Code Review Analysis of Software System using Machine Learning Techniques

Harsh Lal
Aricent, Gurgaon
Email: harshlal028@gmail.com

Gaurav Pahwa
Aricent, Gurgaon
Email: gauravpahwa83@gmail.com

***Abstract*—Code review is systematic examination of a software system's source code. It is intended to find mistakes overlooked in the initial development phase, improving the overall quality of software and reducing the risk of bugs among other benefits. Reviews are done in various forms such as pair programming, informal walk-through, and formal inspections. Code review has been found to accelerate and streamline the process of software development like very few other practices in software development can. In this paper we propose a machine learning approach for the code reviews in a software system. This would help in faster and a cleaner reviews of the checked in code. The proposed approach is evaluated for feasibility on an open source system eclipse. [1], [2], [3]**

***Index Terms*—Code Review, Software development process, Machine learning.**

I. Introduction

An active software development can be seen as a process of sequence of changes which adds to the value of existing software by introducing new features, deleting old ones, enhancing the current implementation, improving the structure of code base as per coding practices, improving scalability and robustness. All of the above mentioned activities risk introducing bugs in the software system. To avert this risk the code is usually peer-reviewed before checking-in the code base. But the peer review being a manual process is prone to errors. Errors may creep in owing to various human reasons like lack of cut-off time for review, workload of the reviewer, domain knowledge of the reviewer, cryptic code etc. Many times it may so happen that code changes are committed directly into the code base without undergoing reviews owing to deadline pressures. So there is a high possibility of the code review being not up to the mark. [5]

This lack of review allows for the bugs to pass unseen to the software code base. These bugs are usually caught at a later time than their inception. These are then assigned to a developer for fixing. In most of the cases the developer has to reacquaint themselves with the code and then fix the bug. This eats up a lot of time and resources in the software development process. Also the fix may be a temporary one and the bug may reappear at a later time, as it is generally a strenuous and time taking task to decipher the root cause of a bug and then fix it at its source.

In large software systems with ever changing requirements, the check-ins are being done at a very frequent rate. Lack of proper review in such cases poses a serious threat of introducing latent bugs. This in turn could causes a lot of issues in the software and would eat up a lot of productive time and resource on rework. To elaborate further the time taken for finding the root cause of the bugs and then fixing it there to avoid further recurrences and maintain the quality of the software system. Had the code changes been carefully scrutinized at the time of commit, the aforesaid bugs would never have had the chance to pass in to the code base thereby saving a ton of resources in the process and improving the quality of software systems as a whole.

In order to deal with the problem mentioned above, a supervised machine learning technique is proposed in conjunction with a version control system and a bug tracking system. Owing to the automated machine learning approach a basic level of code review will always be in place for any check-ins thereby reducing the possibility of a bug passing undetected into the code base. This code review could also be used in addition to the human review and serve as a marker for the human reviewers for the level of attention a code check-in needs thereby making the whole review process more efficient which in turn contributes to better quality software products. [6]

The remainder of paper is structures as follows: We discuss an overview of the systems in section II. We then give a detailed description of problem to which we wish to propose a solution in section III. Then we discuss the possible approaches to solving the problem in section IV. We propose our solution in section V. Moving forward we discuss the results of experiments done in section VI. Section VII talks about the future scope finishing with a conclusion in section VIII.

II. Overview

A. Code Review

Code Review is a systematic examination, which can find and remove the vulnerabilities in the code such as memory

leaks and buffer overflows. A code review allows one or more knowledgeable people to review the code written by another team member. Code reviews allow for a second set of eyes (and perhaps a third and fourth set of eyes) to review the program code. [7]

There are many potential benefits to this activity:

- ***Find bugs faster*** - In many cases, the original programmer cannot catch bugs because he is making assumptions about how events will occur and how the code will respond. He needed to make these assumptions to write the code to begin with. A second person without those same assumptions may turn up potential logic errors that the original programmer did not see. Likewise, the reviewer may see syntax and other common errors that the originator did not see because he was too close.
- ***Enforce construct standards*** - If the organization or team has standards and guidelines to follow in the Construct Phase, the code review can validate that these standards were followed.
- ***Ensure sufficient documentation*** - The person that wrote the program knows what the code does. It is not that obvious to anyone else. The review can ensure that there are sufficient comments to make sure others can understand the logic flow as well.
- ***Cross-training*** - The code review can serve to get others more familiar with programs they did not write.

Of course, there are also obvious costs as well. These fall into two areas. First, it takes time and effort to hold code reviews. This includes preparation time as well as everyone's time that attends the meeting. Second, there can be hurt feelings. When the team is comfortable with code reviews, we become very open to constructive criticism and challenges. However, when we first start, the owner of the code may take offense to other people challenging his work.

B. Review Process

Code reviews mean different things to different people. To some it's a formal meeting with a projector and an entire team going through the code line by line. To others it's getting someone to glance over the code before it is committed. The following describe summarizes the same in general.

Developers generally use branches to implement features and bug fixes. Once the branches are ready for testing, developers request code reviews. Other members of the team review the code from these branches. A list of issues is compiled for each review and passed to the developers. Developers commit additional changes to the branches to fix discovered issues. They then resubmit the review. Once the code review of the branch gets approved then the branch is merged into main code base and shipped to production. [9]

The overall process for code reviews is as follows:

1) Writing code and requesting a review
2) Reviewing code and submitting feedback
3) Fixing the discovered issues and finalizing the review
4) Merging the reviewed code with the code base.

C. Review Tools

There are many code review tools available but for the purpose of study in this paper we focus our attention on Gerrit. Gerrit is a web-based code review tool built on top of the git version control system. Gerrit is intended to provide a lightweight framework for reviewing every commit before it is accepted into the code base. Changes are uploaded to Gerrit but don't actually become a part of the project until they have been reviewed and accepted. Gerrit makes it simple for all committers on a project to ensure that changes are checked over before they are actually applied. Gerrit is firstly a staging area where changes can be checked over before becoming a part of the code base. It is also an enabler for this review process, capturing notes and comments about the changes to enable discussion of the change. This is particularly useful with distributed teams where this conversation cannot happen face to face.

III. Code Review Problem

Code reviews are effective because they provide an alternate perspective to a particular bit of code. These help prevent the chances of getting bad code into the system. But code reviews aren't always perfect, and can often become problematic if they aren't totally understood. There are a bunch of common problems that are being encountered during code reviews as outlined below. [8]

A. Review Problems

The process of code review often turns problematic because of the following main reasons:

1) ***The impersonal nature of code reviews leads to tension and problems*** - Code reviews are almost always conducted via text communications between reviewer and developer. Rarely do the two sit together and review code. This can result in communications challenges between the two parties. In particular, developers are protective of their work. A comment perceived as harsh by the developer can spell disaster and tension between the two parties.
2) ***Code reviews devolve into nit-picking sessions*** - Its easy to descend into a situation where the code review process becomes a nit-picking process, focused on finding every little thing wrong with the code instead of improving the quality of the code and keeping out bad stuff.
3) ***Requests for review are being missed and take days/weeks/months to be considered*** - Because the difference between a developer and a reviewer is usually only their relationship to a particular change proposal, possible reviewers often get busy and miss open patches. And so, review requests can go unanswered for weeks, or even months. This slows down development and wastes everybodys time.

4) ***Code reviews are highly subjective based on who is doing the review*** - Often, developers review code differently from other developers. They have unique nits, different styles, and their own opinions. The same code can be held up by five different people for different reasons, and approved unchanged by a 6th. This is clearly sub-optimal.

B. Basic problem definition

As can be inferred from the above section III-A, there can be a lot of human error involved in the code review process from missed deadlines, to improper reviews, to nit-picking, to interpersonal tension etc. We try to tackle this problem by introducing and automated system for code reviews using machine learning techniques. This would in turn eliminate the human errors and provide a basic level of code review efficiently based on the model generated using earlier bug and review history. Currently we propose a two phase system wherein it is mandatory for the code to be reviewed by the machine learning system and a non-mandatory review by the human agents. The mandatory review serves as a marker for the human agents for the level of severity of a code review. By developing the system and machine learning model further in future we can do away with the non-mandatory review cycle completely.

C. Generic Challenges

One of the main challenges of this paper was generation of a labelled dataset which can be used for model generation. The version control systems (git) do not contain bug information. And the bug tracking systems (Bugzilla) do not contain check-in information. So there is an inherent rift between the two data i.e., commit and bug. No direct mapping is found between the two, so we have to resort to secondary techniques and algorithms for obtaining the same.

Version control system (git) log messages were in turn used for identifying the labels of the initial training data set. But the log messages were found to be inconsistent. Many messages were non descriptive, some even one liners. So it became a tough nut to crack in order to create a labelled training dataset.

Also there were some challenges in scourging version control system for retrieving the needed data. Custom scripts needed to written in order to achieve the task.

IV. Potential Solutions

In this section we outline the approaches that could be used to tackle the problem of code review in the software systems. The main difference lies in type of the data used for model generation. Different algorithms can be used depending on the type of data used. Some of the algorithms that we tried were: 1. Naive Bayes, 2. Maximum entropy, 3. Decision Trees, 4. Support Vector Machines

A. Prediction using Time-series Data

We can build a model around the time-series information of commits and the corresponding labels - ”buggy” or ”clean” as described below in section V. This can be used to show a trend around hours, weekdays or months when the commits made in the version control were most buggy. Based on this information we can predict any incoming check-in for defects.

B. Prediction using Categorical Data

We can build a model around categorical data and the corresponding labels - ”buggy” or ”clean” as described below. Categorical data may include the committer or the author of the change, branch of change etc. This data best captures trends associated with a category. Say for example an author making a lot of buggy commits. Or a branch with complex functionality containing lots of buggy commits. Based on the above information we can predict the probability of a new check-in being buggy.

C. Prediction using Numerical Data

We can build a model using numerical data and the corresponding labels - ”buggy” or ”clean” as described below. Numerical data may include the number of times a file has been changed [10], or the size of the log message or the number of times a particular word appears in the log message. This data can be used to capture the trends and predict the label of a new check-in.

D. Prediction using Textual Data

We can build a model using textual data and the corresponding labels - ”buggy” or ”clean” as described below. Textual data may contain information like snippet of code added or removed. This data captures the change per check-in per file. Thus it can be used as a marker to predict the level of a check-in being buggy or clean.

V. Proposed Approach

As mentioned above a machine learning model can build depending on the type of data used. [17], [18]
In this paper we propose to use the data of the following types:

1) Numeric data
2) Categorical Data
3) Textual Data

We build a separate classifier using the different types of data and then combine the models later using probability summation as describe later in the section.

A. Dataset Labelling

Referring to the white paper on fix inducing changes [6] for software systems we have labelled the check-ins in the code base as ’clean’ or ’buggy’. The labelling was done such that a combination of commit id and filename was considered a unique combination. All such combinations were labelled using the version control system (git) and the bug tracking system (Bugzilla). Here it is assumed that a bug was fixed in a single commit. Also the labelling depended a lot on the

quality of commit message present in the system.

B. Feature Engineering

This section highlights the techniques that has been used to design suitable features to be used for training a machine learning model.
Following are the techniques that has been applied for the task:

1) Data Scraping: As described above the combination of commit id and filename was considered a unique key for each row of the data. Centered around the same data was extracted from the version control system (git) [11] for each such unique combinations. This data consisted of various forms. Some were text, some numeric, some timestamp and some categorical. All such relevant data was extracted to create an initial raw dataset.

2) Feature Selection: Based on the details recorded in the version control system (git) for a particular commit we have used the following fields as features for unique combination of commit id and filename:

- *Revision* - The commit id of the version control system
- *File Name* - The file changed in the commit
- *File Parent* - The hierarchy of the file in the filesystem
- *Type* - The type of change made in the commit (Added/Modified/Deleted/Renamed etc.)
- *Branch Name* - The name of branch on which the change was made.
- *Tag Name* - The name of the tag.
- *Text added* - The source code added in the commit in file
- *Text deleted* - The source code deleted in the commit in file
- *Number of lines added* - The number of added lines in the commit in file
- *Number of lines deleted* - The number of deleted lines in the commit in file
- *Message length* - The length of the commit message.
- *Bug Count* - The number of time word ”bug” appears in the commit message
- *Fix Count* - The number of time word ”fix” appears in the commit message
- *Feature Count* - The number of time word ”feature” appears in the commit message
- *Update Count* - The number of time word ”update” appears in the commit message
- *Committer Id* - The name of the committer of the change.
- *Author Id* - The name of the author of the change.
- *Day of week* - The day of week when the change was committed
- *Hour of day* - The hour of day when the change was committed
- *Times modified* - The number of times a file has been changed prior to the current commit.
- *Label* - The label given to the changed file in a commit (clean/ buggy)

C. Data Pre-processing

Data-gathering methods are often loosely controlled, resulting in out-of-range values (e.g., Income: - 100), impossible data combinations (e.g., Sex: Male, Pregnant: Yes), missing values, etc. Analysing data that has not been carefully screened for such problems can produce misleading results. Thus, the representation and quality of data is first and foremost before running an analysis. [19]

If there is irrelevant and redundant information present or noisy and unreliable data, then knowledge discovery during the training phase is more difficult. Data pre-processing includes cleaning, normalization, transformation, feature extraction and selection, etc. The product of data pre-processing is the final training set.

Following are the techniques that has been applied on the raw dataset for data pre-processing:

1) Imputing values: Imputation is the process of replacing missing data with substituted values. Because missing data can create problems for analyzing data, imputation is seen as a way to avoid pitfalls involved with list-wise deletion of cases that have missing values. Imputation preserves all cases by replacing missing data with an estimated value based on other available information.

Some of the rows of raw data as extracted above contain missing or incorrect data. For example a numeric field such as ”Number of lines added” containing a text value, or a categorical field like ”Type” containing NULL value, or a time-stamp field containing a NULL value etc. Such fields had to be taken care of as they represent spurious data rows.

Following are the ways by which such data values were handled:

1) Ignore the rows containing such spurious values by dropping them.
2) Fill in the missing values using the mean, median or mode of the values that are present for the same column.

Both the techniques were used in our study.

2) Normalizing Values: The range of numeric values in the raw data varies widely. This poses a problem as in some machine learning algorithms, objective functions will not work properly without normalization. For example, the majority of classifiers calculate the distance between two points by the Euclidean distance. If one of the features has a broad range of values, the distance will be governed by this particular feature. Therefore, the range of all features should be normalized so that each feature contributes approximately proportionately to the final distance. Min-Max scaler was used in our study to normalize the numeric values in the raw data.

3) Label Encoding Values: Label encoding is a technique which helps to normalize labels for categorical data such that they contain only values from among the unique class values possible. The unique class values are assigned a number and the categorical data is encoded using these numbers.

4) Vectorizing values: Vectorization is a technique which helps to encode textual data into numeric format using Bag of Words technique. We have used TFIDF vectorization for the purpose of our study. TFIDF (term frequency -inverse document frequency) is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus. It is often used as a weighting factor in information retrieval and text mining. The TFIDF value increases proportionally to the number of times a word appears in the document, but is offset by the frequency of the word in the corpus, which helps to adjust for the fact that some words appear more frequently in general.

Preprocessing was done on the textual columns of raw dataset to make it suitable for feature extraction. Following are the preprocessing steps that were applied:

- Removal of unwanted spaces Removing spaces from feature text.
- Breaking Camel Case words File names usually is written in camel case consisting of different words. All the individual words of the name was broken down.

D. Dimensionality Reduction

It is the process of reducing the number of random variables under consideration, via obtaining a set of "uncorrelated" principal variables. It can be divided into feature selection and feature extraction. We have used Principal Component Analysis for dimensionality feature extraction followed by Select Percentile for feature selection.

E. Solution Approach

As mentioned in section IV the dataset we created consisted of different types of data that can be mainly grouped into three categories: - 1) Numeric, 2) Textual and 3) Categorical

Separate models we generated for each type of data categories and then the result from three generated classifiers were merged to predict the final output. First step included data scraping and imputation. Then the following models were generated:

1) **Numeric Model** - The numeric values were scaled using min-max scaler to normalize the data points. Then a classifier was trained on the data to generate numeric model.
2) **Textual Model** - The textual fields was broken down for camel casing. Then using bag of Words [13] and TFIDF approach the data was vectorized. It was used to train a classifier after passing subjecting it to Principal Component Analysis and Feature selection as mentioned above.
3) **Categorical Model** - The categorical values were encoded using Label Encoding techniques. Then this data was used to train a classifier to generate a categorical model.

Then for any new check-in the above mentioned features V-B2 was extracted and was used to get a probability prediction from all three classifiers. Here we get the probability distribution of the classes "buggy" and "clean" from all the three classifiers. Then we combine these output probabilities from three model by using weighted probabilities wherein the weights for model probabilities were proportional to the number of features (columns) used for model generation. The final probabilities so created are then used to decide the final prediction label. [12], [14]

F. Comparison Metrics

The bug reports (data-points) are classified across the given categories using multi-class classification then the performance of the classifier is evaluated using *precision* and *recall* scores. [15], [16]

- **Precision** is a measure of result relevancy. Precision (P) is defined as the number of true positives (tp) over the number of true positives (tp) plus the number of false positives (fp).

$$P = tp/(tp + fp) \tag{1}$$

- **Recall** is a measure of how many truly relevant results are returned. Recall (R) is defined as the number of true positives (tp) over the number of true positives plus the number of false negatives (fn).

$$R = tp/(tp + fn) \tag{2}$$

VI. Experiments

We tried our experiments with the Semi-Supervised and Unsupervised + Supervised techniques as mentioned above. We tried different algorithms and compared accuracy against the metrics. Below is the description of the experiments done and the results obtained.

Following are the results that has been obtained for various classifiers in tables I, II, III, IV.

TABLE I
NAIVE BAYES

Labels	Precision	Recall
Clean	0.97	1.0
Buggy	1.0	0.76
avg / total	0.98	0.88

VII. Future scope

The current dataset was labelled as "clean" or "buggy" depending mainly on the log messages present in the version control system (git). This process can be improved if there is

TABLE II
MAXIMUM ENTROPY

Labels	Precision	Recall
Clean	0.96	1.0
Buggy	1.0	0.59
avg / total	0.98	0.79

TABLE III
DECISION TREE

Labels	Precision	Recall
Clean	0.97	0.95
Buggy	0.60	0.73
avg / total	0.78	0.84

TABLE IV
SUPPORT VECTOR MACHINES

Labels	Precision	Recall
Clean	0.98	1.0
Buggy	1.0	0.79
avg / total	0.99	0.89

a direct mapping between a bug tracking system (Bugzilla) and the version control system. The reliance on the quality of log messages in the version control can be reduced.

Also in the above mentioned approach we have used a limited number of features available per commit per file. This feature set can be extended to include more direct or indirect features to be available for classification. Here by direct feature we mean the features that can be obtained directly from the version control system, and the indirect features mean the type of features that has to synthesized. One such example of an indirect feature may be the code complexity of the source code added in the commit.

This approach can be tried with multiple classifiers to choose from amongst them for accuracy, precision and recall as the case may be. In future such mature approaches can also be used to eliminate the ambiguities of human review cycle completely.

VIII. CONCLUSION

The main objective of this paper was to propose an automated machine learning approach for the code review cycle in the software systems. This would add an extra level of review and serve as a marker for the level of attention to the review for the human reviewers. We also propose better usage of bug tracking systems for efficient mapping of bug database (Bugzilla in this case) to version control database (git in this case). In this paper we labelled the check-ins made to the code base as 'clean' or 'buggy' based on the commit message in git. When then used the features extracted from version control system (git in this case) to train a classifier. This classifier in turn was used to review the new check-ins made to the code base and got feasible results from the study.

A. Limitation of our approach

We labelled the code check-in as 'clean' or 'buggy' based on algorithm outlined here IV. This process could be made more robust if there a direct mapping of the bug tracking system (git) to the code base (Bugzilla) using some field say branch name or commit id. This would help improve the performance of the classifier and the quality of the software system as a whole.

B. Feasibility of the approach

Although there was no proper labelled dataset to start with, we were able to achieve considerable results using machine learning techniques. We were able to predict a new check-in as clean or buggy based labelling done in accordance with git commit messages.

REFERENCES

[1] https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Quality/Code_Review, Wikipedia, the free encyclopedia

[2] Beller, M; Bacchelli, A; Zaidman, A; Juergens, E (May 2014). *"Modern code reviews in open-source projects: which problems do they fix?"*, Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014).

[3] Bacchelli, A; Bird, C (May 2013). *"Expectations, outcomes, and challenges of modern code review"*, Proceedings of the 35th IEEE/ACM International Conference On Software Engineering (ICSE 2013).

[4] Kemerer, C.F.; Paulk, M.C. (2009-04-17). *"The Impact of Design and Code Reviews on Software Quality: An Empirical Study Based on PSP Data"*, IEEE Transactions on Software Engineering 35 (4)

[5] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, *Brendan Murphy, Change Bursts as Defect Predictors*

[6] Jacek S liwerski, Thomas Zimmermann, Andreas Zeller, *"When Do Changes Induce Fixes?"*

[7] http://www.lifecyclestep.com/browse/435.1CodeReviews.htm, TenStep, Inc. 181 Waterman St. Marietta, GA 30060

[8] http://www.brandonsavage.net/the-pitfalls-of-code-review-and-how-to-fix-them/, Brandon, University of the Pacific in California

[9] http://guides.beanstalkapp.com/code-review/guide-to-code-review.html, Ilya Sabanin, Contributor to Beanstalk Guides

[10] Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl Barr, Premkumar Devanbu, *BugCache for Inspections : Hit or Miss?*, Department of Computer Science - University of California Davis, Davis, CA., USA

[11] Jesus M. Gonzalez-Barahona, Gregorio Robles, Daniel Izquierdo-Cortazar, *The MetricsGrimoire Database Collection*, Universidad Rey Juan Carlos, Bitergia

[12] Lei Xu, Adam Krzyzak, Member, IEEE, and Ching Y. Suen, Fellow, IEEE, *Methods of Combining Multiple Classifiers and Their Applications to Handwriting Recognition*

[13] S. Kim, E. J. W. Jr., and Y. Zhang, *"Classifying software changes: Clean or buggy?"*, IEEE Trans. Software Eng., pp. 181196, 2008.

[14] KE CHEN1, LANWANG, and HUISHENG CHI, *Methods of Combining Multiple Classifiers with Different Features and Their Applications to Text-Independent Speaker Identification*, National Laboratory of Machine Perception and Center for Information Science Peking University, Beijing 100871, China

[15] http://www.cs.cornell.edu/courses/cs578/2003fa/performance_measures.pdf, Performance Measures for Machine Learning

[16] https://www.creighton.edu/fileadmin/user/HSL/docs/ref/Searching_-_Recall_Precision.pdf, Measuring Search Effectiveness

[17] Mattias Liljeson, Alexander Mohlin, *Software defect prediction using machine learning on test and source code metrics*, Faculty of Computing, Blekinge Institute of Technology, SE371 79 Karlskrona, Sweden

[18] Kim Herzig, Sascha Just, Andreas Rau, Andreas Zeller, *Classifying Code Changes and Predicting Defects Using Change Genealogies*, Saarland University

[19] http://scikit-learn.org/stable/modules/preprocessing.html, scikit-learn, Machine Learning in Python