Method Level Change Prediction Using Change History

by

Joseph Heron

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science

in

Computer Science

University of Ontario Institute of Technology

Supervisor: Dr. Jeremy Bradbury

April 2016

Abstract

Project development requires a large amount of changes to be made to a project. Any change to a project can introduce new faults which will cost more time and money to the project owners. We're proposing a technique that will predict whether elements within a project will change in the short term future given the development history of the project. The development history is collected from source code management tools such as GitHub. The predictions are developed using the machine learning approach Support Vector Machine. To validate the results the open source software projects acra, storm, fresco, dagger, and deeplearning4j were selected and analyzed. The prediction results for the specific projects prove to be useful in certain cases to accurately predict future changes of the project.

Acknowledgements

Acknowledgements here

Contents

Abstract i

Acknowledgements ii

Contents iii

List of Figures v

List of Tables vi

Listings vii

Abbreviations viii

1 Introduction 1

1.1 Objective & Methodology . . . 2

1.2 Contributions . . . 5

1.3 Organization . . . 6

2 Literature Review 7

2.1 Data Mining . . . 7

2.1.1 Mining Open Source Software Repositories . . . 8

2.2 Machine Learning . . . 11

2.2.1 Support Vector Machines . . . 12

2.2.2 Random Forests . . . 15

2.3 Software Development Prediction . . . 18

2.3.1 Fault Prediction . . . 19

2.3.2 Change Prediction . . . 19

2.4 Change Analysis . . . 21

3 Approach 22

3.1 Storage . . . 24

3.2 Parsing . . . 27

3.3 Visualization . . . 30

3.3.1 Line Change 30
3.3.2 Method Change 32
3.3.3 Method Statement Change 34

4 Experiments 36
4.1 Sample Data 36
4.1.1 Experiment Factors 39
4.1.2 Experiment 1 41
4.1.3 Experiment 2 42
4.1.4 Experiment 3 42
4.1.5 Experiment 4 43
4.1.6 Experiment 5 43
4.1.7 Experiment 6 45
4.1.8 Experiment 7 46
4.1.9 Experiment 8 47
4.1.10 Experiment 9 47
4.2 Results 51

5 Conclusions 54

Bibliography 55

List of Figures

2.1 Network diagrams 11

3.1 GitHub Data Schema 25
3.2 Project Stats Schema 27
3.3 Newly added method 28
3.4 Removed method 28
3.5 Mixed changed method 29
3.6 Unchanged method 29
3.7 Line Change Visualization for acra 32
3.8 Method Change Visualization for acra 33
3.9 Method Statement Change Visualization for acra 35

4.1 Experiment 1 Data Sample Range 42
4.2 Experiment 6 Data Sample Range 45
4.3 Experiment 7 Data Sample Range 46

List of Tables

2.1 Open Source Software Projects 9

4.1 Experiment projects 36
4.2 Project Change Statistics 37
4.3 Project Change Statistics 2 38
4.4 Project Change Statistics 3 39
4.5 Experiment 5 Results 44
4.6 Experiment 6 Results 45
4.7 Candidate features for Support Vector Machine (SVM) model 48

Listings

Abbreviations

ANN Artifical Neural Network.

API Application Programming Interface.

CTE Common Table Expressions.

DVCS Distributed Version Control System.

IR Information Retrieval.

LD Levenshtein Distance.

MSR Mining Software Repositories.

NLD Normalized Levenshtein Distance.

OSS Open Source Software.

RF Random Forest.

SQL Structured Query Language.

SVM Support Vector Machine.

SVN Apache Subversion.

VCM Version Control Management.

VCS Version Control System.

Chapter 1

Introduction

Software has become wide spread and integrated with mobile devices providing people with a easy to use device that can always be with them. Developers are also able to create applications which can reach a wider audience through the use of application market places such as the Google Play Store, Apple App Store. In other cases such as web development system are expected to be working constantly. The applications must provide maximum availability with minimal number of issues as possible. Developing large scale applications is a difficult task that when executed incorrectly can lead to massive losses for all parties involved in the project.

During the development of a project a large number of changes will be applied to the original source code. These changes can introduce features, issues or fixes to the project. Predicting where changes will occur within the project can help developers of a project keep track of sections of the software project that need more attention. Such a case may also require a reflection on the design of the section to improve the software project.

1.1 Objective & Methodology

Mining of open source projects has been widely used to help research into various software topics relating to project development and quality assurance. This research is vital to improving the development process of software projects. By improving the development of software projects more may succeed in accomplishing their outlined goal. The project development process will take time to complete. The time it takes for the project to be completed relies on numerous factors including project scope, man power, experience. Over the course of the project development changes will be made to project. Changes can be made to almost any part of the project including design, number of developers and type of developers. These changes will in most cases have a measurable impact on the project (or at least they are intended to). In case of adding more developers the intended result may be to increase project capabilities within a shorter span of time than previously. Even with an intended result, the actual result may differ and should be measured to determine the effectiveness of a given change.

The developers of the project must therefore manage changes made to the project to ensure that the changes that are made result in the expected outcome. Keeping track of every change to a project can be difficult because of external changes which are beyond the control of the developers. However for the majority of the changes within the project they can be kept track by using a Version Control System (VCS). With proper use of a VCS the important changes made to the project will be stored. This can help keep previous releases of the software available or even help resolve a bug that was introduced in a recent change. With numerous developers a VCS can also help improve how these developers interact and share the changes that they are making. Some commonly used VCS include Git [1], Apache Subversion (SVN)[2] and Mercurial[3].

The impact of changes can be measured and provide insights into how the project changes. However first the data must be collected and then processed into a usable form. One such change that can be made to the software project would be source code changes. These changes are very fine grain since they will account for almost all functionality changes with the project. The source code level changes with a project can be map directly to functionality changes. Whether the such a change is new, fixed or removed functionality. Simply observing source code line changes can encounter a large amount of noise within which can make tracking the desired changes more difficult. Visualization of the data collected allows for a more accessible look at the data to provide potential insights.

There are two main types of projects that are developed, either closed source or open source. Open Source Software (OSS) projects will provide access to the source code, the ability to change and finally redistribute the changes. OSS is widely used in developing projects of various sizes. In these projects developers are able to contribute towards the project to complete the project to be used by a wider audience. While larger OSS projects may have a small number of developers larger projects can contain developers from numerous geographical positions contributing at different times. The development of OSS has been a focus of research related to software development since the projects are open and freely available. The authors are able to publish and use the data as they wish since it is publicly available. There are also countless OSS projects to study and investigate to apply to software projects in general.

[1] https://git-scm.com/

[2] https://subversion.apache.org/

[3] https://www.mercurial-scm.org/

Data mining is the act of collecting data from one or more sources to make use of. While the actual use of the data once collected can vary greatly from visualizing to modeling. Data can also be collected in several forms including continuous streams of data, sporadic data and one time collection. Depending on what type of data is being collected and the purpose of the collection the means of collection may also vary. Another concern related to data mining is that of big data. If a source provides a wealth of data then extra measures should be taken to manage the size of the data set. Without diligent management a data set can become unwieldy with massive overhead that is entirely avoidable.

Machine learning techniques are widely used to support the completion of difficult tasks. A machine learning algorithm is generally an algorithm that attempts to detect and mimic patterns within a data set. There are numerous different machine learning algorithms including SVM, Random Forest (RF), Artifical Neural Network (ANN). Each technique provides advantages and disadvantages depending on the purpose and the data set in use. The primary focus will be on SVM and RF since they are used as part of the proposed work.

A SVM is a tool algorithm that attempts to classify data into two different categories. This algorithm is a supervised learning technique which requires a training data set to build the model for categorizing. The training set will consists of data samples from each classification. After creating the model for a SVM new data vectors can be provided to the model and be classified into one of the two categories. The model will be constructed by attempting to linearly separate the data into two distinct groups. If the data cannot be separated linearly then the data is mapped to a higher dimension to be properly separated. While separating the data points from each category the model may reclassify data points which are more correctly fix in the other set. This feature allows for some error to be present within the training set without causing further errors.

A RF is another supervised learning technique that requires a training data set to create an prediction model. The foundation of a random forest is that of decision trees. A single decision tree creates a tree structure were each internal node in the tree represents a decision where in the final destination is the outcome. RF extend decision trees to address the tendency for decision trees to overfit the data. A RF uses several decision trees as well as a modified version of bootstrap aggregation to get more robust predictions.

The change prediction process leverages machine learning techniques to train based on the data collected through mining GitHub. Analysis of change data requires extracting data for a large set of data. The model requires a subset of the data to be used for the training of the model and another subset that is distinct from the first to actually use the model.

We propose a tool that assists in managing the development of a software project by predicting which changes will occur. This work explores leveraging change prediction of the source code using the change history to assist in the development of large scale projects. Several large

1.2 Contributions

Our contributions are in mining of OSS, visualization of a project's change history, machine learning change prediction, data collect which can be used and extended.

1.3 Organization

The remainder of the this thesis is organized into 4 more chapters. Literature Review, Approach, Experiments and finally the Conclusion. In chapter 2 more details are given related to the foundation of this work. Primarily this will cover the data that is collected for the analysis. The following chapter 3 discusses the change prediction approach from how the data is collected and stored to what methods are used for to predict change within the project. Chapter 4 reports the experiments conducted and their results. Finally the paper the conclusion summarizes the results and contributions and proposes future work to build of the thesis.

Chapter 2

Literature Review

2.1 Data Mining

Data collection from some original source provides access to a data set that may not be initially available. This data source could also be in a state that is not convenient or feasible for use without leveraging data mining techniques to transform the data to a more accessible state. The source of the data can vary greatly based on the interests for the individual(s) collecting the data. Data mining has mostly focused on single source mining and multiple data sources. Data mining in general has however also taken a large focus on data collection from software repositories which can be either single or multiple source [7,13,15,18,22,35].

Zimmermann et al. collect change the version history of a software project to predict changes that should be made in relation to an initial set of changes. The recommendations their tool provides helps point the developer to make changes that are more common within the project. As well the tool can be used to detect which changes may be missed by a developer when making changes to a project. Maletic and Collard investigate source code changes during a software project's development cycle. The changes are extracted and stored in an more easily usable form to be more easily analyzed. Canfora et al. propose a method for extracting and refining the changes made throughout the life a project to be used in more effective analyses. The changes made to a project are refined through linking lines of source code that are related. Hemmati et al. take a comprehensive review at the research related to Mining Software Repositories (MSR). Several best practices are proposed and areas of future work are identified. Hassan discusses the value of data mining from software repositories. The possible uses of the data collected can be used towards are assisting developers or managers. A benchmark data set of software project development change history is provided by Dit et al. The data set is processed to provide change request description and tracing, where changes that are requested are able to be traced to where they were implemented within the source code. The data set also provides a corpus of various key aspects of the project including files, classes and methods. The data set is targeted to be used for providing a benchmark for tools attempting to improve software maintenance tasks.

2.1.1 Mining Open Source Software Repositories

OSS generally is software that provides with the ability access the source code and make modifications to the source code. While certain licenses provide some restrictions on the ability to redistribute the software the main point of the source code of the software being freely available is key. The scope and capability of OSS projects vary greatly. Several very popular OSS projects are listed in Table 2.1.

The development of large software projects (whether OSS or not) often make use of VCS. A VCS helps the developers of the project manage the changes of the project and facilitate the collaboration between developers. A VCS will keep an current version of the project and keep track of the previous version of the project as well.

Owner	Project	Description
Mozilla	Firefox[a]	Internet Browser
Linux	Linux Kernel[b]	Operation System Kernel
VideoLAN	VLC[c]	Media Player
PostgreSQL	PostgreSQL[d]	Object-Relational Database Management System
git	git[e]	Version Control System

Table 2.1: Open Source Software Projects

[a] `https://www.mozilla.org/en-US/firefox/desktop/`
[b] `https://www.kernel.org/`
[c] `http://www.videolan.org/vlc/index.html`
[d] `http://www.postgresql.org/`
[e] `https://git-scm.com/`

This may be done through keeping a copy of each version of the project or by keeping track of all each change made to the project. SVN and git would be two examples of VCSs.

Git is a Distributed Version Control System (DVCS) and differs greatly from SVN which is a normal VCS. Git will provide the user with a complete copy of the repository that is worked on independent of network connection. The independence of each repository also allows for a repository to be developed without a centralized server. The distributed aspect of git tends to allows for easier use for all involved parities. The one main issue with a DVCS is that while decentralization is useful, developers will require some method to collaborate and communicate to transfer changes made to the repository. Therefore typically one centralized server is used to maintain communication between all interested parties.

Git has grown in popularity since it was created and is at the core of several Version Control Management (VCM) sites such as GitHub [1], BitBucket [2] and GitLab [3]. These platforms tend to be fairly supportive of OSS projects through providing their

[1] `https://github.com/`
[2] `https://bitbucket.org/`
[3] `https://gitlab.com/`

services free of charge. For example, GitHub provides unlimited public repositories completely free. While these projects do not have to be licensed with an open source license typically they will be since they are already publicly visible.

GitHub is the most popular of the VCM websites and hosts numerous very popular OSS projects including, the Linus Kernel, Swift[4] and React[5]. GitHub also provides a public Application Programming Interface (API) to allow for access to the data related to project repositories which is discussed further below. Given the popularity of GitHub for use by developers and the availability of the project data, GitHub is an obvious choice for mining project data. Especially since the goal of mining software is to capture OSS project data to both explore and test analysis methods. Publicly visible projects are also publicly accessible through the API and the majority are open source.

Git provides a simple interface to manage the repository regardless of which site is the central server. Therefore regardless which site the project resides on users can easily interact with the project as long as they know the git interface. Git in essences is a file storage for the project that keeps track of changes made to the project. A *commit* is a set of changes that a developer has made at a certain time. The developer has full control what gets committed, when it gets committed and even modified at a later date.

A branch is a series of commits that are often related. In Figure 2.1, each dot would represent a commit and a set of dots connected by the same colored lines are a branch. Branches can be considered different paths or deviations in the development from each other allowing for different versions of the project to be maintained and developed. The *master* branch is the main branch, represented with black, from

[4] `https://swift.org/`

[5] `https://facebook.github.io/react/`

which all branches usually stem from and is generally where projects are developed on. On a similar note, a *tag* is a branch that is frozen to allow for future reference. Tags are often uses to mark a significant point in the development history such as a project release. Finally, when two differently branches converge into a single dot then the two branches have been *merged*. A merge indicates that the differences between the two branches are consolidated based on the developer's discretion.

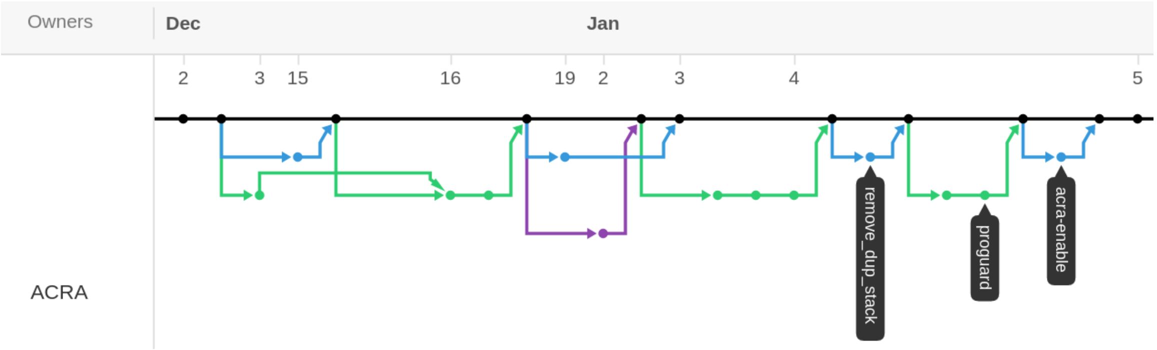


Figure 2.1: Network diagrams

A commit consists of files that have been changed, more specifically a list of *patch* files which each outline the changes made to their corresponding file. The patch file consists of a series of differences between the previous version of the file and this new version of the file. These patch files are key since they contain the actual changes made to the project and thus are the major point of interest.

2.2 Machine Learning

Machine learning is a complex method for software algorithms to attempt to determine patterns within the data. One such problem example would be an algorithm to detect certain people within an image. For an individual such a task may seem trivial however for a software system to detect it is far more difficult. Algorithms that can determine patterns and mimic them from abstract set of data is useful when such patterns are extremely complex. There are numerous algorithms which apply machine learning approaches. Each approach has both advantages or disadvantages. Some examples of machine learning algorithms are SVM, RF and ANN. The three provided examples are also commonly used for data mining [1,4,11,16,17,33]

Bhattacharyya et al. provide a detailed description of RF and SVM.

2.2.1 Support Vector Machines

A SVM is used to predict what type of change will occur based on a set of features provided. A feature is a data extracted from the project represented as a floating point number. In order to be useful a feature must in some way characterize the the category that it is assigned to. The feature must also not rely on the category that it belongs to in order to be calculated. For example, given a category of the method change within the next 5 commits or not, then the features must not rely on knowledge of future changes to the project. If the features fail to effectively characterize the category they are assigned to then the SVM may have poor predictions. It is also necessary for the features to independent of each other to not negatively affect the categorization.

SVM has been widely used for making predictions for various aspects including predicting battery charge state [2], pharmaceutical data [5], software faults [8,10,21,23,24,28], bug localization [26,28], software mutation testing score [17], financial stocks [20], credit score [16], credit card fraud [4], solar power output [34].

Malhotra reviews numerous machine learning techniques, including SVM and RF, used by various studies. The results of which outline where each approaches succeed and falls short. When using a machine learning algorithm it is imperative to use a suitable algorithm for current situation. Kim et al. outline a approach that uses a SVM to predict changes that will occur within the project. By identifying these changes the a project developer can potentially locate a bug within a change and fix it prior to being reported. Erturk and Sezer compare the performance of their proposed method, an Adaptive Neuro Fuzzy Inference System, to that of an SVM for predicting software faults. The models are trained using project metrics as well as the project's historical fault data. Zeng and Qiao use a SVM to provide short-term predict solar power output. The SVM model outperformed both an autoregressive and a neural network model. Anton et al. propose a method for predicting the state of charge of a battery using SVM model. Neuhaus et al. mines vulnerability databases and version archives determine components within the software that were vulnerable. A SVM was then used to predict other component that were also vulnerable. Several feature selection techniques have been assessed by Shivaji et al. for bug prediction methods. Features which are less useful to the prediction are removed to reduce the set to only the essential features. Kim investigates the possible use of SVM as a prediction model for financial forecasting. The model was used to predict whether the stock price would go up or down for the next day.

Bhattacharyya et al. uses RF, SVM, logistic regression to detect credit card fraud. Both RF and SVM are able to predict a large number of fraudulent credit card transactions.

SVM requires all feature data be encoded as floating point numbers. For any numerical data the conversion to floating point is trivial. However, for more complex data the conversion is a little more difficult. Categorical data can be mapped into a unique vector entry per category. For example, if a feature can be 1 of 3 options: 0, 1 or 2 then it can be converted into three entries in the feature vector. Encoding the value 2 the sub-vector of the feature set would be {0, 0, 1} where 1 indicates a field that feature is present in the data for this vector, and 0 indicates the feature is not present. Data that is in the form of a string can be converted to a floating point number by assigning a unique number for each string (similar to hashing). The one downside to this method is that the numbers corresponding to each string maintain no numerical properties. In essence the data becomes categorical, such that if *bob* is mapped to 1 and *sally* is mapped to 2 there is no relationship between 1 and 2. Ideally, this data would then be further converted using the previously described method however if the set of possible strings is large then it may be unreasonable to convert it. For example, if there are 100 possible strings then that would add 100 new entries to a single vector.

The categorization is used for the prediction, where each value of the category relates to a unique prediction type. For example, a simple binary categorization could simply 1 or 0 where 1 predicts the event will occur and 0 predicts that the event will not occur. In essence an SVM is tasked with separating a dataset into two different categories given a sample set of data that has already been categorized into two subsets. Given the categorization of the sample dataset the SVM model is trained to allow for categorization of new data. The categorization of any new vectors (that were not used for training) is called a prediction and is made by the SVM model created through the training. More specifically, the sample dataset is a dataset extracted from the target dataset. The sample dataset is then categorized based on the predetermined criteria (the prediction goal). This dataset along with the categorization for each vector in the dataset is the training dataset, and is then used to *train* the SVM model. Once the model has been trained, the SVM model is ready to be used for making classification predictions. The data for each feature can be extracted from the new dataset, allowing for the model to classify each new vector. Given that the SVM model is accurate and reliable the results can then be used towards making predictions about the dataset. For example if the classification is that of predicting change to occur within the next six commits the developer may wish to be careful with the use of the method or assess the method's quality and determine if any issues within the method need to be addressed.

A lower prediction score often relates to the data from the feature set poorly characterizing the categories. Similarly a warning will be given if the dataset is inseparate. In this case, the dataset for each category may be too similar and cannot be properly split into the two category subsets. In both cases a change to the feature set may help, whether that is a decrease or increase of features in the set. Some features are detrimental to the model, especially two features related to one another.

More details about the specific features used will be given a little later on. Features are descriptive aspects of the dataset that are classified into the predetermined categories. Since these features relate directly to the category understanding of the classification critical and can help determine which features should be used. For example for a classification of whether a change will occur within the next few commits, a useful feature may be the frequency by which a method changes within the project. Picking a descriptive feature set is paramount to providing a strong prediction of future data.

Most of this was done using database queries or user defined functions created in the database language.

2.2.2 Random Forests

RF are a popular machine learning algorithm and is used in numerous areas including predictions for software fault [12, 23, 24], software development effort [24], credit card fraud [4], database indexing [33], malware detection [1].

Malhotra provides an extensive review of studies involving machine learning to predict software faults. The results showed that RF tended to preform better than other machine learning algorithms studied. Moeyersoms et al. made use of RF and SVM as well as a few other data mining approaches to predict software faults and effort estimation. The data mining techniques are used as part of another model, ALPA rule extraction, to improve the predictions and increase traceability. Guo et al. attempt use RF to predict the fault proneness of modules within a project. The RF prediction results for the five sample projects prove more accurate to that of a logistic regression. Yu et al. attempt to use RF to determine a more effective database indexing for video data. The database index are used to provide faster searching of the database for action detection.

RFs are commonly used to on data that has been mined from some source to make predictions [1], [11], [33]. A RF leverages numerous decision trees to provide attempt to improve prediction capabilities. Therefore to fully understand a RF first an understanding of decision trees is necessary. A decision tree is a technique which will create a tree based on a data set that has been classified. Once the decision tree model is created it can be used to predict or categorize data that has not yet to be classified. In the tree model the leafs will be categorizations where as the connections between inner nodes are the decisions by which the categorizations are made.

One issue with decisions trees and more generally machine learning techniques in general is imbalanced data sets for training the model [19]. The data set used rarely provided even sample sizes of each set therefore without taking necessary pro-cautions the algorithm will bias the results. In the worse case the model will classify any input data as the larger data classification.

In case of imbalanced datasets there are several methods to help provide stronger predictions [19]. The most obvious and easiest to attempt would be to sample more data. However if the dataset in general follows this trend then some more advanced techniques can used to improve the model.

The first method would be to *undersample* larger category this will even out both of the categories. This will remove some of the input values within the dataset to reduce the set size. However if there are very few samples of the smaller category the performance will suffer as well. A second method of *oversampling* is useful in the case were the data samples are small. The input data from the smaller category is selected to be duplicated in the set to increase the size of the set. This helpful since it will increase the size of the dataset but could lead to bias based on the data selected from the smaller dataset. The selection method for which input vectors to over or under sample can be based off on the data's statistical distribution or made by random choice. Another advantage of these over and under sampling is that they can also be used together to in the case of a large disparity between the category's set size.

Another feature of RF which is used to help provide more reliable predictions is *Bootstrap Aggregation* [4]. Similar to normal sampling methods it will take the initial dataset. However rather than using the dataset as is the dataset will be uniformly sampled n times and repeated m times to create m datasets of n values. These newly created datasets will then be used to train m models. Finally, when attempting to categorize a new input data it will be given to every model and the prediction result will be aggregated to provide a more accurate results. For some machine learning methods such as SVM this method will improve the results and help with imbalanced datasets.

A RF is a collection of decisions trees trained on random samples of the initial dataset. So the RF will take an input dataset and then train m decisions trees using m randomly sampled sub-datasets of the initial dataset. This helps improve the model created and makes RFs far easier to use. As well RFs have a feature that determines the importance of each feature is assessed during the training of the model [4]. The importance outlines the quality of each feature in providing the prediction [31]. Therefore in order to properly understand the feature importance the accuracy, precision and recall of the model should be determined by running a test dataset to determine the quality of the model.

2.3 Software Development Prediction

The development of large scale projects can take a long time and involve a huge time investment from the developers. The development of the project will cause for the developers to make changes to projects. Changes made to a project may introduce new faults, increase functionality and fix previous problems. Therefore changes to a project can be both positive or negative. The developers of a project must control how a project is changed to attempt to limit the number of negative changes and increase the positive changes. Beyond ensuring that the project is developed correctly the developers typically have a limited amount of time to spend on the project and therefore must allocate their time wisely. Software development prediction models are used to help developers allocate their time more effectively. For example a developer may have a list of features that should be added to the project. However implementing the most fundamental features first will help ensure that these features are more likely to be completed.

Software development prediction contains numerous areas of study which generally attempt to improve projects by focusing on their development and providing feedback to the developers through predictions. Some of these areas include predicting: fault detection [25,27,29,30], mutation score [17], software changes [3,6,9,14,18,32]. While there may be a large overlap in the objective for these studies often they will vary in what is used to make the prediction.

2.3.1 Fault Prediction

Fault prediction is a key area of study for software development since the goal is to provide insight into where issues within the project are located. Identifying these areas can be very beneficial to the developers in saving time from searching for bugs. Rather the developers are able to use their time on fixing those issues. Therefore accurate identification of faulty code improves both development efficiency and software product quality. In order to predict these faults studies used one or more of the following; change metrics [25,27,29], code metrics [25,30], defect history [29], software dependencies [27].

Fault predictions using static and change metrics are studied by Moser et al. The change metrics used outperformed the static metrics in accuracy, and recall. Sisman and Kak alternatively look specifically at change metrics using Information Retrieval (IR) framework to provide the predictions. The prediction framework also uses a time sensitive factor to bias towards more recent changes for predictions. Nagappan and Ball attempted to predict post release project failures for commercial projects. These predictions were done using a software dependency analysis as well as using churn metrics from the project's development. Their method proved to be capable in predicting these failures providing an ability to mitigate these failures from occurring.

2.3.2 Change Prediction

Software projects will have faults within the project especially during the development phase. A project in its early stages may not meet the full set of functionality since it has not been completed yet. Since the development team will known that such features are not yet implemented these faults or fails are not a huge concern. Rather faults that are unknown to the developer team are far more serious. Such cases as a feature was thought to be implemented correct but was not or a feature implementation breaks other features. In both those cases changes made to the project cause the fault to be revealed. Changes to the project are the means by which all development occurs. The ability to analyze and predict changes within a project could give deep insights into the development of a project. A large amount of research as focused on predictions of changes based on changes [3, 6, 9, 14, 18, 32].

Ying et al. present a method that predicts which parts of the system will change given a set of changes or change propagation. The prediction is done using the project's change history. The results of the prediction method were mixed with some projects recording a stronger precision and recall and others recording a far lower results. Kagdi and Maletic also leverage version history changes to perform software change predictions. The actually analysis applied is two fold, through the dependency analysis of the current version and the change analysis of the version history. The data is collected through MSR which is a popular field of study. In a similar work, Hassan and Holt, worked towards predicting change propagation of a given initial change. The main question was to determine given a change to an entity (e.g. function or variable) will propagate to changes in other entities. This work is very related since it tests various methods and leverages presents the best one. Bantelay et al. propose a method that mines the file and method level evolutionary couplings to attempt to predict commits and other interactions within the project. Both methods were used in isolation as well to determine whether the attributes were more helpful when used together. Giger et al. attempt to build off of previous work in change proneness by providing predictions relating to more refined entities. While typical change analysis will involve the use of syntactic changes. However Giger et al. suggest that extracting and tracking semantic change could prove to be more helpful and accessible for developers for predicting future changes within a project.

Chaturvedi et al. attempt to predict the complexity of code changes to a project. The project's change history is analyzed and the entropy is calculated. The future amount of changes necessary, the complexity of code changes, is then predicted.

2.4 Change Analysis

Changes that occur within a project are made to achieve a goal or task. Whether the task is high level such as implement a new features for the program or lower level like fix a syntactical bug. Investigations into how changes are made or used can help provide a better understanding for making a better changes or better use of the changes.

Bieman et al. study the change-proneness of different entities within a software project. In order to provide a deeper understanding visualizations were used as well providing a bit of a different approach from some of the other works. Koru and Liu study and describe change-prone classes found within open source projects. Providing further details into characteristics of different changes that are made to a software project throughout development. Similarly Wilkerson attempts to classify different types of changes that occur to a project throughout development. The classification can then be used to identify the impact that a given change will have on other aspects of the project. Snipes et al. provide a tool that attempts to locate areas within the source code that have a large amount of changes. These areas could be classified as underdevelopment and are likely to be very unstable given the amount of change occurring within them.

Chapter 3

Approach

The goal of the research is to provide change prediction. This is accomplished through mining of software data (covered in introduction), analysis of collected data, candidate feature analysis. After the data has been collected it is further analyzed to extract key features. This data is then visualized to provide insights into the data set. Candidate features are then selected from possible features and analyzed to determine the best feature set.

In order to be able to predict changes within a project some project data must first be collected. The data collection is targeted towards open source projects that use GitHub. Specifically projects that are predominately written in Java. The overall method is not language specific however for the purpose of simplifying the implementation it was restricted to only allow for Java. The data collection simply collects all the project's development history realized through the changes made to the project. This includes the information related to developers, commits, tags (releases) and files in the project.

The data is kept unprocessed and stored directly into a relational database (MySQL) which allows the data to be used and manipulated without requiring access to GitHub again. This was ideal during the more initial phase of the research allowing for various methods of analysis to be applied on the dataset without requiring the data to be download again. The collection of data can take long to perform and depends largely on the size of the project. The collection process also allows for a partial collection of newly added project changes after the initial collection of the project. This allows for the changes made to the project after the initial collection of project data to be collected as well. These maintenance collections will often be much smaller and require a smaller amount of time to collect.

The method chosen for collecting data for GitHub projects was using GitHub's web API. The GitHub API allows for access to the complete set of publicly available information stored in GitHub. Accessing the data through the API allows for the process to be automated and vastly simplifies the process. This dataset can be rather large since it includes a snapshot of the commit, all the change data and developer data related. In order to collect data the repository name and the name of the user who owns the repository must be known.

To actually collect the data from GitHub a ruby script was used. This collection is built around a Ruby library, *github_api*, which is a convenient wrapper for GitHub's web API. The script systematically collects the desired data related from a given GitHub project to be stored locally. As noted above the collection can take a bit of time to complete since it must go commit by commit to collect the necessary data.

Some aspects of the GitHub project's dataset are not collected as they were deemed unnecessary however it could easily be extended to collect the other aspects. The aspects not collected are the issues, branches, forks and pull requests. The issues data outlines the problems reported in the project by users or developers of that project. GitHub allows for issues to be optional and thus some projects do not offer issue reporting through GitHub. Branches are also directly related to the project and they are essentially different workspaces for the developers. They allow for development of different versions (such as a development version compared to a stable version). The simplicities sake this project assumes that the main branch (master) is the development branch and the target of the analysis. Of course other branches could be analyzed however the perspective of the other branches typically originates from the master branch.

A similar sub-data set not collected or used is forks of the repository. For GitHub a fork is an externally created branch. This allows for a developer who does not own the project (but can view it) make a copy of the project and work on it without affecting the original. Forks differ slightly from branches in that they typically denote a deviation from the original project that is unlikely to be reconciled. Finally, pull requests facilitate external developers making small changes which tend to be fixes to problems found or desired feature implementation. The owner of the repository can then decide to integrate the changes made the original repository.

3.1 Storage

As mentioned above the data is stored in a MySQL relational database which leverages Structured Query Language (SQL). There are two databases used for the collection and the analysis. One stores the raw mined data, whereas the second stores the analyzed data in a more convenient layout to be used later. A third database stores the same data as the second however it uses relational database implementation because of some limitations within MySQL. This third database uses PostgreSQL, which has some more advanced features than MySQL and is simply a clone of the second database. The specific limitations that were encountered will be discussed more fully later on in this section.

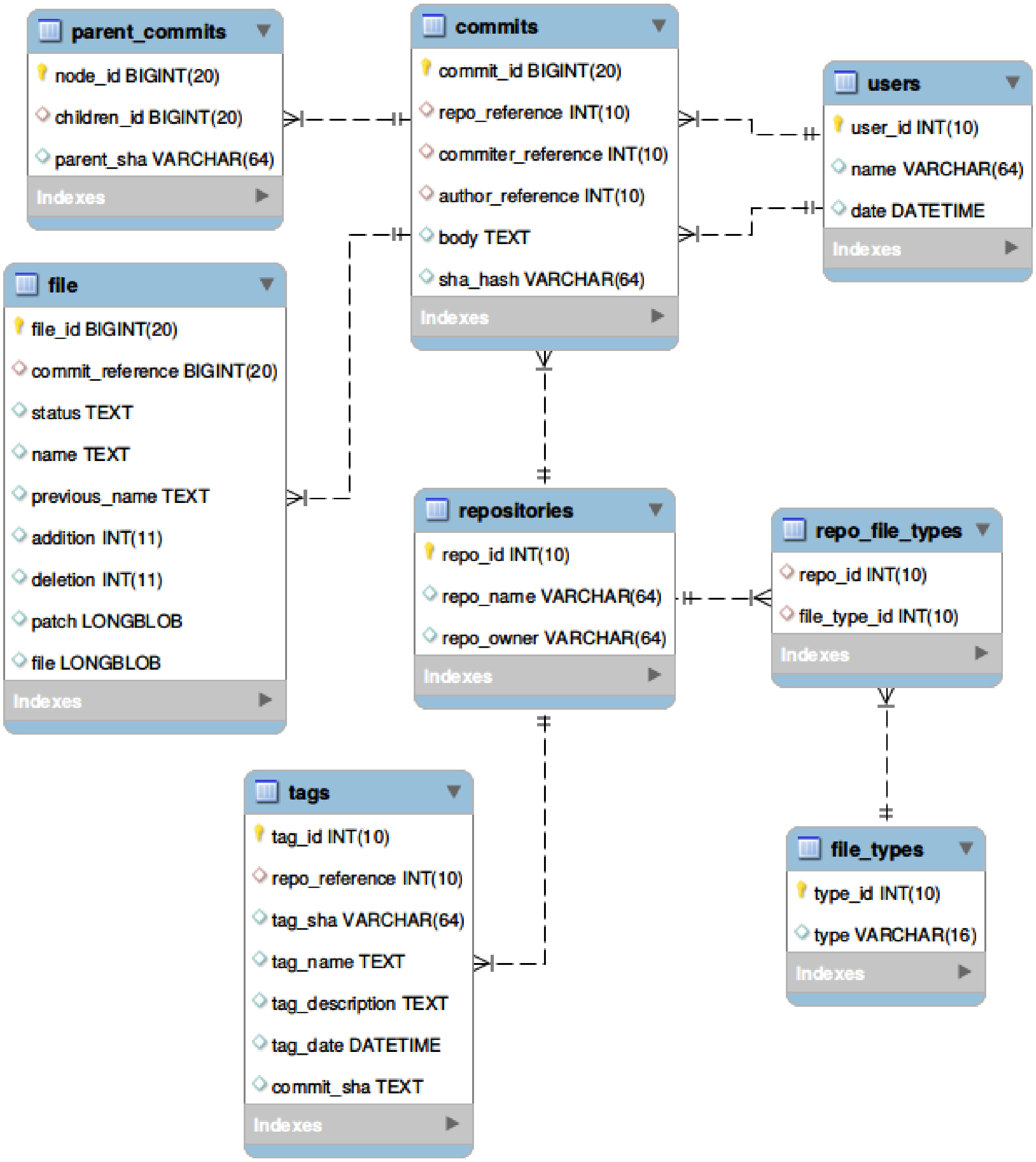


Figure 3.1: GitHub Data Schema

The first database called *github_data* and stores the semi-raw data collected from GitHub's API. This database contains 8 tables which store various aspects about the projects that are considered potentially important for the analysis later on. The tables of primary concern are *repositories*, *commits*, *users*, *files* and *tags* tables. The data collection from GitHub's API collects primary aspects related to the desired analysis. Other aspects are available from the API and if need the database could be extended to store more elements as necessary. In some cases data from the API is not available for one reason or another (usually inaccessible files or such) these are simply removed or a note is made of them depending on their importance. For example, files that do not contain Java code are not essential and if inaccessible are ignored. If a Java file is inaccessible a note is made as this is a greater concern. These files can be retrieved if enough information is available (previous version and corresponding patch file). In the case that insufficient information is available the analysis can still be applied but will likely adversely effected the result.

After storing the data in the *github_data* database, the analysis process is done. The *parsing* script is run next and discussed further in the section 3.2. This database, *project_stats*, is very similar in layout to the first database except some extra tables have been added and a few data items have been removed. Mostly the storage expansions have been to hold change information calculated from the analysis of the data.

The final database uses PostgreSQL because of limitations within the MySQL implementation. Some of the candidate features, discussed in further detail in section 4.1, required a more versatile partitioning function and the ability to perform multiple inner queries. The first of which is more difficult to implement and the second is not available at all MySQL.

In order to transfer the data over to PostgreSQL, a simple program called *pgloader*[1] was used to transfer the MySQL database over to PostgreSQL. Only one difficulty was encountered during the transferring process. One of the tables in the MySQL database was called *user*, however in PostgreSQL this is a reserved table name and

[1] `http://pgloader.io/`

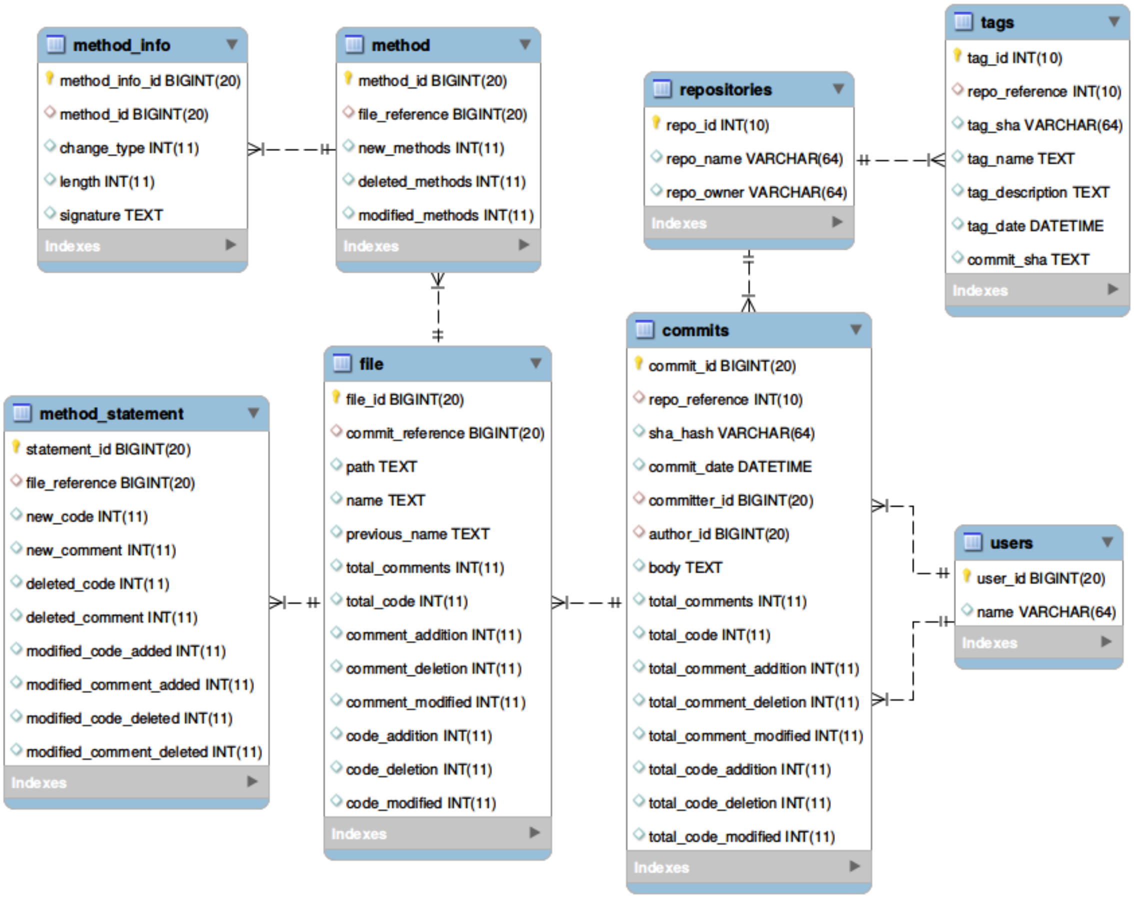
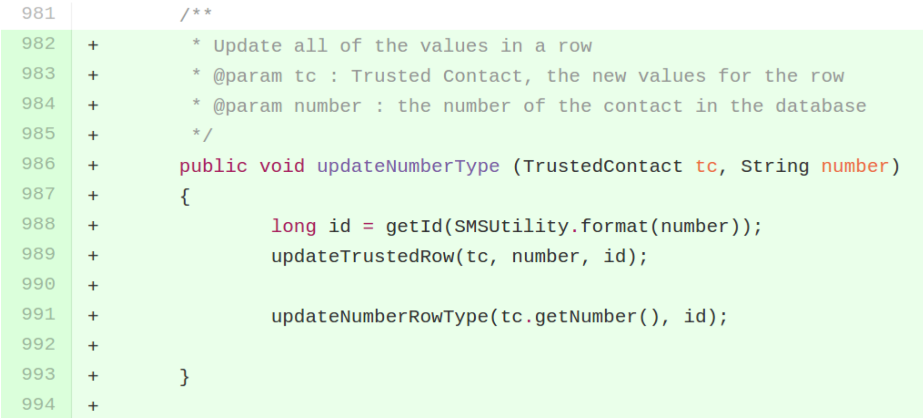


Figure 3.2: Project Stats Schema

therefore the table cannot be interacted with properly. The work around was to simply rename the table in MySQL prior to transferring to avoid any issues with the database. Once the database is copied over to PostgreSQL it is ready to be used for to perform change prediction.

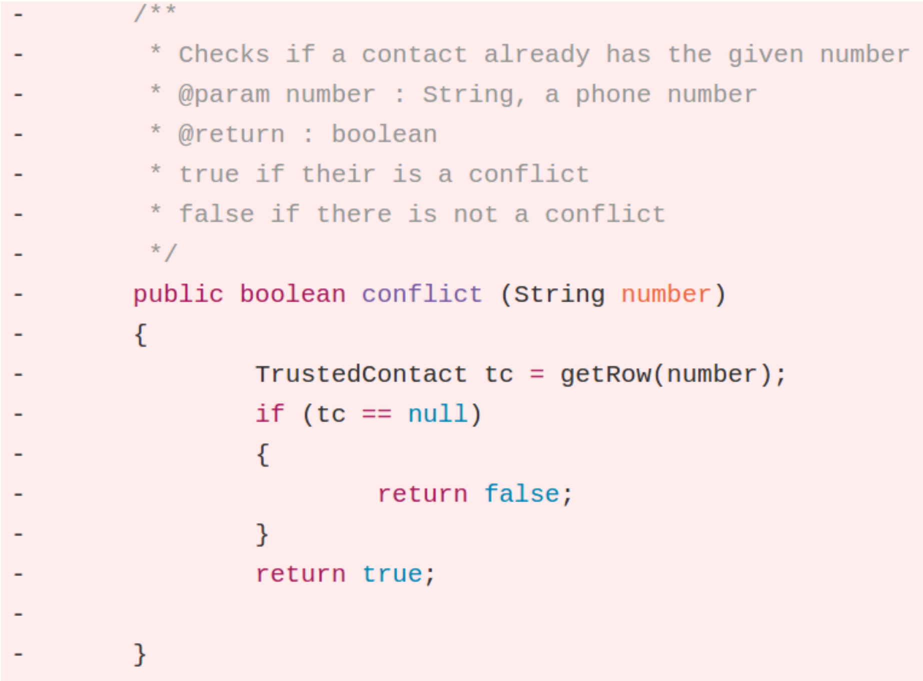
3.2 Parsing

When the data has been collected and stored it can then be analyzed to extract more refined details. The changes made per commit can be analyzed to extract the number of methods added, deleted and modified per commit. The process first requires the changes from a commit, the patches, to be merged into their corresponding full file. A patch is simply a summarized stub of the full file which allows for a quick reference as to which line is changed and what change occurred on that line. The three different types of changes that can appear within a file are deletions, additions and no change. These are represented as a minus sign, plus sign and space respective.



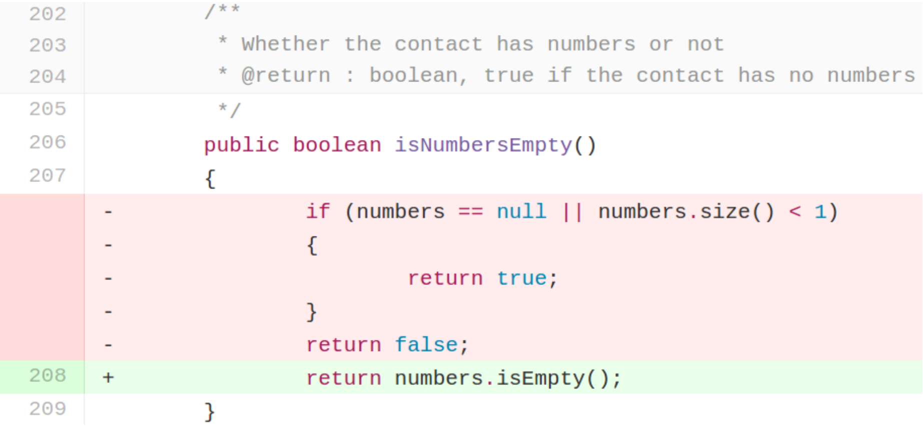
```
        /**
+        * Update all of the values in a row
+        * @param tc : Trusted Contact, the new values for the row
+        * @param number : the number of the contact in the database
+        */
+       public void updateNumberType (TrustedContact tc, String number)
+       {
+               long id = getId(SMSUtility.format(number));
+               updateTrustedRow(tc, number, id);
+
+               updateNumberRowType(tc.getNumber(), id);
+
+       }
+
```

Figure 3.3: Newly added method



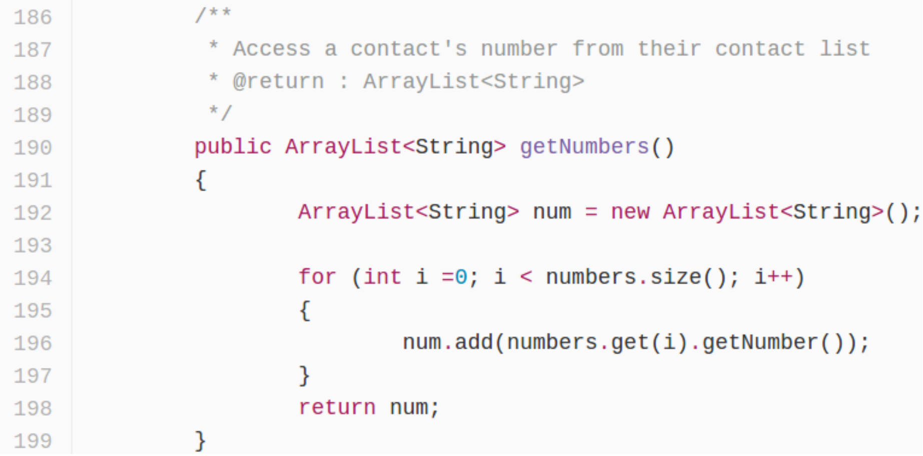
```
-       /**
-        * Checks if a contact already has the given number
-        * @param number : String, a phone number
-        * @return : boolean
-        * true if their is a conflict
-        * false if there is not a conflict
-        */
-       public boolean conflict (String number)
-       {
-               TrustedContact tc = getRow(number);
-               if (tc == null)
-               {
-                       return false;
-               }
-               return true;
-
-       }
```

Figure 3.4: Removed method



```
          /**
           * Whether the contact has numbers or not
           * @return : boolean, true if the contact has no numbers
           */
          public boolean isNumbersEmpty()
          {
-                 if (numbers == null || numbers.size() < 1)
-                 {
-                         return true;
-                 }
-                 return false;
+                 return numbers.isEmpty();
          }
```

Figure 3.5: Mixed changed method



```
        /**
         * Access a contact's number from their contact list
         * @return : ArrayList<String>
         */
        public ArrayList<String> getNumbers()
        {
                ArrayList<String> num = new ArrayList<String>();

                for (int i =0; i < numbers.size(); i++)
                {
                        num.add(numbers.get(i).getNumber());
                }
                return num;
        }
```

Figure 3.6: Unchanged method

Using the patch file a *deleted* file can be reconstructed by removing all lines marked as added from the file and adding the lines marked as deleted back into their original location. This allows for both added and deleted methods to be identified by using the original file for detecting the location of added methods and the *deleted* file to detect deleted methods.

The more difficult method to identify is one which has been modified. Again use of the two files will be necessary, in this case we will identify methods from each which are not entirely additions or deletions respectively. The union of these two sets of methods will be taken to determine the number of methods that have been modified.

For each commit this information is stored to allow for easier access and save time since the analysis of larger datasets can be time intensive. In order to maintain the integrity of the initial dataset this information is stored in a new database.

3.3 Visualization

3.3.1 Line Change

After collecting an analyzing the data the key features are extracted from the collected data. In order to to better understand resulting data it was visualized. The first visualization simply showed the changes recorded on a per line basis. These changes were divided into several closely related subcategories of additions, deletions and modifications. Additions identify changes that are new and do not have a corresponding set of deleted code. Similarly deletions refers to changes that remove code without a corresponding set of additions. Finally modifications are a set of changes which contain a set of additions and deletions that are related.

In a modification the changes are related through the Levenshtein Distance (LD) calculation. This distance calculation will determine the edit distance between two strings. Where edit distance is defined as the number of characters difference between two different strings. For example, LD between *happy* and *mapper* would be 3, since h would be changed to m, y to e and r would be added at the end. While this provides a good initial method for comparison between two string values the value must be normalized to allow for more general use. To calculate Normalized Levenshtein Distance (NLD) the LD would be divided by the larger of the two strings sizes shown in Equation 3.1.

$$NLD(a_i, d_j) = \frac{LD(a_i, d_j)}{\max(|a_i|, |d_j|)} \tag{3.1}$$

Modifications were assumed to only take place in a series of changes that involved both additions and deletions shown in Figure 3.5 and with an NLD below a defined threshold Δ_m.

$$m(a_i, d_j) = NLD(a_i, d_j) < \Delta_m \tag{3.2}$$

In order to account for larger method signatures a threshold α was created to separate small and large method signatures. Therefore the Equation 3.2 was updated accordingly shown in Equation 3.3.

$$m(a_i, d_j) = \begin{cases} NLD(a_i, d_j) < \Delta_s & \text{if } \max(|a_i|, |d_j|) < \alpha \\ NLD(a_i, d_j) < \Delta_l & \text{otherwise} \end{cases} \tag{3.3}$$

Lines that are part of the same block of additions and deletions are selected for the similarity check to determine whether they can be classified as a modification. Modifications will consist of one to many addition lines mapped to one to many lines of deletion. Therefore a modification is more easily referred to as a modification set.

For addition lines that do not meet the threshold of similarity with any deletion line in the change block are classified as additions. Similarly, deletion lines who fail to meet the similarity threshold for any addition lines will be classified as deletions. Therefore a block of changes will contain a set of additions, deletions and modifications any of which may be empty.

The project's tags are shown at the bottom of each graph optionally to potentially provide some context. Since these tags often mark points of significant within the project they can be thought as road signs. The site also provides some options to refine or generalize the graphs. For all of the graphs you are allowed to select the project, package path, and the committers you wish to view. Specifically for the line level graph a further option is provided to condense the data based on a monthly, weekly summary.

As a further guide marker the commit information is provided (when viewing either line at the commit view, method level or statement level). This information allows for a direct link to the project and can be a handy tool for referring back to the software repository.

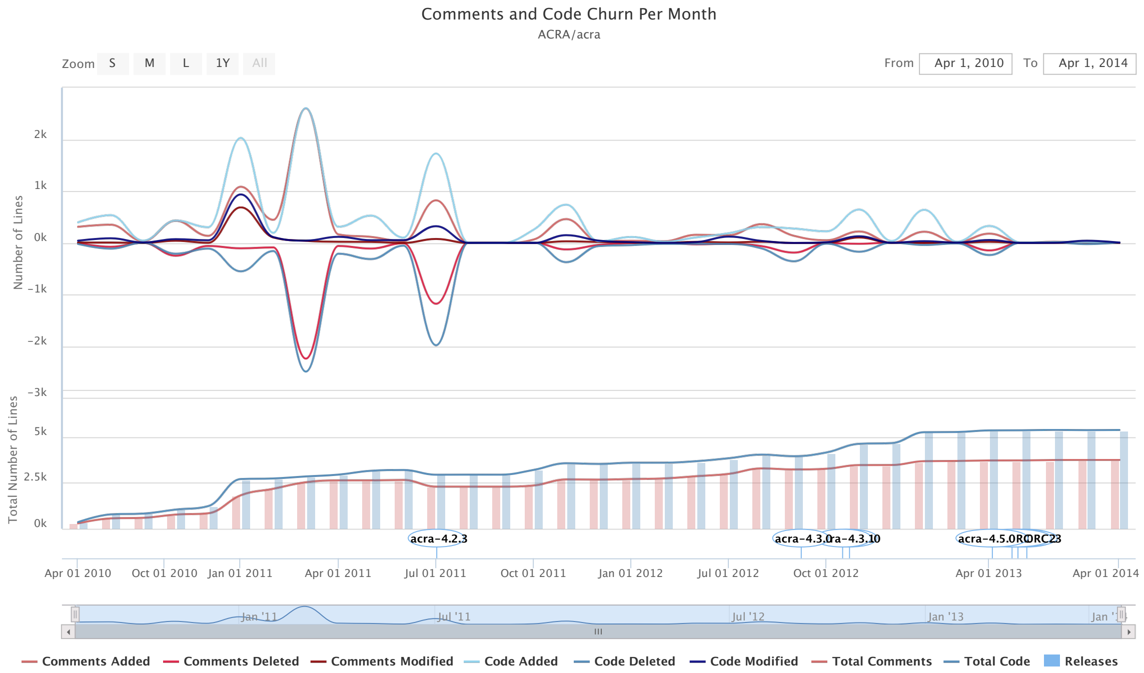


Figure 3.7: Line Change Visualization for acra

3.3.2 Method Change

The visualization of line changes was very noisy and proved difficult to use. Instead of viewing every line of change separately they were grouped together based on the method from which they originate from. Similar classifications are used for method changes however their definitions vary slightly. There are three types of method level changes that can occur. Firstly, a method can be newly added implying that the method had not existed in the previous version. Secondly, a deleted method implying that the method is completely removed from the current version. Thirdly, a method can be modified by containing a set of changes that are not constituting the entire method changing.

It should be noted that at the method level comment changes are ignored. Instead the focus is placed on that of the three types of changes. The visualization for the method level uses a bar graph since it provided a more clear picture of the relationship between commits. Rather than as the first visualization did imply that a relationship was to be drawn between different commits of the same type only changes of the same time are grouped together. The contrast in magnitude between each type of change and each commit is also more clear and defines the visualization.

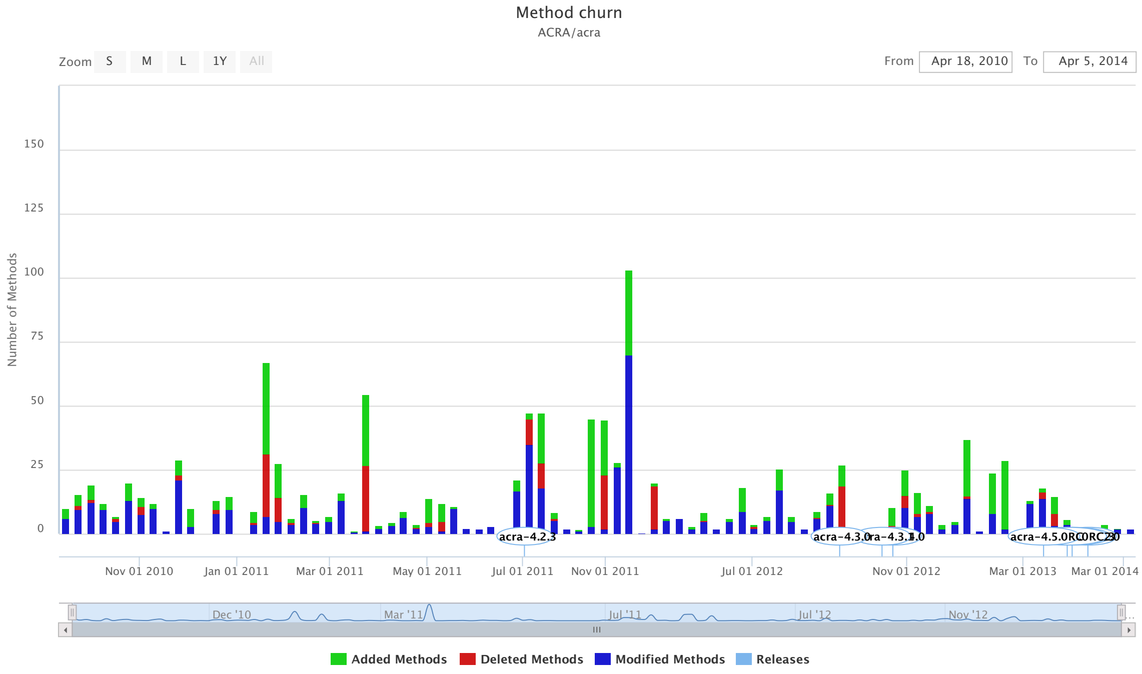


Figure 3.8: Method Change Visualization for acra

3.3.3 Method Statement Change

The method level visualization provided a fairly clear higher level view of the data. However, while collecting that data lower level data was collected as part of the previous analysis. This afforded a combination of the previous two methods. While more data is available and is quite overwhelming the final graph could provide some use when used in conjunction with the previous graph.

The view itself classifies changes into several categories, first their is *Added* changes which comes in the form of both code and comments. Secondly, *Deleted* changes which again is for both code and comments. Similar to that of the method level added or deleted method these statements belong to methods that are either entirely added or deleted from the project. However for this level each statement is counted versus just the method on whole.

The more complicated categories are introduced as part of the modification classifications. These all stem from the method level modifications. A modified method will contain some changes which can be statement additions or deletions. Therefore modifications are divided into modifications that are additions and ones that are deletions. The final filter is again based on statements being either comments are code. So finally we have the categories: *Modified Code Added*, *Modified Comment Added*, *Modified Code Deleted* and *Modified Comment Deleted*.

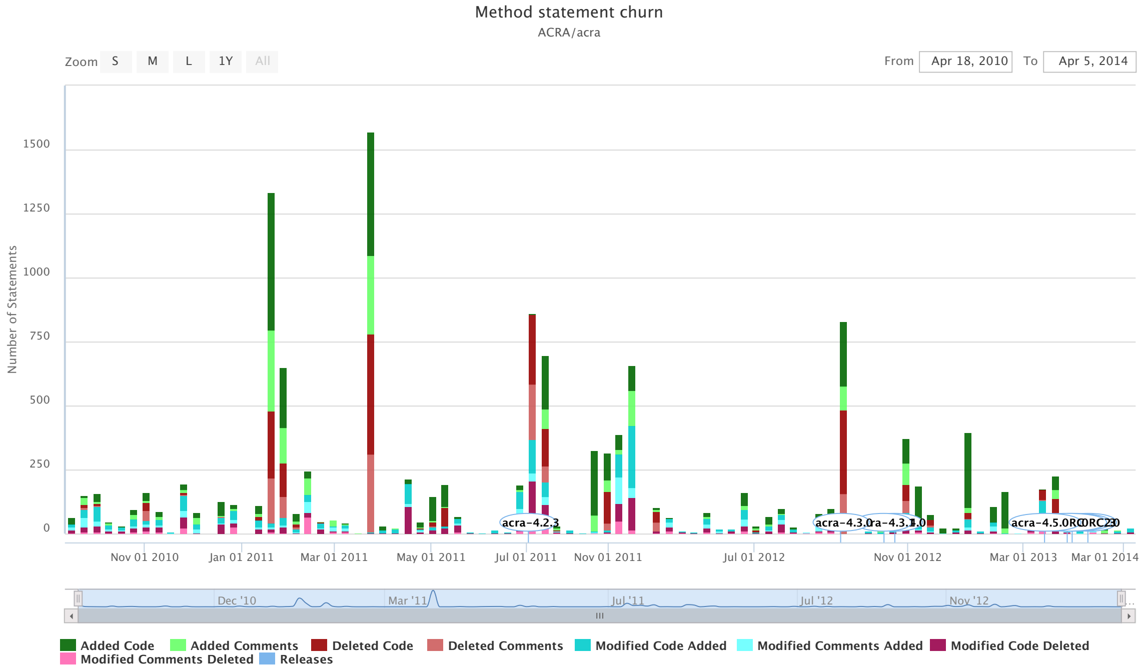


Figure 3.9: Method Statement Change Visualization for acra

Chapter 4

Experiments

4.1 Sample Data

One thing that should be noted for the experiments that were run. Since all of the data was known before the model was even created artificial cut off dates were created to allow for the feature set to be tested as to their effect on the model. A test project, acra (developed by the user ACRA), was chosen to develop the method on.

Owner	Project	Start Date	End Date	# of Commits	# of Developers
ACRA	acra[a]	2010-04-18	2015-06-05	404	32
apache	storm[b]	2011-09-16	2015-12-28	2445	261
facebook	fresco[c]	2015-03-26	2015-10-30	313	47
square	dagger[d]	2012-06-25	2016-01-30	496	39
deeplearning4j	deeplearning4j[e]	2013-11-27	2016-02-13	3523	62

Table 4.1: Experiment projects

[a] https://github.com/ACRA/acra

[b] https://github.com/apache/storm

[c] https://github.com/facebook/fresco

[d] https://github.com/square/dagger

[e] https://github.com/deeplearning4j/deeplearning4j

Project	# of Methods	# of Methods Changes	Avg # of Commits / Year	Avg # of Methods Change / Commit
acra	1309	3605	67.33	9.51
storm	14599	50037	489	24.03
fresco	3463	4139	313	14.73
dagger	1827	6314	99.2	13.70
deeplearning4j	29896	82198	880.75	24.33

Table 4.2: Project Change Statistics

The complete list of projects that were tested is found in Table 4.1. The number of commits excludes any commit that lacked a change to a file containing Java code. Since the primary interest was to parse Java code, files containing Java code were used while all other files are ignored. These measures provide a more accurate description of the project in terms of the analysis and predictions made on it. Secondly, the number of developers does not map effectively to what git uses as committers and authors. Instead, the number of developers includes all individuals (removing duplicates) who committed or authored commits to the current project.

Each of the projects selected on GitHub using the list of Java projects with a large amount of contributions. Open source projects were targeted to simplify any usage concerns. Therefore in order to be selected the program had to clearly use an OSS license. Secondly, the program also needed to have at least a 6 months worth of development and at least 300 commits to provide a large enough dataset to analyze. An effort was also made to pick projects of different sizes to provide better tests of various conditions.

The first project acra is a Android bug logging tool used with Android applications to capture information related to bugs or crashes. The information is sent to the developers to help them address the issues that their clients encounter while using there application. The second project, apache's storm, real time computational

Project	Avg # of Methods Change / Year	Avg # of Changes / Method	Avg # of Commits / Developer	Max Commits / Year	Min Commits / Year
acra	600.83	4.52	13.93	119	33
storm	10007.4	5.93	15.47	948	118
fresco	4139	1.49	156.5	313	313
dagger	1578.5	5.64	16	236	4
deeplearning4j	20549.5	5.69	65.24	2018	65

Table 4.3: Project Change Statistics 2

system for continuous streams of data. This project is one of the larger projects and has a large development community. The third project, facebook's fresco, is the smallest project with the shortest development period. This project provides a library for using images on Android to attempt to solve limited memory issues with mobile devices. The fourth project, square's dagger, is a Java application used to satisfy dependencies for classes to replace the factory model of development. The final project, deeplearning4j, is a distributed neural network library that integrates Hadoop and Spark. This application is the largest of the 5 projects and provides a large wealth of data to analyze.

In order to get a more detailed understand of the selected projects numerous measures were taken. These measures also allow for each projects to be compared to each other in terms of the development of each of the projects. The size of the project is represented through number of commits, methods. The size of the development team is also provided. The length of each project is shown and most of the measures average on a yearly term.

Several average measures were also taken which detail the amount of change that occurs within the project. The average number of commits per project coupled with the average number of changes per commit clearly indicates the amount of changes that are occurring with in the project. The rate at which methods are change provides

Project	Max # of Methods Changed / Year	Min # of Methods Changed / Year	Max # of Change / Method	Min # of Change / Method	Max # of Commits / Developer	Min # of Commits / Developer
acra	1503	183	52	1	229	1
storm	26526	2152	314	1	622	1
fresco	4139	4139	33	1	269	44
dagger	3374	171	65	1	157	1
deeplearning4j	35869	4377	345	1	1987	1

Table 4.4: Project Change Statistics 3

good insight into the growth of a project. While some changes may involve the addition of new methods, others may include the removal of methods or the modification of methods. The other measures relating to the amount of change occurring with a project on average are the number of methods changed per year and the number of changes per method. Each of these further outline how the changes are being made to the project on average.

A few of the measures are related to the number of developers. These while provided are not the primary focus. The information provided by tracking developer interactions with each other or the repository could be integrated into future work.

While the purposed method was being developed ACRA's acra project was primarily used for exploring and initial testing of the approach. After experimenting on acra a few of the potential candidate feature sets were distinguished based on their superior performance. Experiments were then run on other projects using the feature sets that performed better.

4.1.1 Experiment Factors

The sliding window factor is one of core aspects related to extracting samples from the data set. When using the sliding window to sample the data the data is divided as shown in Figure 4.3. The training sample is where the training data set is sampled from. The testing sample is where the testing data is sampled from.

A data set with an extended sampling range will extend the sampling range beyond the original size for either the training sample or the testing sample. The training range can be expanded to include earlier samples to increase the sample space.

The training and testing sampling range are defined as the number of commits from which the samples can be taken. In Figure 4.3, both the training and testing sample ranges are set to 30 commits. These two values can differ from one another but tend to be kept the same for most of the experiments.

Two other factors in the experiments was handling the data set bias. The two methods are oversampling and undersampling of data elements. Most of the time a data set will contain more samples in one category than the other. When the number of samples in each category is very close this will have little effect on the model. However in cases were the data is highly skewed to one classification over the other the model will simply predict the larger classification. Use of oversampling will take more samples (duplicates) from the smaller classification to be closer in size to the larger classification. Alternatively, undersampling will use remove samples from the larger classification to become closer in size to the smaller classification.

Another feature assessed was that of the sample size. The sample size was determined by the sample ratio. When sampling if the ratio is at 50 % then only half of the values retrieved will be used to train or test. For some of the larger data sets sampling 100 % of the data from the range would take a lot longer. Therefore sampling a percentage of the data set is commonly used to decrease the training time. However in the case of using a percentage of the sample range the data should to be sampled randomly to provide a more stable model. Therefore each data entry in the sample has the same chance to be within the training or test data set.

For each project data set there are numerous windows that be can be used. The window number is setting which window is used broadly mapping to the position within the data set that the model will be trained on and then predicted on. As shown in Figure 4.3, the window is shown starting at the 30th commit which would also be called the 30th window.

Finally, the last factor of note is the parameters used to configure each prediction method. RF use a single parameter, the size of the forest. SVM meanwhile uses three parameters; C, gamma and esp.

4.1.2 Experiment 1

The first set of experiments were preliminary and used SVM as the approach for prediction algorithm. They attempted to predict whether a given method would change within the next 6 commits. The test was done one using ACRA's acra (hence forth just called *acra*). The data set was divided into four sections based on the date length. So each section of the data was the project's lifespan divided by 4 long. The sampling method from the data set $data_i$ was to use the first n tuples ordered by date. The value of n was tested at 100 and 1000. The data set $data_i$ contained tuples which had changes and ones that had no changes.

$|data_i| = \frac{|date_f - date_s|}{4}$

The results of these experiments were not particularly promising scoring accuracy, precision, recall and recall around 50% or below. The experiment was run such that $data_i$ would be used for training and $data_{i+1}$ would be used for testing. For example $data_1$ would be used to train $model_{1,2}$ and $data_2$ would be used to test the $model_{1,2}$. A slight variation was also tested where $data_i$ trained $model_i$ and was tested use each data set $data_j$ that had $j > i$. This however still provided poor results similar to the original experiment. Figure 4.1 shows how the data is distributed.

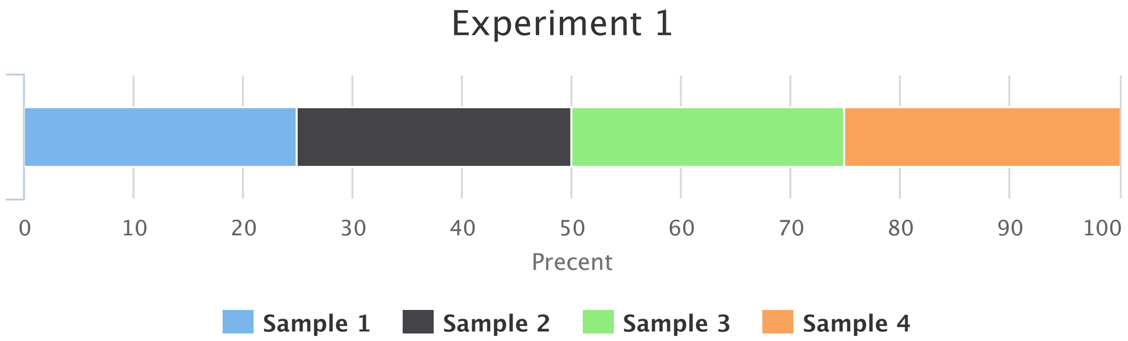


Figure 4.1: Experiment 1 Data Sample Range

4.1.3 Experiment 2

To address the fixed data sampling used previously random sampling was employed. The number of samples n was not changed from the first experiment. The prediction results improved slightly. The SVM model also reported an error relating to *max number of iterations reached*. This error indicates that the SVM model is having trouble effectively separating the two categorizations of data into two distinct sets. Such warnings could mean that the features used for training the model are not linearly separable.

4.1.4 Experiment 3

Further investigation into SVM lead to a research tool that provided grid search for optimizing the parameters used for a SVM. The results of these experiments offered improvements over the previous, however required a long time to find the right parameters and works the best with the dataset $data_i$ that was used to find the parameters.

4.1.5 Experiment 4

Modified the candidate features set to test the SVM with. Added ones like change frequency, average time between commits, number of commits since last change, time since last change and previous change type. Dropped commit author in favor of using just committer. Finally changed the prediction to predict the whether a change will occur within the next 5 commits. In terms of the methodology, 2 variations were tested. The first one was predicting changes to a methods within the same package. The second one was prediction changes to a method within the same file. Neither of these changes provided substantial improvements since they reduced the available sample size, $|data_i|$, down so that the no reasonable predictions could be made from the reduced set.

4.1.6 Experiment 5

The next set of experiments required more complex features which necessitated more complex queries from the database. In fact the database interface in use, MySQL, was unable to implement some of the queries. MySQL only allows 2 levels of nested queries and has a more restrictive data type set. An alternative database interface PostgreSQL was used as a replacement for MySQL. PostgreSQL offers fair more sophisticated data types as well as Common Table Expressions (CTE). The migration from MySQL to PostgreSQL was simplified through the use of pgloader as mentioned in section 3.1.

Another change was the even out the number of samples collected from each category. The data set $data_i$ tended to provide unbalanced categorization of the data. The same number of samples from each $n/2$ was collected to prevent biasing in the data set. A SVM model like most other machine learning algorithms is susceptible

Project	Sample Size (n)	Accuracy
acra	100	70%
acra	1000	52%

Table 4.5: Experiment 5 Results

to category biasing. This will occur when the training set consists of 80% of one category and 20% of another. The model will train such that it always predicts the first category. This works out well if the data is always unbalanced in the same way however it will fail to predict the smaller category entirely. Therefore providing an even sample of each category (50% each) will prevent poor prediction results.

Another result of determining that the data set was unbalanced in terms of the categorization was to calculate both precision and recall of the results. Accuracy alone only provides a very simple measure of how well the model predicted the samples from $data_{i+1}$. A clearer understanding is available with these three measures. Also with each provided an attempt can be made to optimize all three.

Finally the last few tests in this experiment set included a new feature. This feature was the difference in time between the previous commit with a change and the latest commit. This feature was not particularly useful however and caused the data to become inseparable.

At the end of this set of tests it was apparent that a deeper understanding of the candidate features was necessary to improve the results. Therefore an analysis of each candidate feature was performed both on the quality of the feature and the possible relationship with others. SVM models are particularly sensitive towards dependencies between the variables. It was also necessary to properly convert data into a format that could be used by the database. This was talked about in more detail in subsection 2.2.1.

Project	Sample Size (n)	Accuracy
acra	100	60%
acra	1000	47%

Table 4.6: Experiment 6 Results

4.1.7 Experiment 6

After analyzing the candidate features a more ideal set of features was created. The tests were preformed again using sample sizes of 100 and 1000. After the changes were made to the data, the performance improved however some of the features did not prove as useful. However the improvement was marginal and therefore necessitated shift in focus from the features to the prediction method. Specificity the data sampling method was inspected to attempt improve the prediction results. Instead of breaking the data set into four even sets based on the date range the data was divided into two even sets based on date as shown in Figure 4.2

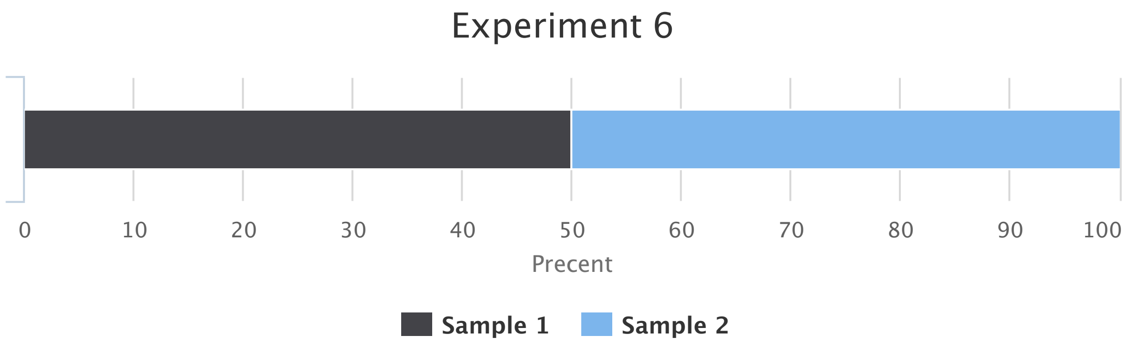


Figure 4.2: Experiment 6 Data Sample Range

The results of the experiment are outlined in Table 4.6. The larger sample with $n = 1000$ is far worse than with $n = 100$. This however was a worse result than when the data set was divided into 4 equal parts.

4.1.8 Experiment 7

Reorganized the data sampling method to sample based on commit ranges rather than date ranges. Instead of splitting the data set into four even sections, the sample range is taken from the current commit c_i to c_{i-m} in the case that $i > m$. m denotes the width in commits of the sample space. For example if the model is predict a change that occurs within the next 5 commits and $m = 30$ then Figure 4.3 shows how the data would be sampled. The training sample would be where data would be collected from to train the model. The prediction gap is to account for the data sampling calculating whether methods at commit 40 will have a change within the next 5 commits. Therefore to properly test it on data that is not used as part of the testing model the offset is needed. The testing sampling section is the same size as the training sampling data set and follows the 5 commit gap.

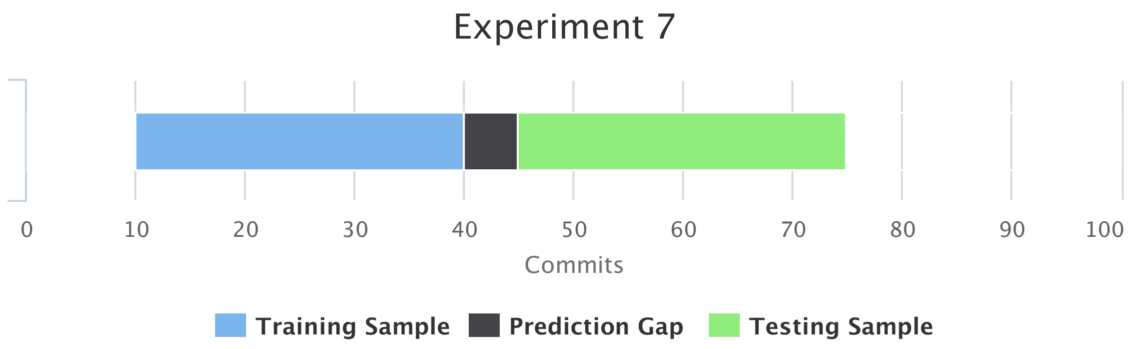


Figure 4.3: Experiment 7 Data Sample Range

Another change to the sampling method was to sample a percentage of the data within the sample rather than a fixed amount. The sample ranges provided a larger range of data to sample and thus sticking to a arbitrary amount of tuples was updated. This however introduced biasing issues with the data set since typically one category or the other had the majority of tuples within the sample. Fixing this issue is talked about in further detail in subsection 2.2.2. Similar to the previous sampling techniques the data set was sampled randomly. Therefore the percentage of data sampled had a large impact on how long it took to train the model but typically also on the prediction results of the model.

4.1.9 Experiment 8

After more or less establishing a data sampling model the candidate feature set was looked at again in an attempt to improve the prediction results. A few of the candidate features were removed and a minimum candidate feature set was determined which provided the best results for the current test project *acra*.

The optimal value of m is a more challenging issue since for project which have a large amount of rapid change occurring in the project larger value seems to provide a more positive result. Where as smaller projects or ones that have a slower rate of change tend to do far better with a smaller value of m.

4.1.10 Experiment 9

After the results of the previous experiments proved to less consistent for other data sets such as *storm* and *fresco* another prediction algorithm was tested. RF as discussed in greater detail in subsection 2.2.2 is more capable with unbalanced datasets and is generally more widely used for performing predictions on mined data.

RF while proving to be at times easier to get better results also had some of the challenges that the SVM model experienced. For example the best features to use in the prediction model and also the best commit width (m) to optimize the results.

The Table 4.7 lists each feature with a more detailed description. An example of each feature is provided to further illustrate them. As stated in the previous subsection 2.2.1, the values need to be first converted into floating point numbers.

Feature	Description	Data	Example Vector
$name$	The name of the file	Main.java	3
$signature$	The method name related to the change details	void work() {	46
$change_i$	Whether the method changed or not at the current commit	3	1
$committer$	The individual who committed the change	bob	5
$freq_{change}$	The number of changes this method has been involved divided by the number of commits up till this point	0.0464	0.0464
$change_{prev}$	A list of whether the method changed or not for the last 5 commits	{3,3,0,3,1}	{1,1,0,1,1}
Δt	A set of time deltas between the last 5 commits that involved the method	{68,416,569,772,898}	{68,416,569,772,898}
$change_{next\_6}$	Identifies whether at least one change occurred within the next 5 commits for the given method	0	0

Table 4.7: Candidate features for SVM model

First the data is extracted from the database as *raw* values as shown in the ***Data*** column. Taking the *name* value, “Main.java” will be mapped to the value 3. This is because 2 other methods have already been mapped and therefore method name is mapped to the next available mapping, 3. Similarly both *signature* and *committer* will be mapped from their respective values “void work() {” and “bob” to 46 and 5. Numerical values are easily converted by casting them to floating point values if they are not already of that type. For spacing reasons all the values in the table that were integers to begin are shown without a “.0”following.

Another small change made to the data to create a vector for the SVM model was to apply Equation 4.1 to the values of $change_i$ and $change_{prev}$. As in Table 4.7, the value of $change_i$ is initially 3 which indicates a modification occurred. Since a modification is a type of change $C(change_i) = 1$ which is the value used by the vector. Likewise this is also applied to each entry in the $change_{prev}$ changing it into a bit vector.

$$C = \begin{cases} 1 & \text{if} change > 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

Both $change_{prev}$ and Δt are actually each 5 features since they are a set of features. $change_{prev}$ shows the type of change that occurred for the last 5 commits. Similarly Δt shows the difference between the current commit time $(t(c_i))$ and the previous commit time $(t(c_{i-1}))$ calculated in Equation 4.2. These two features then expanded to add a new category for each entry in the set. The ordering is maintained since each entry maps to a previous commit in order.

$$\Delta t_i = t(c_i) - t(c_{i-1}), i > 1 \tag{4.2}$$

$freq_{change}$ is calculated as by taking the number commits which involve changes to the current method (c_i) divided by the current number of commits (c_{cur}).

$$freq_{change} = \frac{|c_i|}{|c_{cur}|} \tag{4.3}$$

Another issue that was necessary to address was the arbitrary sample size. For projects that are a lot bigger 100 vectors which map to 100 method changes could be very small. The sampling also seemed like a peculiar approach to picking the data since it would randomly pick values from over a period that could vary from a few months to a few years depending on the size. Therefore instead of dividing the project into four quarters based on time a number of commits is picked. The test is then designed around a given date with the *gap* with t and p commits proceeding it as the range of the test. t is the number of commits that the training dataset will sample from. Alternatively, p is the number of commits that the testing dataset will sample from. In the case that $t = p$ the training commit size and the testing commit size are the same.

The final change that was accounted for was to change the population sample size from a fixed number to a percentage. This allows more flexibility and determining the sample size of a test by allowing for it to scale based on the size of the project.

The initial thought was to provide a few different features that appeared to be unique and potentially provided useful information for whether the method will change within the next 5 commits. Of course since this measurement is calculated, if a vector within the sample set is within the last 5 commits then it will leverage data from the next quarter to provide its prediction. This has not been mitigated and could provide a unrealistic improvement in the prediction score if members of the next sample fall into the first 5 commits. The way to mitigate this would be to provide a buffer between the two sets when the second test is used for testing purposes. The second set would be restricted further, such that the changes must come from after the 6th commit from the start date of the quarter. The first commit would be the one that takes place on or right after (if no commit falls on that date) the start date. The next 5 commits would also be excluded from the test sample set.

4.2 Results

For each experiment where the used random sampling the experiment was performed 5 times to account for variations in the random sample. Therefore if the initial results using the first sample set were not characteristic of the full dataset then running the experiment with more random samples is more likely to represent the true characteristics of the dataset. This required taking five random samples from each quarter, training the model and running the tests on the model to then determine the average prediction score.

The goal of the prediction methods are to provide a good prediction of whether the a given vector will fit in one category or the other. A model's prediction performance can be rated using three measures of accuracy, precision and recall. Accuracy is measured as how often predictions p are classified correctly where a_i represents vector v_i correct classification. The algorithm for calculating a single vectors accuracy is showing in Equation 4.4. The prediction accuracy ($P_{accuracy}$) can then be calculated using Equation 4.5. This simply sums up the accuracy for each vector and then divides it by the total number of vectors (where $n = |v|$).

$$v_i = \begin{cases} 1 & \text{if } p_i = a_i \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

$$P_{accuracy} = \frac{\sum_{i=0}^{n} v_i}{n} \times 100 \tag{4.5}$$

The precision of a model is the measure of how correct the model predicts that a change will occur when it predicts that a change will occur. Given the true positives tp, represents the number of predictions that the model correctly identified as having a change and the false positives fp is the number of times the model incorrect predicted a change to occur when it in fact did not. The equation for calculating precision is show in Equation 4.6.

$$P_{precision} = \frac{tp}{tp + fp} \tag{4.6}$$

The recall of the model is the measure of how correct the model predicts that change will occur out of all the times changes really occurred. Again using tp as the number of true positives, and false negatives fn which is the number of times the model fails to predict that a change will occur. The recall can be calculated using the Equation 4.7

$$P_{recall} = \frac{tp}{tp + fn} \tag{4.7}$$

While initially a larger set of features (the candidate features) was considered, early tests showed poor results and indicated that some of the features may be detrimental. This is not entirely surprising since an SVM is very dependent on the features fitting within specific requirements outlined earlier in subsection 2.2.1. Some of the features appeared at first to be acceptable but with further testing and understanding proved to be determinant to the vector in training the model.

An incrementing unique integer, $commit_id$, was assigned to each commit. Initially this number was used as part of the candidate feature set. However further investigation determined $commit_id$ would only negatively affect the results. Given that each commit was provided a unique incrementing value only methods changed in the same commit would be given the same number. While this may seem initially useful tests showed the opposite.

Other candidate features that were tested more extensively also proved to have a poor effect on the SVM model. The candidate features that appeared to have a negative impact on the SVM model were $committer$, $change_{prev}$, Δt. The fact that these features had a negative impact does not necessary mean that they are unrelated to the changes that occur to methods. However, in conjunction with other candidate features the model created consistently made inaccurate predictions.

While the previous candidate features performed poorly, candidate features $signature$, $change_i$, $freq_{change}$ and $name$ all were apart of feature sets that performed very well.

$name$ was found to not have a large impact and slight detrimental impact on performance but while included still achieved a rather high prediction score.

Chapter 5

Conclusions

Bibliography

[1] Alam, M. S., and Vuong, S. T. Random Forest Classification for Detecting Android Malware. In *IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing* (2013), pp. 663–669.

[2] Antón, J. C. Á., Nieto, P. J. G., Viejo, C. B., and Vilán, J. A. V. Support Vector Machines Used to Estimate the Battery State of Charge. *IEEE Transactions on Power Electronics 28*, 12 (2013), 5919 – 5926.

[3] Bantelay, F., Zanjani, M. B., and Kagdi, H. Comparing and Combining Evolutionary Couplings from Enteractions and Commits. In *Working Conference on Reverse Engineering, WCRE* (2013), pp. 311–320.

[4] Bhattacharyya, S., Jha, S., Tharakunnel, K., and Westland, J. C. Data mining for credit card fraud: A comparative study. *Decision Support Systems 50*, 3 (2011), 601–613.

[5] Burbidge, R., Trotter, M., Buxton, B., and Holden, S. Drug design by machine learning : support vector machines for pharmaceutical data analysis. *Computers and Chemistry 26*, 1 (2001), 5–14.

[6] Chaturvedi, K. K., Kapur, P. K., Anand, S., and Singh, V. B. Predicting the complexity of code changes using entropy based measures. *International Journal of System Assurance Engineering and Management 5*, 2 (2014), 155–164.

[7] Dit, B., Holtzhauer, A., Poshyvanyk, D., and Kagdi, H. A Dataset from Change History to Support Evaluation of Software Maintenance Tasks. In *IEEE International Working Conference on Mining Software Repositories* (2013), pp. 131–134.

[8] Erturk, E., and Akcapinar, E. A comparison of some soft computing methods for software fault prediction. *Expert Systems with Applications 42*, 4 (2015), 1872–1879.

[9] Giger, E., Pinzger, M., and Gall, H. C. Can We Predict Types of Code Changes? An Empirical Analysis. In *IEEE International Working Conference on Mining Software Repositories* (2012), pp. 217–226.

[10] Gondra, I. Applying machine learning to software fault-proneness prediction. *Journal of Systems and Software 81*, 2 (2008), 186–195.

[11] Granitto, P. M., Gasperi, F., Biasioli, F., and Furlanello, C. Modern data mining tools in descriptive sensory analysis: A case study with a Random forest approach. *Food Quality and Preference* (2007), 681–689.

[12] Guo, L., Ma, Y., Cukic, B., and Singh, H. Robust Prediction of Fault-Proneness by Random Forests. In *15th International Symposium on Software Reliability Engineering, 2004. ISSRE 2004* (2004), pp. 417–428.

[13] Hassan, A. E. Mining Software Repositories to Assist Developers and Support Managers. In *IEEE International Conference on Software Maintenance, ICSM* (2006), pp. 339–342.

[14] Hassan, A. E., and Holt, R. C. Predicting Change Propagation in Software Systems. In *IEEE International Conference on Software Maintenance, ICSM* (2004), pp. 284–293.

[15] Hemmati, H., Nadi, S., Baysal, O., Kononenko, O., Wang, W., Holmes, R., and Godfrey, M. W. The MSR Cookbook: Mining a Decade of Research. In *IEEE International Working Conference on Mining Software Repositories* (2013), pp. 343–352.

[16] Huang, C.-l., Chen, M.-c., and Wang, C.-j. Credit scoring with a data mining approach based on support vector machines. *Expert Systems with Applications 33*, 4 (2007), 847–856.

[17] Jalbert, K., and Bradbury, J. S. Predicting Mutation Score Using Source Code and Test Suite Metrics. *2012 First International Workshop on Realizing AI Synergies in Software Engineering (RAISE)* (jun 2012), 42–46.

[18] Kagdi, H., and Maletic, J. I. Combining Single-Version and Evolutionary Dependencies for Software-Change Prediction. In *ICSE 2007 Workshops: Fourth International Workshop on Mining Software Repositories, MSR 2007* (2007).

[19] Khoshgoftaar, T. M., Golawala, M., and Van Hulse, J. An Empirical Study of Learning from Imbalanced Data Using Random Forest. In *19th IEEE International Conference on Tools with Artificial Intelligence* (2007), pp. 310–317.

[20] Kim, K.-j. Financial time series forecasting using support vector machines. *Neurocomputing 55* (2003), 307–319.

[21] Kim, S., Jr, E. J. W., and Zhang, Y. Classifying Software Changes : Clean or Buggy ? *IEEE Transactions on Software Engineering 34*, 2 (2008), 181–197.

[22] Maletic, J. I., and Collard, M. L. Supporting Source Code Difference Analysis. *IEEE International Conference on Software Maintenance, ICSM* (2004), 210–219.

[23] Malhotra, R. A systematic review of machine learning techniques for software fault prediction. *Applied Soft Computing 27* (2015), 504–518.

[24] Moeyersoms, J., Fortuny, E. J. D., Dejaeger, K., Baesens, B., and Martens, D. Comprehensible software fault and effort prediction : A data mining approach. *Journal of Systems & Software 100* (2015), 80–90.

[25] Moser, R., Pedrycz, W., and Succi, G. A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction. In *2008 ACM/IEEE 30th International Conference on Software Engineering* (2008), pp. 181–190.

[26] Murphy, C., Kaiser, G., and Arias, M. An Approach to Software Testing of Machine Learning Applications. In *19th International Conference on Software Engineering & Knowledge Engineering* (2007), pp. 167–172.

[27] Nagappan, N., and Ball, T. Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on* (2007), pp. 364–373.

[28] Neuhaus, S., Zimmermann, T., Holler, C., and Zeller, A. Predicting Vulnerable Software Components. In *14th ACM conference on Computer and communications security* (2007), pp. 529–540.

[29] Sisman, B., and Kak, A. C. Incorporating version histories in Information Retrieval based bug localization. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)* (jun 2012), Ieee, pp. 50–59.

[30] Thwin, M. M. T., and Quah, T.-S. Application of neural networks for software quality prediction using object-oriented metrics. *Journal of Systems and Software 76*, 2 (2005), 147–156.

[31] Verikas, A., Gelzinis, A., and Bacauskiene, M. Mining data with random forests: A survey and results of new tests. *Pattern Recognition 44*, 2 (2011), 330–349.

[32] Ying, A. T. T., Murphy, G. C., Ng, R., and Chu-Carroll, M. C. Predicting Source Code Changes by Mining Change History. *IEEE Transactions on Software Engineering 30*, 9 (2004), 574–586.

[33] Yu, G., Yaun, J., and Liu, Z. Unsupervised random forest indexing for fast action search. In *Computer Vision and Pattern Recognition* (2011), pp. 865 – 872.

[34] Zeng, J., and Qiao, W. Short-term solar power prediction using a support vector machine. *Renewable Energy 52* (2016), 118–127.

[35] Zimmermann, T., Weissgerber, P., Diehl, S., and Zeller, A. Mining Version Histories to Guide Software Changes. *IEEE Transactions on Software Engineering 31*, 6 (2005), 429–445.