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 2015

Abstract

Abstract here

Acknowledgements

Acknowledgements here

Contents

List of Figures

List of Tables

Listings

Abbreviations

API Application Programming Interface.

DVCS Distributed Version Control System.

OSS Open Source Software.

SQL Structured Query Language.

SVM Support Vector Machine.

SVN Apache Subversion.

VCM Version Control Management.

VCS Version Control System.

Chapter 1

Introduction

Introduction here with example Application Programming Interface (API) and citation [1]

Chapter 2

Background

Chapter 3

Approach

3.1 Mining

Open Source Software (OSS) generally is software that is provided 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 3.1.

The development of large software projects (whether OSS or not) have often made use of Version Control System (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. 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. Apache Subversion (SVN) and git would be two examples of VCSs.

Git is a Distributed Version Control System (DVCS) and differs greatly from

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 3.1: 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/

SVN which is a normal VCS. Git will provide the user with a complete copy of the repository to be that can be worked on independent of network connection and thus also without a centralized server. The distributed aspect of git allows for easier use for all involved parities. The one main issue with a DVCS is that the users will require a means to 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 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

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

OSS projects including, the Linus Kernel, Swift[4] and React[5]. GitHub also provides a public 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 a obvious choice for mining project data. Especially since the target for mining software was to capture OSS projects to for testing. 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. Therefore the date of the commit merely means when the developer notified Git that a change was made.

A branch is a series of commits that are often related. In figure 3.1, 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 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. Often tags are 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

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

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

the two branches have been *merged*. A merge indicates that the differences between the two branches are consolidated based on the developer's discretion.

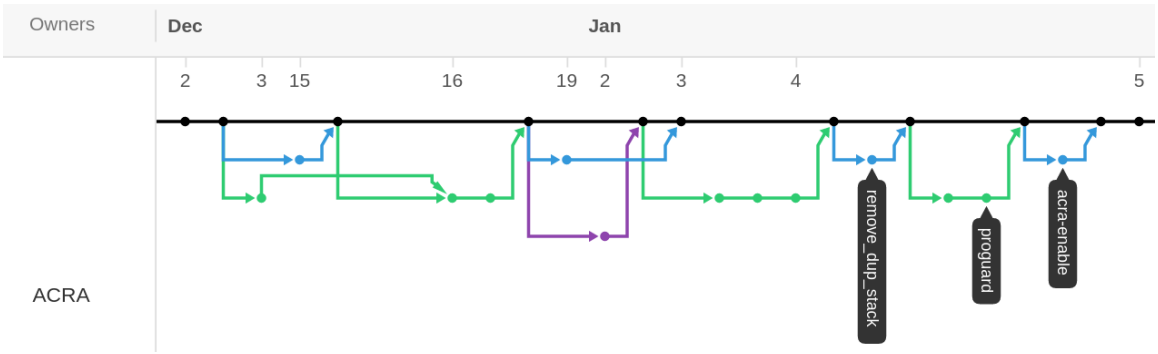

Figure 3.1: Network diagrams

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.

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 data set 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 data set 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.

Some aspects of the GitHub project's data set 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 and are typically fixes to found problems or desired feature implementation. The owner of the repository can then decide to integrate the changes made the original repository.

3.2 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 was later added for the change prediction and uses a different 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.

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 were considered potentially important for the analysis later on. The tables of primary concern are *repositories*, *commits*, *users*, *files* and *tags* tables. While not all of the data available from GitHub's API is collected primary aspects are and the database could be extended to store more of the dataset if necessary. The only real difference between the raw data and the data stored in the *github_data* database

is a slight layout changes and removing invalid or inaccessible data.

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 3.3. This database, *project_stats*, is very similar in layout to the first database 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 more 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* 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 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.

3.3 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.

```
981          /**
982  +        * Update all of the values in a row
983  +        * @param tc : Trusted Contact, the new values for the row
984  +        * @param number : the number of the contact in the database
985  +        */
986  +       public void updateNumberType (TrustedContact tc, String number)
987  +       {
988  +               long id = getId(SMSUtility.format(number));
989  +               updateTrustedRow(tc, number, id);
990  +
991  +               updateNumberRowType(tc.getNumber(), id);
992  +
993  +       }
994  +
```

Figure 3.2: 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 3.3: Removed method

Using the patch file a *deleted* file can be reconstructed by removing all lines marked

```
202          /**
203           * Whether the contact has numbers or not
204           * @return : boolean, true if the contact has no numbers
205           */
206          public boolean isNumbersEmpty()
207          {
-                 if (numbers == null || numbers.size() < 1)
-                 {
-                         return true;
-                 }
-                 return false;
208  +              return numbers.isEmpty();
209          }
```

Figure 3.4: Mixed changed method

```
186          /**
187           * Access a contact's number from their contact list
188           * @return : ArrayList<String>
189           */
190          public ArrayList<String> getNumbers()
191          {
192                  ArrayList<String> num = new ArrayList<String>();
193
194                  for (int i =0; i < numbers.size(); i++)
195                  {
196                          num.add(numbers.get(i).getNumber());
197                  }
198                  return num;
199          }
```

Figure 3.5: Unchanged method

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 data sets can be time intensive. In order to maintain the integrity of the initial data set this information is stored in a new database.

3.4 Analysis

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, if the category is whether the project will change in the future 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 featur es to independent of each other to not negatively affect the categorization.

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 ́bobís mapped to 1

and śallyís 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 than 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 data set that has be categorized into 2 subsets. Once the data set has been separated new data can be provided that has not be categorized to allow for the SVM to predict which category the new data vector belongs too. More specifically a sample data set is extracted from the target data set. It is then categorized based on the predetermined criteria. This data set along with the categorization for each vector in the data set is the training data set, and is then used to *train* the SVM model. Once the model has been trained, the SVM model is ready to be used. The data for each features can be extracted from the new dataset, allowing for the model to classify each new vector. Once classified the developer can take the appropriate course of action. 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 data set is inseparate. In this case, the data set 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 if these features are in some way related to another feature. Other potential steps would be to modify the data to ...

More detail about the specific features used will be given a little later on. Features are descriptive aspects of the data set that are classified into the predetermined categories. Since these features relate directly to the category that they are helping describe any feature must first know what the classification is. 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 methods 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.

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.

Several projects were experimented on to test the purposed method. As noted, acra 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. The complete list of projects that were

Owner	Project	Start Date	End Date	# of Commits	# of Developers
ACRA	acra	2010-04-18	2015-06-05	404	32
apache	storm	2011-09-16	2015-12-28	2445	261
facebook	fresco	2015-03-26	2015-10-30	313	47

Table 4.1: Experiment projects

tested is found in table 4.1. Some simple statistics about the project are also provided such as the start and end date of each project, the number of commits that involved changes to Java source files and the number of developers involved. 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. This gives a more accurate size 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 who committed or authored commits to the current project.

The experimental process for testing the various feature sets on acra involved dividing the repository into four equal quarters (based on time duration). While the number of commits within each period may vary greatly, only a sample is taken.

The SVM model was trained to categorize whether the current method would be changed within the next 6 commits. This of course can generalize to whether a vector will have a commit within the next n commits. Obviously each candidate feature leverages historical or current information and thus a vector can be generated without future information.

The table 4.2 lists each feature with a more detailed description. An example of each feature is provided to further illustrate them. As stated in the previous section 3.4, the values need to be first converted into floating point numbers. 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

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 6 commits for the given method	0	0

Table 4.2: Candidate features for SVM model

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 4.2, 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.

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

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}).

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

Cross validation was then applied, taking a sample from each period of the project.

Two sample sizes were tested, 100 and 1000. It was quickly discovered that maintaining a even distribution vectors categorized as positive (*1*) and negative (*0*) greatly improved the performance of the model. Therefore it is fair more ideal to keep even sets of each category or as close as possible. A simple example of an evaluation set that is even would be one where if the total size of the evaluation set is 100, 50 of the training vectors would be categorized as positive and 50 more would be categorized as negative.

The model was then trained with the sample extracted within the time period of the first period. Once the model was trained the next period's sample set is used to test the model. The prediction made by the model is compared against the actual results. The number of correct predictions is tallied and divided by the sample size.

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 6 commits. Of course since this measurement is calculated, if a vector within the sample set is within the last 6 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 6 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.

It should be noted that while the data set is split into quarters (4 parts) in order for a test to be applied to a quarter it must have data to test with that occurs after. Therefore the final quarter data sample cannot be used to train the model since by

definition no data follows the final quarter.

4.2 Results

The model was trained using various feature sets to determine which feature sets were ideal. Firstly, feature sets that the data could not be separated for any model were marked and not tested further. The vast majority of the tests with the 1000 sample size were inseparable. All of the tests with a 1000 sample size had predictions rates between 40%-70%. Of which most were closer to 50% providing around the same prediction accuracy of that of a coin. While the results from the sample set size of 1000 rather disappointing, the sample size of 100 performed far better. If the candidate feature sets that could not be separated for every quarter are removed the range is 50%-90%. However if the two lowest candidates are dropped the range can be adjusted to 70%-90%. Of this the best candidates can be chosen and focused on further.

The experiment for the top three candidates feature sets were run five times to account for the random sampling. 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. This was performed for the 3 feature sets that scored the highest overall in the first set of experiments. The results proved rather similar with a variation that was to be expected.

The goal of the SVM is to provide a good prediction of whether the a vector will fit in one category or the other. The more often the SVM correctly categorizes different

Project	Quarter 1	Quarter 2	Quarter 3
acra	0.824	0.806	0.87
storm	0.912	0.918	0.912

Table 4.3: Prediction accuracy with sample size of 100

vectors within the data set the better the prediction accuracy. For example, if the SVM is only able to correctly categorize half of the vectors, and fails to categorize the other half, then the model would have a 50% prediction accuracy rate. A prediction of a single vector is verified by using a sample data set of which the actual category can be determined. Since the data set is split into four quarters, the category (of whether a method will be changed in the next 6 commits) is known for the first 3 quarters. The final quarter depending on the size the category is also known except for the changes that occur in the last 5 commits. Therefore at project testing data set can be extracted using the feature data and the category for each quarter (except for the last 5 commits of the 4th quarter). One extracted the model can be trained on one quarter at a time and tested on the following quarter. Since their category has already been categorized the prediction (p_i) that the model provides can be compared against the actual (a_i) value shown in 4.4.

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

The prediction accuracy ($P_{accuracy}$) can then be calculated using 4.5. This simply sums up the accuracy for each vector and then divides it by the total number of vectors (where $n = |v|$).

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

An experiment was run using the feature set of $\{signature, change_i, freq_{change}\}$. The test was run five times with a new random sample each time. The average prediction accuracy of each test was calculated for each project and is shown in table 4.3. The results from the first project are rather standard as numerous other tests were done in order to attempt to find the best feature set for this particular data set. So the particular feature set used performed the best out of all the other feature sets tested. This is just one of the 18 other feature sets that were tested. Of those 19, 8 of them provided a training set which could not be separated and had $43\% \leq P_{accuracy} \leq 68\%$ with an average of 52.7%. These prediction scores are fairly abysmal providing on average a very slight advantage over a simple coin toss. Two more of 18 of candidate feature sets were ruled out because they also had a fairly low score of around $52\% \leq P_{accuracy} \leq 77\%$ and an average of 62.6%. Finally, the remaining candidate feature sets 8 of the 18 had an accuracy of $72\% \leq P_{accuracy} \leq 90\%$ with an average around 80.8%.

The top 3 candidates feature sets were tested and proved to have fairly similar results. The set $\{signature, change_i, freq_{change}\}$ was the smallest feature set and also provided the best results and was selected to test on other repositories. This was to test whether the candidate feature set was generally usable or only worked for the test repository *acra*. This feature set was then tested on other GitHub project repositories. The results for the tests involving the other projects are also shown in table 4.3. While an effort was made to optimize the candidate feature set to perform the best on the *acra* dataset the other project's perform even better.

In particular the project *storm* has a very high prediction score and is much larger than the other two projects shown in table 4.1.

It should be noted that while the SVM model when trained with a random sample size of 100 performed well, when a sample size of 1000 was used the prediction

accuracy was greatly reduced. It would seem that a smaller sample size may prove more helpful regardless of the size of the project.

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 section 3.4. 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

Related Work

Related works here

Chapter 6

Conclusions

Conclusion here

Bibliography

[1] Random, R. How random is everywhere. In *Proc. of the 2nd Work. of Randomness* (Apr. 2012), pp. 34 –41.