M.Sc. Engg. Thesis

Analysing Developers' Sentiments of Code Reviews: An Empirical Study

by

Toufique Ahmed

Submitted to

Department of Computer Science and Engineering

in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science and Engineering

Department of Computer Science and Engineering

Bangladesh University of Engineering and Technology (BUET)

Dhaka 1000

@Replace 2016

The thesis titled “Analysing Developers’ Sentiments of Code Reviews: An Empirical Study”, submitted by Toufique Ahmed, Roll No. **1014052015**, Session October 2014, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering and approved as to its style and contents. Examination held on @Replace.

Board of Examiners

Candidate's Declaration

This is hereby declared that the work titled "Analysing Developers' Sentiments of Code Reviews: An Empirical Study" is the outcome of research carried out by me under the supervision of Dr. Anindya Iqbal, in the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka 1000. It is also declared that this thesis or any part of it has not been submitted elsewhere for the award of any degree or diploma.

Toufique Ahmed

Candidate

Acknowledgment

Foremost, I am thankful to Almighty Allah for his blessings for the successful completion of my thesis. I would like to express my heartiest gratitude, profound indebtedness and deep respect to my supervisor, Dr.Anindya Iqbal, Assistant Professor, Dept. of CSE, BUET, Dhaka, Bangladesh, for his constant supervision, affectionate guidance and great encouragement and motivation. His keen interest on the topic and valuable advices throughout the study was of great help in completing thesis.

I would also want to thank the members of my thesis committee for their valuable suggestions. I thank Dr. M. Sohel Rahman, Dr. Mohammed Eunus Ali, Dr. Rifat Shahriyar and specially the external member Dr. Swakkhar Shatabda. I express my gratitude towards Dr. Amiangshu S. Bosu, Assistant Professor, Department of Computer Science at Southern Illinois University, USA, for the support and guidance he has extended throughout the whole period of this thesis.

I am especially grateful to Department of Computer Science and Engineering (CSE) of Bangladesh University of Engineering and Technology (BUET) for providing their support during the thesis work. My sincere thanks goes to CSE Office staffs for providing logistic support to me to successfully complete the thesis work.

Finally, I would like to thank my family: my parents and all of those who supported me for their appreciable assistance, patience and suggestions during the course of my thesis.

Abstract

The sentiment (i.e., general positive or negative attitude) towards another person, entity or event significantly influences a person's decision-making process. It is one of the most important factors that influences interactions among the stakeholders for different application areas in many domains. Hence, various types of approaches have been proposed in order to detect sentiment accurately. However, it is not formally analyzed whether sentiments can impact the outcomes of that activity, expressed during software development activities, like peer code review. The objective of this study is to identify the factors influencing review comments and the impact of sentiments on the outcomes of associated review requests. On this goal, we manually rated 1000 review comments to build a training dataset and used that dataset to evaluate eight sentiment analysis techniques. We found a model based on Gradient Tree Boosting (GTB), a supervised learning algorithm, providing the best accuracy to distinguish among positive, negative, and neutral review comments. To the best of our knowledge, this is the first approach that implemented supervised learning methods in the context of code review. We achieved as high as 74% accuracy in sentiment detection which is significantly higher than existing lexicon based analyzers (50% accuracy). We have also validated it with human raters.

Using our GTB based model, we classified 10.7 million review comments from 10 popular open source projects. The results suggest that larger code reviews (e.g., measured in terms of number of files or code churn) are more likely to receive negative review comments and those negative review comments not only may increase review interval (i.e., time to complete a code review) but also may decrease code acceptance rate. Based on these findings, we recommend developers to avoid submitting large code review requests and to avoid authoring negative review comments. The results also suggest that the reviewers authoring higher number of negative

review comments are likely to suffer from higher review intervals and lower acceptance rate. We also found that core developers are likely to author more negative review comments than peripheral developers. However, in case of receiving negative review comments, we did not find any discrepancy between the core developers and the peripheral developers.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

A person's sentiment (i.e., positive or negative attitude) towards another person, entity, or event significantly influences his / her decision-making process such as forming relationships, choosing candidates in a local election, selecting commercial products, reviewing movies, or predicting financial condition of a stock market [1]. Sentiments not only influence the quality of relationship between two persons [2] but also have high impacts on productivity, task quality, task synchronization, and job satisfaction of collaborative activities [3] such as software development [4]. Irrespective of application scenarios, positive sentiment is expected to foster team-work whereas negative sentiment in the exchange of mutual comments is likely to create problem.

We are interested in analyzing the sentiments expressed in some large Open-Source Software (OSS) to find out how sentiments can impact outcomes of the projects. OSS (i.e., Android, GO, Wikimedia etc.) is the type of computer software where the source code made available with a license in which the copyright holder provides the rights to study, change, and distribute the software to anyone and for any purpose. OSS may be developed in a collaborative public manner. The open-source model, or collaborative development from multiple independent sources, generates an increasingly more diverse scope of design perspective than any one company is capable of developing and sustaining in long term. Due to the limited availability of face-to-face communications, OSS developers primarily use various text-based tools such as mailing lists, forums, source code repositories, code reviews, and issue tracking tools to manage their collaborations [5]. Previous Software Engineering (SE) research found developers expressing sentiments in commit messages [4], issue tracking systems [6], and mailing-lists [7]. However, it is not formally investigated whether sentiments expressed during a SE activity can impact the outcome of that activity.

Among the various SE activities, this study focuses specifically on peer code review. *Peer code review* is the practice where a developer submits his/her code changes to a peer to judge its eligibility to be included into the main project code-base [8]. Recently many mature OSS projects as well as commercial organizations have adopted peer code review as a mandatory quality assurance gateway [9]. Developers participating in these projects spend around 10-15% of their time preparing code for reviews or reviewing others code [10]. However, negative sentiments expressed in code review comments may often have adversarial effects whereas positive review may strengthen cooperation. For example, carelessness in wording a review comment can lead to negative feelings from the code author and hinder future collaborations [10]. In this research, we plan to formally study the impact of the code review comments on the participants and present our findings to help the process to improve.

1.2 Objectives

It is worth investigating whether review comments correlate with observable code review outcomes. Also, the factors that influence review comments are not studied so far to the best of our knowledge. The impact of sentiment can be identified either by studying negative impact by negative style of comment or by analyzing positive influence of positive wordings. We consider it sufficient to study one of these. Hence, formally we state the objective of this research as follows:

The objective of this study is to identify the factors influencing negative review comments and the impact of negative review comments on the outcomes of associated review requests. We would also like to find out whether the the reviewers authoring higher number of negative review comments are likely to cause higher review intervals and lower acceptance rate. We are also *interested in the fact that whether the core (experienced) developers and the peripheral (novice) developers author and receive negative review comments in similar rates.*

Since the size of code review comments in a large project is huge (e.g., Chrominus has almost 1330 K comments), it is not feasible to detect the sentiment of the code review manually. To address this issue, we aim to introduce an automated sentiment detection technique which can determine the sentiments of code reviews in software development projects accurately in large scale.

There are many techniques for detecting sentiment of any statement. In the context of code review, it is difficult for the previously used lexical based techniques to determine the sentiments with sufficient precision, as the vocabulary used in these methods are not fully compatible with software engineering domain. Existing works [4, 11] failed to achieve acceptable accuracy (50%) using lexical approaches. Therefore, we aim to develop a supervised machine learning based technique with smart pre-processing and insightful feature selection steps. Since there is no dataset for sentiment analysis of code review and this technique requires a labeled training dataset, we need to build a standard dataset which involves collecting, labeling and validating code reviews. Standard techniques has to be followed to build this dataset with good inter rater reliability.

1.3 Research Hypotheses

We investigate the primary objective (Section 1.2) based on five detailed hypotheses (Figure 1.1). The first three hypotheses (i.e., H1, H2 and H3) investigate the influence of three factors (i.e., code churn, the number of files, and the number of patchsets) on negative review comments. The remaining two hypotheses (i.e., H4 and H5) investigate the impact of negative review comments on the outcomes of associated code reviews in terms of review intervals and code acceptance. These are empirically studied in the research and outcomes are analyzed in Chapter 5.

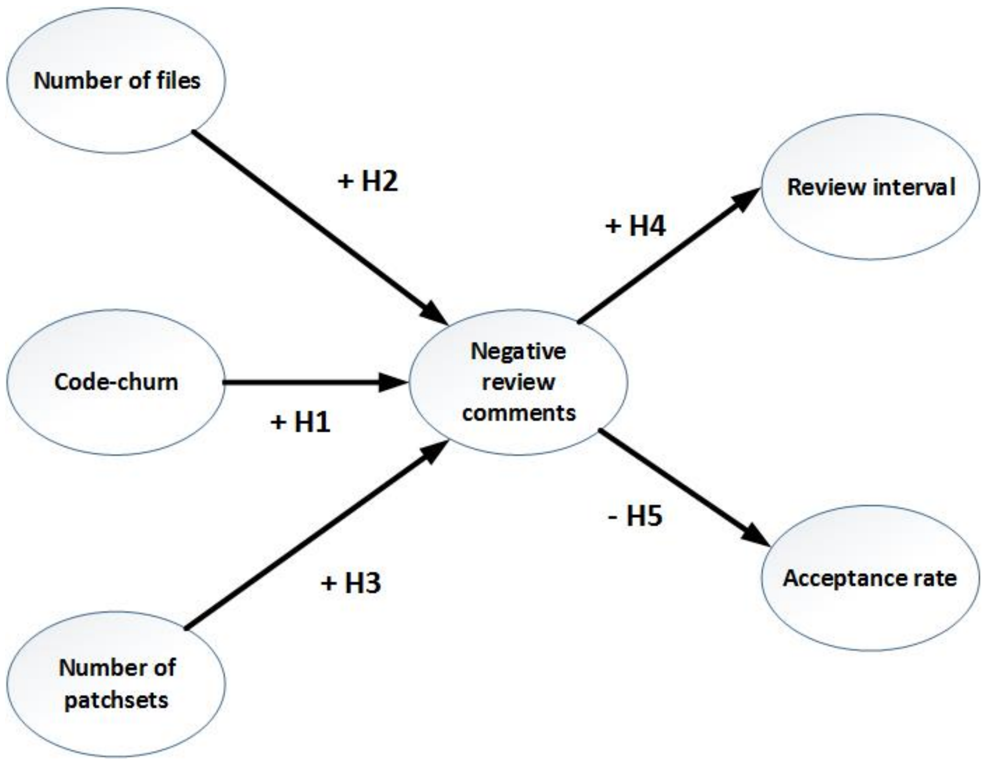


Figure 1.1: Research Hypotheses

1.3.1 Code Churn

Code Churn indicates the total number of lines added, modified or deleted [12] in a review request. Reviewing larger code changes are time-consuming and are often less effective [13]. Reviewers may get annoyed by larger changes and submit negative comments. Moreover, since code churn is also a predictor of defect density [14], larger code changes may have higher number of bugs or improvement scopes increasing the likelihood of negative reviews. Therefore, we hypothesize:

H1: *The code churn is more likely to be higher for a review request receiving at least one negative comment than for a review request receiving no negative comment.*

1.3.2 Number of Files

If there are more files to review, then a thorough review requires more time and effort [13]. A reviewer has to understand the contents and contexts of all files and figure out how the changes work. The extra time and effort required to review higher number of files may displease a reviewer and result in negative comments. Therefore, we pose the following hypothesis:

H2: *The number of files under review is more likely to be higher for a review request receiving at least one negative comment than for a review request receiving no negative comment.*

1.3.3 Number of Patchsets

If a reviewer identifies a problem during code review, s/he suggests changes to resolve the problem. To get the code accepted, the author must upload a new *patchset* (i.e., all files added or modified in a single revision) fixing that problem. The reviewer reviews the new patchset and either accepts it or requests further changes. This process repeats until the reviewer is satisfied with the changes and agrees to accept the change. A negative review comment may indicate the reviewer's objection to a patchset and his/her suggestions for modifications (i.e., a new patchset fixing the issue). Again, since reviewing a patchset is time-consuming, both the author and the reviewer may become annoyed and express a negative sentiment if a code review requires multiple patchsets. Therefore, we hypothesize:

H3: *The number of patchsets is more likely to be higher for a review request receiving at least one negative comment than for a review request receiving no negative comment.*

1.3.4 Review Interval

Review Interval is the time from the beginning to the end of the review process [15]. In this study, we consider a review process to be complete when the patchset status is changed to 'Merged' or 'Abandoned'. An author may take negative review comments personally and may try to defend his / her code. For example, if a reviewer provides constructive criticisms and requests the author to make changes, the author is more likely to oblige. On the other hand, if a reviewer tries to force changes through harsh critique, the author may resist and ask for explanations. Eventually, negative review comments may increase the number of review iterations and therefore increase review intervals. Moreover, negative review comments may degrade the relationship between the author and a reviewer, resulting in delayed responses [16]. Therefore, we pose the following hypothesis:

H4: *The review interval is more likely to be longer for review requests receiving at least one negative comment than for review requests receiving no negative comment.*

1.3.5 Code Acceptance Rate

Code Acceptance Rate is the ratio between the number of review requests submitted and the number 'Merged'. A negative review comment may frustrate the author and dissuade him from completing the suggested changes. Again, a poor quality code, which inherently has higher chance of getting rejected, is more likely to receive negative critiques from the reviewers than a good quality code. Therefore, we hypothesize:

H5: *A review request receiving at least one negative comment is less likely to get accepted than a review request receiving no negative comment.*

1.4 Contribution

We have conducted a three-stage empirical study of code review comments. In the first stage, we mined code review comments from 10 popular OSS projects and manually classified the expressed sentiments in 1000 randomly selected comments to build a training dataset. In the second stage, we evaluated eight sentiment analysis techniques using our dataset and selected a supervised learning technique (i.e., *Gradient Tree Boosting*) as the best performing model. To the best of our knowledge, this is the first approach that implemented supervised learning methods in the context of code review and we achieve as high as 74% accuracy in sentiment detection on validation of dataset. Finally, we used the best performing model to automatically classify 10.7 million review comments from 10 popular OSS projects into positive, neutral, and negative categories. Using this large-scale dataset, we quantitatively investigate the objective of this study.

The primary contributions of this study are:

- An empirically validated automatic model for classifying sentiments expressed in code review comments.
- Empirical evidence on code review outcomes regarding the impact of negative sentiments.
- An empirical study of factors influencing negative review comments.
- Empirical evidence regarding the persons with higher negative review comments influencing the review interval and code acceptance rate.
- An empirical study to see whether the core (experienced) developers and the peripheral (novice) developers author and receive negative review comments in similar rates.
- An empirically built dataset with human validation to train sentiment classification models for SE communications.

1.5 Thesis Organization

The organization of the rest of the thesis is as follows.

In **Chapter 2**, we have also presented relevant background studies on basic vectorization techniques, supervised machine learning algorithms, inter rater agreement and statistical significance tests.

In **Chapter 3**, we have presented literature review on code review and sentiment analysis related to our study.

Chapter 4 briefly explains proposed framework overall technical detail of our methodology.

Chapter 5 focuses on the experimental setups and results. It also illustrates the implications of the results and threats to validity.

Finally, **Chapter 6** concludes our thesis. This chapter also includes the outlines of some future works related to this dissertation.

Chapter 2

Background Study

In this chapter, we have presented background studies related to our research. In Section 2.1, some basic text vectorization techniques are briefly presented. Sections 2.2 and 2.3 discuss about supervised machine learning algorithms and measuring inter rater agreement respectively. Finally, in Section 2.4 statistical significance tests are briefly discussed that will be used in this research for empirical analysis.

2.1 Vectorization Techniques

Vectorization in an important part of pre-processing text data. It is the general process of turning a collection of text documents into numerical feature vectors. It basically determines how text data should be presented to a supervised machine learning algorithm. There are several existing vectorization techniques. Among them TF-IDF vectorization is the most commonly used vectorization technique [17–20]. This section briefly explains TF-IDF vectorization technique which is a combination of count vectorization and TF-IDF transformation.

2.1.1 Count Vectorization

Count vectorization converts a collection of text documents to a matrix of token counts. It counts the occurrences of tokens in each document and presented data in a matrix form. If an a-priori dictionary is not provided and an analyzer is used that does not do some kind of feature selection then the number of features will be equal to the vocabulary size found by analyzing the data.

2.1.2 TF-IDF Transformation

In information retrieval, Term Frequency - Inverse Document Frequency (TF-IDF) 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 TF-IDF 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. It helps to adjust for the fact that some words appear more frequently in general.

In a large text corpus, some words occur very frequently (e.g., “the”, “a”, “is” in English). However, they carry very little meaningful information about the actual contents of the document. If we feed the data count directly to the classifier without using TF-IDF, the low appearance of significant terms (e.g., terms except “a”, “an”, and “The”) will be overshadowed.

TF: Term Frequency, which measures how frequently a term occurs in a document. Since every document is different in length, it is possible that a term would appear much more times in long documents than shorter ones. Thus, the term frequency is often divided by the document length (aka. the total number of terms in the document) as a way of normalization:

$$TF(t) = \frac{(Number\_of\_times\_term\_t\_appears\_in\_a\_document)}{(Total\_number\_of\_terms\_in\_the\_document)} \tag{2.1}$$

IDF: Inverse Document Frequency, which measures how important a term is. While computing TF, all terms are considered equally important. However it is known that certain terms, such as “is”, “of”, and “that”, may appear a lot of times but have little importance. Thus we need to weigh down the frequent terms while scale up the rare ones, by computing the following:

$$IDF(t) = exp\frac{(Total\_number\_of\_documents)}{(Number\_of\_documents\_with\_term\_t\_in\_it)} \tag{2.2}$$

$$TF - IDF(t) = TF(t) * IDF(t) \tag{2.3}$$

2.1.3 TF-IDF Vectorization

TF-IDF vectorizer combines all the options of count vectorizer and TF-IDF Transformer in a single model. Hence, it generates a matrix whose dimensions are equal to the matrix generated by using count vectorization and then it simply replaces the token counts with TF-IDF transformation value.

2.2 Supervised Learning Techniques

The aim of supervised machine learning is to build a model that makes predictions based on evidence in the presence of uncertainty. As adaptive algorithms identify patterns in data, a computer "learns" from the observations. When exposed to more observations, the computer improves its predictive performance. Supervised learning algorithms are very promising for classification and regression problems. Since sentiment detection is a classification problem, we applied different supervised learning algorithms on our dataset (e.g., Naive Bayesian, SVM, SGD, etc.). This section briefly describes some primary supervised machine learning algorithms with ensemble ones (e.g., random forest, adaboosting, and gradient tree boosting algorithms) applied on our dataset.

2.2.1 Primary Supervised Machine Learning Algorithms

Primary machine learning approaches try to learn one hypothesis from training data. In this sub-section, we briefly discuss some primary supervised machine learning algorithms.

Naive Bayes

In machine learning, Naive Bayes classifiers are a family of simple probabilistic classifiers based on applying Bayes' theorem with strong (naive) independence assumptions between the features. Naive Bayes is a simple technique for constructing classifiers: models that assign class labels to problem instances, represented as vectors of feature values, where the class labels are drawn from some finite set. Naive Bayes classifier assumes that the value of a particular feature is independent of the value of any other feature, given the class variable.

Naive Bayes is a conditional probability model. Given a problem instance to be classified, represented by a vector $x = (x_1, x_2, x_3, ..., x_n)$ representing some n features (independent variables), it assigns to this instance probabilities $p(C_k|x_1, x_2, x_3, ..., x_n)$ for each of k possible outcomes or classes C_k.

The problem with the above formulation is that if the number of features n is large or if a feature can take on a large number of values, then developing such a model on probability tables is infeasible. Therefor, reformulate the model to make it more tractable. Using Bayes' theorem, the conditional probability can be decomposed as:

$$p(C_k|x) = \frac{p(C_k)P(x|C_k)}{p(x)} \tag{2.4}$$

In simplified form, using Bayesian probability terminology, the above equation can be written as:

$$posterior = \frac{Prior * likelihood}{evidence} \tag{2.5}$$

Support Vector Machine (SVM)

In machine learning, support vector machines (SVMs, also support vector networks) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall on.

In addition to performing linear classification, SVMs can efficiently perform a non-linear classification using what is called the kernel trick, implicitly mapping their inputs into high-dimensional feature spaces.

In linear support vector machines (SVM-L), kernel is considered as a "linear" equation.

Hence, the categories are divided using straight lines. It scales better with large numbers of samples. This class supports both dense and sparse input and the multi-class support is handled according to a one-vs-the-rest scheme. It uses the full data and solves a convex optimization problem with respect to these data points.

Stochastic Gradient Descent (SGD)

SGD is a very efficient approach to discriminative learning of linear classifiers under convex loss functions. Even though SGD has been around in the machine learning community for a long time, it has received a considerable amount of attention just recently in the context of large-scale learning. This class supports both dense and sparse input and the multi-class support is handled according to a one-vs-the-rest scheme. SGD requires a number of hyper-parameters such as the regularization parameter and the number of iterations. It can treat the data in batches and performs a gradient descent aiming to minimize expected loss with respect to the sample distribution.

2.2.2 Ensemble Learning Algorithms

Ensemble learning is a machine learning paradigm where multiple learners are trained to solve the same problem. In contrast to ordinary machine learning approaches which try to learn one hypothesis from training data, ensemble methods try to construct a set of hypotheses and combine them to use. In many case ensemble learning algorithms improve accuracy. Besides, it also prevents over-fitting. In this sub-section, we briefly discuss some ensemble machine learning algorithms.

Random Forest

In random forests [21], each tree in the ensemble is built from a sample drawn with replacement (i.e., a bootstrap sample) from the training set. while splitting a node during the construction of the tree, the split that is chosen is not the best split among all features. Instead, the split that is picked is the best split among a random subset of the features. As a result of this randomness, the bias of the forest usually slightly increases (with respect to the bias of a single non-random tree) but, due to averaging, its variance decreases. Decrease in variance usually compensates for the increase in bias, hence yielding an overall better model.

AdaBoosting

The core principle of AdaBoost [22] is to fit a sequence of weak learners (i.e., models that are only slightly better than random guessing, such as small decision trees) on repeatedly modified versions of the data. The predictions from all of them are then combined through a weighted majority vote (or sum) to produce the final prediction. The data modifications at each so-called boosting iteration consist of applying weights w_1, w_2, ..., w_N to each of the training samples. Initially, those weights are all set to w_i = 1/*N*, where N denotes number of weak learners, so that the first step simply trains a weak learner on the original data. For each successive iteration, the sample weights are individually modified and the learning algorithm is reapplied to the reweighted data. At a given step, those training examples that were incorrectly predicted by the boosted model induced at the previous step have their weights increased; whereas the weights are decreased for those that were predicted correctly. As iterations proceed, examples that are difficult to predict receive ever-increasing influence. Each subsequent weak learner is thereby forced to concentrate on the examples that are missed by the previous ones in the sequence

Gradient Tree Boosting

Gradient Tree Boosting or Gradient Boosted Regression Trees (GBRT) [23] is a generalization of boosting to arbitrary differentiable loss functions. GBRT is an accurate and effective off-the-shelf procedure that can be used for both regression and classification problems. Gradient Tree Boosting models are used in a variety of areas including Web search ranking and ecology.

Though GBRT-due to the sequential nature of boosting it can hardly be parallelized, it is robust to outliers in output space (via robust loss functions).

Table 2.1: Interpretation of κ value

κ	**Interpretation**
< 0	Poor agreement
0.01 - 0.20	Slight agreement
0.21 - 0.40	Fair agreement
0.41 -0.60	Moderate agreement
0.61 - 0.80	Substantial agreement
0.81 -1.00	Almost perfect agreement

2.3 Measuring Inter Rater Agreement

When a dataset is manually labeled applying for supervised machine learning, it is necessary to measure the agreement among the raters. Fleiss' kappa (named after Joseph L. Fleiss) [24] is a statistical measure for assessing the reliability of agreement between a fixed number of raters when assigning categorical ratings to a number of items or classifying items. This is used when three individuals label the dataset individually. This contrasts with other kappas such as Cohen's kappa, which only work when assessing the agreement between two raters. The measure calculates the degree of agreement in classifications.

If a fixed number of people assign numerical ratings to a number of items, then the kappa will give a measure for how consistent the ratings are. The kappa, κ can be defined as,

$$\kappa = \frac{\bar{P} - \bar{P}_e}{1 - \bar{P}_e} \tag{2.6}$$

The factor $1 - \bar{P}_e$ gives the degree of agreement that is attainable above chance and $\bar{P} - \bar{P}_e$ gives the degree of agreement actually achieved above chance. If the raters are in complete agreement, then κ = 1. If there is no agreement among the raters (other than what would be expected by chance), then $\kappa \leq 0$.

Landis and Koch (1977) [25] gave Table 2.1 for interpreting κ. This s is however by no means universally accepted. Still this is often considered suggestive interpretation.

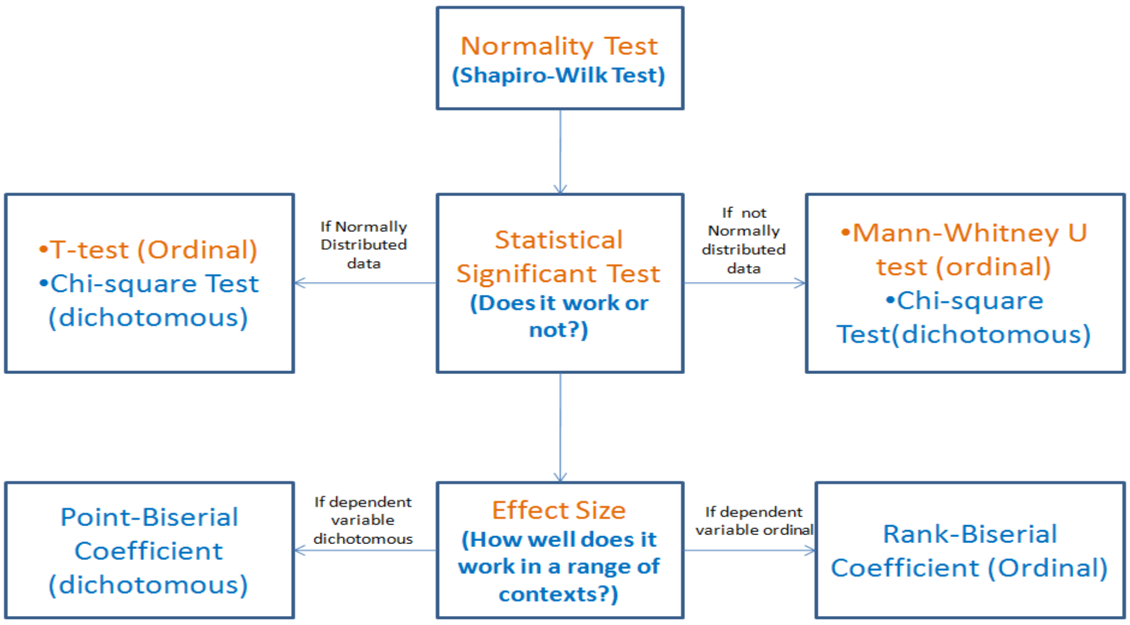

Normality Test
(Shapiro-Wilk Test)
If Normally Distributed data
•T-test (Ordinal)
•Chi-square Test (dichotomous)
Statistical Significant Test
(Does it work or not?)
If not Normally distributed data
•Mann-Whitney U test (ordinal)
•Chi-square Test(dichotomous)
If dependent variable dichotomous
Point-Biserial Coefficient (dichotomous)
Effect Size
(How well does it work in a range of contexts?)
If dependent variable ordinal
Rank-Biserial Coefficient (Ordinal)

Figure 2.1: Consideration for Selecting Statistical Significance Test

2.4 Statistical Significance Test

In statistical significant test, generally two statistical data sets are compared. Alternatively, a data set obtained by sampling is compared against a synthetic data set from an idealized model. It is done to identify whether the two groups originate from same population. To select the proper test method, it is important to identify whether the data follow normal distribution. In Figure 2.1, we have presented different steps of statistical significant test. Depending on the result of normality test (Shapiro-Wilk test), a test is selected to identify whether two groups significantly differ from each other. Finally, determine the magnitude of difference between the two groups, effect size is calculated. The selection of test not only depends on the normality test, but also on the type of variables. For ordinal (continuous) dependent variable (e.g., review interval, code churn etc.), Mann-Whitney U test is applied, whereas for dichotomous (categorical) variable (e.g., code acceptance) chi-square test is applied. In the next sub-section, we present brief discussions on statistical tests relevant to our research.

2.4.1 Shapiro-Wilk test

To make a choice between t-test and Mann-Whitney U test, we need to know whether the dataset follows normal distribution. The Shapiro-Wilk test, proposed in 1965, calculates a W statistic that tests whether a random sample, $x_1, x_2, , x_n$ comes from (specifically) a normal distribution . Small values of W are evidence of departure from normality.

The W statistic is calculated as follows:

$$W = \frac{(\sum_{i=1}^{n} a_i x_{(1)})^2}{\sum_{i=1}^{n} (x_i - \bar{x})^2} \tag{2.7}$$

where $x_{(i)}$ are the ordered sample values ($x_{(1)}$ is the smallest) and a_i are constants generated from the means, variances and covariances of these order statistics of a sample of size n from a normal distribution.

2.4.2 Mann-Whitney U test

Mann-Whitney U test is a non-parametric test that is used to compare two population means that come from the same population. It is also used to test whether two population means are equal or not. It is used for equal sample sizes, and is used to test the median of two populations. Usually the Mann-Whitney U test is used when the data is **ordinal**. Rank of the sample size is used to estimate the U value. Wilcoxon rank sum, Kendalls, and Mann-Whitney U test are similar tests and in the case of categorical data, it is equivalent to the chi-square test.

Mann-Whitney U, being a non-parametric test, does not make any assumptions related to the distribution. There are, however, some assumptions as stated below.

- The sample drawn from the population is random.
- There exits independence within the samples.
- Ordinal measurement scale is applied.

It is estimated as:

$$U = n_1 n_2 + \frac{n_2(n_2+1)}{2} - \sum_{i=n_1+1}^{n_2} R_i \tag{2.8}$$

where U=Mann-Whitney U test,

n_1= sample size one,

n_2= Sample size two,

R_i = Rank of the sample size.

Mann-Whitney U test is frequently used in psychology, medical/nursing, and business. For example, in psychology, it is used to compare attitude or behavior. In medicine, it is used to analyze the effect of two medicines and whether they are equal or not. It is also used to analyze whether or not a particular medicine cures an ailment. In business, it can be used to know the preferences of different people and also to see if that changes depending on location. Different confidence levels can be considered for Mann-Whitney U test and that is indicated by variable p (e.g., p=.05 means 95% confidence level).

2.4.3 Chi-Squared test

Mann-Whitney and independent t-test can be used when the outcome is a ordinal variable. If there is a binary outcome (yes or no), then a Pearson' chi-square test (or Likelihood Ratio) is required to test if the difference between the two groups is significant.

A chi-squared test, also written as χ^2 test, is a statistical hypothesis test wherein the sampling distribution of the test statistic is a chi-squared distribution when the null hypothesis is true. Without other qualification, 'chi-squared test' often is used as short for Pearson's chi-squared test.

The chi-squared test is used to determine whether there is a significant difference between the expected frequencies and the observed frequencies in one or more **categories**. Chi-squared tests are often constructed from a sum of squared errors, or through the sample variance. Test statistics that follow a chi-squared distribution arise from an assumption of independent normally distributed data, which is valid in many cases due to the central limit theorem. A chi-squared test can be used to attempt rejection of the null hypothesis that the data are independent.

A chi-squared test is asymptotically true, meaning that the sampling distribution (if the null hypothesis is true) can be made to approximate a chi-squared distribution as closely as desired by making the sample size large enough. Different confidence levels can be considered for chi-square test and that is indicated by variable p.

2.4.4 Effect Size

To determine the magnitude of difference between the two groups, effect size is calculated. As mentioned earlier, we want to determine whether two groups are different. If two groups are significantly different, effect size can inform the magnitude of difference.

In statistics, an effect size is a quantitative measure of the strength of a phenomenon. Effect sizes generally indicate the correlation between two variables, the regression coefficient in a regression, the mean difference, or even the risk with which something happens. For example, such as how many people survive after a heart attack for every one person that does not survive may be estimated. For each type of effect size, a larger absolute value always indicates a stronger effect. Effect sizes complement statistical hypothesis testing and play an important role in power analyses, sample size planning, and in meta-analyses. They are the first item (magnitude) for evaluating the strength of a statistical claim.

We have presented two ways to determine the effect size. We use rank-biserial correlation coefficient (r_rb) to estimate the effect size (magnitude of difference between the two groups) between a dichotomous vs. an ordinal variable. For a dichotomous vs. another dichotomous variable, we use point-biseral correlation coefficient (r_pb) to estimate the effect size.

Point-Biserial Coefficient

The point-biserial correlation coefficient, referred to as r_{pb} is a special case of Pearson in which one variable is quantitative and the other variable is dichotomous and nominal. The calculations simplify since typically the values 1 (presence) and 0 (absence) are used for the dichotomous variable. This simplification is sometimes expressed as follows:

$$r_{pb} = (Y_1 - Y_0) * \sqrt(pq)/\sigma_Y \tag{2.9}$$

where Y_0 and Y_1 are the Y score means for data pairs with an x score of 0 and 1, respectively. $q = 1 - p$ and p are the proportions of data pairs with x scores of 0 and 1, respectively, and σ_Y is the population standard deviation for the y data. An example usage might be to determine if one gender accomplished some task significantly better than the other gender. In this example, gender variable is dichotomous.

Rank-Biserial Coefficient

The rank-biserial correlation coefficient, r_{rb} is used for dichotomous nominal data vs rankings (ordinal). The formula is usually expressed as

$$r_{rb} = 2 * (Y_1 - Y_0)/n \tag{2.10}$$

where n is the number of data pairs, and Y_0 and Y_1, again, are the Y score means for data pairs with an x score of 0 and 1, respectively. These Y scores are ranks. This formula assumes no tied ranks are present. An example usage might be to determine if one group take more time to accomplished some task significantly than the other group.

2.5 Summary

In this chapter, we have discussed about vectorization techniques, supervised machine learning algorithms, inter rater agreement and statistical significant tests. The next chapter will briefly discuss on contemporary research works on code review and sentiment analysis.

Chapter 3

Literature Review

Sentiment is one of the most important factors that influences interactions for different application areas among stakeholders in many domains. Hence, various types of approaches have been proposed in order to detect sentiment accurately. *Peer code review* is the practice where a developer submit his code changes to a peer to judge its eligibility to be included into the main project code-base [8]. This research aims to analyze impact of sentiment on code review. In this chapter, the contemporary researches on the code review and sentiment analysis have been briefly explained in Sections, 5.6.1 and 3.2, respectively.

3.1 Code Review

Compared with the traditional heavy-weight inspection process [26], peer code review is more informal, tool-based, and used regularly in practice [9]. There are basically two levels of inspection process (e.g., I_1 and I_2) in traditional inspection process (Figure 3.1). Each level of inspection process consists of five steps.

1. **Overview (whole team):** The designer first describes the overall area being addressed an then the specific area he has designed in detail- logic, paths, dependencies, etc. Documentation of design is distributed to all inspection participants on conclusion of the overview. (For an I_2 inspection, no overview is necessary, but the participants should remain the

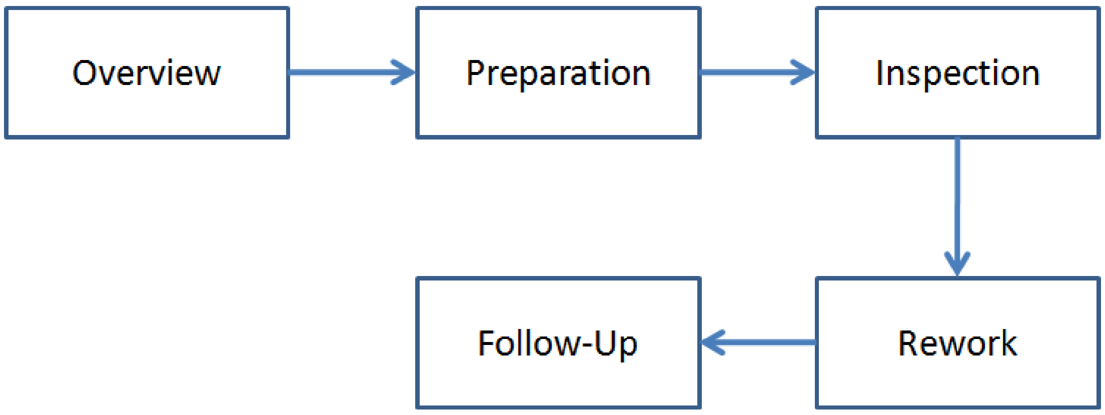


Figure 3.1: Traditional Inspection Process in Code Review

same. Preparation, inspection, and follow-up proceed as for I_1 but, of course, using code listing and design specifications as inspection material. Also, at I_2 the moderator should flag for special scrutiny those area that were reworked since I_1 errors were found and other design changes made.)

2. **Preparation (individual):** Participants, using the design, its intent and logic. (Sometimes flagrant errors are found during this operation, but in general, the number of errors found is not nearly as high as in the inspection operation.) To increase their error detection in the inspection, the inspection team should first study the ranked distribution of error types found by recent inspections. This study will prompt them to concentrate on the most fruitful areas.

3. **Inspection (whole team):** A “reader” chosen by moderator (usually the coder) describes how he will implement the design as expressed by the designer. Every piece of logic is converted at least once, and every branch is taken at least once. All high-level documentation, high-level design specifications, logic specifications,etc., and macro and control block listing at I_2 must be available and present during the inspection. Now the design in understood, the objective is to find the errors. (Note that an error is defines as any con-

dition that causes malfunction or that precludes the attainment of expected or previously specified results. Thus, deviations from specifications are clearly tempered errors.) The finding of errors is actually done during the implementation/ coder's discourse. Questions raised are pursued only to the point at which an error is recognized. It is noted by the moderator: its type is classified, and the inspection is continued. Often the solution of a problem is obvious. If so, it is noted, but no specific solution hunting is to take place during inspection. A team is most effective if it operates with only one problem at a time. Within one day of conclusion of the inspection, the moderator should produce a written report of the inspection and its findings to ensure that all the issues raised in the inspection will be addressed in the rework and follow-up operations.

4. **Rework:** All errors or problems noted in the inspection are resolved by the designer or coder/ implementor.

5. **Follow-UP:** It is imperative that every issue, concern, and error be entirely resolved at this level, or errors that result can be 10 to 100 times more expensive to fix if found later in the process (programmer time only, machine time not included). It is the responsibility of the moderator to see that issues, problems, and concerns discovered in the inspection operation have been resolved by the designer in the case of I_1, or the coder/implementor for I_2 inspections. If more than five percent of the material has been reworked, the team should reconvene and carry out a 100 percent inspection. Where less than five percent of the material has been reworked, the moderator at his discretion may verify the quality of the rework himself or reconvene the team to reinspect either the complete work or just the rework.

The steps mentioned above is certainly time consuming and require several iterations to come into a conclusion. Therefore, many OSS projects have adopted peer code review into their development process since it has been established to be effective [27] and more easy going than the traditional inspection process. Known benefits of peer code reviews include:

1. Detecting defects [28],

2. Maintaining the integrity of a project codebase [27],
3. Improving relationships among the review participants [8],
4. Spreading knowledge, expertise, and development techniques [29],
5. Facilitating the identification and removal of security vulnerabilities [30].

To make peer code reviews more efficient, teams use automated support tools such as Gerrit[1], Phabricator[2], and ReviewBoard[3]. A tool-based code review process starts when an author creates a *patchset* (i.e. all files added or modified in a single revision) along with a description of the changes and submits that information to a code review tool. To facilitate reviews, code review tools highlight the changes between revisions in a side-by-side display (Figure 3.2).

Both the reviewers and the author can insert comments pointing out issues, suggesting improvements, or clarifying the code. After the review, the author may upload a new patch-set addressing the review comments and initiate a new review iteration. This review cycle repeats until either the reviewers approve the changes or the author abandons it. Code review tools capture the interactions (a.k.a. review comments) between the author and a reviewer to facilitate *post-hoc* analyses.

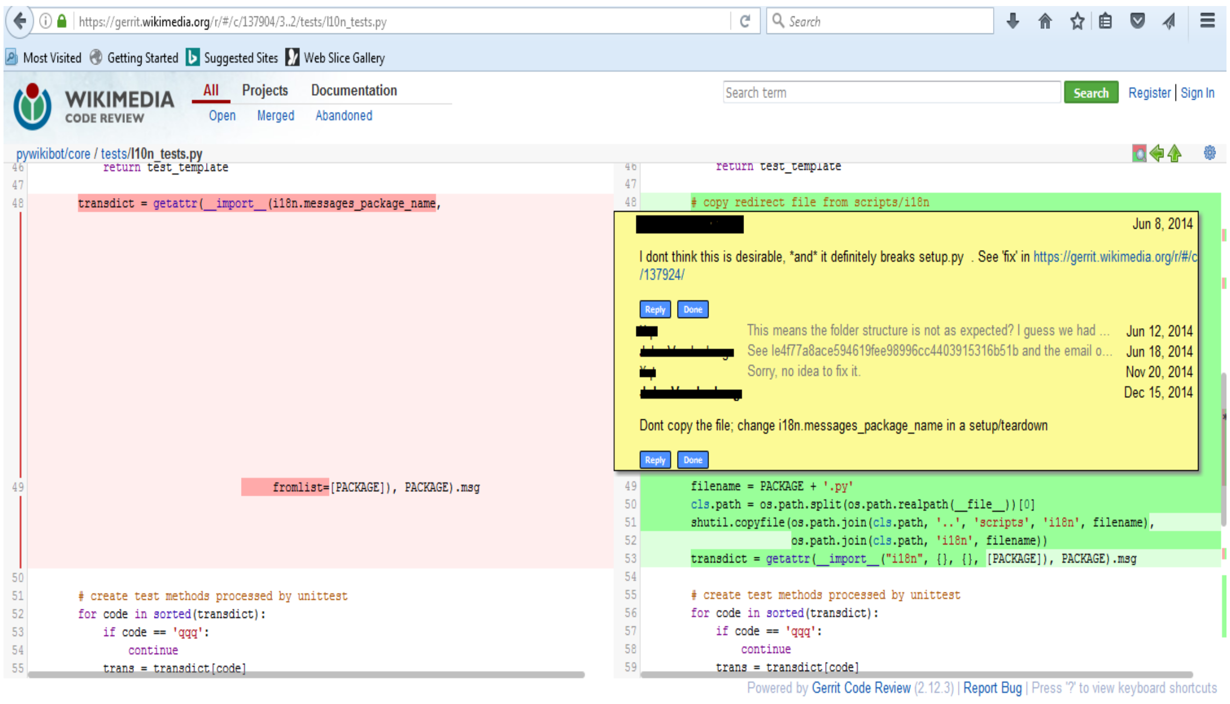
3.2 Sentiment Analysis

Sentiment Analysis is a natural language processing technique that analyzes the attitude of a speaker or an author of a body of text towards entities such as products, services, organizations, individuals, issues or events [31]. Sentiment analysis techniques aim to identify polarity (i.e., positive, negative, or neutral) in a sentence or a paragraph. It also aims to find the factors influencing sentiments and its impact on respective fields.

Researchers have primarily used two types of sentiment analysis techniques.

1. Lexicon-based analyzers.

[1] `https://code.google.com/p/gerrit/`
[2] `http://phabricator.org/`
[3] `https://www.reviewboard.org/`



```
        return test_template

    transdict = getattr(__import__(i18n.messages_package_name,
                                   fromlist=[PACKAGE]), PACKAGE).msg

    # create test methods processed by unittest
    for code in sorted(transdict):
        if code == 'qqq':
            continue
        trans = transdict[code]
```

```
        return test_template

    # copy redirect file from scripts/i18n
    filename = PACKAGE + '.py'
    cls.path = os.path.split(os.path.realpath(__file__))[0]
    shutil.copyfile(os.path.join(cls.path, '..', 'scripts', 'i18n', filename),
                    os.path.join(cls.path, 'i18n', filename))
    transdict = getattr(__import__("i18n", {}, {}, [PACKAGE]), PACKAGE).msg

    # create test methods processed by unittest
    for code in sorted(transdict):
        if code == 'qqq':
            continue
        trans = transdict[code]
```

[redacted] Jun 8, 2014

I dont think this is desirable, *and* it definitely breaks setup.py . See 'fix' in https://gerrit.wikimedia.org/r/#/c/137924/

Reply Done

[redacted] This means the folder structure is not as expected? I guess we had ... Jun 12, 2014
[redacted] See le4f77a8ace594619fee98996cc4403915316b51b and the email o... Jun 18, 2014
[redacted] Sorry, no idea to fix it. Nov 20, 2014
[redacted] Dec 15, 2014

Dont copy the file; change i18n.messages_package_name in a setup/teardown

Reply Done

Powered by Gerrit Code Review (2.12.3) | Report Bug | Press '?' to view keyboard shortcuts

Figure 3.2: An Example Snapshot of Code Review in Gerrit

2. Supervised machine learning based techniques.

Supervised machine learning based techniques that are able to adapt and create trained models for specific purposes and contexts can be used in conjunction with any of the exiting supervised learning methods (e.g., Naïve Bayes, and SVM) [32,33]. Instead of using a standard supervised technique, researchers have also proposed several custom techniques specifically for sentiment classification that take into account the contexts of words expressing sentiments [1]. However, supervised learning techniques require a labeled training dataset, which might be costly or even prohibitive.

On the other hand, lexicon-based analyzers that do not require a training dataset identify the sentiment for a document from the semantic orientation of words or phrases in the document [34]. In a lexicon-based analyzer the sentiment polarity of each lexicon is taken from a predefined dictionary and sentiment is determined from those values. Although lexical methods do not rely on labeled data, it is hard to create a unique lexical-based dictionary to be used for different contexts and often require customization of the dictionary for each domain [35]. Creation of

Table 3.1: An Application based Taxonomy of Existing Research on Sentiment Analysis

Application	**References**
Social Media	[32,36–42]
Product and Movie Reviews	[34,43,44]
Financial Activities	[1,45]
Question Anwering	[46–49]
Software Engineering	[4,6,11,50]

context dependent dictionary is critical and time consuming. Besides, same lexicons may convey different sentiments on different contexts in the same domain. An application based taxonomy of existing research on sentiment analysis is presented in Table 3.1.

Most of the prior research focused on sentiment analysis on social media posts. In [36], the authors used lexicon based analyzer to process twitter data. They introduced POS-specific prior polarity features.Twitter is a social networking and micro-blogging service that allows users to post real time messages, called tweets. Tweets are short messages, restricted to 140 characters in length. Due to the nature of this micro-blogging service (quick and short messages), people use acronyms, make spelling mistakes, use emoticons and other characters that express special meanings. Therefore, the author gave importance on emoticons, target and hashtags. In [37], the authors introduced a novel approach of adding semantics as additional features into the training set for sentiment analysis of twitter data to achieve greater accuracy.

AdaBoost ensemble learning algorithm has been applied to detect sentiments of twitter data in [38]. They used lexicons and feature hashing as the features of the classifier. According to their opinion, sentiment analysis of social media data is challenging because of noisy text, irregular grammar and orthography, highly specific lingo, and others. Moreover, temporal dependencies can affect the performance if the training and test data have been gathered at different times. In [39], another ensemble learning algorithm Gradient Tree Boosting is applied to dataset and depending on the sentiments, a user classification technique in twitter has been developed. The authors applied profile features and tweeting behavior as features of the classifier.

In [32, 40], the authors applied Naive Bayes as classifiers. Sentiment detection is also done using subjectivity analysis [40]. In the community of sentiment analysis, supervised learning techniques have been shown to perform very well. When transferred to another domain, however, a supervised sentiment classifier often performs extremely bad. This is so-called domain-transfer problem. In [32], the authors attempt to attack this problem by making the maximum use of both the old-domain data and the unlabeled new-domain data. To leverage knowledge from the old-domain data, they proposed an effective measure, i.e., Frequently Co-occurring Entropy (FCE), to pick out generalizable features that occur frequently in both domains and have similar occurring probability. To gain knowledge from the new-domain data, they proposed Adapted Nave Bayes (ANB), a weighted transfer version of Naive Bayes Classifier.

In [41, 42], sentiment analysis is done on twitter data using several supervised learning algorithms (i.e. SVM, SGD etc.). A comparative study has been done using supervised learning algorithms in [51]. From this study, it has been established that lexicon based analyzers are not efficient enough to detect the sentiment of software engineering domain and more research is required to achieve better accuracy.

The application of sentiment analysis is not restricted to only social media data, it has been also applied in applications like product review, movie review etc. [34] presents a simple unsupervised learning algorithm for classifying reviews as recommended (thumbs up) or not recommended (thumbs down). The classification of a review is predicted by the average semantic orientation of the phrases in the review that contain adjectives or adverbs. A phrase has a positive semantic orientation when it has good associations (e.g., “subtle nuances”) and a negative semantic orientation when it has bad associations (e.g., “very cavalier”). The semantic orientation of a phrase is calculated as the mutual information between the given phrase and the word ”excellent” minus the mutual information between the given phrase and the word ”poor”. A review is classified as recommended if the average semantic orientation of its phrases is positive. The algorithm achieves an average accuracy of 74% when evaluated on 410 reviews from Epinions, sampled from four different domains (reviews of automobiles, banks, movies, and travel destinations). The accuracy ranges from 84% for automobile reviews to 66% for movie reviews.

[43] presents a new method for sentiment classification based on extracting and analyzing

appraisal groups such as “very good” or “not terribly funny”. An appraisal group is represented as a set of attribute values in several task-independent semantic taxonomies, based on Appraisal Theory. Semi-automated methods were used to build a lexicon of appraising adjectives and their modifiers. Movie reviews have been classified using features based upon these taxonomies combined with standard “bag-of-words” features, and report state-of-the-art accuracy of 90.2%. The authors applied machine learning techniques to detect the sentiment of movie review in [44].

In [45], the authors detect the sentiment of financial news. They introduced a semantically enhanced methodology for the annotation of sentiment polarity in financial news. The proposed methodology is based on an algorithm that combines several gazetteer lists and leverages an existing financial ontology. The financial related news are obtained from RSS feeds and then automatically annotated with positive or negative markers. The outcome of the process is a set of news organized by their degree of positivity and negativity. Sentiment analysis is also used in predicting stock market [1].

Question answering is another area where sentiment analysis was proved useful [46–48]. For example, opinion-oriented questions may require different treatment. Alternatively, Lita *et. al.* [48] suggest that for definitional questions, providing an answer that includes more information about how an entity is viewed may better inform the user.

In [49], by sentiment analysis, the authors refer to the problem of assigning a quantitative positive/negative mood to a short bit of text. They used a sentiment extraction tool to investigate the influence of factors such as gender, age, education level, the topic at hand, or even the time of the day on sentiments in the context of a large online question answering site. They extended this basic analysis by investigating how properties of the (asker, answerer) pair affect the sentiment present in the answer and showed that the best answers differ in their sentiments from other answers.

SE researchers have recently applied sentiment analysis to analyze security discussions [11]. In this research, the authors applied NLTK [52] as sentiment analyzer. In [4], the author applied SentiStrength for sentiment analysis. Both NLTK and SentiStrength are lexicon based analyzers and they are not fully compatible with software engineering domain. Sentiment analysis is also applied to project artifacts [6], and activity of contributors in the Gentoo community [50].

3.3 Summary

In this chapter, we have discussed contemporary research works on code review and sentiment analysis. The next chapter will briefly discuss on the automatic sentiment detection scheme, that we developed with supervised learning methods.

Chapter 4

Automatic Sentiment Detection with Supervised Learning Methods

SentiStrength[1] [4, 7, 11] and NLTK[2] [52] are lexicon based analyzers [50]. However, a recent study [53] observed not only poor accuracies but also significant disagreements among these tools, which could potentially lead to contradictory conclusions. Since those tools are not trained using a SE dataset, their inaccuracies are not surprising. A reliable sentiment analyzer for the SE domain requires either a supervised model trained on a SE dataset or a customized lexicon-based dictionary or both. In this chapter, we present an automatic sentiment detection approach using supervised learning. The whole project comprises three components as depicted in Figure 4.1, i.e., training dataset generation, classifier selection and empirical study. We discuss training dataset generation in Section 4.1. Section 4.2 describes pre-processing of this dataset. Different classifier development and their performance comparison is presented in Section 4.3.

4.1 Training Dataset Generation

In absence of any training dataset, we had to develop one. For any supervised machine learning algorithm, labeled dataset is one of the fundamental elements. To train and validate a sentiment

[1] `http://sentistrength.wlv.ac.uk/`
[2] `http://www.nltk.org/`

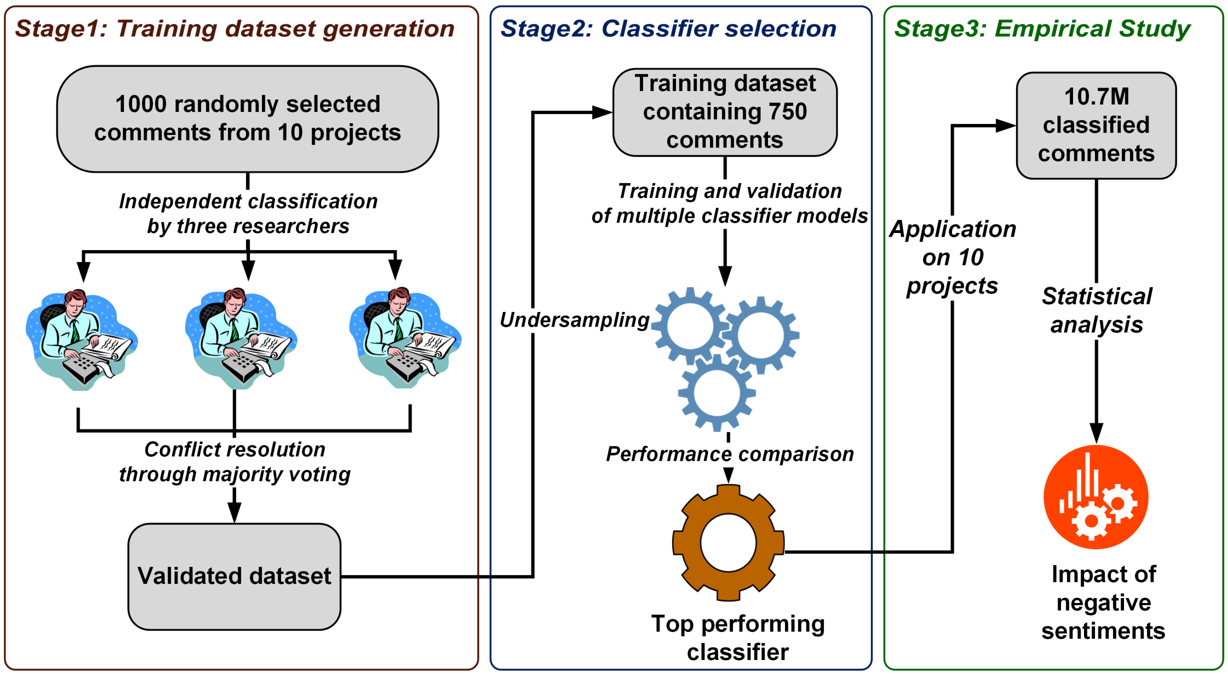

Stage1: Training dataset generation
1000 randomly selected comments from 10 projects
Independent classification by three researchers
Conflict resolution through majority voting
Validated dataset
Stage2: Classifier selection
Training dataset containing 750 comments
Training and validation of multiple classifier models
Undersampling
Performance comparison
Top performing classifier
Stage3: Empirical Study
10.7M classified comments
Application on 10 projects
Statistical analysis
Impact of negative sentiments

Figure 4.1: An overview of our three-stage research method

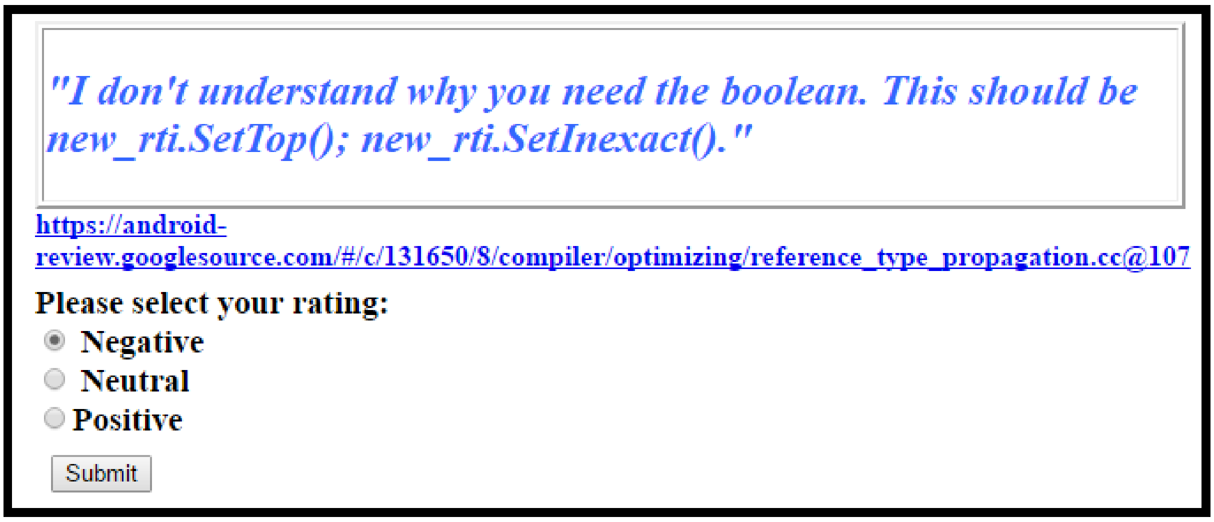
classifier, we manually labeled a dataset using the following process.

1. In April 2016, we used the Gerrit-Miner [30] tool to mine the code review repositories of 18 popular OSS projects. This tool provides detail information about the review requests and the associated software developers.

2. To ensure that we had sufficient data for our analysis, we chose the ten projects (Table 4.1) that each contained more than 10,000 code review requests each.

3. A manual inspection of the comments posted by some accounts (e.g., 'Qt Sanity Bot' or 'BuildBot) suggested that those accounts were automated bots rather than humans. These accounts typically contain one of the following keywords: 'bot', 'auto', 'CI', 'Jenkins', 'integration', 'build', 'travis', or 'verifier'. Because we wanted only code review comments from actual reviewers, we excluded these bot accounts after a manual inspection had confirmed that the comments were automatically generated.

4. We randomly selected total 1000 review comments each having at least 50 characters from the selected ten projects (100 comments from each project).


"I don't understand why you need the boolean. This should be new_rti.SetTop(); new_rti.SetInexact()."

https://android-review.googlesource.com/#/c/131650/8/compiler/optimizing/reference_type_propagation.cc@107

Please select your rating:

[✓ Negative]
Neutral
Positive

Figure 4.2: Web app to we developed manually label the review comments

5. We developed a web-app (Figure 4.2) to manually label the selected review comments. The app shows one review comment at a time and a link to view the comment in the code context. In case of a confusion, a rater could follow the link to better understand the sentiment of a review comment based on the associated context.

6. Two experienced computer science academics and a student independently labeled each of the review comment as ‘positive’, ‘negative’ or ‘neutral’.

7. We used majority voting to determine the final rating of a review comment. When a dataset is manually labeled applying for supervised machine learning, it is necessary to measure the agreement among the raters. We used Fleiss’ Kappa [24] (useful for more than two raters) to measure the level of agreement among the three raters. Because $\kappa = 0.408$ ($p < 0.001$), indicates a “moderate agreement” [25] (see Table 2.1), we could use this manual categorization as a ‘gold set’ for the subsequent analysis. The distributions of the labeled comments were: 8.3% (‘positive’), 17.1% (‘negative’), and 74.6% (‘neutral’).

In the next section, we describe the pre-processing of the dataset.

Table 4.1: Overview of the projects

Project	**Domain**	**Technology**	**Using Gerrit since**	**Requests mined***	**Total Comments**	**+ve comments**	**-ve comments**
Android	Mobile OS	C, C++, Java	October, 2008	80,460	566,863	4.16%	6.66%
Chromium OS	Desktop OS	C, C++	February, 2011	153,484	1,392,877	3.73%	6.51%
Go	Programming	Go, Assembly	November, 2014	10,569	94,736	4.49%	8.23%
ITK / VTK	Visualization Toolkit	C++	August, 2010	20,104	87,120	6.49%	8.52%
LibreOffice	Office Suite	C++	March, 2012	20,627	83,457	5.25%	5.48%
OpenStack	Cloud Computing	Python, JavaScript	July, 2011	281,197	5,627,585	8.35%	5.72%
OVirt	Virtualization	Java	October, 2011	50,824	734,949	2.36%	7.27%
Qt Project	UI framework	C, C++	May, 2011	133,494	868,126	3.50%	6.50%
Typo3	CMS	PHP, JavaScript	August, 2010	37,609	259,771	4.30%	6.43%
WikiMedia	Wiki System	PHP, JavaScript	September, 2011	270,033	1,082,430	4.07%	6.25%
			Total:	**1,058,401**	**10,797,914**		

*Mined during April, 2016

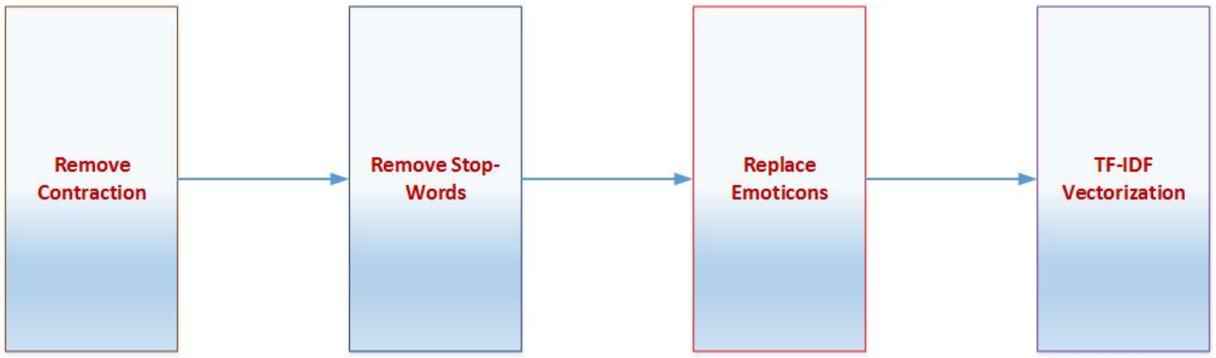


Figure 4.3: Data Pre-Processing Stages

4.2 Data Pre-Processing

Once the dataset is developed, we need to perform some pre-processing to improve the quality of the dataset with an objective to gain good performance in sentiment detection. The text of a review comment differs from articles, books, or even spoken language. For example, review comments often contain word contractions, emoticons, URLs, and code snippets. Therefore, we performed data cleansing steps before training our models. We used a three-step approach (as follows) to clean a document before vectorization (Figure 4.3).

i. Contractions, which are shortened form of one or two words, are widely used in informal written communications. Some commonly used contractions and their expanded forms include: I'm $\rightarrow$ I am, doesn't $\rightarrow$ does not, and don't $\rightarrow$ do not. By creating two different lexicons of the same term, contractions increase the number of unique lexicons and misrepresent the real characteristics of a dataset. We replaced the commonly used 124 contractions, each with its expanded version (appendix C).

ii. Many stopwords (usually non-semantic words such as articles, prepositions, conjunctions, and pronouns) do not play significant roles to express sentiments. Popular natural language processing tools, such as NLTK [52] and Stanford CoreNLP [54] provide lists of stopwords. However, some of the words (e.g., 'no', 'not', 'why', and 'what') in those lists are influential in expressing sentiments. We used a customized stopword list (Table 4.2) and removed words belonging to that list from the review comments.

Table 4.2: List of stopwords

i, me, my, myself, we, our, ours, ourselves, you, your, yours, yourself, yourselves, he, him, his, himself, she, her, hers, herself, it, its, itself, they, them, their, theirs, themselves, this, that, these, those, am, is, are, was, were, be, been, being, have, has, had, having, do, does, did, doing, a, an, the, and, if, or, as, until, while, of, at, by, for, between, into, through, during, to, from, in, out, on, off, then, once, here, there, all, any, both, each, few, more, other, some, such, than, too, very, s, t, can, will, don, should, now

iii. Emoticons are widely used in informal written communication and are very influential in expressing sentiments. Similar to a prior study [36], we replaced each of the emoticons with its sentiment polarity by looking up a emoticon dictionary (prepared by Agarwal et al. [36]). For example, both the happy face emoticon - ':)' and laughing emoticon - ':D' were replaced with 'PositiveEmoticon'. (Appendix D presents the list of emoticons we have considered to replace in our dataset)

Next, we applied different classifiers using this as training dataset and identify the one that performs best in this context.

4.3 Classifier Selection

The poor performance of the lexicon-based analyzers in [4, 7, 11] motivated us to build a sentiment classification model based on supervised learning techniques. We evaluated six commonly used supervised learning algorithms mentioned below.

1. Adaptive Boosting (AdaBoost) [38],
2. Gradient Tree Boosting (GTB) [39],
3. Naïve Bayes [32, 40],
4. Random Forest [51],
5. Linear Support Vector Machine (SVM-L) [32, 41], and
6. Stochastic Gradient Descent (SGD) [42].

Short descriptions of these algorithms are given in Section 2.2.

4.3.1 Design of the Classifiers

Similar to prior studies [17–20], we computed TF-IDF of words (Term Frequency - Inverse Document Frequency) to extract the features for classification. We used sublinear tf scaling, i.e. replace tf with $1 + log(tf)$. It seems unlikely that higher occurrences of a term in a document truly carry higher times the significance of a single occurrence. Accordingly, there has been considerable research into variants of term frequency that go beyond counting the number of occurrences of a term. We used the logarithm of the term frequency. A detail description on vectorization techniques has already presented in Section 2.1.

We used the Scikit-learn [55] implementations of the six commonly used supervised machine learning algorithms. Since our training dataset was imbalanced (i.e., almost 75% comments were 'neutral'), we observed high classification error for the non-neutral comments. Prior studies have used undersampling (i.e., randomly excluding a subset of the majority class) to combat imbalanced training dataset [56,57]. We used undersampling to exclude 250 'neutral' comments from our 'gold set'. The training dataset substantially reduced the classification errors for the non-neutral comments. We validated each of the algorithms using a 10-fold cross-validation [58] and repeated the process 100 times.

4.3.2 Performance Evaluation of the Supervised Learning Methods

Figure 4.4 shows boxplots and means ('X' denotes the mean) based on the accuracies of one hundred 10-fold cross-validations for each of the six supervised learning techniques. The results suggest the traditional methods (i.e., Naïve Bayes and L-SVM) did not perform well. Ensemble technique GTB performed the best (74% accuracy) among the supervised ensemble learning[3] methods (i.e., AdaBoost, GTB, and RandomForest). The performance of SGD was very close to GTB.

We recruited two participants from software industry who had published their email address to externally validate our sentiment detection approach. We randomly selected email addresses in batches of 5. It took 10 emails to recruit these two participants (response rate 20%). The study

[3] methods using multiple learning algorithms to obtain better predictive performance than could be obtained from any of the constituent learning algorithms alone.

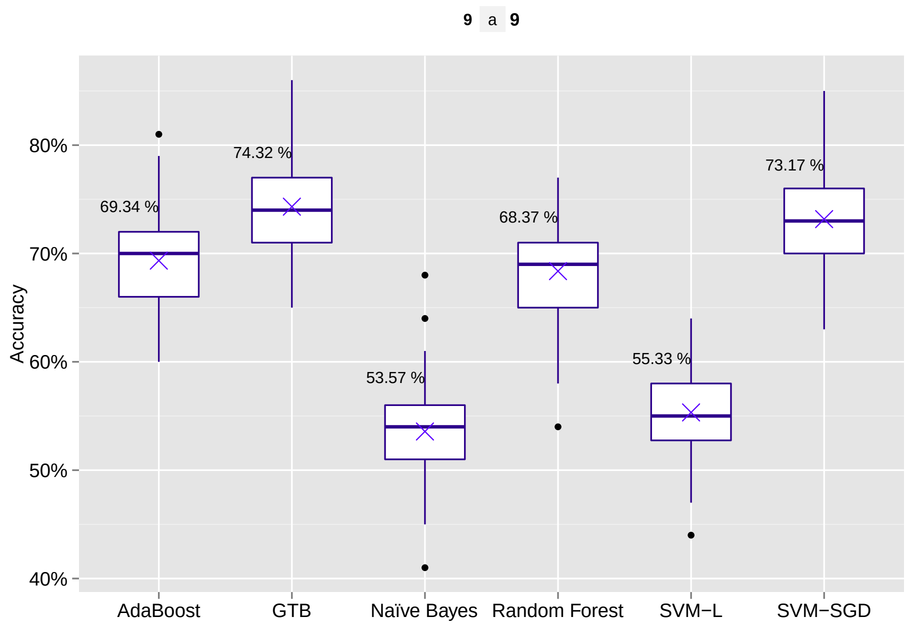


Figure 4.4: Sentiment Detection Accuracy Comparison of the Supervised Learning Algorithms

was conducted using Google Forms and there were no time constraints. To minimize bias, we did not explain the research goal to participants. Each participant was given 100 review comments and they manually rated the sentiments of those comments. Our approach showed 70% and 76% match respectively in detecting sentiments rated by our participants which is very consistent with our cross-validation accuracy (74%).

After the selection of best performing algorithm (GTB), we measured the precision and recall for negative review comments.

Precision (also called positive predictive value) is the fraction of retrieved instances that are relevant. It indicates the proportion of correct positive classifications over all positive classifications. Since our hypotheses are based on negative review comments, we consider negative review comments as relevant instances and classification of negative review comments as positive classification. The precision can be measured as:

$$Precision_i = \frac{M_{ii}}{\sum_j M_{ji}} \tag{4.1}$$

That is, precision is the fraction of events where we correctly declared i out of all instances where the algorithm declared i.

Recall (also known as sensitivity) is the fraction of relevant instances that are retrieved. It suggests the proportion of positive examples that were classified correctly.

The recall can be measured as:

$$Recall_i = \frac{M_{ii}}{\sum_j M_{ij}} \tag{4.2}$$

Hence, recall is the fraction of events where we correctly declared i out of all of the cases where the true of state of the world is i.

The precision and recall are 72.94% and 72.18% respectively. The values are very close to our accuracy. They are acceptable enough to use this classifier for further analysis.

4.3.3 Comparison with Lexicon Based Classifiers

We compared GTB, the best performing method with two lexicon-based analyzers i.e., SentiStrength and a lexicon based analyzer provided by the NLTK [52] which were previously used in software development domain [4, 7, 11]. Both the SentiStrength (accuracy: 52.5%) and the NLTK (accuracy: 50.1%) performed poorly on our dataset because the dictionary used in lexicon based classifiers are not fully compatible with software development domain.

To compare the techniques, we performed each technique 100 times on random set of data and each time our method performs better than SentiStrength and NLTK (Figure 4.5).

To develop an insight on the reason of our better performance, we manually investigated 200 review comments and identified why our proposed supervised learning method works better than SentiStrength in most cases. We also report the cases where StentiStrength performed better than our proposed method. Among the 200 review comments, both our method and SentiStrength classified 99 comments correctly. 58 of the remaining review comments were correctly classified by ours and SentiStrength failed. On the other hand, StentiStrength correctly classified 15 examples that we could not. Neither our method nor SentiStrength could correctly detect the sentiments of 28 comments. Thus we achieved 37% better result compared to SentiStrength.

In appendix A, we present some examples of review comments classified correctly by our

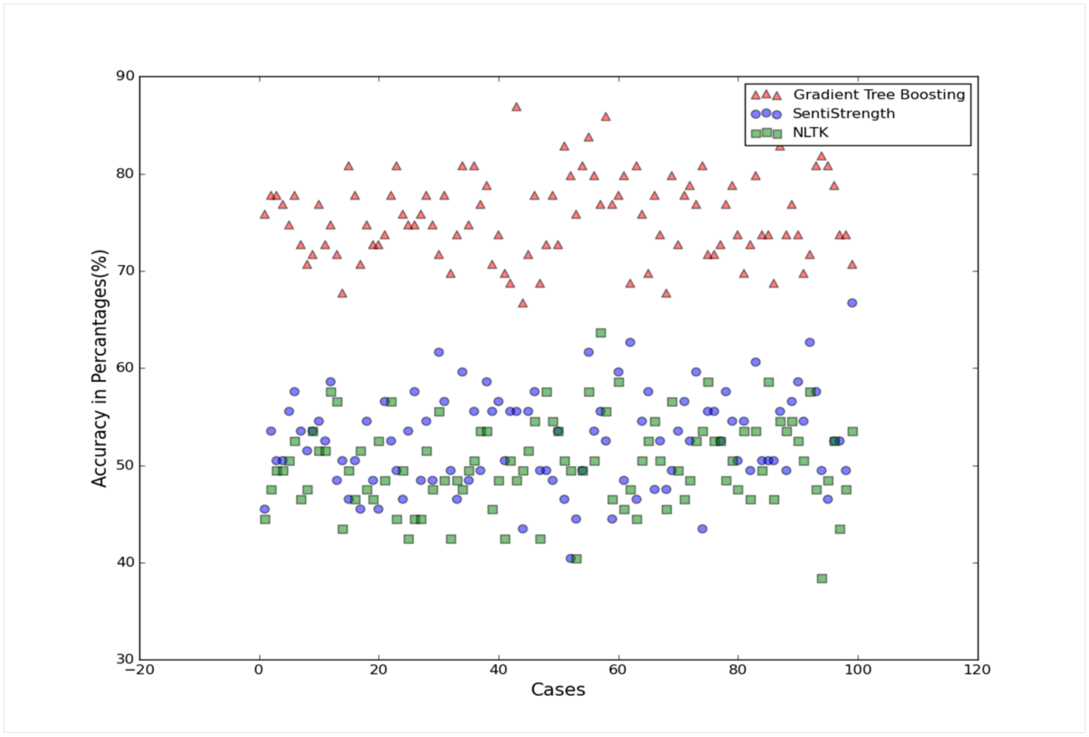


Figure 4.5: Comparing our method with SentiStrength and NLTK on Our Dataset

method but incorrectly by SentiStrength with proper reasonings. In these cases, we have seen that SentiStrength can not get the context of software engineering properly. For example, the word "like" has two common uses in different contexts. One is to indicate preference and the other one is to indicate similarity. In most of the cases in review comments, the word "like" is used to indicate similarity. In all these cases, StrentiStrength wrongly considers it as preference. We found similar problem with words like "understand", "sorry", etc. This is an inherent limitation of lexicon based classifiers that multiple emotions of a single word in different contexts cannot be assigned. Though we are not using contexts directly in our method, it has performed better in these cases since the classifier has learned the examples of both contexts. Moreover, the other features helped our method to perform better than SentiStrength. Hence, we prefer supervised learner for sentiment detection.

In appendix B, we also present some examples of review comments classified incorrectly by our method but correctly by SentiStrength. In these cases, the reason of failure of our method is basically 5W1H tokens (who, what, when, where, which, how). Most of the examples in our labeled dataset with 5W1H are labeled as negative comments. However, in test dataset, some 5W1H carried neutral meaning. Our method also failed to detect sentiment of review comments having “weird” because the examples having “weird” did not fell into training dataset due to random sampling. Such cases may rarely happen as the training dataset is not very large.

4.4 Summary

In this chapter, we have discussed about training dataset generation, data pre-processing and classifier selection. The next chapter will briefly discuss final step of this research, i.e. experimental results, implications of the study, and threats to validity.

Chapter 5

Empirical Studies

In the previous chapter, we have selected a supervised automatic sentiment detection method that performed best in our context. Now, we use it for empirical analysis of code review comments (10.7 million) collected from 10 OSS projects. Design of experimental setup is presented in Section 5.1. We present analyses of factors influencing negative review comments and impact of negative review comments on code review outcomes in Sections 5.2 and 5.3. Sections 5.4 and 5.5 show the impact of individual's sentiment on project outcomes and investigate the relation between the sentiment and the experience of the developers respectively. Section 5.6 presents some extended analysis and Section 5.7 discusses about the implication. Finally, threats to validity of our results are discussed in Section 5.8.

5.1 Design of the Experimental Framework

We designed a database (Figure 5.1) which is used to store data collected from Gerrit. This database stores information about the patch details, request details, author information, in-line comments, and review comments which indicates that a tool-based code review process can store every single detail of code review process. The attribute named "message" both in "in-line_comments" and "review_comments" helps to capture the interactions. We analyzed these comments to work on the hypotheses mentioned in Section 1.3.

We identified and merged multiple Gerrit accounts belonging to the same developer using an

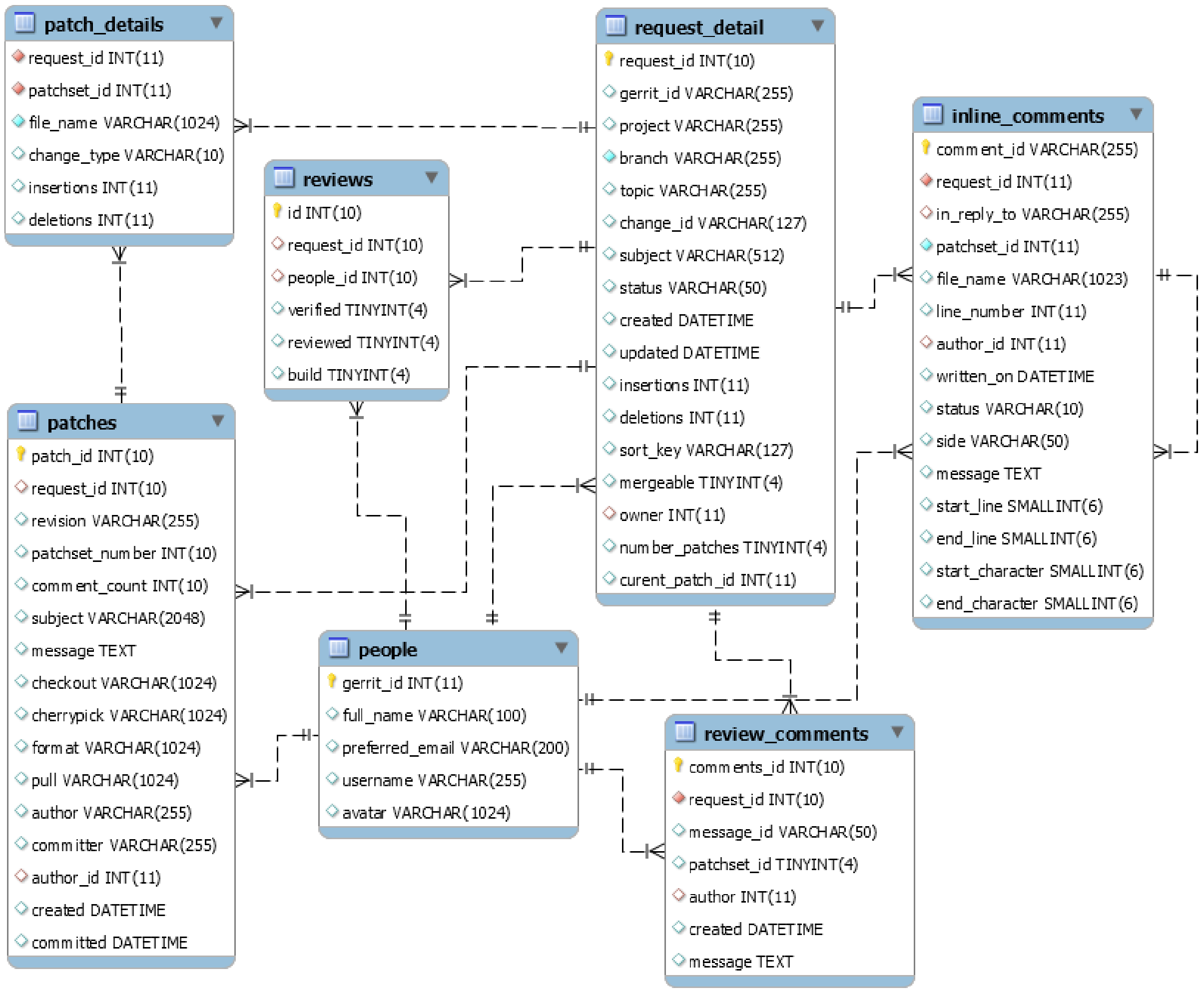


Figure 5.1: ER Diagram of the Code Review Storage

existing approach [59]. Since GTB had the highest mean accuracy among the eight evaluated techniques presented in Chapter 4, we selected a GTB based model trained using the 750 comments for our large-scale analysis. To enable a large-scale empirical study, we applied the GTB model on 10.7 million review comments belonging to around one million code review requests mined from 10 OSS projects. The projects (Table 4.1) present a wide-range of product domains and technologies. We found more than 85% review comments in each of the ten projects classified as neutrals. Among the remaining comments, 2-8% comments were positives and 5-8% comments were classified as negatives. Nine (except OpenStack) out of the ten projects had more negative comments than positives, suggesting negative comments may be more prevalent than positive ones during code reviews in OSS projects. We used **Scikit-learn** for our experimental setup[1]. It is an open source Python library that implements a range of machine learning, preprocessing, cross-validation and visualization algorithms.

We performed Shapiro-Wilks normality tests [60] on all the metrics and found all the distributions significantly differing from a normal distribution. Therefore, we used non-parametric hypothesis tests (Mann-Whitney U and Chi-Square) for each of the hypotheses introduced in Section 1.3 and use median to report the central tendencies. We use rank-biserial correlation coefficient (r_{rb}) [61] to estimate the effect size (magnitude of difference between the two groups) [62] between a dichotomous vs. an ordinal variable. For a dichotomous vs. another dichotomous variable (Receiving at least one negative comments or not vs. code acceptance), we use point-biseral correlation coefficient (r_{pb}) to estimate the effect size. McGrath and Meyer [63] recommends biserial coefficients to be interpreted as follows: $|r_b| \geq 0.10 \rightarrow$ small effect, $|r_b| \geq 0.24 \rightarrow$ moderate effect, and $|r_b| \geq 0.37 \rightarrow$ large effect.

We use Beanplots [64] to visualize and compare the distribution density for multiple samples along the y-axis. Beanplots are best for a large range of non-normal data as they show the entire distribution (they essentially show the full distribution drawn vertically, and show whether there are peaks and valleys in a distribution). We also used it to visualize the highly skewed distributions of review intervals and code churns. The horizontal line in the middle indicates median of the distribution. We use boxplots to visualize highly concentrated data (i.e., number

[1] `http://scikit-learn.org/`

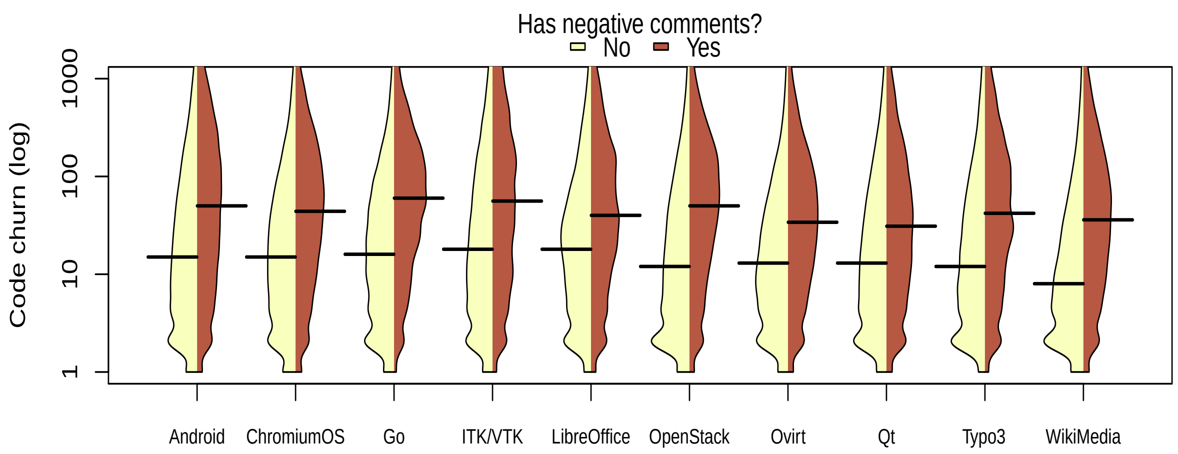


Figure 5.2: Code churn vs. Negative review comments

of files and number of patchsets). For statistical analysis, we used **R**[2]. It is a free software environment for statistical computing and graphics.

Table 5.1 shows the number of negative comments with associated number of review requests. The number of review requests having negative comments is too low and most of the review requests (more than 12%) with negative comment(s) have only one negative comment. Hence, we divided the review requests into two groups on the basis of having at least one negative review comment for our hypotheses.

5.2 Factors Influencing Negative Review Comments

As mentioned earlier in Chapter 1, three factors (i.e. code churn, number of files under review and number of patchsets) are expected influence negative review comments. In the following subsections, we will briefly discuss about those factors.

We define *'negative review ratio'*, as the percentage of review requests that include at least one negative review comment. Using this definition, the following subsections present the results

[2] `https://www.r-project.org/`

Table 5.1: Number of Review Requests for Each Number of Negative comments

Number of Negative Comments per Review Requests	**Number of Review Requests**	**Proportion of Review Requests(%)**
0	722577	75.69493113
1	115649	12.11503146
2	46776	4.9001090523
3	23718	2.484624305
4	13318	1.395152479
5	8370	0.87681530623
6	5515	0.5777343386
7	3955	0.4143135647
8	2880	0.3016998903
9	2106	0.2206180448
10	1544	0.1617446634

of the three hypotheses ($H1$, $H2$ and $H3$) regarding the factors influencing negative review comments.

5.2.1 The code churn is more likely to be higher for a review request receiving at least one negative comment than for a review request receiving no negative comment (H1).

Code Churn indicates the total number of lines added, modified or deleted in a review request. Table 5.2 and Figure 5.2 show the median code churn for the review requests with and without negative review comments on all projects. The code churn is significantly higher (Mann-Whitney U, $p < 0.001$ on all the projects) for the review requests with negative comments on all the projects, supporting H1. The beanplots (Figure 5.2) show lower peaks but longer tails for the review requests with negative comments suggesting code reviews with larger code churn were highly likely to receive negative reviews. The effect sizes, estimated using the rank-biserial coefficient (r_{rb}), suggest moderate-sized effect on two projects and small-sized effect on the remaining eight projects.

Figure 5.3 shows a representative example line-chart (for the OpenStack project) comparing code churn against negative review ratio. The chart indicates negative review ratio increases with code churns and majority of the code reviews with more than 140 line code churns in the OpenStack project encountered negative comments.

5.2.2 The number of files under review is more likely to be higher for a review request receiving at least one negative comment than for a review request receiving no negative comment (H2).

Table 5.3 and Figure 5.4 show the median number of associated files for the review requests with and without negative review comments on each project. The number of associated files is significantly higher (Mann-Whitney U, $p < 0.001$ for all the projects) for the review requests with negative comments on all the projects, supporting H2. The effect sizes, estimated using the

Table 5.2: Hypothesis tests: Code churn (H1)

Project	**Mann-Whitney U**	**Code churn (median)**		**Effect size (r_{rb})**
		-ve comments (no)	**-ve comments (yes)**	
Android	$3.84 \times 10^{8*}$	15	50	0.18^{*}
Chromium OS	$1.52 \times 10^{9*}$	15	44	0.20^{*}
Go	$7.57 \times 10^{6*}$	16	60	0.28^{*}
ITK/VTK	$2.25 \times 10^{7*}$	18	56	0.18^{*}
LibreOffice	$1.80 \times 10^{7*}$	18	40	0.13^{*}
OpenStack	$5.48 \times 10^{9*}$	12	50	0.28^{*}
OVirt	$2.02 \times 10^{8*}$	13	34	0.21^{*}
Qt	$1.18 \times 10^{9*}$	13	31	0.16^{*}
Typo3	$2.03 \times 10^{7*}$	12	42	0.21^{*}
WikiMedia	$1.52 \times 10^{9*}$	8	36	0.23^{*}

$*$Statistically significant at $p < 0.001$ level

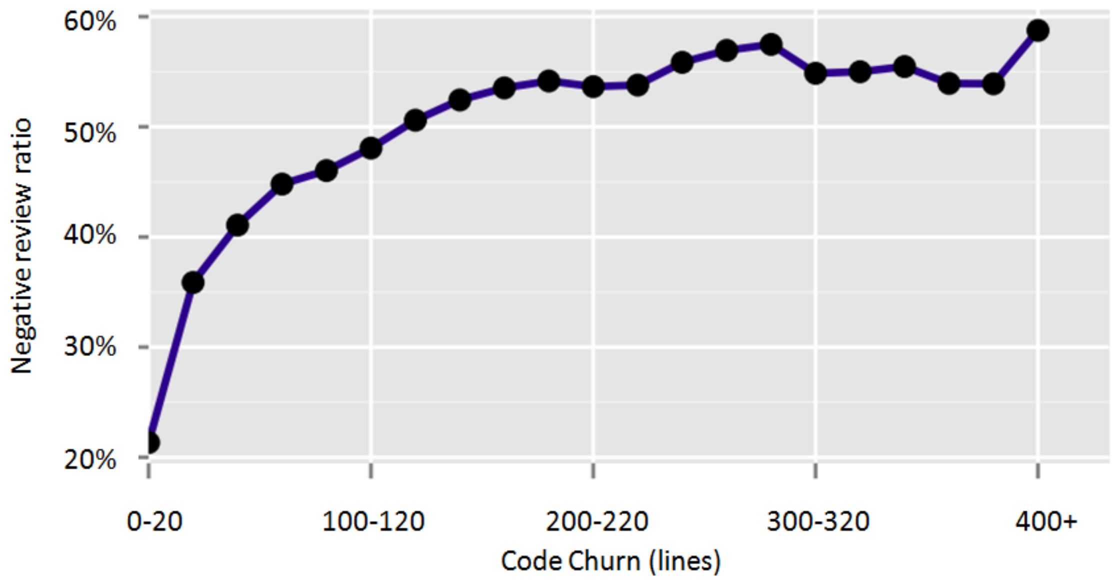


Figure 5.3: Code churn vs. Negative review ratio (OpenStack)

rank-biserial coefficient (r_{rb}) suggest small-sized effect on all the ten projects.

Figure 5.5 shows a representative example line-chart (for the Ovirt project) comparing number of files in a changeset against negative review ratio. The chart indicates negative review ratio increases with the number of files in a changeset and more than one-third of the code reviews with more than two files in the Ovirt project encountered negative comments.

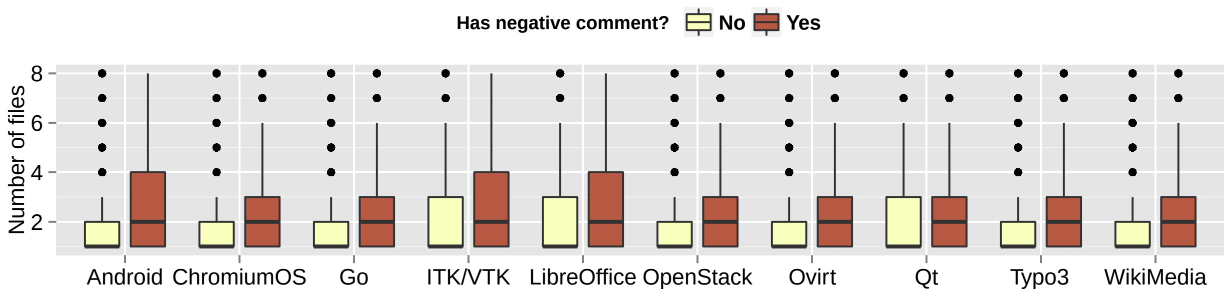


Figure 5.4: Number of files vs. Negative review comments

Table 5.3: Hypothesis tests: Number of Files(H2)

Project	**Mann-Whitney U**	**# of files (median)**		**Effect size (r_{rb})**
		-ve comments (no)	**-ve comments (yes)**	
Android	$4.17\times 10^{8*}$	1	2	0.14*
Chromium OS	$1.67\times 10^{9*}$	1	2	0.15*
Go	$8.66\times 10^{6*}$	1	2	0.21*
ITK/VTK	$2.33\times 10^{7*}$	2	3	0.15*
LibreOffice	$1.93\times 10^{7*}$	2	3	0.10*
OpenStack	$6.32\times 10^{8*}$	1	2	0.20*
OVirt	$2.20\times 10^{8*}$	1	2	0.17*
Qt	$1.28\times 10^{9*}$	2	2	0.11*
Typo3	$2.36\times 10^{7*}$	1	2	0.13*
WikiMedia	$1.73\times 10^{9*}$	1	2	0.19*

$*$Statistically significant at $p < 0.001$ level

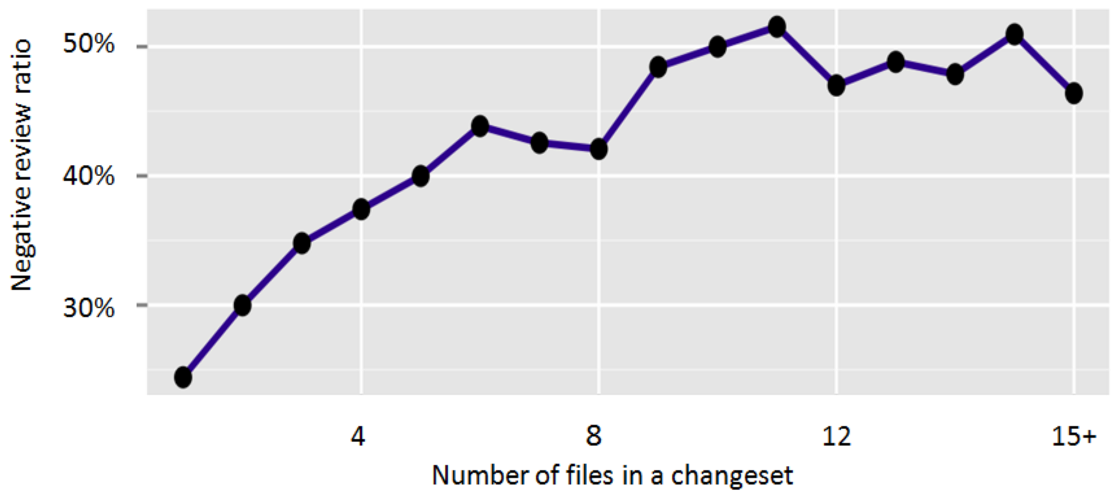


Figure 5.5: Number of files vs. Negative review ratio (OVirt)

5.2.3 The number of patchsets is more likely to be higher for a review request receiving at least one negative comment than for a review request receiving no negative comment (H3).

If a reviewer identifies a problem during code review, s/he suggests changes to resolve the problem. To get the code accepted, the author must upload a new *patchset* (i.e., all files added or modified in a single revision) fixing that problem. The reviewer reviews the new patchset and either accepts it or requests further changes. This process repeats until the reviewer is satisfied with the changes and agrees to accept the change. Table 5.4 and Figure 5.6 show the median number of patchsets for the review requests with and without negative review comments on each project. The number of patchsets is significantly higher (Mann-Whitney U, $p < 0.001$ for all the projects) for the review requests with negative review comments on all projects, supporting H3. The effect sizes, estimated using the rank-biserial coefficient (r_{rb}), suggest large-sized effect on five projects, moderate-sized effect on four projects, and small-sized effect on the remaining one project.

Figure 5.7 shows a representative example line-chart (for the Android project) comparing number of patchsets against negative review ratio. The chart indicates negative review ratio

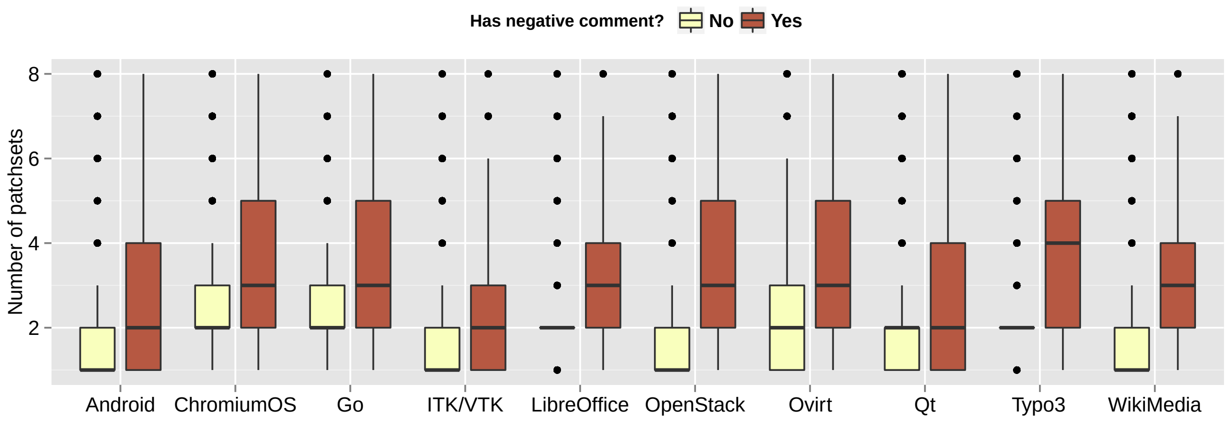


Figure 5.6: Number of patchsets vs. Negative review comments

increases with the number of patchsets and majority of the code reviews with more than four patchsets in the Android project encountered negative comments.

5.3 Impact of Negative Review Comments on Code Review Outcomes

In chapter 1, we mentioned two impacts (i.e. review interval and code acceptance rate) of negative review comments. The following subsections present the results of the two hypotheses ($H4$ and $H5$) regarding the impact of negative comments on code review outcomes.

5.3.1 The review interval is more likely to be longer for review requests receiving at least one negative comment than for review requests receiving no negative comment (H4).

Review Interval is the time from the beginning to the end of the review process. In this study, we consider a review process to be complete when the patchset status is changed to 'Merged' or 'Abandoned'. Table 5.5 shows the median review intervals for the review requests with and without negative review comments on each project. The median review interval is significantly higher (Mann-Whitney U, $p < 0.001$ for all the projects) for the review requests with negative

Table 5.4: Hypothesis tests: Number of patchsets (H3)

Project	**Mann-Whitney U**	**# of patchsets(median)**		**Effect size (r_{rb})**
		-ve comments (no)	**-ve comments (yes)**	
Android	$2.83\times 10^{8*}$	1	2	0.37^*
Chromium OS	$1.26\times 10^{9*}$	2	3	0.30^*
Go	$6.94\times 10^{6*}$	2	4	0.34^*
ITK/VTK	$2.05\times 10^{7*}$	1	2	0.25^*
LibreOffice	$1.80\times 10^{7*}$	2	3	0.15^*
OpenStack	$3.78\times 10^{9*}$	1	4	0.47^*
OVirt	$1.52\times 10^{8*}$	2	4	0.37^*
Qt	$1.05\times 10^{9*}$	2	2	0.24^*
Typo3	$1.27\times 10^{7*}$	2	4	0.40^*
WikiMedia	$1.06\times 10^{9*}$	1	3	0.39^*

$*$Statistically significant at $p < 0.001$ level

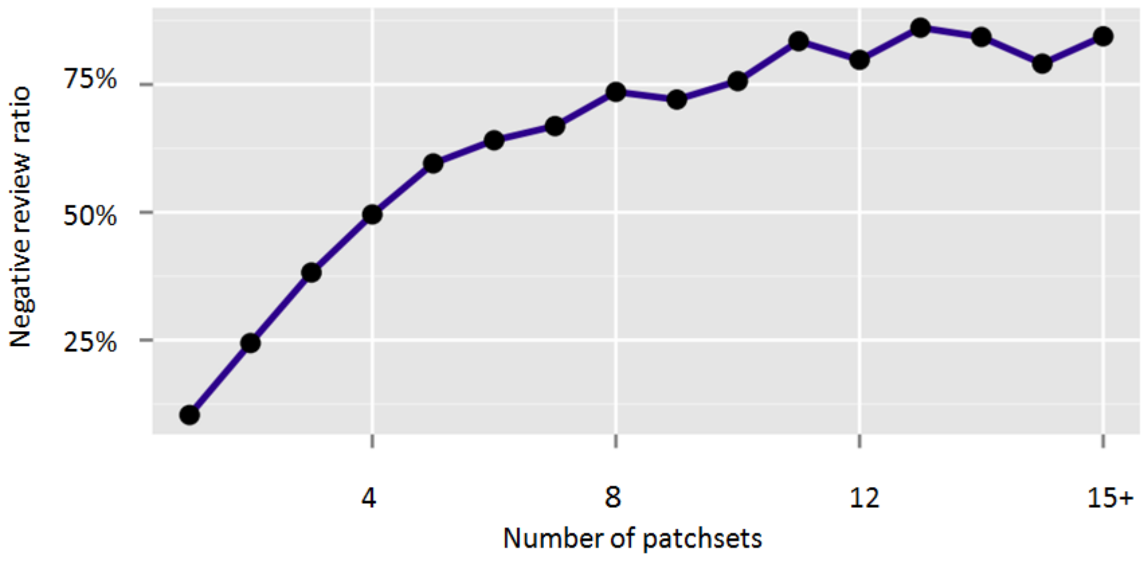


Figure 5.7: Number of patchsets vs. Negative review ratio (Android)

comments on all the projects, supporting H4. The 'Ratio' column in the Table 5.5 shows that in general code reviews with negative comments required between 4 to 232 times more (or 50 to 290 hours longer) to complete the review process. The beanplots (Figure 5.8) of review intervals show long tails for the review requests with negative comment, indicating many review requests with negative sentiments require substantially longer time to complete. The effect sizes, estimated using the rank-biserial coefficient (r_{rb}), suggest moderate-sized effect on two projects and small-sized effect on the remaining eight projects.

5.3.2 A review request receiving at least one negative comment is less likely to get accepted than a review request receiving no negative comment (H5).

Code Acceptance Rate is the ratio between the number of review requests submitted and the number 'Merged'. Table 5.6 shows the acceptance rate for the review requests with and without negative review comments on each project. The acceptance rate is significantly lower (Chi-

Table 5.5: Hypothesis tests: Review interval (H4)

Project	**Mann-Whitney U**	**Median Review Interval (hrs)**		**Ratio**	**Effect size (r_{rb})**
		-ve comments (no)	**-ve comments (yes)**		
Android	$3.84 \times 10^{8*}$	2.9	92.8	32.0	0.18*
Chromium OS	$1.52 \times 10^{9*}$	20.8	87.4	4.2	0.20*
Go	$7.57 \times 10^{6*}$	5.8	57.5	9.9	0.29*
ITK/VTK	$2.26 \times 10^{7*}$	24.4	103.7	4.3	0.18*
LibreOffice	$1.80 \times 10^{7*}$	15.7	80.6	5.1	0.13*
OpenStack	$5.49 \times 10^{9*}$	30.7	246.0	8.0	0.28*
OVirt	$2.03 \times 10^{8*}$	23.9	175.3	7.3	0.21*
Qt	$1.19 \times 10^{9*}$	20.6	113.8	5.5	0.16*
Typo3	$2.03 \times 10^{7*}$	0.4	293.2	732.5	0.21*
WikiMedia	$1.52 \times 10^{9*}$	1.2	114.8	95.7	0.23*

$*$Statistically significant at $p < 0.001$ level

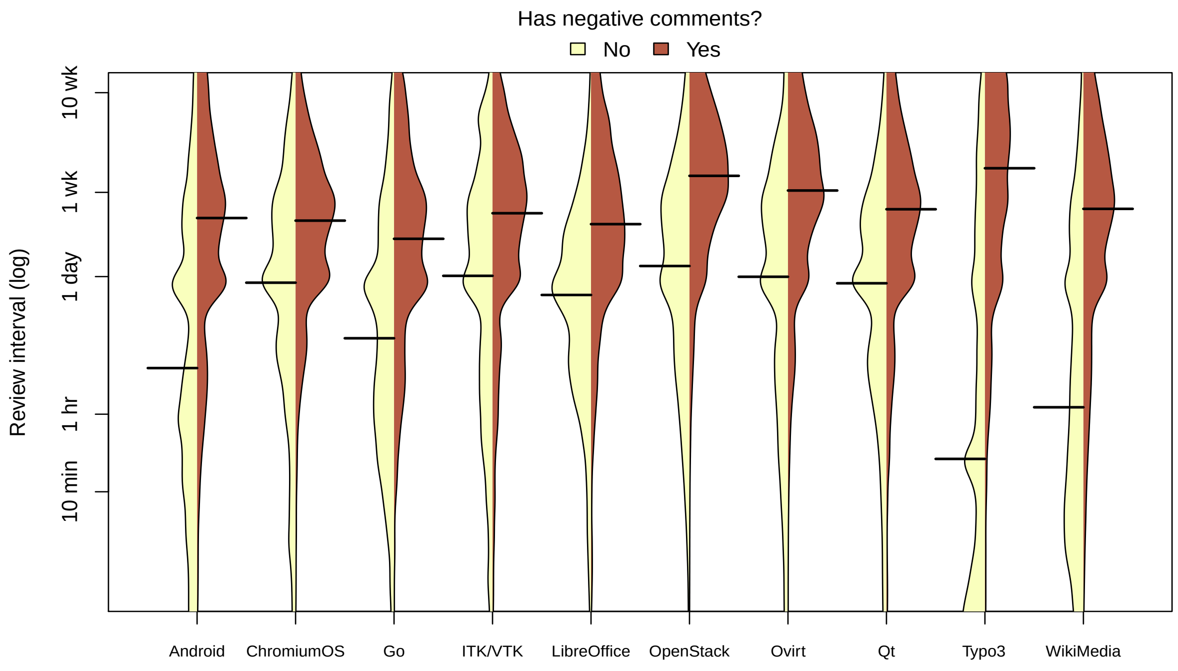


Figure 5.8: Negative review comments vs. Review intervals

Square, $p < 0.001$ on all the projects) for the code reviews with negative comments on all projects, supporting H5. The effect sizes, estimated using the point-biserial coefficient (r_{pb}), suggest moderate-sized effect on one project, small-sized effect on eight projects, and negligible effect on the remaining one project.

5.4 Investigating Individual's Sentiment Pattern

The corporate software companies monitor individual developer's review pattern and make arrangement to help them improving sentiment or if necessary transfer them to another department/role. OSS community also may be benefited if individual reviewer knows his/her relative position in terms of frequency of making harsh comments and develops self-awareness. From this motivation, we study the sentiment pattern of individual reviewer's comments. For 10 projects we are studying, we do not try to identify same person in multiple projects. Rather we consider a reviewer's all comments of a single project for this analysis.

Table 5.6: Hypothesis tests: Acceptance rate (H5)

Project	**Chi Square** (χ^2)	**Acceptance rate**		**Effect size** (r_{pb})
		-ve comments (no)	**-ve comments (yes)**	
Android	1188.2*	84.9%	73.4%	-0.12*
Chromium OS	421.8*	86.2%	81.8%	-0.05*
Go	206.7*	92.2%	82.8%	-0.14*
ITK/VTK	801.8*	83.4%	62.9%	-0.20*
LibreOffice	1141.9*	91.4%	68.7%	-0.24*
OpenStack	3627.1*	84.5%	75.0%	-0.11*
OVirt	833.02*	89.5%	80.0%	-0.13*
Qt	5762.4*	88.4%	70.2%	-0.21*
Typo3	915.3*	94.7%	79.4%	-0.22*
WikiMedia	4192.7*	92.6%	80.7%	-0.15*

$*$Statistically significant at $p < 0.001$ level

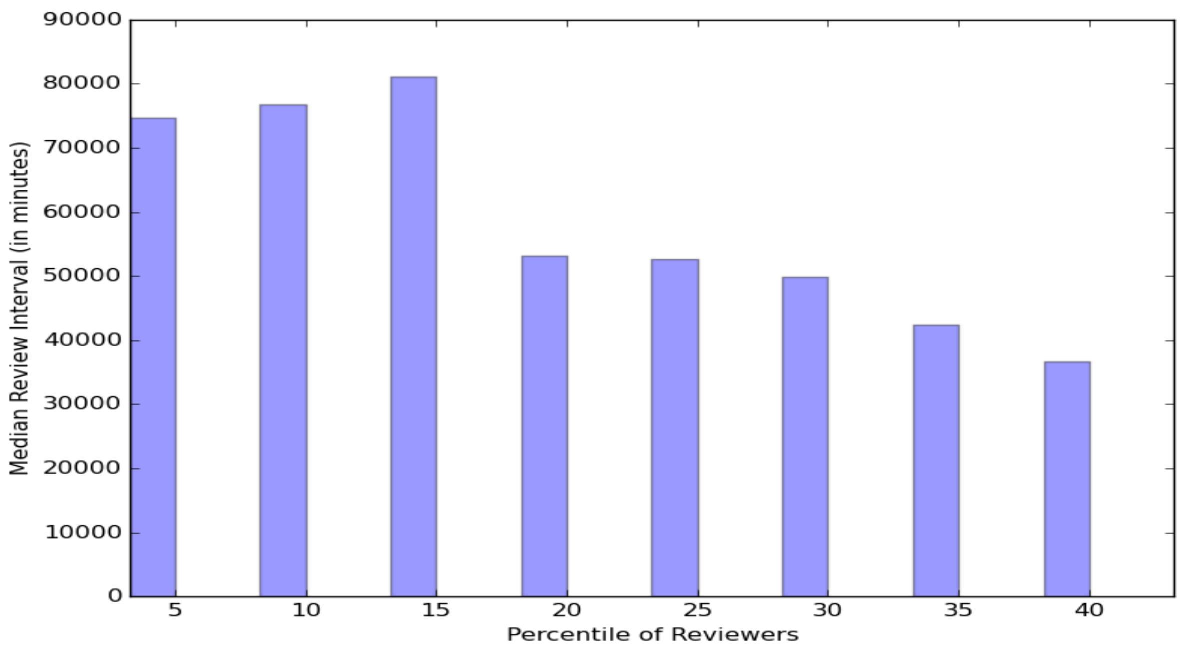


Figure 5.9: Review Interval Required for Different Percentile of Reviewers in Terms of Negative Comments

Figure 5.9 shows the impact of individual's negative sentiment on review interval. For this experiment, we selected the developers who authored more than 5% negative review comments and worked on more than 100 review requests. We sort them in descending order according to their percentage of authored negative review comments and divided them into 20 groups each consists of equal number of developers. In Figure 5.9, we found that the people in first 15 percentile author review requests having more review intervals than others. The review interval gradually decreases for the developers with lower percentage of authored negative review comments. Therefore, developers authoring more negative review comments are likely to increase the development time. We reported median value of review interval for each percentile group.

Figure 5.10 shows the impact of individual's sentiment on code acceptance rate. We selected the developers on the basis of similar restrictions we applied while selecting developers for analyzing review interval. In Figure 5.10, we found that the people in first 5 percentile involve in review requests having lower code acceptance rate than others. The code acceptance rate gradually increases for the developers with lower percentage of authored negative review comments.

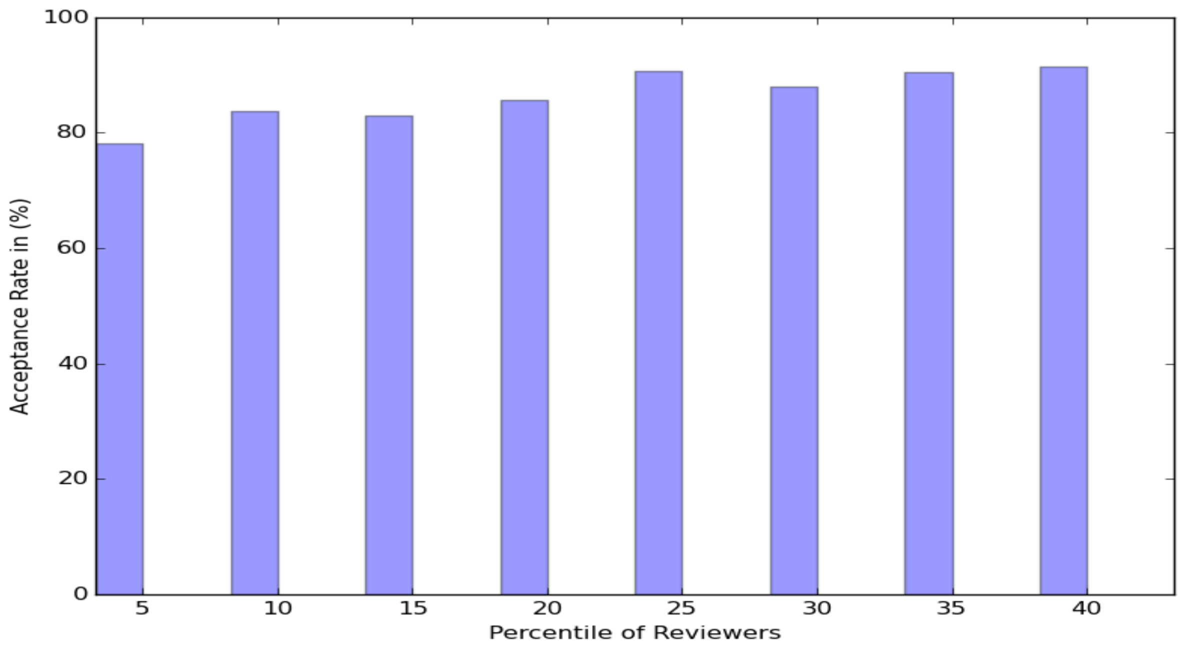


Figure 5.10: Code Acceptance Rate for Different Percentile of Reviewers in Terms of Negative Comments

Therefore, developers authoring more negative review comments are likely to involve in review requests with lower code acceptance code.

In Figure 5.11, we present the distribution of developers according to their authored percentage of negative comments. We found that most of the developers (authoring more than 5 % negative comments) authored negative review comments around 5%-10% of the total comments made by them. Another observation is that the developers who have worked in more than 100 review comments have authored maximum 36% negative comments. We presented how many reviewers may fall into five percentile, ten percentile, and fifteen percentile. The reviewers authoring more than 16.56%, 13.76%, and 12.26% of negative comments likely to fall into five percentile, ten percentile, and fifteen percentile respectively. Moreover, they author on average 20.43%, 15.17%, and 12.94% of negative review comments. Reviewers up to first fifteen percentile likely to increase the review interval of review request. Therefore, proper initiative can greatly reduce the review interval.

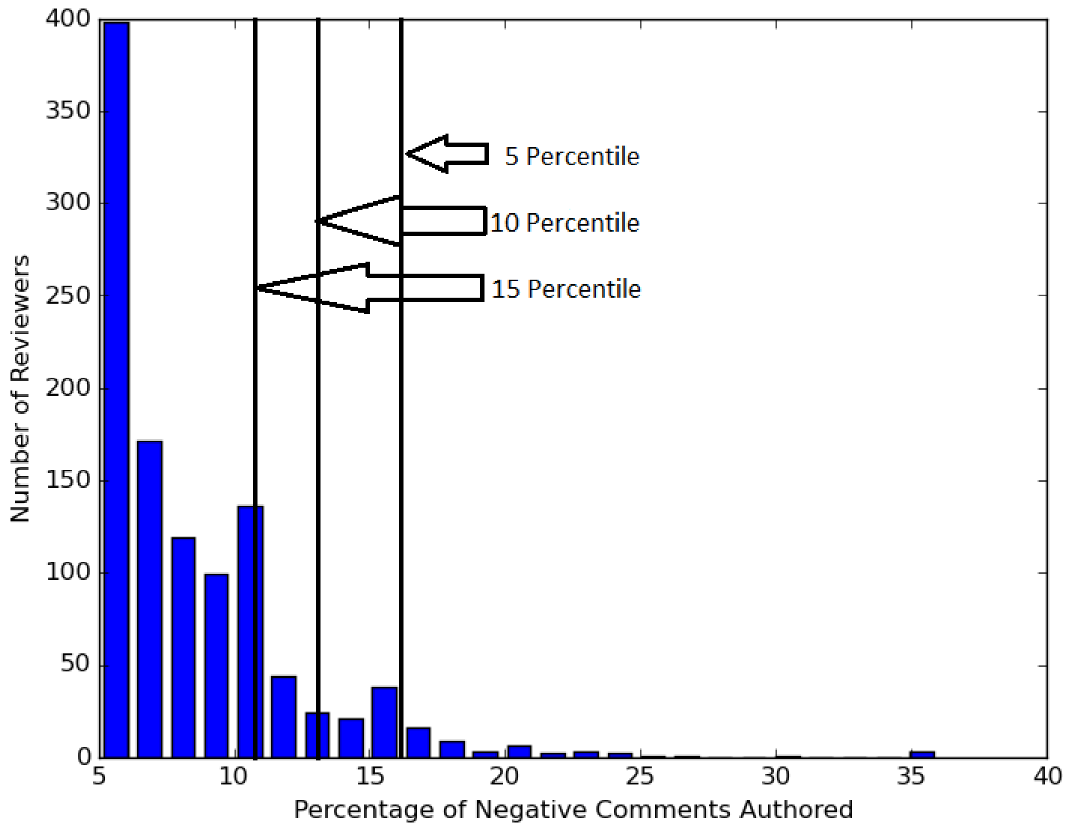


Figure 5.11: Distribution of Developers According to their Authored Percentage of Negative Comments

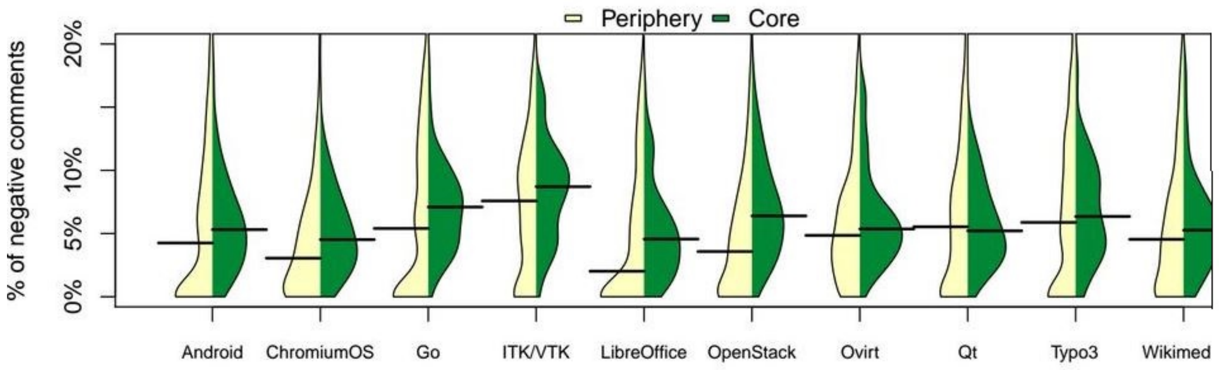


Figure 5.12: Core Developers likely to Author more Negative Comments than the Peripheral Developers

5.5 Authoring and Receiving Negative Comments by the Core (Experienced) and the Peripheral (Novice) Developers

The small set of core developers are those who have been involved with the OSS project for a relatively long time and make significant contributions to guide the development and evolution of the project. The larger set of peripheral developers occasionally contribute to the project, mostly interact with the core developers, and rarely interact with other peripheral developers. Figure 5.12 shows that in 9 project the core developers authored more percentage of negative review comments than the peripheral developers. Only in Qt project, the peripheral developers authored more percentage of negative comments than the core developers. Therefore, we can suggest that the core developers are likely to author more negative comments than the peripheral developers.

From Figure 5.13, we found that in 5 projects the core developers received more percentage of negative comments than the peripheral developers and in other 5 projects the peripheral developers received more percentage of negative comments than the core developers. Therefore, we can not develop any conclusive observation on this issue.

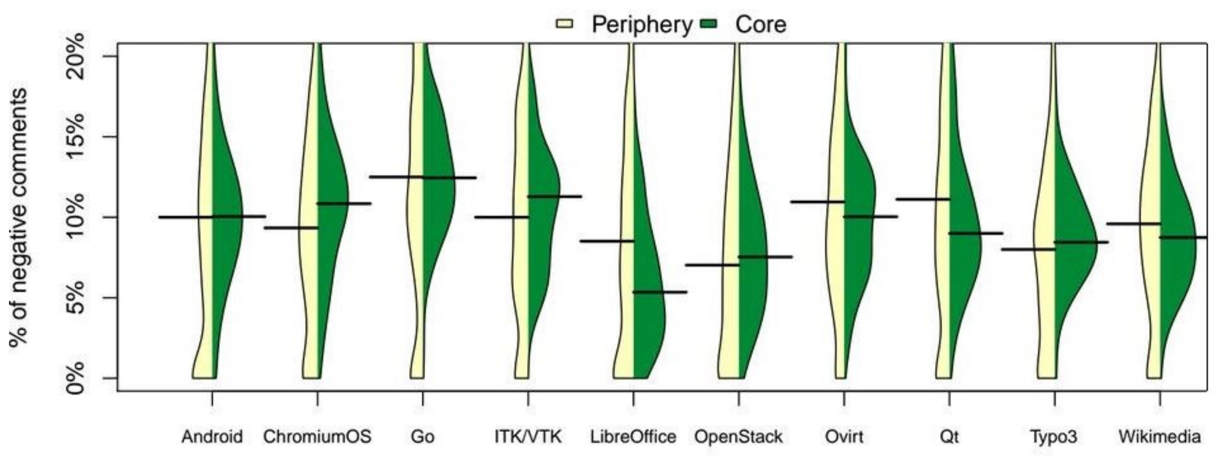


Figure 5.13: Core Developers and Peripheral Developers likely to Receive Equal Percentage of Negative Review Comments

5.6 Extended Analysis

In this research, we have discussed about three factors (i.e., code churn, number of files under review and number of patchsets) that are expected to influence negative review comments and two impacts (i.e. review interval and code acceptance rate) of negative review comments. In the following subsections, we investigate the relations among the factors influencing negative review comments and the impacts.

5.6.1 Code Churn vs. Review Interval

In Figure 5.14, we present the relation between code churn and review interval. We divided the review requests into six groups. The review requests of the first group do not contain any negative comment and those of the other groups contain different number of negative review comment(s) (i.e., 1, 2, 3 etc.). We worked with code churn up to 500 lines. For our analysis, we divided the review requests on the basis of code churn into 20 bins (i.e., 1-25, 25-50,..., 476-500 etc.) and estimated the median review interval of each bin for all six groups. From Figure 5.14, we can suggest that the review intervals for the review requests without any negative comment increase negligibly with code churn compared to the requests with negative review comment(s). The review interval increases consistently with number of negative comments. Therefore, it may be

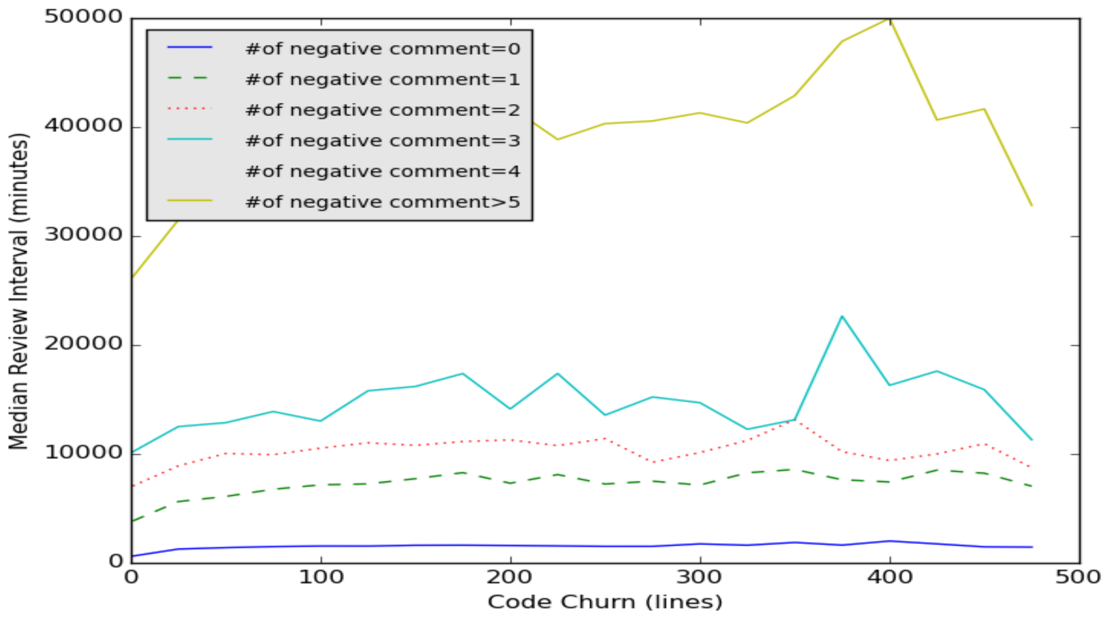


Figure 5.14: Code Churn vs. Review Interval

concluded that negative review comment(s) play(s) a significant role in deciding the relationship between code churns and review intervals.

5.6.2 Code Churn vs. Code Acceptance Rate

In Figure 5.15, we present the relation between code churn and code acceptance rate. We followed a similar experimental setup mentioned in Section 5.6.1 for this analysis. We observe that the code acceptance rate decreases with increase in code churn for review requests without any negative comment. However, the acceptance rate hardly varies and no trend is found for review requests with negative review comments. Hence, negative review comments offset the impact of increase in code churn and code churn by itself does not have considerable impact on acceptance rate. From Figure 5.15, we observe that the code acceptance rate without any negative comment remains higher than review requests with negative comment(s) up to code churn with 250 lines. In our dataset, 89% review requests have code churn less than 250. Hence, the whimsical pattern after this demarcation line is due to insufficient sample. Therefore, it can be concluded that negative review comments decrease the code acceptance rate irrespective of churn size.

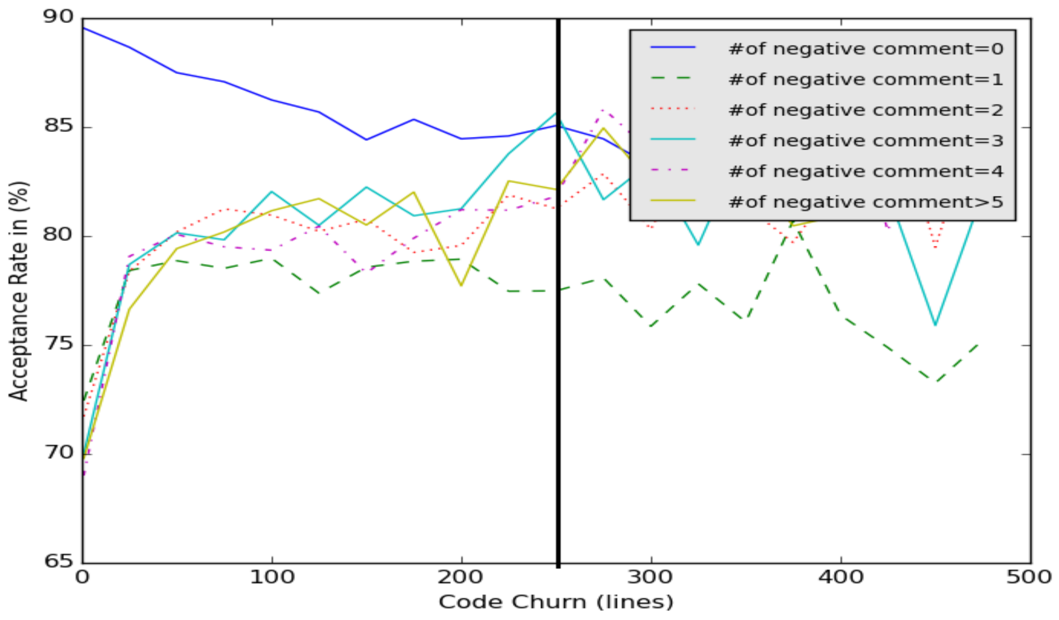


Figure 5.15: Code Churn vs. Code Acceptance Rate

5.6.3 Number of Files vs. Review Interval

In Figure 5.16, we present the relation between the number of files under review at a time and review interval. We divided the review requests into six groups. The review requests of the first group do not contain any negative comment and that of the other groups contain different number of negative review comment(s) (i.e., 1, 2, 3 etc.). We analyzed number of files up to 25 in one request. For our analysis, we divided the review requests on the basis of number of files into 25 bins (i.e., 1, 2,..., 25 etc.) and estimated the median review interval of each bin for all six groups. From Figure 5.16, we observe that the review intervals for the review requests without any negative comment does not increase much with number of files compared to the rate in case of review requests with negative review comment(s). The review interval increases consistently with number of negative comments. Therefore, negative review comment(s) play(s) a significant role in deciding the relationship between the number of files and review intervals.

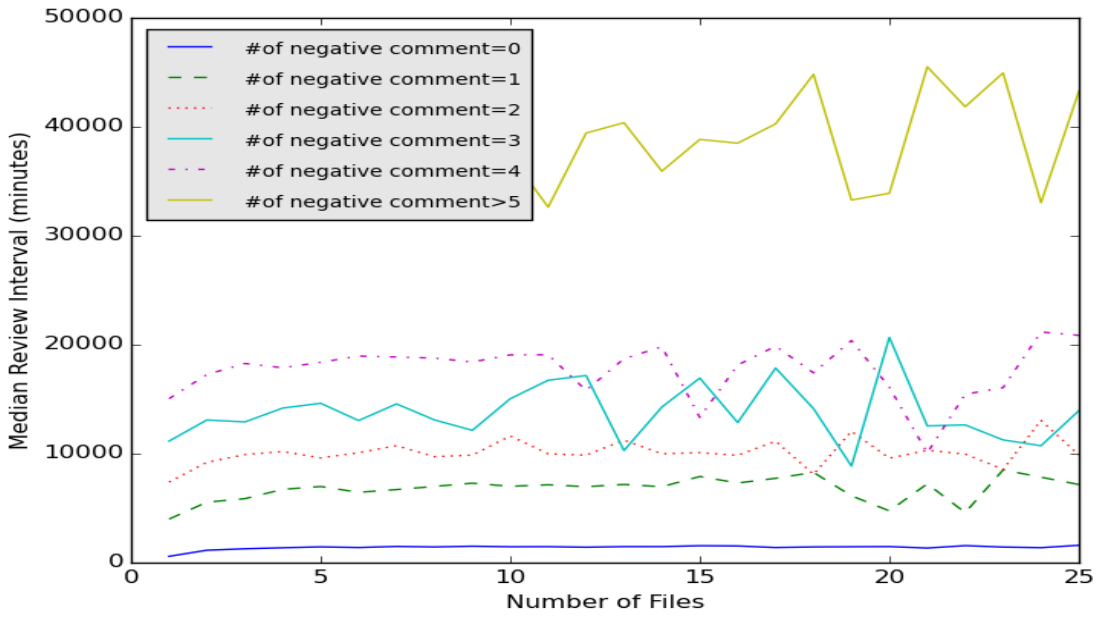


Figure 5.16: Number of Files vs. Review Interval

5.6.4 Number of Files vs. Code Acceptance Rate

In Figure 5.17, we present the relation between the number of files under review at a time and code acceptance rate. We followed a similar experimental setup mentioned in Section 5.6.3 for this analysis. From Figure 5.17, we see that the code acceptance rate decreases with number of files for review requests without any negative comment and increases for review requests with negative review comments. However, the acceptance rate hardly varies and no trend is found for review requests with negative review comments. Hence, negative review comments offset the impact of increase in number of files and number of files by itself does not have considerable impact on acceptance rate. From Figure 5.17, we observe that the code acceptance rate without any negative comment remains higher than review requests with negative comment(s) up to 13 files. In our dataset, 94% review requests have less than 13 files. Hence, the whimsical pattern after this demarcation line can be considered due to insufficient sample. Therefore, it can be concluded that negative review comments decrease the code acceptance rate irrespective of number of files. However, there is no impact of *difference in number* of negative comments on acceptance rate.

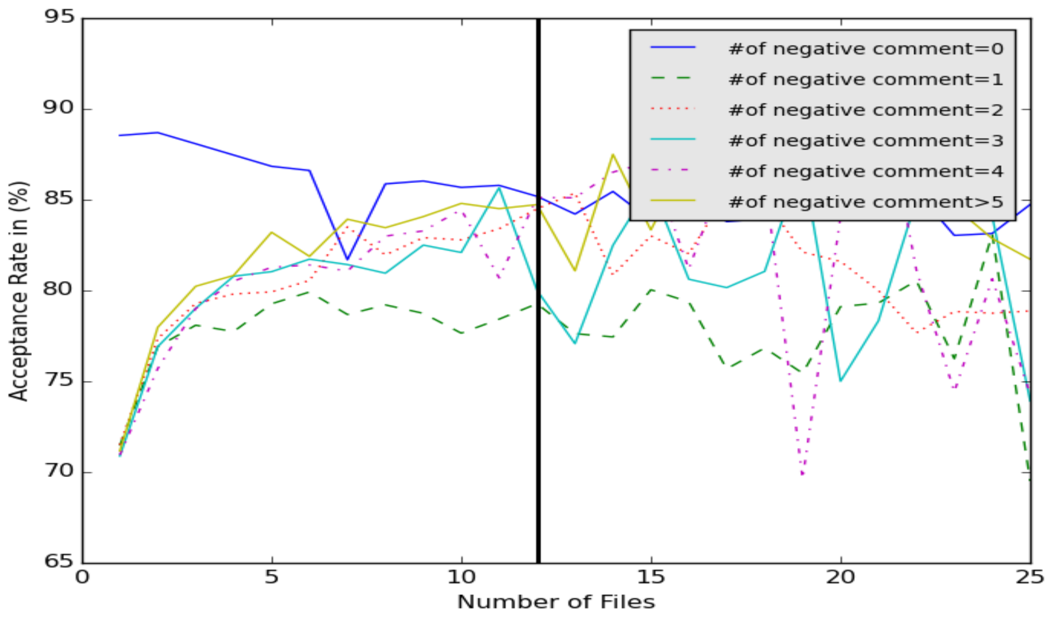


Figure 5.17: Number of Files vs. Code Acceptance Rate

5.6.5 Number of Patchsets vs. Review Interval

In Figure 5.18, we present the relation between the number of patchsets and review interval. We divided the review requests into six groups. The review requests of the first group do not contain any negative comment and that of the other groups contain negative review comment(s) (i.e., 1, 2, 3 etc.). We analyzed number of patchsets up to 25 in one request. For our analysis, we divided the review requests on the basis of number of patchsets into 25 bins (i.e., 1, 2,..., 25 etc.) and found out the median review interval of each bin for all six groups. From Figure 5.18, we observe that the review intervals for the review requests without any negative comment increase with number of patchsets like the review requests with negative review comment(s). However, the review intervals of review requests with negative review comments always maintain higher review interval than review requests without any negative review comment.

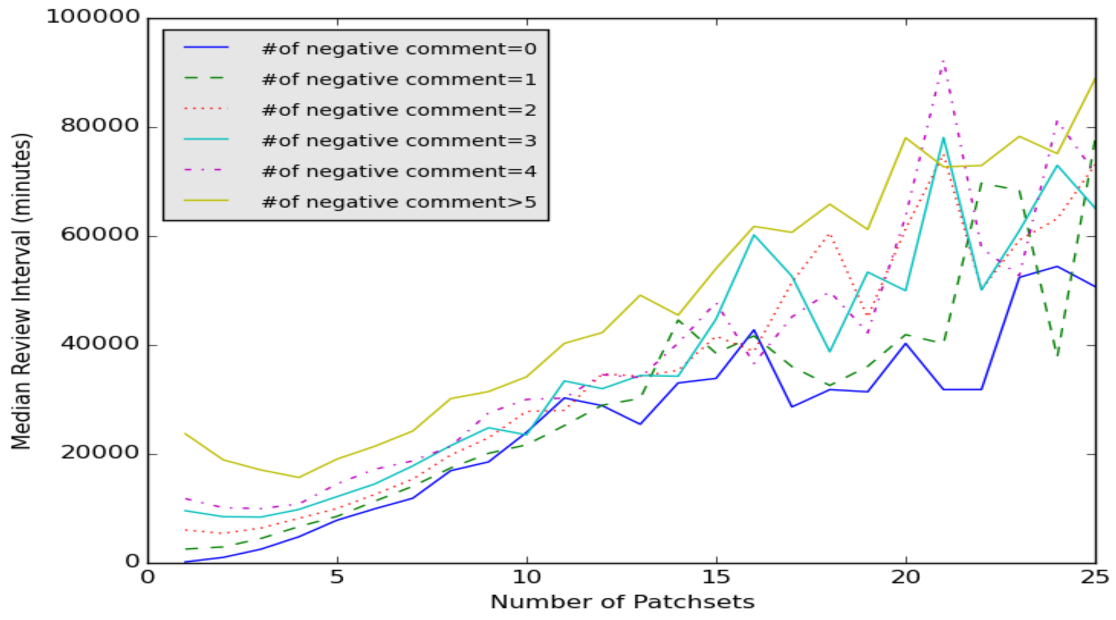


Figure 5.18: Number of Patchsets vs. Review Interval

5.6.6 Number of Patchsets vs. Code Acceptance Rate

In Figure 5.19, we present the relation between the number of patchset and code acceptance rate. We followed a similar experimental setup mentioned in Section 5.6.5 for this analysis. From Figure 5.19, we estimated that the code acceptance rate decreases with number of patchsets for review requests without any negative and increases for review requests with negative review comment(s). However, the acceptance rate hardly varies and no trend is found for review requests with negative review comments. Hence, negative review comments offset the impact of increase in number of patchsets and number of patchsets by itself does not have considerable impact on acceptance rate. From Figure 5.19, we also observe that the code acceptance rate without any negative comment remains higher than review requests with negative comment(s) up to 8 patchsets. In our dataset, 94% review requests have less than 8 patchsets. Therefore, it can be deduced that negative review comments decreases the code acceptance rate irrespective of patchsets. Acceptance rate is clearly lower for requests with more than five negative comments. However, for 1-4 negative comments, this impact is not clearly distinguishable.

In final analysis, we may deduce that unlike code churn or number of files, patchset has

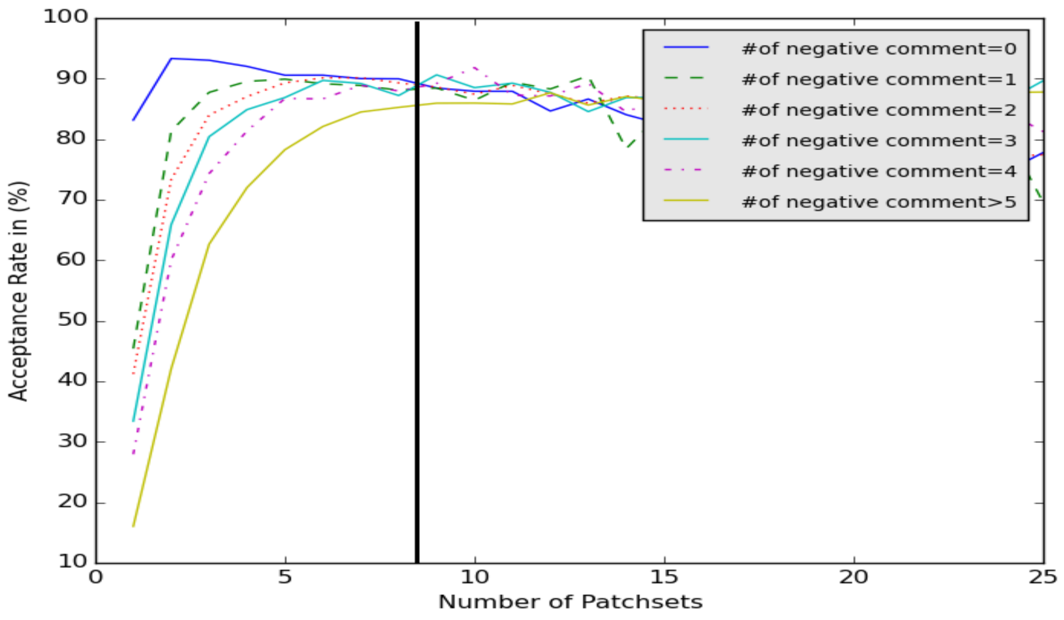


Figure 5.19: Number of Patchsets vs. Code Acceptance Rate

considerable impact on both review interval and code acceptance rate.

5.6.7 Number of Negative Comments vs. Review Interval

In Section 5.3.1, we have already found that negative comments significantly impact review interval. Here we would observe how this impact varies with the number of negative comments.

In Figure 5.20, we present the relation between the number of negative review comments and review interval. We analyzed number of negative comments up to 25. For our analysis, we divided the review requests on the basis of number of negative comments into 25 bins (i.e., 1, 2,..., 25 etc.) and estimated the median review interval of each bin for both groups of review requests. From Figure 5.20, we may deduce that the review intervals increase almost linearly with the number of negative comments which imply a strong relation between these two.

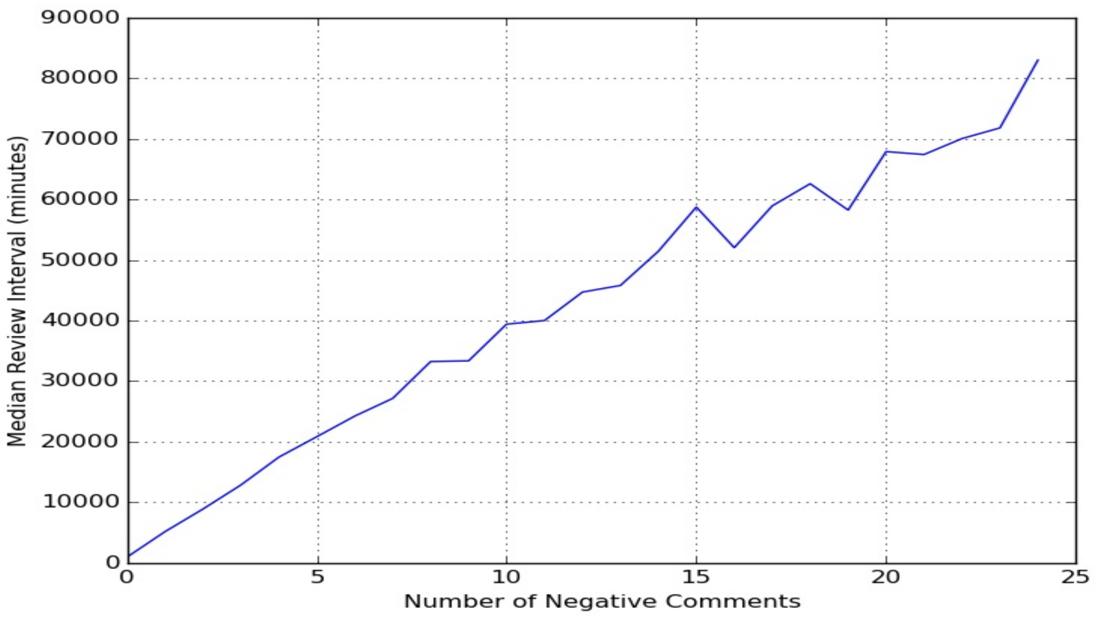


Figure 5.20: Number of Negative Comments vs. Review Interval

5.7 Implications of the Study

The results of this study have several implications for both code review participants and researchers.

1. Managing changesets.
2. Impacts on future collaborations.
3. Impact on project outcomes.
4. Avoiding negative comments.

The following subsections describe these implications.

5.7.1 Managing Changesets

Results of this study suggest that the likelihood of a code review including negative comments increase with associated code churns or number of files in the changeset. Since reviewing large code changes are time-consuming and annoying for the reviewers, this result may not be surprising. Prior research has also found large code reviews less effective [13]. Since the ability to find defects diminish beyond 400 lines of code, best practices recommend reviewing less than 200-400 lines of code at a time [27]. Our results add to the arguments against large code reviews. Based on these results, we recommend that developers submit smaller and incremental changes for reviews whenever possible, in contrast to waiting for a large feature to be completed.

5.7.2 Impacts on Future Collaborations

A negative review comment may lead to negative feelings from the recipient. Though a constructive criticism helps an author to make the required changes quickly, an opposite picture can be found if the review comments are viewed as an attack. Reaching a consensus may become very difficult, if an author and a reviewer start arguing with each other. Since code reviews have significant impacts on building relationships among developers [8], the relationship between a pair of arguing review participants may deteriorate. Moreover, relationship with the other developer is one of the most important considerations for an OSS developer during collaborations [10, 65]. Therefore, negative review comments may have longer lasting effects as both review participants may want to avoid future collaborations.

5.7.3 Impact on Project Outcomes

Our results suggest that negative review comments indeed impact code review outcomes by increasing review intervals and by decreasing acceptance rates. Therefore, negative review comments may increase project duration and may eventually increase project cost.

Moreover, integration of prospective newcomers is crucial for the success of an OSS project. OSS projects that cannot attract and more importantly retain newcomers cannot survive [66]. Since prospective newcomers may turn away if treated poorly [67], negative review comments to newcomers will inversely impact their continuation in the project and ultimately hurt project success.

5.7.4 Avoiding Negative Comments

A developer may be very busy and unintentionally express a negative sentiment. However, there has not be much research on how to prevent negative review comments. For example, using a sentiment classification model similar to ours, it is possible to provide feedback to an author regarding the possible perceived sentiment of a review comment before submission. Implementation of such a tool warrants further research.

5.8 Threats to validity

There are several threats which can challenge the validity of the research work. There are four common threats to validity.

1. Internal validity.
2. Construct validity.
3. External validity.
4. Conclusion validity.

The following subsections describe these four common threats to validity in this study.

5.8.1 Internal Validity

The primary threat to internal validity in this study is *project selection*. We included 10 publicly accessible OSS projects that practice tool-based code reviews supported by the same tool (i.e., Gerrit). Though, it is possible that projects supported by other code review tools (e.g., ReviewBoard, Github pull-based reviews, and Phabricator) could have behaved differently, sentiments expressed in review comments may not depend on any feature that is exclusive to Gerrit only. We think this threat is minimal for three reasons: 1) all code review tools support the same basic purpose, i.e. detecting defects and improving the code, 2) the basic workflow (i.e. authors posting code, reviewers commenting about code snippets, and code requiring approval from reviewer before integration) of most of the code review tools are similar, and 3) we did not use any Gerrit-specific feature/attribute in this study. Therefore, we believe the project selection threat is minimal.

5.8.2 Construct validity

We attempted to validate the model training data and the results of the model's classification in multiple ways, checking consistency with inter-rater reliability, manually cross-checking classified comments, and using 10-fold cross validations. While the selected model achieves high levels of accuracy, the classification error is around 25%. It is possible that those incorrect classifications may have altered our results, however this can only lead to incorrect findings and conclusions if there is a systematic relationship between the comments that the model incorrectly classifies and the factors examined during our large-scale empirical study (if, for example, our model incorrectly predicts sentiments of comments in large reviews far more than those with fewer files). We have no reason to believe that such a relationship exists, however there is no empirical evidence against this possibility.

Second, while our manually labeled dataset had around 75% neutral comments, our model labeled 85-90% comments as neutrals for the ten projects, suggesting potential bias in favor of the neutral class. An investigation found a large number of one word comments. For example, in the Chromium OS project, 5% of the review comments say, 'done'. Majority of the one word comments were classified as neutrals. To build our training dataset, we selected only the review comments with more than 50 characters (to ensure enough features). We found that the percentages of 'neutral' classified comments consisting more than 50 characters (Table 5.7) were similar to the training sample. Therefore, we believe this threat is minimal.

Third, overfitting is a potential threat for any supervised learning based models. To combat overfitting, we employed 10-fold cross validations of our models. We randomly selected 100 review comments that the model had not seen before and manually classified them. Against our manual classification, the model had 77% accuracy, suggesting no performance degradation.

Table 5.7: Percentage of neutral comments consisting more than 50 characters

Project	**% Neutral**
Android	73.48%
Chromium OS	74.25%
Go	64.48%
ITK / VTK	73.78%
LibreOffice	76.37%
OpenStack	79.42%
OVirt	81.36%
Qt Project	78.32%
Typo3	79.89%
WikiMedia	76.12%

Therefore, we do not think our model was overfitted.

Finally, the results show that code review metrics measured in this study (i.e. review interval, acceptance rate, number of patchsets, code churn and number of files) are influenced by the presence of negative comments. However, there may be number of unmeasured confounding factors (e.g., availability of developers, code complexity, reputation of the author, and relationship between an author and the reviewer) that could influence those review metrics. There is no evidence that our classification model or our measurements would be systematically biased based on any of these confounding factors. Furthermore, the fact that the results were similar across all ten projects provides additional confidence that any confounding factors did not significantly impact the study results.

5.8.3 External validity

The OSS projects in this study vary across domains, languages, age, and governance. In addition, we analyzed a large number of code review requests for each project. Therefore, we believe these results can be generalized to many other OSS projects. However, because of the wide variety of

OSS project characteristics, we do not claim that these results are universal for all OSS projects. All the projects in this study belong to successful and matured OSS projects. Nine out of the ten included projects are practicing Gerrit-based reviews for more than five years. The results may differ in a small-scale or less-matured project. Drawing a more general conclusion would require a family of experiments [68] that included OSS projects of all types. To encourage replications, we publish our scripts and training dataset in a supplementary website[3].

5.8.4 Conclusion validity

The use of large dataset drawn from ten projects, which produced similar results across those ten different projects, boosts confidence about the validity of the results. We tested all data for normality prior to conducting statistical analyses and performed appropriate tests based upon the results of the normality test. We used widely used implementations of the commonly used machine learning techniques for this study. Finally, the effect size, estimated with biserial correlation coefficients showed small to moderate, yet, significant effects in almost all of the cases. Therefore, our study does not have any serious conclusion validity threat.

5.9 Summary

In this chapter, we have discussed about the experimental results, implications of the study and threats to validity. . The next chapter will conclude the thesis with some future plans.

[3] `http://amiangshu.com/sentiment/index.html`

Chapter 6

Conclusion

In this study, we have built a standard code review training dataset with good inter rater agreement based on 1000 randomly selected comments. We have developed a scheme for automatic sentiment detection using supervised learning technique with a smart pre-processing steps and achieved around 74% accuracy. Using this detection model, we found sentiments of 10.7 million comments from 10 OSS projects and analyzed our research hypotheses. We have developed five hypotheses to investigate factors that influence sentiment of code reviews and to formally analyze the impact of sentiment on the stakeholders.

This study identified three factors (i.e., code churn, number of files, and number of patchsets) influencing negative review comments in OSS projects and also investigated the impact of those comments on the outcomes of associated review requests. The findings indicate that larger code reviews are more likely to encounter negative comments. According to our study, it is very difficult and time consuming to review large code segment. Therefore, it may influence the tone of the reviewer and result in negative comment. Majority of the code reviews with more than 140 line code churns receive at least one negative review comments and more than one-third of the code reviews with more than two files encounter negative comments. Moreover, negative review comments not only increase review intervals 4 to 32 times (i.e., time to complete a code review) but also are likely to decrease code acceptance rate. Since code reviews influence the relationship between an author and a reviewer, negative review comments may degrade their relationship. An author may be hurt by the harsh tone of a reviewer and be tempted to leave the project. On the other hand, if a developer receives constructive and persuasive reviews, he will be further motivated to contribute to the project.

In the future, we plan to extend our work achieving the followings:

- Extending our training dataset by adding more review comments.
- Including texts from different types of other SE communications (e.g., bug, mailing list, and IRC) to build a generalized sentiment classifier.
- Studying the differences between the level of sentiments expressed on various SE communication channels.
- Identifying the impacts of sentiments on the outcomes of other project activities (e.g., bug resolution).

Based on our findings, we recommend developers to avoid submitting large code review requests and to avoid authoring negative review comments. We believe that the findings of this study would provide valuable guidance to the OSS code review participants to improve their code review process. Also this study expected to improve team-effort or collaboration in the open source community and benefit this noble movement to reduce digital divide.

Appendices

Appendix A

Examples Misclassified by SentiStrength

Review Comments	**Human Raters**	**Our Classifier**	**Senti-Strength**	**Comments**
if so there will be some problems here - i mean, to authorise we need password/token - token is really not supported in that version of nova_client. dunno what password to use here - where we are using ctx from trust. using service user creds here is not really a good idea.	-1	-1	0	Though "problem" and "not supported" are negative tokens, "trust" and "good' are considered positive. "trust" is wrongly classified in this context. Here, "not" did not qualify "good" because of "really".
per our discussion, make this a class that is just the combobox? maybe like androidapicombo?	0	0	1	"like" is wrongly considered as positive token is this context.

i think you also need to add -.7em to this 24px. that number comes from the .fulltext-search left and right margins. otherwise there is a horizontal scroll on the page.	0	0	−1	The word “number” is considered as negative in SentiStrength. We did not understand the reason.
this seems a bit error prone and hacky to me. what if $contentcontext-¿getinaccessiblecontentshown() was true before. it would be false afterwards (probably not relevant here, but maybe elsewhere). what about a second argument in ”getnode()” that allows to even fetch hidden/protected nodes?	−1	−1	0	The dictionary does not contain “hacky” and gives over emphasis on “true” which is not applicable in this context.
if we have got this, do we still need the flag? this field should probably be called something more like “allocated_stack_size”.	0	0	1	“like” is wrongly considered as positive token is this context.
also explain in this file why the compiler was failing, and dx’s smart/dumb behavior.	−1	−1	0	Gives over-emphasis on “smart”.

no hsc in exceptions please. the exception handler does take care of proper encoding	0	0	1	Gives over-emphasis on “care”.
tdf... is unnecessary. rather document why this is done, if necessary.	-1	-1	0	Gives equal emphasis on “unnecessary” and “necessary”. In this context, “unnecessary” is more important than necessary.
i am sorry :d order these	1	1	-1	Gives over-emphasis on “sorry” and considered it as negative token.
:(so we need to either port this file over to kernel style or not use goto’s. since we started it before everyone decided kernel style was ok, we have been using a hacked-down google c++ style which forbids goto	-1	-1	0	“hacked-down”, “forbids” are not included in dictionary.

the variable you should probably use here is u_boot_fdt_use, defined in make.conf. i agree the name is not great - we can change it if you like. but the intention is that this variable tells you which fdt to use, so there should be no need for this sort of logic in ebuilds.	1	1	-1	Gives over-emphasis on “not great”.
i do not understand why you need the boolean. this should be new_rti.settop(); new_rti.setinexact().	-1	-1	0	“not understand” is considered as neutral. If we include “understand” as positive token, then it will work but “understand” is not a positive token.

Appendix B

Examples Misclassified by Our Proposed Method

Review Comments	Human Raters	Our Classi-fier	Senti-Strength	Comments
reconsider why is that these variables should be ”protected” instead of ”pri-vate”	0	-1	0	Use of “why”.
gsm is not in the original list which is why it is not in the enum.	0	-1	0	Use of “which” and “why”.
if there is only one developer path, do not append ’0’ to the xcode name. it looks weird.	-1	0	-1	Training set missed the examples with “weird”. Therefore, proposed method failed in this case.

”when i was suggesting flipping the boolean, i did also wonder whether we would not be better off just passing in the length. if it is ¡= 0, interpret that as ””do not know””? that would then move 8192 down out of the callers too. or... i really like the way you went for readfully rather than implementing the missing read()s, which is what i would actually imagined. how about we go one further and just have a static fromfile or similar? then it could do the stat, and have everything it needs. you could even use libcore.os open/fstat/read/close to avoid the randomaccessfile without increasing the overall amount of code.”	1	-1	1	Though there is a negative lexicon “missing”, over emphasized “like” has made it possible for SentiStrength to detect the sentiment correctly. Our method failed because of “W” s.

Appendix C

Contractions

List will be completed as soon as possible.

Contractions	**Expansions**	**Contractions**	**Expansions**	**Contractions**	**Expansions**
ain’t	am not	aren’t	are not	ain’t	am not
can’t	cannot	can’t’ve	cannot have	aren’t	are not
cause	because	could’ve	could have	can’t	cannot
couldn’t	could not	couldn’t’ve	could not have	can’t’ve	cannot have
didn’t	did not	doesn’t	does not	cause	because
don’t	do not	hadn’t	had not	could’ve	could have
hadn’t’ve	had not have	hasn’t	has not	couldn’t	could not

Appendix D

Emoticons

List will be completed as soon as possible.

Emoticons	**Sentiments**	**Emoticons**	**Sentiments**
%-(	Negative	%-)	Positive
(-:	Positive	(:	Positive
(^ ^)	Positive	(^-^)	Positive
(^ . ^)	Positive	(^_^)	Positive

Bibliography

[1] B. Liu and L. Zhang, "A survey of opinion mining and sentiment analysis," in *Mining text data*. Springer, 2012, pp. 415–463.

[2] J. A. Davis and S. Leinhardt, "The structure of positive interpersonal relations in small groups." 1967.

[3] M. De Choudhury and S. Counts, "Understanding affect in the workplace via social media," in *Proceedings of the 2013 conference on Computer supported cooperative work*. ACM, 2013, pp. 303–316.

[4] E. Guzman, D. Azócar, and Y. Li, "Sentiment analysis of commit comments in github: an empirical study," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 352–355.

[5] M.-A. Storey, C. Treude, A. van Deursen, and L.-T. Cheng, "The impact of social media on software engineering practices and tools," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 2010, pp. 359–364.

[6] A. Murgia, P. Tourani, B. Adams, and M. Ortu, "Do developers feel emotions? an exploratory analysis of emotions in software artifacts," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 262–271.

[7] E. Guzman and B. Bruegge, "Towards emotional awareness in software development teams," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, 2013, pp. 671–674.

[8] A. Bosu and J. C. Carver, “Impact of peer code review on peer impression formation: A survey,” in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 133–142.

[9] A. Bacchelli and C. Bird, “Expectations, outcomes, and challenges of modern code review,” in *Proceedings of the 2013 International Conference on Software Engineering*. San Francisco, CA, USA: IEEE Press, 2013, pp. 712–721.

[10] A. Bosu, J. C. Carver, C. Bird, J. Orbeck, and C. Chockley, “Process Aspects and Social Dynamics of Contemporary Code Review: Insights from Open Source Development and Industrial Practice at Microsoft, year=2016, volume=PP, number=99, pages=1-1,,” *IEEE Transactions on Software Engineering*.

[11] D. Pletea, B. Vasilescu, and A. Serebrenik, “Security and emotion: sentiment analysis of security discussions on github,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 348–351.

[12] J. C. Munson and S. G. Elbaum, “Code churn: A measure for estimating the impact of code change,” in *Proceedings of the 1998 International Conference on Software Maintenance*. IEEE, 1998, pp. 24–31.

[13] A. Bosu, M. Greiler, and C. Bird, “Characteristics of useful code reviews: An empirical study at Microsoft,” in *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 2015, pp. 146–156.

[14] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” in *Proceedings. 27th International Conference on Software Engineering*. IEEE, 2005, pp. 284–292.

[15] P. C. Rigby, D. M. German, and M.-A. Storey, “Open source software peer review practices: a case study of the Apache server,” in *Proceedings of the 30th International Conference on Software Engineering*, 2008, pp. 541–550.

[16] A. Bosu and J. C. Carver, “Impact of Developer Reputation on Code Review Outcomes in OSS Projects: An Empirical Investigation,” in *Proceedings of the 2014 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, 2014, pp. 33:1–33:10.

[17] D. Isa, L. H. Lee, V. Kallimani, and R. Rajkumar, “Text document preprocessing with the bayes formula for classification using the support vector machine,” *IEEE Transactions on Knowledge and Data engineering*, vol. 20, no. 9, pp. 1264–1272, 2008.

[18] D. Isa, L. L. Hong, V. Kallimani, and R. Rajkumar, “Text document pre-processing using the bayes formula for classification based on the vector space model,” *Computer and Information Science*, vol. 1, no. 4, p. 79, 2008.

[19] L. H. Lee, C. H. Wan, R. Rajkumar, and D. Isa, “An enhanced support vector machine classification framework by using euclidean distance function for text document categorization,” *Applied Intelligence*, vol. 37, no. 1, pp. 80–99, 2012.

[20] T. W. Chow, H. Zhang, and M. Rahman, “A new document representation using term frequency and vectorized graph connectionists with application to document retrieval,” *Expert Systems with Applications*, vol. 36, no. 10, pp. 12 023–12 035, 2009.

[21] L. Breiman, “Random forests,” *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[22] Y. Freund and R. E. Schapire, “A desicion-theoretic generalization of on-line learning and an application to boosting,” in *European conference on computational learning theory*. Springer, 1995, pp. 23–37.

[23] J. H. Friedman, “Greedy function approximation: a gradient boosting machine,” *Annals of statistics*, pp. 1189–1232, 2001.

[24] J. L. Fleiss, “Measuring nominal scale agreement among many raters.” *Psychological bulletin*, vol. 76, no. 5, p. 378, 1971.

[25] J. R. Landis and G. G. Koch, “The measurement of observer agreement for categorical data,” *biometrics*, pp. 159–174, 1977.

[26] M. E. Fagan, “Design and code inspections to reduce errors in program development,” *IBM Syst. J.*, vol. 15, no. 3, pp. 182–211, Sept. 1976.

[27] J. Cohen, E. Brown, B. DuRette, and S. Teleki, *Best kept secrets of peer code review*. Smart Bear, 2006.

[28] K. E. Wiegers, *Peer reviews in Soft.: A practical guide*. Addison-Wesley Boston, 2002.

[29] P. C. Rigby and C. Bird, “Convergent contemporary software peer review practices,” in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 202–212.

[30] A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni, “Identifying the characteristics of vulnerable code changes: An empirical study,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 257–268.

[31] B. Pang and L. Lee, “Opinion mining and sentiment analysis,” *Foundations and trends in information retrieval*, vol. 2, no. 1-2, pp. 1–135, 2008.

[32] ——, “A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts,” in *Proceedings of the 42nd annual meeting on Association for Computational Linguistics*. Association for Computational Linguistics, 2004, p. 271.

[33] E. Boiy and M.-F. Moens, “A machine learning approach to sentiment analysis in multilingual web texts,” *Information retrieval*, vol. 12, no. 5, pp. 526–558, 2009.

[34] P. D. Turney, “Thumbs up or thumbs down?: semantic orientation applied to unsupervised classification of reviews,” in *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 2002, pp. 417–424.

[35] M. Taboada, J. Brooke, M. Tofiloski, K. Voll, and M. Stede, "Lexicon-based methods for sentiment analysis," *Computational linguistics*, vol. 37, no. 2, pp. 267–307, 2011.

[36] A. Agarwal, B. Xie, I. Vovsha, O. Rambow, and R. Passonneau, "Sentiment analysis of twitter data," in *Proceedings of the workshop on languages in social media*. Association for Computational Linguistics, 2011, pp. 30–38.

[37] H. Saif, Y. He, and H. Alani, "Semantic sentiment analysis of twitter," in *International Semantic Web Conference*. Springer, 2012, pp. 508–524.

[38] N. F. F. d. Silva, E. R. Hruschka, E. R. Hruschka Junior, *et al.*, "Biocom_usp: tweet sentiment analysis with adaptive boosting ensemble," in *International Workshop on Semantic Evaluation, 8th*. ACL Special Interest Group on the Lexicon-SIGLEX, 2014.

[39] M. Pennacchiotti and A.-M. Popescu, "Democrats, republicans and starbucks afficionados: user classification in twitter," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2011, pp. 430–438.

[40] S. Tan, X. Cheng, Y. Wang, and H. Xu, "Adapting naive bayes to domain adaptation for sentiment analysis," in *European Conference on Information Retrieval*. Springer, 2009, pp. 337–349.

[41] A. Pak and P. Paroubek, "Twitter as a Corpus for Sentiment Analysis and Opinion Mining." in *LREc*, vol. 10, 2010, pp. 1320–1326.

[42] A. Bifet and E. Frank, "Sentiment knowledge discovery in twitter streaming data," in *International Conference on Discovery Science*. Springer, 2010, pp. 1–15.

[43] C. Whitelaw, N. Garg, and S. Argamon, "Using appraisal groups for sentiment analysis," in *Proceedings of the 14th ACM international conference on Information and knowledge management*. ACM, 2005, pp. 625–631.

[44] B. Pang, L. Lee, and S. Vaithyanathan, "Thumbs up?: sentiment classification using machine learning techniques," in *Proceedings of the ACL-02 conference on Empirical methods*

in natural language processing-Volume 10. Association for Computational Linguistics, 2002, pp. 79–86.

[45] J. M. Ruiz-Martínez, R. Valencia-García, and F. García-Sánchez, "Semantic-based sentiment analysis in financial news," in *Proceedings of the 1st International Workshop on Finance and Economics on the Semantic Web*, 2012, pp. 38–51.

[46] S. Somasundaran, T. Wilson, J. Wiebe, and V. Stoyanov, "Qa with attitude: Exploiting opinion type analysis for improving question answering in on-line discussions and the news." in *ICWSM*, 2007.

[47] V. Stoyanov, C. Cardie, and J. Wiebe, "Multi-perspective question answering using the opqa corpus," in *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing.* Association for Computational Linguistics, 2005, pp. 923–930.

[48] L. V. Lita, A. H. Schlaikjer, W. Hong, and E. Nyberg, "Qualitative dimensions in question answering: Extending the definitional qa task," in *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, vol. 20, no. 4. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2005, p. 1616.

[49] O. Kucuktunc, B. B. Cambazoglu, I. Weber, and H. Ferhatosmanoglu, "A large-scale sentiment analysis for yahoo! answers," in *Proceedings of the fifth ACM international conference on Web search and data mining.* ACM, 2012, pp. 633–642.

[50] D. Garcia, M. S. Zanetti, and F. Schweitzer, "The role of emotions in contributors activity: A case study on the Gentoo community," in *Cloud and Green Computing (CGC), 2013 Third International Conference on.* IEEE, 2013, pp. 410–417.

[51] A. Gupte, S. Joshi, P. Gadgul, A. Kadam, and A. Gupte, "Comparative study of classification algorithms used in sentiment analysis," *International Journal of Computer Science and Information Technologies*, vol. 5, no. 5, pp. 6261–6264, 2014.

[52] S. Bird, "NLTK: The Natural Language Toolkit," in *Proceedings of the COLING/ACL on Interactive presentation sessions*. Association for Computational Linguistics, 2006, pp. 69–72.

[53] R. Jongeling, S. Datta, and A. Serebrenik, "Choosing your weapons: On sentiment analysis tools for software engineering research," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE, 2015, pp. 531–535.

[54] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The Stanford CoreNLP Natural Language Processing Toolkit." in *ACL (System Demonstrations)*, 2014, pp. 55–60.

[55] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[56] X.-Y. Liu, J. Wu, and Z.-H. Zhou, "Exploratory undersampling for class-imbalance learning," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 39, no. 2, pp. 539–550, 2009.

[57] C. Drummond, R. C. Holte, *et al.*, "C4. 5, class imbalance, and cost sensitivity: why under-sampling beats over-sampling," in *Workshop on learning from imbalanced datasets II*, vol. 11. Citeseer, 2003.

[58] L. Breiman and P. Spector, "Submodel selection and evaluation in regression. The X-random case," *International statistical review/revue internationale de Statistique*, pp. 291–319, 1992.

[59] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *Proceedings of the 2006 international workshop on Mining software repositories*. ACM, 2006, pp. 137–143.

[60] S. S. Shapiro and M. B. Wilk, “An analysis of variance test for normality (complete samples),” *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965.

[61] E. E. Cureton, “Rank-biserial correlation,” *Psychometrika*, vol. 21, no. 3, pp. 287–290, 1956.

[62] C. O. Fritz, P. E. Morris, and J. J. Richler, “Effect size estimates: current use, calculations, and interpretation.” *Journal of Experimental Psychology: General*, vol. 141, no. 1, p. 2, 2012.

[63] R. E. McGrath and G. J. Meyer, “When effect sizes disagree: the case of r and d.” *Psychological methods*, vol. 11, no. 4, p. 386, 2006.

[64] P. Kampstra *et al.*, “Beanplot: A boxplot alternative for visual comparison of distributions,” *Journal of statistical software*, vol. 28, no. 1, pp. 1–9, 2008.

[65] A. Bosu, J. Carver, R. Guadagno, B. Bassett, D. McCallum, and L. Hochstein, “Peer impressions in open source organizations: A survey,” *Journal of Systems and Software*, vol. 94, pp. 4–15, 2014.

[66] K. Crowston, H. Annabi, and J. Howison, “Defining open source software project success,” *Proceedings of the 24th International Conference on Information Systems*, pp. 327–340, 2003.

[67] I. Steinmacher, T. Conte, M. A. Gerosa, and D. Redmiles, “Social barriers faced by newcomers placing their first contribution in open source software projects,” in *Proceedings of the 18th ACM conference on Computer supported cooperative work & social computing*. ACM, 2015, pp. 1379–1392.

[68] V. R. Basili, F. Shull, and F. Lanubile, “Building knowledge through families of experiments,” *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 456–473, 1999.