14) below depicts the behavior of the overall strength of an actual file obtained from our dataset where 27c1b0a4-3e41-11ea-9288-482ae32cf5b4 is the file_id, the date is the commit_time of a file, followed by the overall strength of a file. More specifically in (14) the file ’27c1b0a4-3e41-11ea- 9288-482ae32cf5b4’ has overall strength value at time ’2015- 10-11 16:42:54+00:00=2.1333332’ (corresponds to a commit) equal to 2.1333332, at time 2015-10-28 04:30:37+00:00 equal to 2.7932, and so on.

The values depicted in Eq. (14) are straight-up computations of the overall strength without applying decay, and only for the commits, the file participates. The same values but when we consider all commits in the project and applying decay are depicted Eq. (15).

27c1b0a4-3e41-11ea-9288-482ae32cf5b4= {2015-10-11 16:42:54+00:00=2.1333332, 20 15-10-28 04:30:37+00:00=2.7932, 2015-12-05 15:36:23+00:00=2.6787488, 2017 -07-07 05:53 :48+00:00=2.522886, 2017-10-11 22:28:07+00:00=6.329699, 20 18-09-27 00 :24: 19+00:00=0.9582231, 2018-11-11 22:17:32+00: 00=23.295345, 2019-04-08 17:53: 59 + 00:00=16 .482807, 2019-04-08 23:34:13+00:00=0.47562602} (14)

The listing (15) below depicts the decayed overall strength of the same file listed in (14). If we compare the corresponding values in (14) and (15), we see that in commit with timestamp 2015-10-16 01:00:56+00:00 file ’27c1b0a4- 3e41-11ea-9288-482ae32cf5b4’ is not committed (see (14), while in (15) the overall strength of the file ’27c1b0a4-3e41-11ea-9288-482ae32cf5b4’ is decayed from value 2.1333332 (see (15), to value 1.9753835 (see (14)).

27c1b0a4-3e41-11ea-9288-482ae32cf5b4= 2015-10-11 16:42:54+00:00=2.1333332, 201 5 -10-16 01:00:56+00:00 =1.9753835, 2015-10-16 01:01:07+00:00=1.8493395, 2015 -10 -26 23:16:08+00:00=1.7466254, 2015-10-28 04:30: 37+00: 00=2.7 93932, 2015 -12-05 1 5:36:23+00:00=2.6787488, 2016-08-04 00:59:09+00:00=2.267512, 2016-09-16 01:16:3 4 +00 :00=1.9533783, 2017-02-18 03:30:39+00:00=1.7080456, 2017-02-23 21:41:55 + 00 :00=1.512738, 2017-07-07 05:53:48+ 00:00=2.5 22886, 2017-09-08 10:20:09 +00:00 =2.1201575, 2017-10-11 22:28:07+00:00=6.329699, 2017-11-29 17:13:33+ 00:00 = 5.18 2319 , 2018-01-15 18:49:41+00:00=4.308705, 2018-09-26 01:25:27+00:00=3.6316864, 2018-09-27 00:24: 19+00:00= 0.9582231,2018-11-11 22:17:32+00:00= 23. 295345, 20 19 -01-0822:05:31+00:00=18.44734, 2019-02-26 03:09 :37+00:00 =14.82983, 2019-04-08 17:53: 59+00 :00= 16.48 28 07, 2019-04-08 23:34:13+00:00=0.47562602, 2019-04-13 15:24:13+ 00:00= 0.36501044, 2019-04-23 15:30:30 +00:00 =0.28438988, 2019-05-2120:37:07+00:00=0.22466981, 2019-10-07 13 :44:09+ 00: 00 =0.17976652} (15)

Segment Information: As discussed in Section 3.4.3, the segment width for file A is the mean number of commits elapsed between two consecutive commits of file A. This value depends on the frequency file A is committed over the project’s operational life. Finally, we have normalized the obtained value.

In our model, the number of overall strength instances in the segment coincides with segment width of a file. In (16) the structure of a segment for file ‘27c1fee8-3e41-11ea-8184-482ae32cf5b4’ is as follows: start commit date of the segment (here 2009-09-09 00: 43:47+00:00), the start commit_id of the segment (here 0a13d17542bbf5f714b334b9814459dfb2b2e29e), the slope value of the segment (here 0.0), the slope qualifier (i.e., ‘U’ if the segment’s slope is positive and ‘D’ if it is negative or 0) (here ‘D’), a boolean value to indicate the error proneness of the file in the subsequent segment (here ‘true’), the end commit date of the segment (here 2015-10-1601:00:56+00:00), and the end commit_id of the segment (here 01d63daa7-70abfdbc8ef 30316d595b6b72b5dcff). The instance of this segment is depicted in (16) below:

27c1fee8-3e41-11ea-8184-482ae32cf5b4=[[2009-09-0900:43:47+00:00, 0a13d17542bbf 5f714b334b9814459dfb2b2e29e, 0.0, D, true, 2015-10-16 01:00:56 +00:00, 01d63daa7-70abfdbc8ef 30316d595b6b72b5dcff]] *(16)*

Chapter 4 – Results Evaluation, Analysis, and Conclusion

This chapter discusses the results obtained by analyzing the dependency score trends collected from the designed framework to assess the health status of a file and deduce whether a file will participate in a bug fixing or non-bug fixing commit in the project's subsequent future commits.

4.1 Experimental Setup and Objective

In order to create a system that helps to predict the behavior of a file based on the process metrics, we used the following hardware and software components:

Software components:

1) We have used IntelliJ IDEA IDE to build our project.
2) We have used Java-11 as a programming language and its features like lambda, and streams to improve performance.
3) We have developed the complete system in spring boot version 2.4.0 to avoid all the manual work of writing boilerplate code, annotations, and complete XML configuration.

Hardware components:

1) A Laptop and a Desktop computer with 8GB RAM with 552GB SSD running the Windows 10 operating system.

4.1.1 Procedure to run the system

For our experiments, we have analyzed 21 open source projects mined from KDE.bugzilla and Github repositories. The systems we have analyzed are listed in Table II sorted by their size in total lines of code. For each of the systems we have applied a reconciliation process, that is to identify the most probable buggy file in a bug-fixing GitHub commit by reconciling this information with information obtained from Bugzilla repositories, and use this information as a *gold standard* for evaluation purposes. Note that a GitHub bug-fixing commit may involve several files, not all of which are the root causes of the problem. The reconciliation process aims to identify which file or files are the buggy ones in a bug-fixing commit. The reconciliation process is discussed in detail in section 3.3.3 and is part of work conducted in a related project [79]. The reconciled csv is parsed, and a strength scores (*Binary Strength*, *Overall Strength*) are calculated for each file in a GitHub commit. The calculated scores are decayed as new commits appear. Based on the dependency scores, and in particular the *Overall Strength* score we examine the trend of each file as time elapses, and we try to reason on whether this file will be participating in a bug fixing commit in the immediately following commits using reconciliation information. The designed system processes the parsed information and generates a .csv that contains the numbers associated with different scenarios, i.e., scenario 1 (i.e., the health status of a file in one segment predicts its health status on the second segment), scenario 2 (i.e., two-segments predict the third), scenario 3 (i.e., three segments predict the fourth), and scenario 4 (i.e., four segments predict the fifth).

4.2 Experimental Results

The results we have obtained are reported a) per strategy and b) per scenario. For each of the three analysis strategies presented in Section 3.4.2, we consider four different scenarios. The first scenario is to analyze the behavior of a file in one segment and predict its behavior on the next segment (note that a segment consists of a series of commits). The second scenario is to analyze the behavior of a file in two segments and predict its behavior on the next (i.e. third) segment. The third scenario is to analyze the behavior of a file in three segments and predict its behavior on the next (i.e. fourth) segment. Finally, the fourth scenario is to analyze the behavior of a file in four segments and predict its behavior on the next (i.e. fifth) segment. The following Tables starting from Table VI depict the results we have obtained form 21 open source systems and for which GitHub and Bugzilla data were available for the reconciliation process to be applied.

In these tables, a U indicates positive segment slope, while D indicates a negative slope. For example, in Table XIV and for the system *akregator,* we report that in 12.6 cases the slope of the segment is positive (i.e. upward trend) and the immediate next segment is buggy, whereas in cases 468.3 the slope of the segment is negative (i.e. downward trend) and the immediate next segment is non-buggy. These values are in decimal points because we have applied the analysis for each system 10 times starting from different segments as an initial point, and we took the average value of the results of all runs. Similarly, Table XV contains the summation of given values of all the systems defined in table VI (Strategy1 – Scenario 1) with the corresponding value of true and false. The value of true/false indicates the ratio of buggy to non-buggy segments.

In Table VIII, we report results for Strategy 1, Scenario 2 and so on. In Table VIII, UU means two consecutive segments both having a positive slope, UD means the first segment has a positive slope while the second consecutive segment has a negative slope, DU means the first segment has a negative slope while the second consecutive segment has a positive slope, and DD means the two consecutive segments has a negative slope.

4.3 Result analysis of Strategy 1

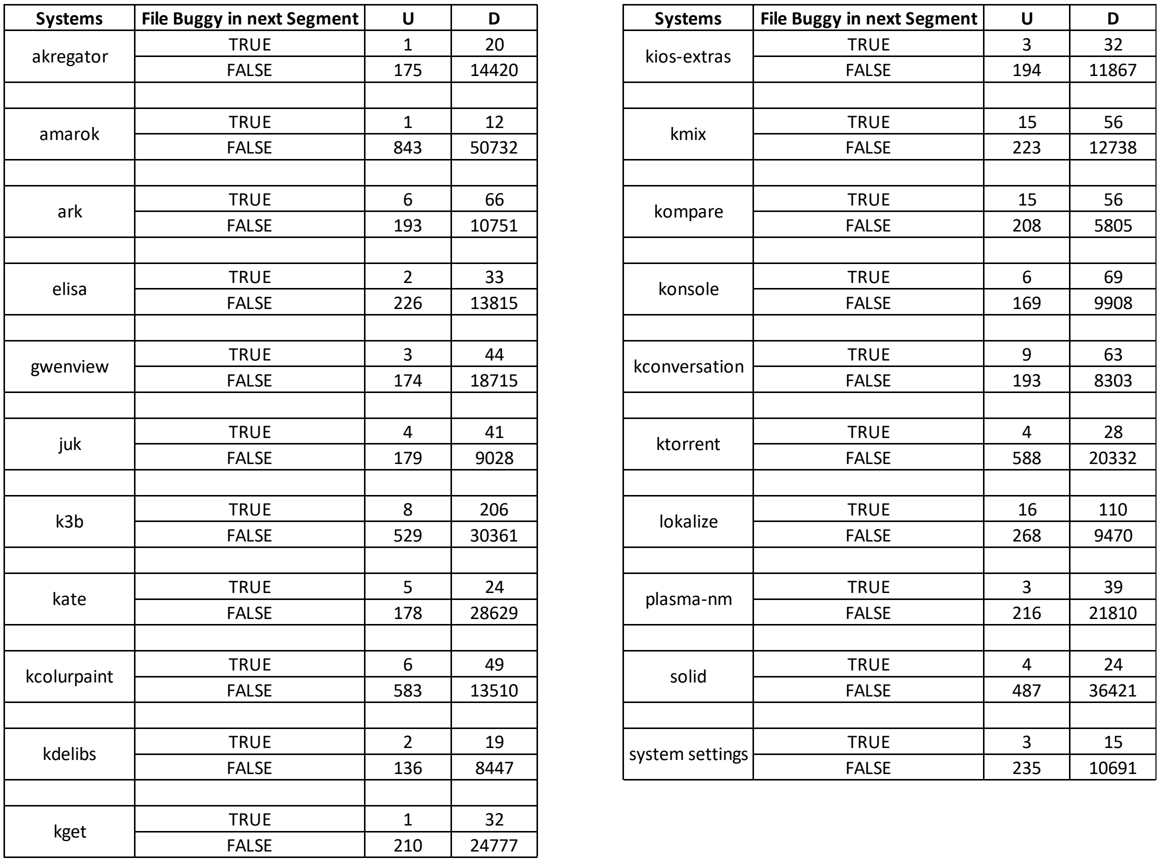
4.3.1 Strategy 1 - Scenario 1

Table VI contains the number of instances of the next segments that appear in the project as bug fixing commit and non-bug fixing commit based on the trend. As depicted in Table VI, for scenario 1 (i.e. examining the previous segment's behavior and predict the file's behavior in the next), we observe that the upwards slope is not a good predictor of whether the file will be buggy on the next segment. On the other hand, we observe that if the trend is downward then this is a very good predictor of whether the file will be non-buggy in the next segment.

In Table VII, we observe that if the trend is downward then this is a very good predictor of whether the file will be non-buggy in the next segment. Table VII depicts that 1,038 cases being buggy vs. 370,530 being non-buggy in the next segment if the trend is D. On the other hand, if the trend is U then this is not a good indicator of whether the file will be buggy in the next segment (i.e. if the trend is U in the previous segment, in 117 cases the file was buggy in the next segment, and 6207 cases the file was not buggy in the next segment)

However, the ratio of the file being buggy versus being non-buggy in the next segment is 6.7 times higher (i.e. 0. 01885/0. 002801) when the previous segment has a upward slope compared to when segment has downward slope which is an encouraging result and requires further analysis.

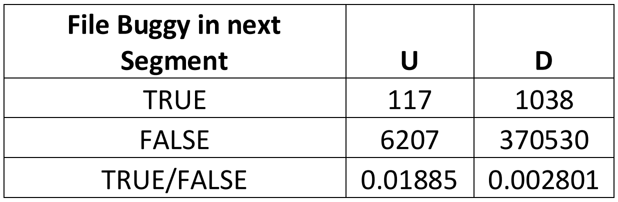
Table VI: Scenario 1- Instances of Individual Project Where One Segment Predicts Second



Systems	File Buggy in next Segment	U	D
akregator	TRUE	1	20
	FALSE	175	14420
amarok	TRUE	1	12
	FALSE	843	50732
ark	TRUE	6	66
	FALSE	193	10751
elisa	TRUE	2	33
	FALSE	226	13815
gwenview	TRUE	3	44
	FALSE	174	18715
juk	TRUE	4	41
	FALSE	179	9028
k3b	TRUE	8	206
	FALSE	529	30361
kate	TRUE	5	24
	FALSE	178	28629
kcolurpaint	TRUE	6	49
	FALSE	583	13510
kdelibs	TRUE	2	19
	FALSE	136	8447
kget	TRUE	1	32
	FALSE	210	24777

Systems	File Buggy in next Segment	U	D
kios-extras	TRUE	3	32
	FALSE	194	11867
kmix	TRUE	15	56
	FALSE	223	12738
kompare	TRUE	15	56
	FALSE	208	5805
konsole	TRUE	6	69
	FALSE	169	9908
kconversation	TRUE	9	63
	FALSE	193	8303
ktorrent	TRUE	4	28
	FALSE	588	20332
lokalize	TRUE	16	110
	FALSE	268	9470
plasma-nm	TRUE	3	39
	FALSE	216	21810
solid	TRUE	4	24
	FALSE	487	36421
system settings	TRUE	3	15
	FALSE	235	10691

Table VII: Scenario 1- Total of One Segment Predict Second



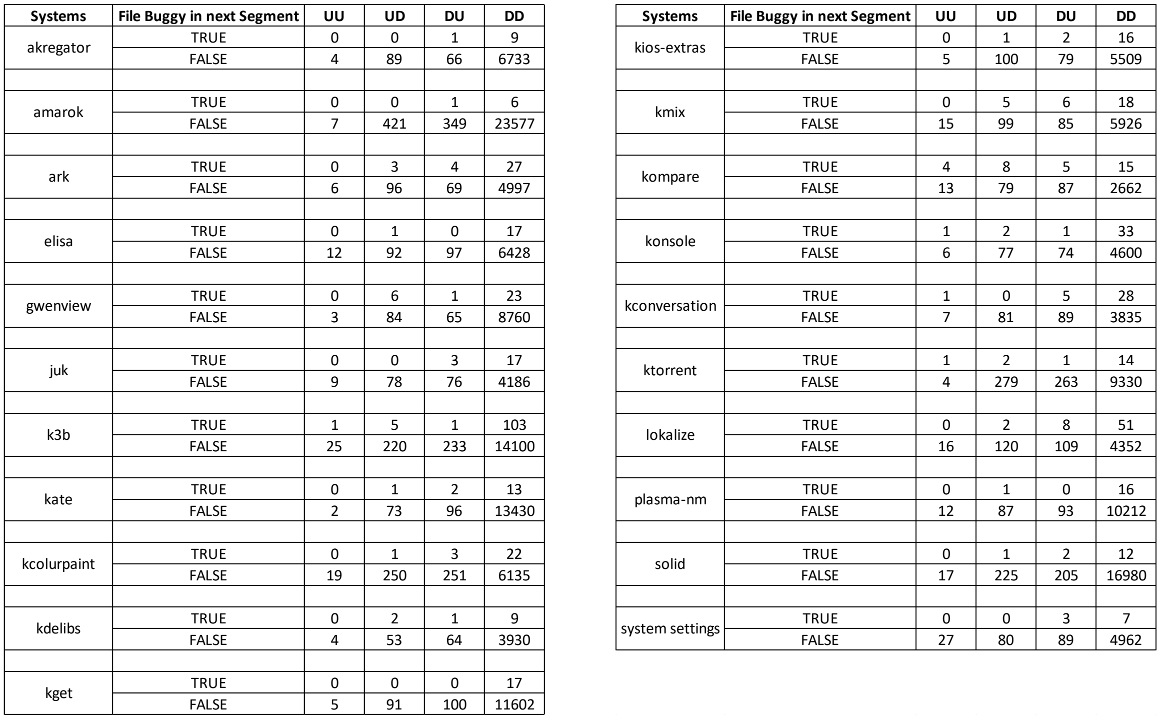
File Buggy in next Segment	U	D
TRUE	117	1038
FALSE	6207	370530
TRUE/FALSE	0.01885	0.002801

In this scenario, we were expecting the number of segments in [TRUE, U], i.e., 117 must be more as compared to number of segments in [FALSE, D] to satisfy our hypothesis. On the other side, in the case of [FALSE, D], i.e., 370530 we have achieved good numbers if compared to [TRUE, D], i.e., 1038.

4.3.2 Strategy 1 - Scenario 2

Table VIII depicts the number of instances of the next segments that appear in the project as bug fixing commit or non-bug fixing commit based on the trend. As given in Table VIII, for scenario 2 (i.e., examining the previous two segments' behavior and predict the file's behavior in the third segment), we observe that the upwards slope is not a good predictor of whether the file will be buggy on the next segment. On the other hand, we observe that if the trend is downward then this is a very good predictor of whether the file will be non-buggy in the next segment. Table IX depicts the overall behavior of systems using strategy 2 defined in Table II.

Table VIII: Scenario 2- Instances of Individual Project Where Two Segment Predicts Third



Systems	File Buggy in next Segment	UU	UD	DU	DD
akregator	TRUE	0	0	1	9
	FALSE	4	89	66	6733
amarok	TRUE	0	0	1	6
	FALSE	7	421	349	23577
ark	TRUE	0	3	4	27
	FALSE	6	96	69	4997
elisa	TRUE	0	1	0	17
	FALSE	12	92	97	6428
gwenview	TRUE	0	6	1	23
	FALSE	3	84	65	8760
juk	TRUE	0	0	3	17
	FALSE	9	78	76	4186
k3b	TRUE	1	5	1	103
	FALSE	25	220	233	14100
kate	TRUE	0	1	2	13
	FALSE	2	73	96	13430
kcolurpaint	TRUE	0	1	3	22
	FALSE	19	250	251	6135
kdelibs	TRUE	0	2	1	9
	FALSE	4	53	64	3930
kget	TRUE	0	0	0	17
	FALSE	5	91	100	11602
kios-extras	TRUE	0	1	2	16
	FALSE	5	100	79	5509
kmix	TRUE	0	5	6	18
	FALSE	15	99	85	5926
kompare	TRUE	4	8	5	15
	FALSE	13	79	87	2662
konsole	TRUE	1	2	1	33
	FALSE	6	77	74	4600
kconversation	TRUE	1	0	5	28
	FALSE	7	81	89	3835
ktorrent	TRUE	1	2	1	14
	FALSE	4	279	263	9330
lokalize	TRUE	0	2	8	51
	FALSE	16	120	109	4352
plasma-nm	TRUE	0	1	0	16
	FALSE	12	87	93	10212
solid	TRUE	0	1	2	12
	FALSE	17	225	205	16980
system settings	TRUE	0	0	3	7
	FALSE	27	80	89	4962

In Table IX, we observe that if the trend is downward then this is a very good predictor of whether the file will be non-buggy in the next segment depicts 473 cases being buggy vs. 172,246 being non-buggy in the next segment. On the other hand, if the trend is upward then this is not a good indicator of whether the file will be buggy in the next segment.

Furthermore, the ratio of the file being non-buggy versus being buggy in the next segment is 18 times higher (i.e. 0. 036/0. 002) when the previous segment has a downward slope.

Table IX: Scenario 2- Total of Two Segment Predict Third

File Buggy in next Segment	UU	UD	DU	DD
TRUE	8	41	50	473
FALSE	218	2774	2639	172246
TRUE/FALSE	0.036697	0.01478	0.018947	0.002746

4.3.3 Strategy 1 - Scenario 3

Table X contains the number of instances of the next segments that appear in the project as bug fixing commit and non-bug fixing commit based on the trend. As given in Table X, for scenario 3 (i.e., examining the previous three segments' behavior and predict the file’s behavior in the fourth segment), we observe that the upwards slope is not a good predictor of whether the file will be buggy on the next segment. On the other hand, we observe that if the trend is downward then this is a very good predictor of whether the file will be non-buggy in the next segment. Table X depicts the overall behavior of systems using strategy 3 defined in Table II.

The result of scenario 3 in Table XI, (i.e. three segments and predict the file’s behavior in the fourth) we verify the above observations i.e. the upward slope is not a good predictor of the file being buggy, while the downward slope is an excellent predictor of the file being non-buggy.

In Table XI, we observe that if the trend is downward then this is a very good predictor of whether the file will be non-buggy in the next segment (293 cases being buggy vs. 109,804 being non-buggy in the next segment). On the other hand, if the trend is upward then this is not a good indicator of whether the file will be buggy in the next segment. Here we also observe that the ratio of the file being non-buggy versus being buggy in the next segment is 26 times higher (i.e. 0.052/0.002) when the previous segment has a downward slope.

Table X: Scenario 3- Instances of individual project where three segments predict fourth

Systems	File Buggy in next Segment	UUU	UDD	DDU	DUD	UUD	DUU	UDU	DDD
akregator	TRUE	0	1	1	0	0	0	0	3
	FALSE	0	40	56	45	1	3	3	4313
amarok	TRUE	0	0	0	0	0	0	0	5
	FALSE	0	254	262	212	6	5	4	15015
ark	TRUE	0	4	0	1	0	0	0	15
	FALSE	0	53	46	57	4	3	0	3183
elisa	TRUE	0	1	0	1	0	0	1	6
	FALSE	2	64	48	52	6	6	4	4110
gwenview	TRUE	0	0	2	2	0	0	0	12
	FALSE	0	49	46	36	2	3	6	5628
juk	TRUE	0	1	0	4	0	0	0	11
	FALSE	3	35	59	34	4	4	5	2667
k3b	TRUE	0	1	2	0	0	0	2	70
	FALSE	0	146	140	146	16	7	6	8968
kate	TRUE	0	0	2	0	0	0	0	9
	FALSE	0	53	50	53	1	2	2	8639
kcolurpaint	TRUE	0	1	1	1	0	0	0	15
	FALSE	1	128	160	165	9	13	14	3815
kdelibs	TRUE	0	1	0	0	0	0	0	4
	FALSE	2	24	37	41	2	1	4	2513
kget	TRUE	0	0	0	0	0	0	0	11
	FALSE	0	52	61	65	2	4	3	7447
kios-extras	TRUE	0	0	0	2	0	0	0	8
	FALSE	1	50	55	61	1	4	4	3510
kmix	TRUE	0	5	4	3	2	0	1	12
	FALSE	1	61	43	45	9	8	7	3781
kompare	TRUE	1	2	2	3	1	1	0	5
	FALSE	2	44	43	42	10	11	9	1683
konsole	TRUE	0	3	0	0	0	0	0	21
	FALSE	0	50	55	36	6	4	2	2925
kconversation	TRUE	0	1	4	2	0	1	0	26
	FALSE	0	59	45	49	2	5	4	2420
ktorrent	TRUE	0	3	2	1	0	1	0	12
	FALSE	0	173	158	171	5	4	4	5868
lokalize	TRUE	0	0	5	2	0	0	0	25
	FALSE	1	66	68	76	10	7	5	2749
plasma-nm	TRUE	0	0	0	0	0	0	0	11
	FALSE	2	67	54	47	5	6	1	6550
solid	TRUE	0	0	3	0	0	0	0	7
	FALSE	2	123	130	136	11	10	5	10859
system settings	TRUE	0	0	1	0	0	0	0	5
	FALSE	2	46	53	39	11	14	12	3161

Table XI: Scenario 3- Total of Three Segment Predict Fourth

File Buggy in next Segment	UUU	UDD	DDU	DUD	UUD	DUU	UDU	DDD
TRUE	1	24	29	22	3	3	4	293
FALSE	19	1637	1669	1608	123	124	104	109804
TRUE/FALSE	0.052632	0.014661	0.017376	0.013682	0.02439	0.024194	0.038462	0.002668

4.3.4 Strategy 1 - Scenario 4

Table XII contains the number of instances of the next segments that appear in the project as bug fixing commit and non-bug fixing commit based on the trend. As given in Table XII, for scenario 4 (i.e., examining the previous four segments' behavior and predict the file's behavior in the fifth segment), we observe that the upwards slope is again not a good predictor of whether the file will be buggy on the next segment. On the other hand, we observe that if the trend is downward then this is a very good predictor that the file will be non-buggy in the next segment. Table XII depicts the overall behavior of systems using strategy 4 defined in Table II.

The result of scenario 4 in Table XIII, (i.e. fourth segments and predict the file's behavior in the fifth) we verify the above observations i.e. the upward slope is not a good predictor of the file being buggy, while the downward slope is an excellent predictor of the file being non-buggy.

Here we also observe that the ratio of the file being non-buggy versus being buggy in the next segment is 166 times higher (i.e. 0.333/0.002) when the previous segment has a downward slope.

Table XII: Scenario 4- Instances of individual project where fourt segments predict fifth

Systems	File Buggy in next Segment	UUUU	DUUU	UDUU	UUDU	UUUD	DDUU	UDDU	UUDD	DUUD	UDUD	DUDU	UDDD	DUDD	DDUD	DDDU	DDDD
akregator	TRUE	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	4
	FALSE	0	0	0	0	0	1	0	3	2	3	2	44	38	44	27	3485
amarok	TRUE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
	FALSE	0	0	0	1	0	3	7	3	3	3	4	206	186	227	176	12077
ark	TRUE	0	0	0	0	0	0	0	0	0	0	0	1	1	2	2	14
	FALSE	0	1	0	1	0	2	4	2	3	3	3	43	35	46	29	2562
elisa	TRUE	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	4
	FALSE	0	0	0	3	0	5	3	4	3	4	3	36	41	49	47	3315
gwenview	TRUE	0	0	0	0	0	0	0	0	1	0	0	0	1	1	0	11
	FALSE	0	0	1	0	0	2	2	0	3	6	1	33	33	45	31	4563
juk	TRUE	0	0	0	1	0	0	0	0	0	0	1	0	2	0	0	9
	FALSE	0	2	0	0	0	4	1	2	8	3	3	35	30	33	30	2149
k3b	TRUE	0	0	1	0	0	0	0	0	0	1	0	1	0	0	1	52
	FALSE	0	0	2	2	1	7	2	13	4	6	3	114	104	102	125	7235
kate	TRUE	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	6
	FALSE	0	0	0	0	1	0	2	1	1	2	0	32	45	35	54	7028
kcolurpaint	TRUE	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	11
	FALSE	0	1	1	3	1	8	12	7	4	11	15	102	108	132	114	3006
kdelibs	TRUE	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1	5
	FALSE	0	0	0	0	1	1	2	2	1	3	4	27	24	23	31	2024
kget	TRUE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
	FALSE	0	0	1	1	0	1	1	3	3	0	4	35	48	54	44	6056
kios-extras	TRUE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10
	FALSE	0	0	0	1	0	1	1	3	1	3	4	49	41	47	31	2832
kmix	TRUE	0	0	0	0	0	0	0	0	1	0	0	1	1	0	0	7
	FALSE	1	1	2	0	1	5	9	4	7	6	4	31	35	47	35	3060
kompare	TRUE	1	0	0	1	0	2	0	0	2	2	0	0	0	1	0	6
	FALSE	0	1	0	1	3	3	6	4	7	3	5	30	29	31	42	1341
konsole	TRUE	0	0	0	0	0	0	0	0	0	0	0	0	1	2	0	11
	FALSE	0	0	0	2	0	3	2	2	3	2	2	36	30	35	37	2370
kconversation	TRUE	0	0	0	0	0	1	0	0	0	0	0	1	1	0	1	14
	FALSE	0	1	0	1	0	2	4	3	7	1	4	32	41	42	34	1952
ktorrent	TRUE	0	0	0	0	0	0	0	0	0	1	0	0	1	0	1	9
	FALSE	0	0	0	0	0	2	1	3	5	6	3	140	147	134	127	4658
lokalize	TRUE	0	0	0	0	0	0	0	0	0	0	0	0	1	0	4	24
	FALSE	0	1	0	1	1	3	6	11	2	5	4	52	53	53	50	2195
plasma-nm	TRUE	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	7
	FALSE	0	1	1	0	1	5	4	4	4	2	0	32	41	50	48	5316
solid	TRUE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	5
	FALSE	0	1	2	1	1	6	0	6	3	3	4	116	106	107	102	8770
system settings	TRUE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	2
	FALSE	2	2	1	3	2	5	3	11	3	4	9	35	31	30	37	2553

Table XIII: Scenario 4- Total of four segments predicting fifth

File Buggy in next Segment	UUUU	DUUU	UDUU	UUDU	UUUD	DDUU	UDDU	UUDD
TRUE	1	0	1	2	0	3	0	0
FALSE	3	12	11	21	13	69	72	91
TRUE/FALSE	0.333333	0	0.090909	0.095238	0	0.043478	0	0

File Buggy in next Segment	DUUD	UDUD	DUDU	UDDD	DUDD	DDUD	DDDU	DDDD
TRUE	4	5	1	6	11	9	15	216
FALSE	77	79	81	1260	1246	1366	1251	88547
TRUE/FALSE	0.051948	0.063291	0.012346	0.004762	0.008828	0.006589	0.01199	0.002439

4.4 Result analysis of Strategy 2

4.4.1 Strategy 2 - Scenario 1

Table XIV helps to examine the previous segment's behavior and predict the file's behavior in the next using strategy 2. As defined in Table V, we have used different segment combinations and prediction scenarios to evaluate the results.

Table XIV contains the number of instances of the next segments that appear in the project as bug fixing commit and non-bug fixing commit based on the trend. As given in Table XIV, for scenario 1 (i.e. examining the previous segment's behavior and predict the file's behavior in the next), we observe that the upwards slope is not a good predictor of whether the file will be buggy on the next segment. On the other hand, we observe that if the trend is downward then this is a very good predictor that the file will be non-buggy in the next segment.

Table XIV: Scenario 1- Instances of individual project where one segment predicts second

Systems	File Buggy in next Segment	U	D
akregator	TRUE	12.6	54.2
	FALSE	468.3	13994.5
systemsettings	TRUE	19.4	70.2
	FALSE	399	8594.2
kompareData	TRUE	56.8	198.7
	FALSE	425	5129.1
kmix	TRUE	21.8	165.7
	FALSE	571.5	23206.2
juk	TRUE	49.3	213.7
	FALSE	366.8	8528.6
solid	TRUE	7	41.8
	FALSE	1098	25995.5
lokalize	TRUE	130.6	428
	FALSE	603.4	8522
plasma-nm	TRUE	29.2	207.4
	FALSE	823.3	20702.1
kios-extras	TRUE	12	68
	FALSE	374.7	8644.2
ark	TRUE	70.8	242.2
	FALSE	553	10034.8
kget	TRUE	21.1	163.1
	FALSE	571.1	23281.9

Systems	File Buggy in next Segment	U	D
kcolourpaint	TRUE	11	44.6
	FALSE	821.7	5116
elisa	TRUE	43.3	125.6
	FALSE	562.9	13218.2
ktorrent	TRUE	23.6	53.7
	FALSE	1194.2	14597.7
kate	TRUE	14	196.3
	FALSE	790.7	26821.4
kconversation	TRUE	14	115.6
	FALSE	371.8	5530.7
gwenview	TRUE	17.8	141.2
	FALSE	559.6	18037.4
konsole	TRUE	56.9	251.7
	FALSE	520.8	9161.2
k3b	TRUE	45.5	739
	FALSE	2266.7	26154.8
kdelibs	TRUE	3	20.3
	FALSE	122.3	4175.5
amarok	TRUE	2	10
	FALSE	1906.9	22636.1

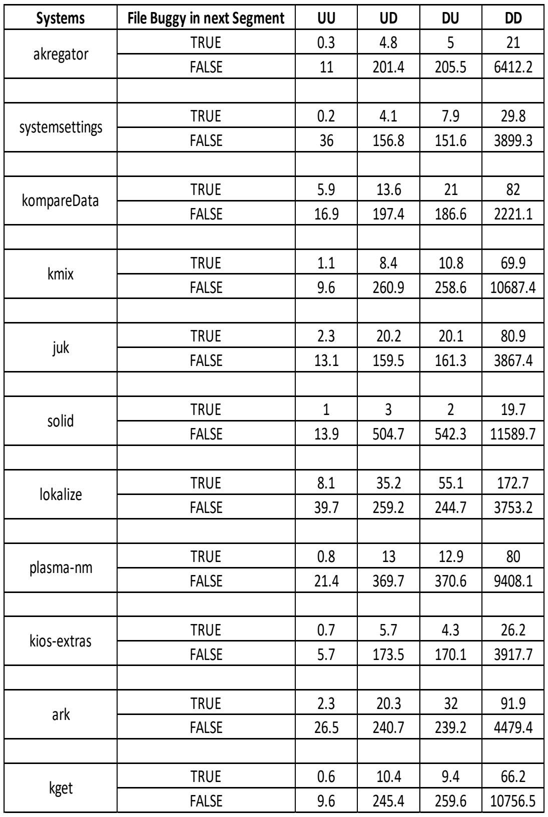
Table XV: Scenario 1- Total of One Segment Predict Second

File Buggy in next Segment	U	D
TRUE	661.7	3551
FALSE	15371.7	302082.1
TRUE/FALSE	0.043047	0.011755

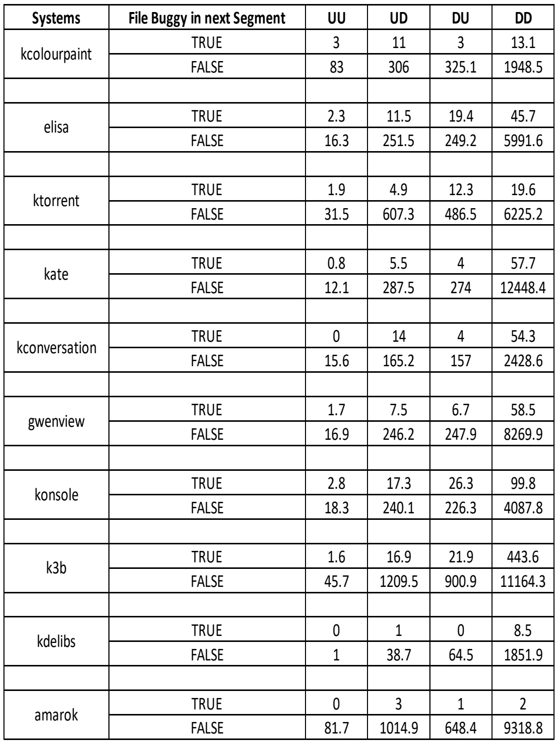
In Table XV, we observe that if the trend is downward then this is a very good predictor of whether the file will be non-buggy in the next segment depicts 3,551 cases being buggy vs. 302,081 being non-buggy in the next segment. On the other hand, if the trend is upward then this is not a good indicator of whether the file will be buggy in the next segment. Furthermore, the ratio of the file being non-buggy versus being buggy in the next segment is 3.6 times higher (i.e. 0.043/0.0117) when the previous segment has a downward slope.

4.4.2 Strategy 2 - Scenario 2

Table XVI: Scenario 2- Instances of individual project where two segments predict third



Systems	File Buggy in next Segment	UU	UD	DU	DD
akregator	TRUE	0.3	4.8	5	21
	FALSE	11	201.4	205.5	6412.2
systemsettings	TRUE	0.2	4.1	7.9	29.8
	FALSE	36	156.8	151.6	3899.3
kompareData	TRUE	5.9	13.6	21	82
	FALSE	16.9	197.4	186.6	2221.1
kmix	TRUE	1.1	8.4	10.8	69.9
	FALSE	9.6	260.9	258.6	10687.4
juk	TRUE	2.3	20.2	20.1	80.9
	FALSE	13.1	159.5	161.3	3867.4
solid	TRUE	1	3	2	19.7
	FALSE	13.9	504.7	542.3	11589.7
lokalize	TRUE	8.1	35.2	55.1	172.7
	FALSE	39.7	259.2	244.7	3753.2
plasma-nm	TRUE	0.8	13	12.9	80
	FALSE	21.4	369.7	370.6	9408.1
kios-extras	TRUE	0.7	5.7	4.3	26.2
	FALSE	5.7	173.5	170.1	3917.7
ark	TRUE	2.3	20.3	32	91.9
	FALSE	26.5	240.7	239.2	4479.4
kget	TRUE	0.6	10.4	9.4	66.2
	FALSE	9.6	245.4	259.6	10756.5



Systems	File Buggy in next Segment	UU	UD	DU	DD
kcolourpaint	TRUE	3	11	3	13.1
	FALSE	83	306	325.1	1948.5
elisa	TRUE	2.3	11.5	19.4	45.7
	FALSE	16.3	251.5	249.2	5991.6
ktorrent	TRUE	1.9	4.9	12.3	19.6
	FALSE	31.5	607.3	486.5	6225.2
kate	TRUE	0.8	5.5	4	57.7
	FALSE	12.1	287.5	274	12448.4
kconversation	TRUE	0	14	4	54.3
	FALSE	15.6	165.2	157	2428.6
gwenview	TRUE	1.7	7.5	6.7	58.5
	FALSE	16.9	246.2	247.9	8269.9
konsole	TRUE	2.8	17.3	26.3	99.8
	FALSE	18.3	240.1	226.3	4087.8
k3b	TRUE	1.6	16.9	21.9	443.6
	FALSE	45.7	1209.5	900.9	11164.3
kdelibs	TRUE	0	1	0	8.5
	FALSE	1	38.7	64.5	1851.9
amarok	TRUE	0	3	1	2
	FALSE	81.7	1014.9	648.4	9318.8

Table XVI contains the number of instances of the next segments that appear in the project as bug fixing commit and non-bug fixing commit based on the trend. As given in Table XVI, for scenario 2 (i.e., examining the previous two segments' behavior and predict the file's behavior in the third segment), we observe that the upwards slope is not a good predictor of whether the file will be buggy on the next segment or not. On the other hand, we observe that if the trend is downward then this is a very good predictor that file will be non-buggy in the next segment. Table XVII depicts the overall behavior of systems using strategy 2 defined in Table II.

Table XVII: Scenario 2- A total of two segments predict third

File Buggy in next Segment	UU	UD	DU	DD
TRUE	37.4	231.3	279.1	1543.1
FALSE	525.5	7136.1	6369.9	134727
TRUE/FALSE	0.07117	0.032413	0.043815	0.011454

As depicted in Table XVII, the downward trends are very good predictors of the file being healthy in the next segment (1,543.1 being buggy vs. 134,727 being non-buggy). Also, as before, the ratio of the file being non-buggy versus being buggy in the next segment is 6 times higher (i.e. 0.0711/0.0114) when the previous segment has a downward slope.

4.4.3 Strategy 2 - Scenario 3

Table XVIII reports the number of instances of the next segments that appear in the project as bug fixing commit segment (i.e. it contains a bug fixing commit) and non-bug fixing commit segment based on the trend. As depicted in Table XVIII, for scenario 3 (i.e., examining the previous three segments' behavior and predict the file's behavior in the fourth segment), we observe that the upwards slope is not a good predictor of whether the file will be buggy on the next segment. On the other hand, we observe that if the trend is downward then this is a very good predictor of whether the file will be non-buggy in the next segment. Table XVIII depicts the overall behavior of systems using strategy 3 define in Table II.

Table XVIII: Scenario 3- Instances of individual project where three segments predict fourth

Systems	File Buggy in next Segment	UUU	UDD	DDU	DUD	UUD	DUU	UDU	DDD
Akregator	TRUE	0.2	1.2	2.6	3.3	0.3	0	0.8	11.8
	FALSE	0.9	122.5	114.8	128.4	5.4	5.5	13.1	4028.8
systemsettings	TRUE	0	3	5.4	1.7	0.3	0.1	1.2	16.1
	FALSE	3.1	81.7	77.8	82.5	20.5	17.3	21.7	2442.9
kompare	TRUE	0.2	7.7	11.7	7.3	1.3	3.2	2.5	49.9
	FALSE	2.4	106	104.3	121.6	11.6	10.1	15.4	1324.2
kmix	TRUE	0	3.4	6.8	5.4	0.3	0.8	0.6	40.1
	FALSE	0.2	154.6	160.6	163.4	5.3	4.8	9.7	6766.7
juk	TRUE	0.2	10.8	11.6	11.3	2.1	1	4.2	40.8
	FALSE	0.7	90.2	89.9	92.3	8.4	9.2	11.1	2414.6
solid	TRUE	0	1	0	5	0	0	0	10.3
	FALSE	1	230.5	269.7	358.4	25.1	4.4	12.7	7432.8
lokalize	TRUE	1.3	19	24.6	18.7	4.6	4.5	7.2	90.9
	FALSE	3.4	148.8	142.6	151.4	21.4	18.5	17.8	2284.3
plasma-nm	TRUE	0	9.2	6.6	7.3	0.5	0.1	0.7	46.1
	FALSE	0.7	219.3	219.6	227.3	13.7	12.7	19	5866.7
kio-extras	TRUE	0	0	2	1.3	0	1.2	1.3	18.7
	FALSE	1.5	78	113.7	134.3	3.5	3.2	6.2	2426.8
ark	TRUE	0.8	13.7	18.1	12.3	1.9	0.5	3.2	46.6
	FALSE	1.5	149.1	141.9	143.5	14	13.2	12.7	2757.8
kget	TRUE	0	3.5	4.8	4.4	0.4	0.7	0.8	42.2
	FALSE	0.1	165.5	157.1	153.6	6.1	6	9.1	6790.4
kolourpaint	TRUE	2	1	2	5	5	0	0	6.2
	FALSE	14	134.9	176.6	188.9	70	24	23	1027.8
elisa	TRUE	0.1	4.2	9.4	6.4	1.1	1.7	2.9	23.2
	FALSE	1.3	149.9	149.1	151.4	11.3	10.1	11.5	3728.9
ktorrent	TRUE	0	1.1	6.4	2.9	0.1	0	1.8	12.9
	FALSE	3.8	269.7	307.2	416.2	31.3	16.4	17.3	3763.9
kate	TRUE	0	3.2	3.1	3.6	0.3	0.3	0.5	49.4
	FALSE	0.2	217.7	216.4	222	10.6	8.9	14.4	7766.9
konversation	TRUE	0	4	5	6	0	0	0	27.2
	FALSE	0	94.5	95.2	112.9	10.7	4.2	5.4	1481.6
gwenview	TRUE	0.1	3.5	4.2	4.7	0.7	0.8	0.9	32.5
	FALSE	1.3	146	146.8	143.2	11	10.7	12.9	5211.7
konsole	TRUE	0	11.7	12.3	12.7	2.2	2.2	2.6	52.2
	FALSE	0.3	141.2	141.6	137.6	9.8	10.8	14.5	2500.8
k3b	TRUE	0.5	4.5	12.7	8.9	0.9	0.4	1.5	174.5
	FALSE	0.2	582.9	714.2	683.9	16.2	22.1	64.1	6804.5
kdelibs	TRUE	0	0	1	0	1	1	0	5.5
	FALSE	0	41.5	23.3	32.4	0	0	0	1167.2
amarok	TRUE	0	0	1	0	0	0	0	1
	FALSE	10	701.9	342.3	334.1	36.3	26.2	66	5635.5

The result of scenario 3 in Table XIX, (i.e. three segments and predict the file's behavior in the fourth) we verify the above observations i.e. the upward slope is not a good predictor of the file being buggy, while the downward slope is an excellent predictor of the file being non-buggy.

Also, the ratio of the file being non-buggy versus being buggy in the next segment is 12 times higher (i.e. 0.115/0.0095) when the previous segment has a downward slope.

Table XIX: Scenario 3- Total of three segments predicting fourth

File Buggy in next Segment	UUU	UDD	DDU	DUD	UUD	DUU	UDU	DDD
TRUE	5.4	105.7	151.3	128.2	23	18.5	32.7	798.1
FALSE	46.6	4026.4	3904.7	4179.3	342.2	238.3	377.6	83624.8
TRUE/FALSE	0.11588	0.026252	0.038748	0.030675	0.067212	0.077633	0.0866	0.009544

4.4.4 Strategy 2 - Scenario 4

Table XX contains the number of instances of the next segments that appear in the project as bug fixing commit and non-bug fixing commit based on the trend. As given in Table XX, for scenario 4 (i.e., examining the previous four segments' behavior and predict the file's behavior in the fifth segment), we observe that the upwards slope is not a good predictor of whether the file will be buggy on the next segment. On the other hand, we observe that if the trend is downward then this is a very good predictor of whether the file will be non-buggy in the next segment. Table XX depicts the overall behavior of systems using strategy 4 defined in Table II.

The result of scenario 4 in Table XXI, (i.e. fourth segments and predict the file's behavior in the fifth) we verify the above observations i.e. the upward slope is not a good predictor of the file being buggy, while the downward slope is an excellent predictor of the file being non-buggy.

Also, the ratio of the file being non-buggy versus being buggy in the next segment is 6 times higher (i.e. 0.068/0.010) when the previous segment has a downward slope.

Table XX: Scenario 4- Instances of individual project where four segments predict fifth

Systems	File Buggy in next Segment	UUUU	DUUU	UDUU	UUDU	UUUD	DDUU	UDDU	UUDD	DUUD	UDUD	DUDU	UDDD	DUDD	DDUD	DDDU	DDDD
akregator	TRUE	0	0.1	0	0	0	0.1	0.3	0.2	0	0.3	0.2	0.5	1.7	2.5	2.3	8.7
	FALSE	0.3	0.3	0.3	1.3	0.5	4.7	6.8	4.1	3.7	12	8.5	90.6	98.7	95.9	98.2	3189.6
systemsettings	TRUE	0	0	0	0.2	0	0.1	0.8	0.5	1	0.1	0.3	1.5	1.4	1.2	2.6	11.9
	FALSE	0.1	2.5	4.7	4.8	1.6	12.8	10.1	11.1	12.9	12.9	11	56.3	55.9	55.4	55.1	1941.9
kompare	TRUE	0	0.2	0.2	0	0	2.8	2.4	0.8	0.5	1.5	1.4	13.1	5.3	4.9	6.3	28
	FALSE	0.3	1.8	1.2	3.1	1.4	4.1	13.7	6.5	8.4	10.9	11.4	75.5	78.8	73.8	70.4	1020.7
kmix	TRUE	0	0	0.1	0.1	0	0.8	0.8	0	0.1	0	0.6	3.8	3.9	3.8	3.4	28.4
	FALSE	0	0	0.9	0.5	0.3	4.4	6.4	3.8	5.2	8	6.8	124.1	125.5	124.3	125	5410.3
juk	TRUE	0	0.1	0.6	0.3	0.4	0.7	2.6	0.4	1.5	2.2	2	4.6	7.8	7.4	6.1	29.4
	FALSE	0.1	0.4	1.2	1.4	0.4	5.6	6.6	4.9	6.7	7.1	9.5	70.2	71.1	65.8	68.7	1903.8
solid	TRUE	0	0	0	0	0	0	0	0	0	0	0	1	3.1	2	0	6
	FALSE	0	1	2	2	0	5.4	35	4.4	8	10.7	12.3	186.4	191.8	184.8	276.3	5425.3
lokalize	TRUE	0.2	0.5	1	0.6	0.4	1.9	4.4	2.8	3	2.3	6	11.1	13.3	11.8	19.1	61.6
	FALSE	0.6	2.2	2.5	3	2.1	15.6	15.6	14.4	14	15.7	9.3	104.6	109.2	114.1	100.7	1757.4
plasma-nm	TRUE	0	0	0	0.1	0	0.3	0.7	0.6	0.3	1.2	0.3	4.7	6.4	6.6	7.1	31.6
	FALSE	0	0.4	0.7	1.4	0.6	9.1	17.9	9.4	10.7	15.5	12.6	160.5	177.2	167.3	155.9	4641.4
kio-extras	TRUE	0	0	0.2	0	0	0	0	0	0	0	0.3	0	1	1.8	2	11.1
	FALSE	0	1.1	0.4	0	0.4	4	3	0.5	5.2	6	3.2	86.5	98.7	76.1	56.4	1871.8
ark	TRUE	0.3	0.2	0	0.6	0.4	0.7	1.8	1.4	0.8	1.4	1.7	6.7	7.9	6.7	11.9	31.6
	FALSE	0.8	0.7	1.9	2.3	1.2	10.7	12.5	7.3	8.3	9.3	7.7	106.4	114.4	109.5	106.8	2151.3
kget	TRUE	0	0	0	0	0	0.1	0.7	0.1	0.4	0.6	0.3	2.6	2.9	4.2	3.5	27.9
	FALSE	0	0.4	0.8	0.1	0.3	4.1	6.1	4.7	4.6	7.6	7.4	119.7	130.1	122.3	128.7	5429.1
kolourpaint	TRUE	0	0	1	0	2	0	0	0	0	0	0	0	3	1	2	5.1
	FALSE	4	3	7	3	11	38	27.4	14	9	28	12	120.5	140.9	65.2	116.8	724.2
elisa	TRUE	0	0.1	0	0	0.4	1	0.5	0.3	0.5	0.1	2.4	3.3	4.2	5.9	8.3	18.3
	FALSE	0	0.8	0.8	0.6	0.5	8.2	11.3	7.9	9.1	9.8	9.4	110.9	112.3	117.1	113.6	2929.9
ktorrent	TRUE	0	0	0	0	0	0	0	0	0	0.9	0.1	1.2	1.1	1.2	4.7	6.7
	FALSE	0	0	0.1	0.9	2.9	7	14.2	22.7	14.8	19.3	18.8	253.3	250.1	306.1	179.8	2721
kate	TRUE	0	0.2	0	0	0	0.2	0.1	0	0.1	0.4	0.4	4.6	1.6	3.4	2.7	44.6
	FALSE	0.1	0.2	1.4	0.2	0.3	8.2	9.4	6.8	7.2	11	12.7	167.5	159.8	188.9	148.8	6157.6
konversation	TRUE	0	0	0	0	0	0	0	0	0	2	1	3	3	5	1	18.4
	FALSE	0	0	0	0	0	7.6	8	8.2	5.2	2	2.1	70.4	51.3	47.9	87.3	1102.4
gwenview	TRUE	0	0.1	0.2	0	0.1	0.7	0.4	0.8	0.3	0.6	0.6	2.9	2.7	3.8	2.8	25.3
	FALSE	0	0.9	0.9	1.1	0.6	8.2	7.8	6.1	8.5	9.1	8.9	108.4	110.9	115	115.2	4146.1
konsole	TRUE	0	0.2	0.6	0.9	0	0.5	1.8	0.9	1.5	1	2.3	8.2	10.1	6.1	9	35.5
	FALSE	0	0.2	1.6	1.9	0.6	7.1	12.9	7.1	5.6	10.7	9.7	98	101.8	109	94.3	1958
k3b	TRUE	0	0.1	0	0.5	0.6	0.1	0.4	0	0.2	2.5	1.9	19.7	4.6	6.1	8.3	223.9
	FALSE	0	1	9.2	1.2	0.2	20	38.1	15.2	7.9	43.7	40.6	460.2	446.8	576.2	337.1	4855.4
kdelibs	TRUE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4.5
	FALSE	0	0	0	0	0	1	0	0	1	0	0	25.7	35.2	22.2	28.8	926.2
amarok	TRUE	0	0	0	0	0	0	0	0	0	1	0	1	0	2	1	1
	FALSE	1	2	3	3	9	21.3	55	41.3	14	74	10	494.2	237.9	300.4	313.9	4075.5

Table XXI: Scenario 4- Total of four segments predicting fifth

File Buggy in next Segment	UUUU	DUUU	UDUU	UUDU	UUUD	DDUU	UDDU	UUDD
TRUE	0.5	1.8	3.9	3.3	4.3	10	17.7	8.8
FALSE	7.3	18.9	40.6	31.8	33.9	207.1	317.8	200.4
TRUE/FALSE	0.068493	0.095238	0.096059	0.103774	0.126844	0.048286	0.055695	0.043912

File Buggy in next Segment	DUUD	UDUD	DUDU	UDDD	DUDD	DDUD	DDDU	DDDD
TRUE	10.2	18.1	21.8	93.5	85	87.4	104.1	659.5
FALSE	170	323.3	223.9	3089.9	2898.4	3037.3	2777.8	64338.9
TRUE/FALSE	0.06	0.055985	0.097365	0.03026	0.029327	0.028776	0.037476	0.01025

4.5 Result analysis of Strategy 3

4.5.1 Strategy 3 - Scenario 1

Table XXII helps to examine the previous segment's behavior and predict the file's behavior in the next using strategy 2. As defined in Table V, we have used different segment combinations and prediction scenarios to evaluate the results.

Table XXII contains the number of instances of the next segments that appear in the project as bug fixing commit and non-bug fixing commit based on the trend. As given in Table XXII, for scenario 1 (i.e. examining the previous segment's behavior and predict the file's behavior in the next), we observe that the upwards slope is not a good predictor of whether the file will be buggy on the next segment. On the other hand, we observe that if the trend is downward then this is a very good predictor of whether the file will be non-buggy in the next segment.

In Table XXIII, we observe that if the trend is downward then this is a very good predictor of whether the file will be non-buggy in the next segment depicts 709 cases being buggy vs. 194,124 being non-buggy in the next segment. On the other hand, if the trend is upward then this is not a good indicator of whether the file will be buggy in the next segment because we have only 79 cases for the file being buggy in next segment vs. 3628 being non-buggy.

Furthermore, the ratio of the file being non-buggy versus being buggy in the next segment is 5.9 times higher (i.e. 0.02/0.003) when the previous segment has a downward slope.

Table XXII: Scenario 1- Instances of individual project where one segment predicts second

Systems	File Buggy in next Segment	U	D
akregator	TRUE	1	12
	FALSE	260	14343
ark	TRUE	22	78
	FALSE	203	10713
elisa	TRUE	1	26
	FALSE	194	13855
juk	TRUE	10	40
	FALSE	144	9058
k3b	TRUE	4	212
	FALSE	553	30335
kcolurpaint	TRUE	5	50
	FALSE	592	13501
kdelibs	TRUE	1	18
	FALSE	134	8451

Systems	File Buggy in next Segment	U	D
kget	TRUE	1	32
	FALSE	195	24792
kmix	TRUE	13	73
	FALSE	204	12742
kompare	TRUE	8	42
	FALSE	152	5882
konversation	TRUE	9	55
	FALSE	185	8319
kTorrent	TRUE	1	23
	FALSE	562	20366
plasma-nm	TRUE	3	48
	FALSE	250	21767

Table XXIII: Scenario 1- Total of One Segment Predict Second

File Buggy in next Segment	U	D
TRUE	79	709
FALSE	3628	194124
TRUE/FALSE	0.021775	0.003652

4.5.2 Strategy 3 - Scenario 2

Table XXIV contains the number of instances of the next segments that appear in the project as bug fixing commit and non-bug fixing commit based on the trend. As given in Table XXIV, for scenario 2 (i.e., examining the previous two segments' behavior and predict the file's behavior in the third segment), we observe that the upwards slope is not a good predictor of whether the file will be buggy on the next segment.

On the other hand, we observe that if the trend is downward then this is a very good predictor of whether the file will be non-buggy in the next segment. Table XXV depicts the overall behavior of systems using strategy 2 defined in Table II.

As depicted in Table XXV, the downward trends are very good predictors of the file being healthy in the next segment (297 being buggy vs. 90084 being non-buggy).

Also, as before the ratio of the file being non-buggy versus being buggy in the next segment is 14 times higher (i.e. 0.04/0.003) when the previous segment has a downward slope.

Table XXIV: Scenario 2- Instances of individual project where two segments predict third

Systems	File Buggy in next Segment	UU	UD	DU	DD
akregator	TRUE	0	1	0	5
	FALSE	16	111	104	6665
ark	TRUE	2	5	10	27
	FALSE	9	87	86	4976
elisa	TRUE	0	0	1	11
	FALSE	6	84	85	6460
juk	TRUE	2	2	3	15
	FALSE	6	72	58	4211
k3b	TRUE	0	2	2	107
	FALSE	20	243	236	14078
kcolurpaint	TRUE	0	3	2	15
	FALSE	19	283	232	6127
kdelibs	TRUE	0	0	0	8
	FALSE	3	61	62	3929

Systems	File Buggy in next Segment	UU	UD	DU	DD
kget	TRUE	0	0	0	13
	FALSE	6	90	84	11622
kmix	TRUE	2	6	2	29
	FALSE	9	82	93	5931
kompare	TRUE	0	2	5	17
	FALSE	9	66	60	2714
konversation	TRUE	0	3	3	22
	FALSE	8	83	78	3849
kTorrent	TRUE	0	0	0	9
	FALSE	9	272	247	9357
plasma-nm	TRUE	0	0	0	19
	FALSE	8	116	113	10165

Table XXV: Scenario 2- A total of two segments predict third

File Buggy in next Segment	UU	UD	DU	DD
TRUE	6	24	28	297
FALSE	128	1650	1538	90084
TRUE/FALSE	0.046875	0.014545	0.018205	0.003297

4.5.3 Strategy 3 - Scenario 3

Table XXVI reports the number of instances of the next segments that appear in the project as bug fixing commit and non-bug fixing commit based on the trend. As given in Table XXVI, for scenario 3 (i.e., examining the previous three segments' behavior and predict the file's behavior in the fourth segment), we observe that the upwards slope is not a good predictor of whether the file will be buggy on the next segment. On the other hand, we observe that if the trend is downward then this is a very good predictor of whether the file will be non-buggy in the next segment. Table XXVII depicts the overall behavior of systems using strategy 3 define in Table II.

Table XXVI: Scenario 3- Instances of individual project where three segments predict fourth

Systems	File Buggy in next Segment	UUU	UDD	DDU	DUD	UUD	DUU	UDU	DDD
akregator	TRUE	1	1	0	1	0	0	0	2
	FALSE	2	76	64	44	8	5	9	4253
ark	TRUE	0	1	5	2	0	1	0	21
	FALSE	0	58	53	47	6	8	5	3159
elisa	TRUE	0	1	0	0	0	0	0	6
	FALSE	1	55	44	49	4	3	3	4135
juk	TRUE	1	1	2	0	0	2	0	13
	FALSE	1	47	34	36	7	2	1	2680
k3b	TRUE	0	0	2	0	0	0	0	74
	FALSE	1	156	143	142	8	12	11	8955
kcolurpaint	TRUE	0	1	0	0	1	0	0	14
	FALSE	3	130	151	169	10	11	18	3815
kdelibs	TRUE	0	1	0	1	0	0	0	3
	FALSE	0	29	37	41	2	1	5	2509
kget	TRUE	0	0	0	0	0	0	0	5
	FALSE	0	48	56	61	5	1	3	7466
kmix	TRUE	0	2	2	1	1	1	2	20
	FALSE	0	50	53	55	4	5	6	3780
kompare	TRUE	0	2	3	2	1	0	0	8
	FALSE	0	42	33	36	7	4	4	1717
konversation	TRUE	0	1	2	3	0	0	2	11
	FALSE	0	48	46	43	6	5	2	2449
kTorrent	TRUE	0	1	1	1	0	0	0	3
	FALSE	1	158	165	172	3	1	6	5890
plasma-nm	TRUE	0	1	0	0	0	0	1	13
	FALSE	0	72	63	64	4	6	5	6514

The result of scenario 3 in Table XXVII, (i.e. three segments and predict the file's behavior in the fourth) we verify the above observations i.e. the upward slope is not a good predictor of the file being buggy, while the downward slope, i.e., 57322 is an excellent predictor of the file being non-buggy.

Also, the ratio of the file being non-buggy versus being buggy in the next segment is 66 times higher (i.e. 0.22/0.003) when the previous segment has a downward slope.

Table XXVII: Scenario 3- Total of three segments predicting fourth

File Buggy in next Segment	UUU	UDD	DDU	DUD	UUD	DUU	UDU	DDD
TRUE	2	13	17	11	3	4	5	193
FALSE	9	969	942	959	74	64	78	57322
TRUE/FALSE	0.222222	0.013416	0.018047	0.01147	0.040541	0.0625	0.064103	0.003367

4.5.4 Strategy 3 - Scenario 4

Table XXVIII reports the number of instances of the next segments that appear in the project as bug fixing commit and non-bug fixing commit based on the trend. As given in Table XXVIII, for scenario 4 (i.e., examining the previous four segments' behavior and predict the file’s behavior in the fifth segment), we observe that the upwards slope is not a good predictor of whether the file will be buggy on the next segment.

On the other hand, we observe that if the trend is downward then this is a very good predictor of whether the file will be non-buggy in the next segment. Table XXIX depicts the overall behavior of systems using strategy 4 defined in Table II.

The result of scenario 4 in Table XXIX, (i.e. fourth segments and predict the file’s behavior in the fifth) we verify the above observations i.e. the upward slope is not a good predictor of the file being buggy, while the downward slope is an excellent predictor of the file being non-buggy as we observed 46181 cases for the file being non-buggy vs 149 cases for file being buggy.

Table XXVIII: Scenario 4- Instances of individual project where fourt segments predict fifth

Systems	File Buggy in next Segment	UUUU	DUUU	UDUU	UUDU	UUUD	DDUU	UDDU	UUDD	DUUD	UDUD	DUDU	UDDD	DUDD	DDUD	DDDU	DDDD
akregator	TRUE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
	FALSE	2	2	0	0	1	4	5	5	4	3	6	48	40	55	47	3428
ark	TRUE	0	0	0	1	0	1	2	0	2	0	0	1	1	3	7	10
	FALSE	0	0	0	0	0	5	5	4	4	3	0	42	39	37	40	2547
elisa	TRUE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	7
	FALSE	0	1	0	1	0	1	2	3	4	4	5	43	34	31	42	3340
juk	TRUE	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	9
	FALSE	0	1	0	0	1	3	1	2	4	2	2	32	20	33	31	2170
k3b	TRUE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	53
	FALSE	0	1	1	1	1	13	6	5	10	8	4	123	117	103	108	7220
kcolurpaint	TRUE	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	9
	FALSE	1	1	4	5	0	5	18	3	6	17	12	122	101	115	100	3015
kdelibs	TRUE	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	5
	FALSE	0	0	0	0	1	0	2	2	0	2	2	29	32	28	25	2021
kget	TRUE	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	6
	FALSE	0	1	2	1	0	1	1	1	1	3	2	37	37	45	45	6070
kmix	TRUE	0	0	0	0	0	0	0	1	0	0	0	0	2	1	1	15
	FALSE	0	1	2	0	0	5	2	4	2	6	7	38	37	33	42	3059
kompare	TRUE	0	0	0	0	0	0	1	0	0	1	1	0	0	0	2	8
	FALSE	0	1	1	1	0	6	3	2	2	4	3	33	21	20	29	1382
konversation	TRUE	0	0	0	0	0	0	1	0	0	0	0	1	0	1	0	11
	FALSE	0	0	1	0	1	5	2	2	3	1	2	41	34	39	40	1957
kTorrent	TRUE	0	0	0	0	0	0	0	0	0	0	0	1	2	0	0	1
	FALSE	0	0	2	0	0	3	1	4	2	3	7	141	108	132	133	4698
plasma-nm	TRUE	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	11
	FALSE	0	0	0	0	1	2	3	5	2	4	3	53	65	51	42	5274

Table XXIX: Scenario 4- Total of four segments predicting fifth

File Buggy in next Segment	UUUU	DUUU	UDUU	UUDU	UUUD	DDUU	UDDU	UUDD
TRUE	0	0	0	1	1	1	4	1
FALSE	3	9	13	9	6	53	51	42
TRUE/FALSE	0	0	0	0.111111	0.166667	0.018868	0.078431	0.02381
File Buggy in next Segment	**DUUD**	**UDUD**	**DUDU**	**UDDD**	**DUDD**	**DDUD**	**DDDU**	**DDDD**
TRUE	2	1	1	7	6	7	15	149
FALSE	44	60	55	782	685	722	724	46181
TRUE/FALSE	0.045455	0.016667	0.018182	0.008951	0.008759	0.009695	0.020718	0.003226

4.6 Comparison of Different Strategies Results

With the implementation of Strategy-1, we have observed the peculiarity of numbers in false positive when the slope is going upward. If the file is not participating, we were expecting its slope should be downward. However, it was not the case with obtained values. It means the file is participating in the commit, but they are not buggy. It may be the case they are related files that only participate to support other files in the deployment. After a meticulous investigation of Strategy-1 and its result, we had improvised our Strategy-1 for Strategy 2. However, the results remain the same for all the analyzed scenarios. We didn’t observe any significant changes in the results. Thus, we again began our quest of improving the strategy by including source code metrics. We have included the static coupling between files that are participating in the commit to advance our metrics. After investigating 21 projects with Strategy-3 we haven’t seen any significant improvement in the numbers of false positives. After careful analysis of different strategies, we have observed, there is no significant difference between the values obtained from different strategies. However, we have observed that the three strategies behave similarly in the case of the prediction of healthy files. With the help of these three strategies, we can easily segregate or filter the healthy files in their immediate future commits as there is a very low likelihood of a file being the root cause of fault-proneness.

4.7 Conclusion

Error prone files constitute a major threat to maintaining and evolving a software system. Over the past few years, we have seen a number of approaches that utilize machine learning and source code metrics to classify a module or a file as error prone or not. In this thesis, we take a different approach and we examine quantitative trends of process metrics in order to perform the same task. More specifically, for each file in a software system and for each commit it participates in, we compute a file-to-file dependence with all its co-committed files, to which we refer as binary strength of the file. This value is then used to compute an overall strength for the file. By examining how this value behaves in a window of consecutive commits, we investigate whether we can predict the error proneness of the file in the next immediate window of commits. The approach has been applied to 21 open source systems for which we had access to both GitHub and Bugzilla repositories. The results indicate that a file's buggy behavior and its strong dependencies with other co-committed files are not good predictors of whether the file will exhibit buggy behavior in the immediate near future. On the other hand, a healthy behavior of a file is an excellent predictor of the file remaining healthy in its immediate future commits. In this respect, the technique can be directly used to filter out files with very low likelihood of being the root causes of an observed failure.

Chapter 5 – Discussion and Future Work

5.1 Discussion

The results and findings reported in the previous Chapter can be classified into two categories:

1. The first is whether the “healthy” behavior of a file (i.e., few co-commits, low interaction with other files) can predict that the file will remain fault-free in the immediate future commits.
2. The second, and most important category, is whether the “erratic” behavior of a file (i.e., several and frequent co-commits, high or fluctuating interaction with other buggy files) can predict that the file will be faulty in the immediate future commits.

Our findings indicate that it is highly probable that a file exhibiting a healthy behavior will remain healthy, and it is highly unlikely that this file will be the root cause of a failure in immediate future commits. However, for the second area, our findings suggest that there is not conclusive indication that an observed ‘erratic’ behavior of a file with respect to its commit profile and process metrics can serve as a predictor for the file been the root cause of a failure observed in the immediate future.

Currently, by analyzing the results, we can say that the proposed framework in its current form provides a very good technique for identifying and filtering out, with a high degree of probability, the healthy files of the system. Thus, we can separate files that exhibit low risk by using this system. Files with two or more consistent upward trends were expected to be classified as faulty, but our results do not support that (we are still experimenting with different strength metrics).

For this work, we have solely focused on file process metrics based on information that can be extracted from a version control repository and not the source code of the system itself. There are pros and cons in using source code-related data to identify file-to-file dependencies and compute strength values and segment slopes. The pros are that such dependencies will better depict the actual runtime interactions between files. The cons are that the prediction system will require language specific parsers and source code extractors, a task which is very complex, not to mention potentially expensive both in terms of time as well as money, for systems written in various languages or for systems written in languages for which there are no extractors available.

5.1.1 Threats to validity

We identify the following threats to validity:

1. The first threat has to do with the accuracy of the gold standard data which is used to provide us with the absolute frequency and location (in terms of commit) of bug fixing file changes. The proposed approach relies on Bugzilla records and the reconciliation process heuristic implemented in order to deduce and classify whether a file change is a bug-fixing change or not and consequently form the gold standard against which the results are obtained. If the reconciliation is not accurate or the heuristic introduces some form of bias, it may skew the results.
2. The second threat is introduced when there are errors in the measurement. As we use file churn to compute the strength of a file, a case can be made that that developers have made merges or squashed edits to a file, where in this case we cannot retrieve the exact incremental churn value.
3. The third threat has to do with the volume of the analyzed systems. We have analyzed 21 open-source systems for which we could find both Bugzilla and GitHub repositories. It would be useful to experiment with more systems, even some industrial ones, to compare these results with those we have obtained so far.
4. The proposed technique utilized process metrics in a way that failed to conclusively predict the error proneness of a file in its immediate future commits but succeeded on asserting that a file will remain healthy in its immediate future commits.
5. Most projects in the early 2000s did not use Bugzilla as a bug tracking system and GitHub as the version control repository. Thus, we are not sure whether every commit and bug is reported to GitHub and Bugzilla, respectively. Therefore, a lack of data may have skewed our results. In this respect, more systems can be considered for analysis and in order to verify the results obtained by these 21 open source systems.

5.2 Future Work

The research presented as part of our thesis can be stretched into four major research areas that are defined as follows:

1. Improving the reconciliation of data can help in improving predicting technique results since, if we are able to achieve a 100% true reconciliation rate, this can greatly contribute in improving the prediction of fault-prone/buggy files through the various frameworks set forth for this task.
2. Considering more advanced process metrics or a hybrid metrics approach can help to produce better results.
3. Lastly, we can investigate how Technical Debt measures can affect the behavior of a file, and also how such measures can contribute to a variant of the current or another similar framework in order to improve their efficacy.

5.2.1 Improving Data Reconciliation

As part of this thesis, data is collected from GitHub and KDE.bugzilla repositories and reconciled to identify which particular files were the faulty and healthy ones in a GitHub commit. However, as technology advances, software developers rely on multiple tools to report and track bugs. Thus, we need to reconcile a corpus of information originating from numerous tools to form our datasets. There are various tools available in the market that keeps track of bugs, like, Trac, Bugzilla, BugHerd, ReQtest, JIRA, Mantis, and many more. Also, there is a high probability that each bug is reported in more than one bug tracking tool. Hence, we need to track how many repositories are used in a system to track defects. Thus, we need a strategy that will encompass a more universal approach towards issue tracking systems to find the bug fixing commits/file-changes of a project. Moreover, we need to work towards a filtering tool able to perform an initial separation between fixing commits and development commits.

5.2.2 Advanced Metrics and Projects

Another proposed addition to this work is to opt for the hybrid metrics approach, i.e., a combination of process, source code, and model metrics. For this work, we have solely focused on file-to-file process dependencies based on information extracted only by the repository and not the source code of the system itself.

There are pros and cons to using source code-related information to spot file-to-file dependencies to compute strength values and segment slopes. The pros are that such dependencies will better depict the real interactions between files. The cons are that the prediction system would require language parsers and ASCII text file extractors, a difficult task for systems written in various languages or for systems written in languages that there aren't any extractors available. Moreover, the system interacts with the files which will be written in several languages. Thus, again it'll be more difficult to put in writing a parser. We can also consider the projects within which bugs are reported in JIRA, not in Bugzilla. With the assistance of this, we will compare the system efficiency and integrity.

5.2.3 Dynamic Coupling

As part of our effort, we have included only static level coupling. However, there are other dependencies pertaining to the dynamic behavior of modules that can only be inferred at run-time. For example, in object-oriented software systems, it is difficult to determine the receiver and sender classes because of polymorphism and inheritance. The actual relationship between the classes and method invocations can only be determined at run-time. Hence, it is not possible to extract the same information from the static coupling. Thus, including dynamic coupling instead of static coupling needs consideration. On the other side, the dynamic coupling is always a more difficult and expensive metric to calculate as compared to static coupling. Thus, we need to investigate projects and see the potential benefits of dynamic coupling if it outweighs the collection cost.

References

[1] Jane Radatz, Anne Geraci, and Freny Katki. Ieee standard glossary of software engineering terminology. IEEE Std, 610121990(121990):3, 1990.

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

[3] K. Dejaeger T. Verbraken and B. Baesens, “Toward comprehensible software fault prediction models using Bayesian network classifiers”. Software Engineering, IEEE Transactions, Volume 39, Issue 2, pp 237 – 257, Feb. 2013.

[4] C. Jin, S. -. Jin and J. -. Ye, "Artificial neural network-based metric selection for software fault-prone prediction model," in IET Software, vol. 6, no. 6, pp. 479-487, Dec. 2012, doi: 10.1049/iet-sen.2011.0138.

[5] M. Shepperd, D. Bowes and T. Hall, "Researcher Bias: The Use of Machine Learning in Software Defect Prediction," in IEEE Transactions on Software Engineering, vol. 40, no. 6, pp. 603-616, 1 June 2014, doi: 10.1109/TSE.2014.2322358.

[6] Kan, S. & Parrish, J. & Manlove, D. (2001). In-process metrics for software testing. IBM Systems Journal. 40. 220 - 241. 10.1147/sj.401.0220.

[7] Deb Barma, Dr. Mrinal & Kar, Nirmalya & Saha, Ashim. (2012). Static and dynamic software metrics complexity analysis in regression testing. 2012 International Conference on Computer Communication and Informatics, ICCCI 2012. 10.1109/ICCCI.2012.6158825.

[8] D. Romano, M. Pinzger, “Using Source Code Metrics to Predict Change- Prone Java Interfaces”, 27th International Conference on Software Maintenance, 2011.

[9] F. Toure, M. Badri, L. Lamontagne, “Predicting different levels of the unit testing effort of classes using source code metrics: a multiple case study on open-source software”, Innovations in Systems and Software Engineering 14, pp. 15-46, 2018.

[10] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A Systematic Literature Review on Fault Prediction Performance in Software Engineering", IEEE Trans. on Software Eng., 38(6): pp. 1276 – 1304, 2012.

[11] N. Nagappan, T. Ball, A. Zeller, "Mining metrics to predict component failures," Proceedings of the 28th international conference on Software engineering, 2006.

[12] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, K. Matsumoto. "The Impact of Automated Parameter Optimization on Defect Prediction Models". IEEE Trans. Software Eng., 45(7): pp. 683 - 711, 2018.

[13] C. Tantithamthavorn, A. E. Hassan, K. Matsumoto. "The Impact of Class Rebalancing Techniques on the Performance and Interpretation of Defect Prediction Models". IEEE Trans. Software Eng., DOI 10.1109/TSE.2018.2876537, IEEE, 2018.

[14] M. Kondo, C.P. Bezemer, Y. Kamei, A. E. Hassan, and O. Mizuno. "The Impact of feature reduction techniques on defect prediction models." Empirical Software Engineering 24: pp. 1925 - 1963, 2019.

[15] J. Jiarpakdee, C. Tantithamthavorn, A. E. Hassan. "The Impact of Correlated Metrics on the Interpretation of Defect Prediction Models." IEEE Trans. Software Eng. early access, DOI 10.1109/TSE.2019.2891758, 2019.

[16] Khoshgoftaar, T.M., Allen, E.B., Jones, W.D., Hudepohl, J.I.: Classification Tree Models of Software Quality over Multiple Releases. In: Proceedings of the 10th International Symposium on Software Reliability Engineering, pp. 116–125 (1999)

[17] N. Shakhovska, V. Yakovyna, N. Kryvinska, "An Improved Software Defect Prediction Algorithm Using Self-Organizing Maps Combined with Hierarchical Clustering and Data Preprocessing," Database and Expert Systems Applications", 2020.

[18] J. Sayyad Shirabad and T.J. Menzies, "The PROMISE Repository of Software Engineering Databases", 2005

[19] Rhmann, Wasiur & Pandey, Babita & Ansari, Gufran & Pandey, Devendra. (2019). Software fault prediction based on change metrics using Hybrid algorithms: An empirical Study. Journal of King Saud University - Computer and Information Sciences. 32.

[20] Radjenović, Danijel & Hericko, Marjan & Torkar, Richard & Živkovič, Aleš. (2013). Software fault prediction metrics: A systematic literature review. Information and Software Technology. 55. 1397–1418. 10.1016/j.infsof.2013.02.009.

[21] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models. ESE, 13(5):539–559, 2008.

[22] C. Tantithamthavorn, A. E. Hassan, K. Matsumoto. "The Impact of Class Rebalancing Techniques on the Performance and Interpretation of Defect Prediction Models". IEEE Trans. Software Eng., DOI 10.1109/TSE.2018.2876537, IEEE, 2018.

[23] Rhmann, Wasiur & Pandey, Babita & Ansari, Gufran & Pandey, Devendra. (2019). Software fault prediction based on change metrics using Hybrid algorithms: An empirical Study. Journal of King Saud University - Computer and Information Sciences. 32. 10.1016/j.jksuci.2019.03.006.

[24] S. Majumder, P. Mody, T. Menzies, "Revisiting Process versus Product Metrics: A Large-Scale Analysis," preprint, August 2020

[25] Mitchell, Tom (1997). Machine Learning. New York: McGraw Hill. ISBN 0-07-042807-7. OCLC 36417892.

[26] Koza, John & Bennett III, Forrest & Andre, David & Keane, Martin. (1998). Automated Design Of Both The Topology And Sizing Of Analog Electrical Circuits Using Genetic Programming. 10.1007/978-94-009-0279-4_9.

[27] Alpaydin, Ethem (2010) Introduction to Machine Learning. MIT Press. p. 9. ISBN 978-0-262-01243-0.

[28] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, H. Leung, Hareton,“Effort-Aware Just-in-Time Defect Prediction: Simple Unsupervised Models Could Be Better than Supervised Models,” Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016.

[29] Stehman, Stephen V. (1997). "Selecting and interpreting measures of thematic classification accuracy". Remote Sensing of Environment. 62 (1): 77–89. Bibcode:1997RSEnv.62...77S. doi:10.1016/S0034-4257(97)00083-7.

[30] Kumar, Sunil & Chong, Ilyoung. (2018). Correlation Analysis to Identify the Effective Data in Machine Learning: Prediction of Depressive Disorder and Emotion States. International Journal of Environmental Research and Public Health. 15. 2907. 10.3390/ijerph15122907.

[31] Proceedings, 2016 IEEE 8th International Workshop on Managing Technical Debt: 4 October 2016, Raleigh, North Carolina. (2016). New York: IEEE.

[32] T. Besker, A. Martini, and J. Bosch, “Managing architectural technical debt: A unified model and systematic literature review,” Journal of Systems and Software, vol. 135, pp. 1–16, 2018.

[33] J. D. Morgenthaler, M. Gridnev, R. Sauciuc and S. Bhansali, "Searching for build debt: Experiences managing technical debt at Google," 2012 Third International Workshop on Managing Technical Debt (MTD), Zurich, 2012, pp. 1-6.

[34] N. A. Ernst, "On the role of requirements in understanding and managing technical debt," 2012 Third International Workshop on Managing Technical Debt (MTD), Zurich, 2012, pp. 61-64.

[35] H. F. Soares, N. S. R. Alves, T. S. Mendes, M. Mendonça and R. O. Spínola, "Investigating the Link between User Stories and Documentation Debt on Software Projects," 2015 12th International Conference on Information Technology - New Generations, Las Vegas, NV, 2015, pp. 385-390.

[36] P. Rodríguez, A. Yagüe, P. Alarcón, and J. Garbajosa, “Some findings concerning requirements in agile methodologies,” in Bomarius, F., Oivo, M., Jaring, P., and Abrahamsson, P., editors, Product-Focused Software Process Improvement, v. 32 of Lecture Notes in Business Information Processing, Springer Berlin Heidelberg, pp. 171–184, 2009.

[37] M. Khelghatdost, Z. Hajilary, and H. Pourkarim, “Requirements in agile software development,” Life Science Journal, pp. 326–330, 2013.

[38] Ç. ALDAN and E. Demir, "Detection of Duplicate Bug Reports in Jira and Bugzilla Tools," 2020 Turkish National Software Engineering Symposium (UYMS), Istanbul, Turkey, 2020, pp. 1-4.

[39] "Github Number of Repositories". GitHub. Retrieved October 5,2020.

[40] "User search". GitHub. Retrieved January 29, 2019. Showing 40,206,691 available users.

[41] T. M. Khoshgoftaar and N. Seliya, "Tree-based software quality estimation models for fault prediction," Proceedings Eighth IEEE Symposium on Software Metrics, Ottawa, Ontario, Canada, 2002, pp. 203-214.

[42] Z.Jun,“Cost-sensitive boosting neural networks for software defect prediction,”Expert Systems with Applications, 2010.

[43] M.M. Thet Thwin, T.S. Quah, “Application of neural networks for software quality prediction using object-oriented metrics,” Journal of systems and software, 2005.

[44] T. Menzies, J. Greenwald, A. Frank,“Data mining static code attributes to learn defect predictors,”IEEE Trans. on Software Engineering, 2007.

[45] H. Peng, L. Bing, L. Xiao, C. Jun, M. Yutao, “An empirical study on software defect prediction with a simplified metric set,”Information and Software Technology, 2015.

[46] D. Gray et. al, “Using the support vector machine as a classification method for software defect prediction with static code metrics,” International Conference on Engineering Applications of Neural Networks, 2009.

[47] A. Okutan, Y. Olcay Taner, “Software defect prediction using Bayesian networks,” Empirical Software Engineering 2014.

[48] C. Catal, B. Diri, “Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem,”Information Sciences, 2009.

[49] Catal, C., & Diri, B. (2009). Investigating the effect of dataset size, metrics sets, and feature selection techniques on software fault prediction problem. Information Sciences, 179(8), 1040-1058.

[50] E. Erturk and E. A. Sezer, “A comparison of some soft computing methods for software fault prediction,” Expert Systems with Applications, 2014.

[51] K. Dejaeger, T. Verbraken, and B. Baesens, “Prediction Models Using Bayesian Network Classifiers,” vol. 39, no. 2, pp. 237–257, 2013.

[52] A. Shanthini, “Applying Machine Learning for Fault Prediction Using Software Metrics,” vol. 2, no. 6, pp. 274– 278, 2012.

[53] C. W. Yohannese and T. Li, “A Combined-Learning Based Framework for Improved Software Fault Prediction,” vol. 10, pp. 647–662, 2017.

[54] X. Yang, D. Lo, X. Xia, Y. Zhang and J. Sun, "Deep Learning for Just-in-Time Defect Prediction," 2015 IEEE International Conference on Software Quality, Reliability and Security, Vancouver, BC, 2015, pp. 17-26.

[55] Guisheng Fan, Xuyang Diao, Huiqun Yu, Kang Yang, Liqiong Chen, "Software Defect Prediction via Attention-Based Recurrent Neural Network", Scientific Programming, vol. 2019, Article ID: 6230953, 14 pages, 2019.

[56] Zazworka, Nico & Spínola, Rodrigo & Vetro, Antonio & Shull, Forrest & Seaman, Carolyn. (2013). A Case Study on Effectively Identifying Technical Debt. ACM International Conference Proceeding Series. 42-47.

[57] V.U.B. Challagulla, F.B. Bastani, I.L. Yen, R.A. Paul,"Empirical assessment of machine learning based software defect prediction techniques," International Journal on Artificial Intelligence Tools, 2008.

[58] S.Wang, X.Yao, "Using class imbalance learning for software defect prediction,"IEEE Transactions on Reliability, 2013.

[59] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, B. Murphy, "Crossproject defect prediction: a large scale experiment on data vs. domain vs. process,"Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering", 2009.

[60] A.E. Hassan, "Predicting faults using the complexity of code changes,"Proceedings of the 31st Intl. Conf. on Software Engineering. IEEE Comp. Society, 2009.

[61] S. Majumder, P. Mody, T. Menzies, "Revisiting Process versus Product Metrics: a Large Scale Analysis," preprint, August 2020

[62] Zazworka, Nico & Vetro, Antonio & Izurieta, Clemente & Wong, Sunny & Cai, Yuanfang & Seaman, Carolyn & Shull, Forrest. (2013). Comparing Four Approaches for Technical Debt Identification. SOFTWARE QUALITY JOURNAL. 22. 1-24.

[63] C. Izurieta and J. M. Bieman, "Testing Consequences of Grime Buildup in Object Oriented Design Patterns," 2008 1st International Conference on Software Testing, Verification, and Validation, Lillehammer, 2008, pp. 171-179.

[64] Xuan, Jifeng & Hu, Yan & He, Jiang. (2012). Debt-Prone Bugs: Technical Debt in Software Maintenance. International Journal of Advancements in Computing Technology. 2012, pp. 453-461.

[65] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. An in-depth study of the promises and perils of mining github. Empirical Software Engineering, 21(5):2035–2071, 2016.

[66] Z. Toth, P. Gyimesiand R. Ferenc. “A Public Bug Database of GitHub Projects and Its Application in Bug Prediction”. Springer, Intern.Conf. on Computational Sciences and Its Applications pp. 625 - 638, 2016.

[67] L. Pascarella, F. Palomba, A. Bacchelli, “Fine-grained just-in-time defect prediction”, Journal of Systems and Software 150: pp.22-36, 2018.

[68] https://figshare.com/s/23b8e010f573bb46f487.

[69] https://mvnrepository.com/artifact/com.google.guava/guava.

[70] https://mvnrepository.com/artifact/com.univocity/univocity-parsers.

[71] Graves, T. L., Karr, A. F., Marron, J. S., & Siy, H. (2000). Predicting fault incidence using software change history. IEEE Transactions on Software Engineering, 26(7), 653–661.

[72] Ostrand, T. J., & Weyuker, E. J. (2002). The distribution of faults in a large industrial software system. In ISSTA’02: Proceedings of the 2002 ACM SIGSOFT international symposium on software testing and analysis (pp. 55–64). New York, NY: ACM.

[73] Ostrand, T. J., Weyuker, E. J., & Bell, R. M. (2004). Where the bugs are. In ISSTA’04: Proceedings of the 2004 ACM SIGSOFT international symposium on software testing and analysis (pp. 86–96). New York, NY: ACM.

[74] Ostrand, T. J., Weyuker, E. J., & Bell, R. M. (2005). Predicting the location and number of faults in large software systems. IEEE Transactions on Software Engineering, 31(4), 340–355.

[75] Schro¨ter, A., Zimmermann, T., Premraj, R., & Zeller, A. (2006). If your bug database could talk. In Proceedings of the 5th international symposium on empirical software engineering, volume II: Short papers and posters (pp. 18–20).

[76] Arisholm, E., & Briand, L. C. (2006). Predicting fault-prone components in a java legacy system. In ISESE'06: Proceedings of the 2006 ACM/IEEE international symposium on empirical software engineering (pp. 8–17). New York, NY: ACM.

[77] N. Nagappan and T. Ball, ''Using software dependencies and churn metrics to predict field failures: An empirical case study,'' in Proc. 1st Int. Symp. Empirical Softw. Eng. Meas. (ESEM), Madrid, Spain, Sep. 2007, pp. 364–373

[78] Wahono, Romi & Suryana, N. & Ahmad, Sabrina. (2014). A Comparison Framework of Classification Models for Software Defect Prediction. Advanced Science Letters. 20. 1945-1950. 10.1166/asl.2014.5640.

[79] M. Grigoriou, A. Giammaria, C Brealey, "Continuous Compliance Data Science for Software Systems", IBM Center for Advanced Studies, IBM Toronto Lab, IBM Austin Lab.

[80] A. Schr¨oter, T. Zimmermann, R. Premraj, A. Zeller. If your bug database could talk. In Proceedings of the 5th International Symposium on Empirical Software Engineering, Volume II: Short Papers and Posters, pp. 18–20, 2006.

[81] T. Illes-Seifert, B. Paech. Exploring the relationship of a file's history and its fault-proneness: An empirical method and its application to open source programs. Information and Software Technology, 52(5):539–558, 2010.

[82] S. Matsumoto, Y. Kamei, A. Monden, K. ichi Matsumoto, M. Nakamura. An Analysis of Developer Metrics for Fault Prediction. PROMISE '10: Proceedings of the Sixth International Conference on Predictor Models in Software Engineering, pp. 18:1–18:9. ACM, 2010.

Curriculum Vitae

Microservices and Backend Java Developer with 4+ years of experience in developing highly scalable systems and REST API's using Advanced Java Frameworks, i.e., Spring Boot.

EXPERIENCE

Western University – London, Ontario
Teaching and Research Assistant *Jan 2020 – May 2021*

A Technique for Evaluating the Health Status of a Software Module Using Process Metrics

- **Java** Bean Components developed using **Spring Framework** for implementing business logic.
- Write the multi-table joins and conditional **SQL** queries to fetch data from the database
- Strong working knowledge of **Object-Oriented Programming** (**OOP**) techniques.
- Extensively worked with Java **Collection** classes like Set, List, HashTable, and HashMap.
- 21 projects are collected and analyzed from Bugzilla and GitHub.

Infosys Limited – Bengaluru, India
Technology Analyst *July 2018 – August 2019*

Order Management-OPL It was an upgrade project of converting existing PL/SQL utilities to Java Spring Service.

- Transformed legacy application into a suite of cloud hosted **Microservices** using **Spring Boot** and laid the groundwork for x10 traffic scale.
- Strictly adhere to coding practices and implemented at least 6 **design patterns**, like factory, façade, strategy, etc. to achieve maximum application extensibility.
- Involved in the development of **CRUD** functionality for various administrative system related tables and product components using **Spring Data JPA** and Hibernate.
- Coordinated with the team of 15 **offshore** and three **onsite** members to meet the deadlines of the Project.
- Functional and design documents are prepared for every quarterly/ monthly release to demonstrate the newly added functionalities of the system.

Infosys Limited- Hyderabad, India
Senior System Engineer *July 2015 – June 2018*

- Worked on Web Applications based on **Spring MVC** and **Microservices** architecture.
- Experienced in **Continuous Integration (CI) and Continuous Delivery (CD)** with **Jenkins**.

- Implemented the singleton instance of **Log4j** for logging the application, log of the running system to trace the errors, and certain automated routine functions.
- Good Knowledge on working with **OAUTH2.0** to provide authentication and authorization to RESTful services.
- Created test plans and test cases for unit testing, integration, and user acceptance testing. Responsible for Production support of the application.

Tools and Technologies

Technologies*: Core Java, Advanced Java, Collections, RESTful/SOAP Web-Services, Apache CXF, Design Patterns, Apache Tomcat, GitHub, Jenkins, Jira, Spring Boot, PL/SQL, Microservices, Spring Security, OAuth 2.0, Kafka, RabbitMQ*

Tools*: SOAP UI, IntelliJ IDEA, Eclipse, Jenkins, Jira, MySQL, GitHub, Postman, Oracle, Swagger*

Languages Known*: English (S, R, W) Hindi (S, R, W)*

Awards and Achievements

- Achieved a 30% reduction in memory consumption by eliminating strings via Garbage Collection Logs analysis and refactoring the code.
- The start-up time of the system is reduced by 70% by introducing hash maps for organizing data.
- Participated in the competitive programming organized by Microsoft Student Partner in 2020.

EDUCATION

Western University – London, Canada *May 2021*
Master of Computer Science

The Technological Institute of Textile & Sciences – Haryana, India *May 2015*
Bachelor of Computer Science