2015 12th Working Conference on Mining Software Repositories

Characteristics of Useful Code Reviews: An Empirical Study at Microsoft

Amiangshu Bosu*, Michaela Greiler†, and Christian Bird†
*Department of Computer Science
University of Alabama, Tuscaloosa, Alabama
Email: asbosu@ua.edu
†Microsoft Research, Redmond, WA USA
Email: {mgreiler, cbird}@microsoft.com

***Abstract*—Over the past decade, both open source and commercial software projects have adopted contemporary peer code review practices as a quality control mechanism. Prior research has shown that developers spend a large amount of time and effort performing code reviews. Therefore, identifying factors that lead to useful code reviews can benefit projects by increasing code review effectiveness and quality. In a three-stage mixed research study, we qualitatively investigated what aspects of code reviews make them useful to developers, used our findings to build and verify a classification model that can distinguish between useful and not useful code review feedback, and finally we used this classifier to classify review comments enabling us to empirically investigate factors that lead to more effective code review feedback.**

In total, we analyzed 1.5 millions review comments from five Microsoft projects and uncovered many factors that affect the usefulness of review feedback. For example, we found that the proportion of useful comments made by a reviewer increases dramatically in the first year that he or she is at Microsoft but tends to plateau afterwards. In contrast, we found that the more files that are in a change, the lower the proportion of comments in the code review that will be of value to the author of the change. Based on our findings, we provide recommendations for practitioners to improve effectiveness of code reviews.

I. Introduction

In recent years, many open source software (OSS) and commercial projects have adopted peer code review [1], the process of analyzing code written by another developer on the project to judge whether it is of sufficient quality to be integrated into the main project codebase. The formal variant of peer code review, which is better known as software inspection or Fagan-inspection [2], has been an effective quality improvement practice for a long time [3]. Even with the benefits offered by software inspections, their relatively high cost and formal requirements have reduced the prevalence with which software teams adopt them [4], [5]. On the other hand, projects have recently adopted lightweight, informal, and tool-based code review practices, which researchers have termed *modern* or *contemporary code review* [6].

This modern code review has shown increasing adoption both in industrial and open source contexts [7], [8], [9] For example, in January of 2015, more than 50,000 Microsoft employees practiced tool-based code review. A recent survey [10] found that on average developers spend about six hours per week preparing code for review or reviewing others' code. Therefore increasing the effectiveness of code review practices is beneficial for ensuring developers' time (both the author and the reviewer) is spent wisely. The main building blocks of code reviews are comments that reviewers add as feedback and suggestions for change that the code review author can address. The usefulness of those comments highly influence the effectiveness of the code review practices. Comments that point out bugs, suggest improved ways of solving problems, or point out violations of team practices can help the author submit a higher quality change to the codebase, improve the author's development skills, or both. Other comments may contain incorrect information or may ask questions that are not relevant and require the author's time to respond to without improving the code.

As the primary goal of code review is to ensure that a change is free from defects, follows team conventions, solves a problem in a reasonable way, and is of high quality [6], we consider review feedback useful if it is judged useful by author of the change to enable him or her to meet these goals. We have observed that one of the top requests from teams using code review in their development process is a way to track the effectiveness of code review feedback and identify what they can do to improve effectiveness. As one team manager indicated, he'd like to know *"was this an impactful review, a useful comment on the review? You know, not just a comment, but did it result in a change that wouldn't have been there before."* The primary purpose of this paper is to address both challenges. Our objective is *to identify the factors that impact the usefulness of code reviews, and to derive recommendations for effectiveness improvements*.

To achieve this goal, we conducted a three-stage empirical study of code review usefulness at Microsoft. In the first stage, we conducted an exploratory study in which we interviewed developers to understand their perception of what "useful" means in the context of code review feedback. In the second stage, we used our findings from the first stage to build and verify an automated classifier able to distinguish between useful and not useful review comments. Finally, we applied the classifier to ∼1.5 million code review comments from five Microsoft projects, distinguishing comments that were useful from those that were not. Using this large-scale dataset, we quantitatively investigated factors that are associated with useful review comments.

The primary contributions of this study are:

- An exploratory qualitative analysis of authors' perceptions on useful code review comments.

978-0-7695-5594-2/15 $31.00 © 2015 IEEE
DOI 10.1109/MSR.2015.21

- An empirically validated automatic model for classifying the usefulness of review comments.
- An empirical study of factors influencing review comment usefulness.
- A set of implications and recommendations for teams using code review to achieve a high rate of useful comments during review.

In this paper we start (Section II) by providing a brief overview of the code review process at Microsoft. We then (Section III) introduce the research questions that drive the three stages of our study. Section IV, V, and VI describes the methodology and results for the three stages. We then address the threats to the validity of our findings (Section VII), discuss the implications of the results (Section VIII), and position our work relative to prior studies on code review (Section IX).

II. Code Review at Microsoft

Most Microsoft developers practice code review using CodeFlow, an internal tool for reviewing code, which is under active development and regularly used by more than 50,000 developers. CodeFlow is a collaborative code review tool similar to other popular review tools such as Gerrit [11], Phabricator [12], and ReviewBoard [13].

The single desktop view of CodeFlow (shown in Figure 1) features several panes to display important information about the code review. Such information includes the list the files involved in the change (A), the reviewers and their status (B), diff-highlighted content of the file currently selected by the user (C), a summary of all the comments made during the review (D), and tabs for the individual *iterations* (explained below) of the review (E). Bacchelli and Bird provide a more detailed description of CodeFlow [6].

The workflow in CodeFlow is relatively straightforward. An author submits a change for review and reviewers are notified via email and can examine the change in the tool. If they would like to provide feedback, they highlight a portion of the code and type a comment which is then overlayed in the user interface with a line to the highlighted code as shown in Figure 1-F and seen by all involved in the review. For example, the comment shown is for the highlighted portions of line 66. These comments can start threads of discussion and are the interaction points for the people involved in the review. Each such thread has a status that participants can modify over the course of the review. The status is initially 'Active', but can be changed to 'Pending', 'Resolved', 'Won't Fix', and 'Closed' by anyone. There is no proscribed universal definition for each status label and no enforced policies to enforce resolving or closing threads of discussion. Many teams find these useful for tracking work status and decide which labels to use and how to use them independently. The author may take feedback in comments, update the change, and submit the updated change for additional feedback. In CodeFlow parlance, each updated change submitted for review is termed an *iteration* and constitutes another review cycle. It is not unusual to see two, three, or four iterations before a change is ready to check into the source code repository. In the review shown, there are five iterations (indicated by the tabs labeled "1", "2", etc.), with the original change in iteration 1, an updated change in iteration 2, and the final change in iteration five. Reviewers can continue to provide feedback in the form of comments on each iteration and this process repeats until the reviewers are happy with the change. Once a reviewer is comfortable that a change is of sufficient quality, he or she indicates this by setting their status to "signed off". After enough people sign off (sign off policies differ by team), the author checks the changes into the source code repository.

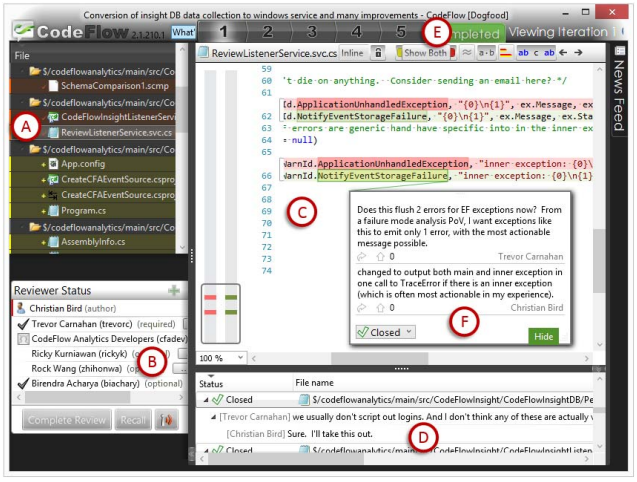


Fig. 1: Example of Code Review using CodeFlow

III. Research Questions

The goal of our study is to derive insight regarding what leads to high quality reviews in an effort to help teams understand the impact of and change (if needed) their code reviewing practices and behaviors so that their reviews are most effective.

We accomplish this by identifying how characteristics of reviewers performing the review, of changes under review, and of temporal aspects of the review, influence usefulness of review comments.

We decompose this high level objective into three concrete research questions.

RQ1. *What are the characteristics of code review comments that are perceived as useful by change authors?*

RQ2. *What methods and features are needed to automatically classify review comments into useful and not useful?*

RQ3. *What factors have a relationship with the density of useful code review comments?*

Each of the following three sections focuses on one research question. We describe the study methods and findings separately for each. These three questions represent high level steps in our study. We first aimed to understand what constitutes usefulness from the developer perspective (RQ1), then we used these insights as we set out to build an automatic classifier to distinguish between useful and not useful code review comments (RQ2). Finally we used this classifier to classify over one million comments that we then investigated quantitatively to help uncover the characteristics of reviewers and their team and the code under review that influence the usefulness of code review comments (RQ3). Figure 2 shows an overview of our three-stage research methodology.

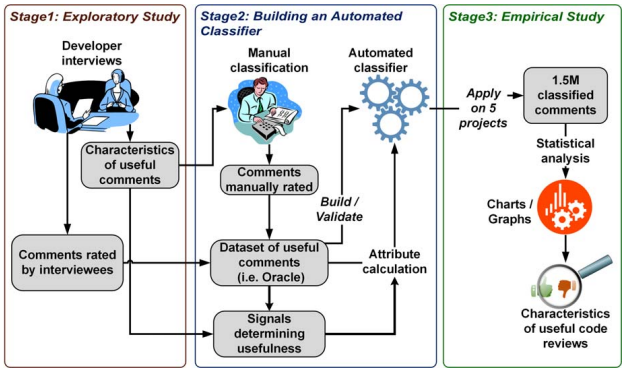


Fig. 2: Three-stage Research Method

IV. Qualitative Exploratory Study on Comment Usefulness

In the first phase, we conducted an exploratory qualitative study to understand what code review comment usefulness means to developers. Further, with the help of the interviewees we identified signals suited to distinguish between useful and not useful code review comments. In the following subsections, we describe the study method and our findings. We deliberately focused on understanding comment usefulness from the perspective of the authors of source code changes because they are the people that actually make changes to the code based on feedback and they are the primary target for code review comments.

A. Developer Interviews

We chose interview research because interviewing is a frequently used exploratory technique used to gain insight into the opinions and thoughts of the participants – which cannot be obtained by quantitative measures, such as data mining [14]. We conducted *semi-structured individual interviews* with seven developers, selected from four different Microsoft projects based on their varying level of development / code-review experiences. Semi-structured interviews allowed us to combine *open-ended questions* to elicit information on their perception on code reviews and comments with *specific questions* revealing the perceived usefulness of individual code review comments. Each interview lasted 30 minutes and was structured into three phases. First ($\approx$ 5 minutes), we asked the interviewee about their job role, their experience, and what role comments play during code reviewing. In the second phase ($\approx$ 20 minutes), we showed the interviewee $20 \sim 25$ randomly selected review comments from recently submitted code reviews for which the interviewee was the author. We asked the interviewee to rate the usefulness of each comment and categorize it based on a classification scheme adopted from a prior study to identify the types of issues detected during code reviews [15] (see 3 for the list of categories). We used the ratings from the author because they are the only ones with the "ground truth" of the usefulness of the comment (i.e. the author is the best judge of whether the comment was helpful to him or her and/or the change). For each of these comments, we asked the interviewees to

1) rate the comment on a 3-point scale (1- *Not useful*, 2- *Somewhat useful*, and 3- *Useful*),
2) briefly explain why they selected that particular rating for the comment, and
3) assign the comment to a comment category (e.g., comment about documentation, functional, and structure).

To assist with the classification, we supplied the interviewee with a printed copy of the comment classification scheme including a brief description for each category. In the last phase ($\approx$ 5 minutes), we asked the interviewee if they could indicate other types of useful comments that were not covered in the interview. Prior to our interviews we performed pilot interviews with a separate set of developers to assess whether the question were clear and timing was appropriate. During the interviews, we wrote down all answers, categories and ratings of the review comments on a printed interview form and immediately after the interview completed the notes with further details and observations. We used purposeful sampling until we reached saturation (new interviews were not providing any new information). During the interviews, the seven participants rated and categorized 145 review comments.

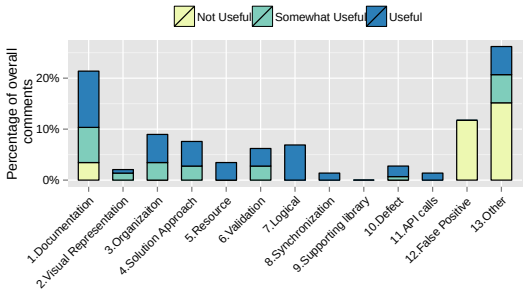


Fig. 3: Distribution of comment categories

B. Insights from the Interviews

Prior literature has found that the primary goal of code reviews is most often to improve the quality of software by identifying defects, identify better approaches for a source code change or help to improve the maintainability of code [16]. There are, however, other secondary benefits of code reviews such as knowledge dissemination, and team awareness [6]. We found similar sentiments at Microsoft, as developers at Microsoft consider a code review effective if the review comments help to improve the quality of code. On the other hand, comments whose sole purpose can be attributed to knowledge dissemination and team awareness are perceived as less useful by developers.

Figure 3 shows the distribution of comments categorized by the interviewees and their ratings. The interviewees rated almost 69% comments as either useful or somewhat useful. Most of the comments identifying functional defects (categories from 5 to 11 in Figure 3) were rated as 'Useful'. More than 60% of the "Somewhat Useful" comments belong to the first four categories: documentation in the code, visual representation of the code (e.g. blank lines, indentation), organization of the code (e.g. how functionality is divided into methods), and solution approach. These four all belong to the class termed "evolvability defects" as classified by Mantyla et al. [15] and

represent issues that affect future development effort rather runtime behavior.

On the other hand, most of the 'Not useful' comments are either false positives (e.g., when a reviewer incorrectly indicates a problem in the code, perhaps due to lack of expertise) or don't fall into a predefined category, which we therefore label as 'Other'. Although, most of the review comments ($\approx$ 73%) fell into one of the categories from Mantyla et al.'s study, we identified four additional categories based on recurring labels interviewees gave to comments in the 'Other' category. The new categories were: 1) questions from reviewers, 2) praise for the implementation, 3) suggestions for alternate output or error messages, and 4) discussions on design or new features. The interviewees did not consider comments from the first two categories useful, while in most of the cases they considered the last two categories as useful comments.

Based on the interviewee answers and additional post-interview analysis of the rated comments we gained insights about participants' perception of usefulness and identified suitable signals, as described below. We first describe our insights regarding the three usefulness ratings.

1) *'Useful' comments*
- Authors consider identification of any functional issues [15] as useful review comments. However, most of the review comments are unrelated to any types of functional defects.
- Sometimes reviewers identify validation issues or alternate scenarios (i.e. corner cases) where the current implementation may fail. If the author lacks knowledge about particular execution environments or scenarios, an experienced reviewer may help him through validating the implementation. Authors indicated that review comments related to such validations are very useful.
- Code review is very useful for the new members to learn the project design, constraints, available tools, and APIs. Authors that were new to the team considered comments with suggestions regarding APIs to use, designs to follow, team coding conventions, etc. to be useful.

2) *'Somewhat Useful' comments*
- Many of the review comments identify what some developers refer to as "nit-picking issues" (e.g., indentation, comments, style, identifier naming, and typos). Some of the interviewees rated nit-picking issues as 'Somewhat useful', while others rated those as 'Useful'. As a rationale for those ratings, interviewees indicated that resolving nit-picking issues may not be essential, but the identification and resolution of nit-picking issues help long-term project maintenance.
- Sometimes review feedback / questions help the authors to think about an alternate implementation or a way to refactor the code to make it more comprehensible (even if the current implementation may be correct). Authors consider those comments 'Somewhat useful'.

3) *'Not Useful' comments*
- Sometimes reviewers comment on a code segment not to point out an issue in the code, but rather to ask questions to understand the implementation. Those comments may be useful to the reviewers for knowledge dissemination, but are not considered useful by the author as they do not improve the code.
- Some of the review comments praise code segments. Those comments may help building positive impressions between the team members, and encourage good coding, but interviewees rated those as 'Not useful'.
- Some comments pointed out work that needed to occur in the future, but not during the current development cycle. In a few cases, these comments were about code that was not related to the change at all, but simply existed in the changed files. If immediate actions based on these comments are not foreseeable, authors rated such comments as not useful.

During the classification of the useful/not useful comments by the interviewees, we looked for signals that could be used to automatically classify comment usefulness. We observed that useful comments very often trigger changes in close proximity to the comment-highlighted lines (recall that a comment is associated with a highlighted portion of the code) in subsequent iterations. We refer to these comments that appear to induce a change in the code as *change triggers*. Therefore, automatically identifying these change triggers can provide a valuable feature for classifying the usefulness of review comments.

We found that examining the status of comments can give some indication of comment usefulness. When a comment was marked as 'Won't Fix', authors often also perceived the comment as not useful (though occasionally authors flag useful comments as 'Won't Fix' to defer a change). On the other hand, most of the times that the status was set to 'Resolved', it indicated that the author perceived the comment as useful and addressed it.

The insights gathered from the interviews were crucial for our understanding of review authors' perception of usefulness. In the next research phase, we leverage our findings as they allow us to manually classify a large amount of review comments independent from developers. Further, we identified two valuable features (whether a comment has changes in close proximity in subsequent iterations and comment status) that can be computed automatically and used in an automated classifier for useful comments.

V. Automated Classification of Comment Usefulness

In the second phase of the study, we built upon the findings from the interviews to create an automated classifier to distinguish between useful and not useful comments. The purpose behind this is that an accurate automatic classifier allows us to classify a large number of review comments, enabling a large scale quantitative study of review comment effectiveness (as we conduct and describe in Section VI), as manual classification of code review comments is time consuming and does not scale. Note that our goal in building a classifier is *not* to predict comment usefulness at the time that the comment is written. Such a classifier would not be useful to developers (an author of a change under review likely wants to read *every* comment on the review) and the features that

such a classifier could use are constrained (e.g., the status of the comment isn't known at the time a comment is made).

In order to build a classifier, we have to first identify potential signals (metrics), then select a classification approach, and finally implement, train and evaluate the classifier. A crucial part of training and evaluating a classifier is obtaining a standard dataset which contains information about whether or not a code review comment is useful, a so called oracle. Relying on the 145 comments rated during the interviews is not sufficient as that dataset is too small to build a reliable classifier. To rectify this, we manually analyzed and separately validated an additional 844 comments to enhance the oracle. In the following, we describe the manual classification, the oracle generation process, the selected signals and their calculation, the classifier, and the validation of our model. We end this section by highlighting the results of the classification approach.

A. Manual Classification

Based on our insights from the developer interviews, we manually classified 844 review comments from five projects across Microsoft: Azure, Bing, Exchange, Office, and Visual Studio.

We classified each of the comments into one of the two categories: *Not Useful*, and *Useful*. Although our exploratory interviews included three classifications, authors in those interviews indicated that *Somewhat Useful* comments still were valuable enough to improve their code. For example, as we observed disagreements between authors related to how they classify nit-picking comments (Section IV-B), we discussed with them how to rate nit-picking comments. Since nit-picking comments are useful for long-term project maintenance and often led to changes in the code, we agreed to rate them as 'Useful'. Similar discussions happened for the other types of somewhat useful comments, thus we consider comments that would fall into the *Somewhat Useful* category as described in Section IV-B to be useful.

For the manual analysis, we randomly selected reviews with at least two iterations and two comments. Multiple iterations allow us to determine if a comment is in proximity to, and thus likely caused, a code change (i.e., we can tell if a comment in iteration 1 caused a code change or not based on the code submitted in iteration 2).

We read each comment thread and examined the code that the comment thread was referring to in an effort to understand the question or concern raised by the reviewer and how the author responded to it. If the author did not participate in the comment thread, we also examined the subsequent iteration(s) to determine if the suggested changes were implemented by the author. Our assessment of comment usefulness was primarily based on insights drawn from interviews such as the categories that comments fell into, the topics addressed in the comments and the types of action that the comments suggested. For example, if the author pointed out a valid defect and a suggested fix, we classified it as useful, whereas a question about how the changed component was tested was termed not useful.

To assess validity and subjectivity, we used inter-rater reliability. A sample of one hundred comments was selected randomly from the set of 844 comments and each of the three authors classified the hundred comments independently, using the findings from Section IV-B as a guide. Our classifications differed for only three out of the hundred. Since we used more than two raters for the transcriptions we calculated inter-rater reliability using Fleiss' Kappa [17] on the individually coded responses. The inter-rater reliability kappa κ value was 0.947, which Landis and Koch classify as "almost perfect agreement" [18]. Thus we have confidence that the ratings are valid and consistent.

We combined the comment classifications from the interviews and the manual classification into one dataset that we used as our oracle to build our comment usefulness classifier.

B. Attributes of Useful Comments

Based on the insights from the interviews and our manual analysis, we identified eight attributes of comments which we anticipate distinguish useful from not useful comments.

Some of these attributes of comments are not known at the time that the comment is made such as whether a comment will be a change trigger. Using attributes that are not available until a review is complete would be a problem if we were trying to build a predictor of comment usefulness to be used at the time a comment was made. However, since our goal is to classify the usefulness of a large number of comments in an effort to enable an empirical investigation of what leads to useful comments, this is not an issue for our study.

The first attribute we considered was the *thread status* (discussed in Section IV-B). Further, to measure the engagement and activity of a comment we count the *number of participants* commenting on one thread (including the author), the *number of comments* constituting the thread, whether or not there was a *reply from the author* to a comment, and the *number of iterations* in the code review.

In addition, we hypothesize three more signals that may be good candidates for our classifier: from the interviews we saw that a comment that was a *change trigger* was often deemed useful. We also observed anecdotally that 'Useful' comments frequently contained a different set of *keywords* (e.g., 'fixed', 'bug' or 'remove') than 'Not useful' comments. Finally we believe that *the sentiment* of a comment (i.e. whether or not a comment is formulated in a positive or negative tone) may related to comment usefulness.

C. Attributes calculation

While the first set of signals can be derived easily from the code review data itself, the latter three signals (i.e. if a comment is a change trigger, the keywords in a comment, and comment sentiment) require additional work to compute. We briefly describe our approaches below.

Change trigger. From the manual analysis and interviews, we observed that most changes triggered by a comment happen in close proximity to the comment-highlighted line(s). To see if a comment triggered a change we look at differences between the file containing the comment and versions of the file submitted in subsequent iterations of the code review. For example, if a comment is made on a particular line of code in *foo.cs* in the first iteration of a code review, then we track the position of that same line of code in subsequent versions of *foo.cs* in the

subsequent iterations, even if the location of the line is affected by other additions or deletions. If there was a change in any later iteration that is in close proximity to the line of code associated with the comment then the comment is considered a change trigger.

Sometimes review comments trigger changes before or after the code highlighted by the comment, as illustrated by Figure 4(a). We investigated this by varying the level of proximity in our analysis (i.e., the number of lines a change is away from the code associated with a comment). At the lowest level, we only associated changes on the same line as the highlighted code, while at the highest level, we considered changes within ten lines of the comment. Based on analysis of the false positives (sometimes changes close to a comment are not related to the comment) and false negatives (if a change is more than a few lines away from the comment, we might not categorize it as a change trigger) at each level of proximity, we found that the best results occurred when we associated code changes to comments that were at most one line away from the change.

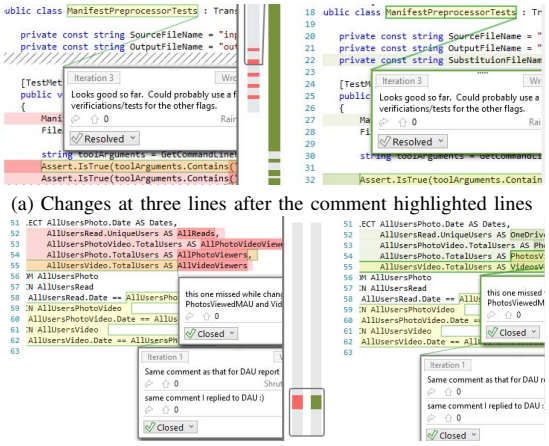
Keyword-based Classifier. As *Naive Bayes classifiers* have been used to successfully classify natural language in different domains (e.g. filtering spam emails [19]), we implemented such a classifier based on the multi-variate Bernoulli document model [20] to classify comments. As the training corpus, we used comments from our previously established oracle. We performed standard pre-processing of the corpus [21] by removing white-space, punctuation, and numbers, converting all words to lower case, applying a Porter-stemmer [22] to generate stems of the words and removing common English stop-words. We also limited keywords to those appearing in at least ten comments. After these steps, there were 349 unique keywords. After the Naive Bayes model is trained using our oracle, the frequencies of each keyword in a new comment are fed into the model to generate a classification of useful or not useful. This classification is then used as a feature in our main classifier.

Comment sentiment. Sentiment analysis has been widely used to classify natural language text [23]. To use the sentiment of a comment as a probable predictor of usefulness, we used the Microsoft Research Statistical Parsing and Linguistic Analysis Toolkit (MSR-Splat) [24] service to calculate the sentiment of the comments. MSR-Splat calculates the probability of a comment having a positive sentiment as a floating point number between 0.0 and 1.0. Similar to prior literature [23], we categorized the comments based on sentiment probability into five categories, each category spanning a 0.2 probability range from the previous category: extremely negative, somewhat negative, neutral, somewhat positive, and strongly positive. The sentiment category for the text of the comment was used as an input feature for our main classifier.

D. Classification process and validation

After calculating the attributes (section V-B) for the comments in our oracle, we used a classification tree algorithm[1] [25] to build a decision tree model based on the discussed features

[1] A machine learning technique in which the goal is to predict the value of a target variable based on several input variables



(a) Changes at three lines after the comment highlighted lines

(b) Two comments within a close proximity

Fig. 4: Identifying changes triggered by review comments

(Section V-B) of the comments in our oracle. To validate our tree-based model, we employed 10-fold cross-validation [26] and repeated the process 100 times.

Finally, to validate our model via review participants, we sent five developers (three of them were not part of the interviews) a list of review comments they received for their recent code reviews and asked them to classify each comment as 'Useful' or 'Not Useful'. We then compared these classifications with the classifications produced by the decision tree. The results of this validation are discussed below (Section V-F).

E. Effects of individual attributes

We examined each of the features individually to understand the decision characteristics of attributes and the strength of relationship with usefulness. Due to space restriction we report some interesting observation, but omit detail correlation outcomes.

If a comment triggered changes within one line distance to comment-highlighted lines, it was highly likely to be useful (precision: 88% and recall: 78%). If the author did not participate in a comment-thread, it was more likely to be useful (88%) than those threads where the author did (49%). A manual investigation of comments showed that absence of the author in a comment-thread very often indicated an implicit acknowledgment by the author and a useful comment. On the other hand, author participation indicated either a useful comment with explicit acknowledgment (e.g., 'done', 'fixed', and 'nice catch') from the author or a not useful question / false positive, which the author responded to.

Comment-threads with only one comment or participant were more likely to be useful (88%), than those with more than one comment or participant (51%). The explanation is similar as the explanation for author participation, as participation of several engineers indicated a discussion which might or might not be useful. No discussion often indicated implicit agreement. Our keyword based classification also showed promising results. Table I shows keywords, which belonged to at least 15 comments in our oracle and are at least twice more likely to be in a particular class of comments (i.e. either useful or not useful) than the other. We found that comments

TABLE I: Keywords distribution

Useful	Not Useful
assert, int, big, expand, least, nit, space, log, fix, match, action, line, rather, please, correct, should, remove, may be, move	leave, yes, message, store, doesn't, keep, result, first, let, default, actual, which, why, current, happen, time, else, exist, reason, type, work, how, item, want, really, not, fail, test, already

with command verbs (assert, expand, fix, remove, and move) or request (e.g., please, should, may be) are more likely to be useful. On the other hand, questions (e.g., why, which, and how), acknowledgment (e.g., yes, already) or denial (doesn't, and not) are more likely to be in not useful comments.

Most of the comments (83%) are made during the first two iterations. Comments made after the first two iterations are less likely to be useful. Also, comments where a change within one line of the comment highlighted text could be detected where much more likely to be useful. Most of the comments (92%) with Won't fix status are Not Useful, while most of the comments with Resolved/ Closed status are ($\approx 80\%$) Useful.

Finally, although majority (51%) of the comments had 'Extremely Negative' tones, only 57% of those comments were useful. On the other hand, comments with 'Neutral' or 'Somewhat Negative' tones were more likely to be 'Useful' ($\approx 79\%$ were considered useful).

F. Model performance

Figure 5 shows the decision tree model built from our oracle. The root node of the tree is *ChangeTrigger_1* (i.e. changes within one line distance to a comment-highlighted line).

Comment status is the second most important attribute for the classification. These two most important signals were evident to us during the first stage of this study (Section IV-B), and our model provides confirmation. The other useful attributes for the model are *number of comments*, *iteration number*, and *sentiment group*.

Based on one hundred 10-fold cross-validations of the decision tree model, our model had a mean precision of 89.1%, mean recall of 85.1%, and mean classification error of 16.6%. Finally, the review participants that we contacted rated 97 out of the 125 comments as Useful. Our model classified 105 of those review comments as 'Useful', of which 91 were correct. In this step, our model had 86.7% precision, and 93.8% recall.

VI. Empirical Study of Factors influencing Review Usefulness

The ultimate goal of our study is to understand the influences of different factors on the usefulness of code reviews feedback. Specifically, we investigate two types of factors: 1) characteristics of the reviewers and their team and 2) characteristics of the changeset under review. The selection of those factors was guided by prior studies on software inspection [27], [28], and suggestions for code review practices [29], [16]. To identify the influence of each factor on the usefulness of code review comments we used our trained decision tree to classify review comments in useful and not useful comments (as described in Section V-C) and then examined the relationship of various factors with usefulness. In total, we analyzed ≈ 1.5 million comments from 190,050 review requests from five major

TABLE II: Comment usefulness density

Project	Domain	# of Reviews	# of Comments	# of Useful Comments	Usefulness Density
Azure	Cloud software	15,410	126,520	86,914	68.6%
Bing	Search engine	92,987	664,619	426,513	64.2%
Visual Studio	Development tools	12,802	113,208	75,378	66.6%
Exchange	Email server	29,272	246,566	155,971	63.3%
Office	Office suite	33,351	299,919	204,045	68.0%
	Total	**190,050**	**1,496,340**	**979,440**	**65.5%**

Microsoft projects, i.e., Azure, Bing, Visual Studio, Exchange and Office, the same projects that the comments used to train the decision tree were drawn from. We selected those projects as they represent a wide range of domains, development practices, and include both services and traditional desktop applications. Each project has a substantial code base, comprising millions of lines of code. Based on these data sets, we examine the relationship of *comment usefulness density* (i.e. the proportion of comments in a review that are considered useful) with a number of factors related to the reviewers and what is being reviewed. Table II provides summary information for each of the five projects, including the overall comment usefulness density. Interestingly, all projects have a similar comment usefulness density between 64% and 68%. In this section, we explore the influence of the two aforementioned factors on comment usefulness.

A. Reviewer characteristics

Prior studies on software inspection found wide variation in the effectiveness of different inspectors (i.e., the person examining the code), even when they are using the same technique on the same artifact [27]. Similarly Rigby et al. suggested using experienced members or co-developers as reviewers [29]. Since those studies suggest that reviewer characteristics can have an influence on review usefulness, we studied the following three aspects of reviewer characteristics: 1) experience with the artifacts in the review, 2) experience in the organization, and 3) being in the same team as the change author. We also examined one aspect of the project the reviews belongs to: how the effectiveness of comments in an entire project changes over time.

1) Do reviewers that have prior experience with a software artifact give more useful comments?: We investigated two aspects of experience: first, experience in *changing*, and second experience in *reviewing* an artifact. We used a source code file as the level of granularity for experience. We compared the density of useful comments for developers who had previously made *changes* to the files in a review to the density of useful comments made by developers who had not made changes. The developers who had made prior changes to files in a change under review had a higher proportion of useful comments in four out of the five projects (all but Exchange which shows marginal increases), but we did not see a difference in effectiveness based on the number of times that a developer had worked on a file. That is, comments from developers who had changed a file ten times had the same usefulness density as from developers how had only changed a file once. In detail, experience in changing a file at least once increases the density of useful comments from 66% to 74% for Azure, from 60%

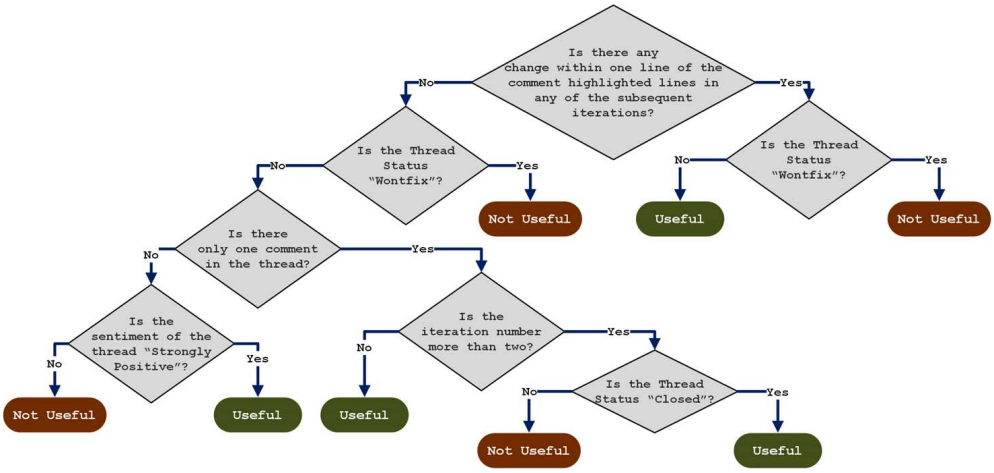


Fig. 5: Decision Tree Model to Classify Useful Comments

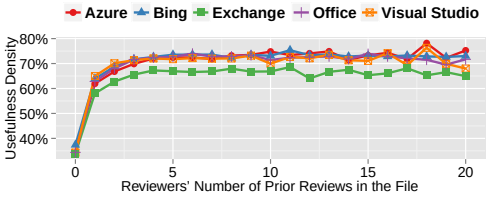


Fig. 6: Prior experience reviewing the artifact vs. comment usefulness density

to 71% for Bing, from 66% to 70% for Visual Studio, from 67% to 72% for Office and from 62% to 63% for Exchange.

Our analysis of the effect of experience in *reviewing* a file showed strong effects on the density of useful comments (Figure 6). For all the five projects, reviewers who had reviewed a file before were almost twice more useful (65% -71%) than the first time reviewers (32% -37%). Comment usefulness densities also show an increasing trend with the number of prior reviews up to around five reviews, after which the usefulness density plateaued between 70% and 80%.

Based on these results, we conclude that developers who have either changed or reviewed an artifact before give more useful comments. One possible explanation for these results is that reviewers who have changed or reviewed a file before have more knowledge about the design constraints and the implementation. Therefore, they are able to provide more relevant comments. Also, a first time reviewer may not know the design and context, they may ask questions to understand the implementation, or identify false issues based on their incorrect assumption. Unsurprisingly, first time reviewers of an artifact are providing less valuable feedback.

We assume that review experience shows more drastic effects on comment usefulness than change experience because many teams have a practice of letting new developers first review the code before they are allowed to change the code. Therefore, a developer who makes the first change to a file has most likely already reviewed it before.

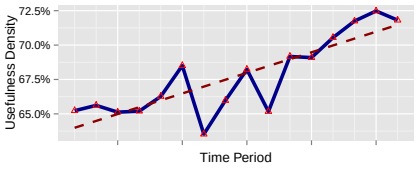
We calculated a reviewer's experience based on his or her tenure at Microsoft. In four out of the five projects (all but Exchange), reviewers that spend more time in the organization have a higher density of useful comments. The effect is especially visible for new hires, who in the first three months had the lowest density of useful comments. During the first three quarters, the usefulness density increases the most, and stays relatively stable after the first year. The first year at Microsoft is often considered "ramp up" time for the new hires. During that time employees become more familiar with the code review process, project design, and coding practices at Microsoft. After the ramp up period, they can be as useful reviewers as their senior Microsoft peers. In detail, we saw for Azure an increased density of useful comments from 60% to 66%, for Bing from 62% to 67%, for Visual Studio from 60% to 70% and for Office from 60% to 68% after the first year. For Exchange, we could not see a steady trendline, and usefulness ratios vary between 60% to 65%.

2) Do reviewers from the same team give more useful comments?: We hypothesize that a reviewer may give more useful comments to member of his or her own team because they are familiar with that person, their abilities, and are more invested in the quality of the code that the team ships. A team in this sense is a group of usually four to ten developers all working under the same manager or developer lead (i.e., each project in our analysis comprises many teams). We found that roughly three quarters (76%) of review comments come from reviewers on the same team as the author. Although cross-team reviewers were less frequent, we found that reviewers from different teams gave slightly more useful comments than reviewers from the same team in all the five projects. As Table III shows however, the magnitude of the difference is quite small (under 1.5% for all but one project) and is statistically significant only because of the large sample used (over one million comments in total). Based on this, we conclude that there is no noticeable difference in comment usefulness density between reviewers who are on the same team or on different teams than the author.

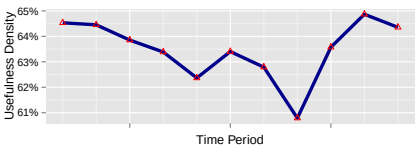
3) How do comment usefulness densities vary over time?: Porter et al. found in their study on software inspection that effectiveness and defect discovery rates vary over different time periods [28]. We investigated whether reviewers are becoming

TABLE III: Usefulness density vs. team

Reviewer and Author in the same team	Azure	Bing	Visual Studio	Exchange	Office
Yes	68.0%	67.8%	66.0%	62.5%	66.7%
No	69.4%	68.6%	69.1%	63.7%	66.9%



(a) Office



(b) Bing

Fig. 7: Temporal Trends of Usefulness density

more efficient in making comments on the same project over time by looking at the the comment usefulnesss density of the entire project for different periods of time. We found that for four out of the five projects the density of useful comments increases over time and we suspect this can be attributed to both increased experience with the project, similar to our findings in Section VI-A1 and also refinement of the code reviewing process (more training of developers, better tracking of code review data, etc.). Figure 7 (a) shows the temporal trend for a snapshot of time for the Office project. Even though the long term trend shows an increase density of useful comments, for three out of the five projects we noticed peaks and valleys in the density of useful comments for limited periods. For example, Figure 7:(b) shows a temporary drop for the Bing project. Examining trends of usefulness density can help managers determine whether or not code review practices are improving. For example, during times where the usefulness density drops, managers can be alerted and can easily "drill-down" to the specific reviews, reviewers, changes, or components, that are contributing to the drop and can take corrective action.

B. Changeset characteristics

Porter et al. found that software inspection effectiveness depends on code unit factors such as code size, or functionality [28] and Rigby et al., suggested that reviews should contain small, incremental and complete changesets [29]. Therefore, we investigated whether size of the changeset or type of file under review has any effect of review usefulness.

1) Do larger code reviews (i.e., with higher number of files) get less useful comments?: Figure 8 illustrates how comment usefulness density change with the number of files in a change under review. The trendline shows that as number of files in the change increases, the proportion of comments that are useful drops. This result supports Rigby's recommendation for smaller changesets. Developers have indicated that if there are more files to review, then a thorough review takes more time and effort. As a result, reviewers may opt for cursory review of large changesets and may miss some changes. This may lead

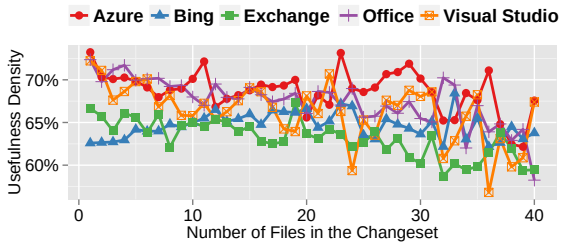


Fig. 8: Usefulness density vs. Number of files

to false positives or more questions to the author in an effort to understand the change, causing lower usefulness densities.

2) Do the types of files under review have any effect on comment usefulness?: We grouped the files into four groups based on the purpose of the file: 1) Source code (e.g., C#, C++, Visual Basic or C-header files), 2) Scripts (e.g., SQL or command line scripts), 3) Configuration (e.g., .Config or .INI) files), and 4) Build (e.g., Project or make files). We observed that source code files had the highest density of useful comments (70%). On the other hand, build files had the lowest comment usefulness densities (65%). As notable outliers, Visual studio solution files (a type of configuration file) (57%) and make files (53%) had a low proportion of useful comments. We expect that this may be due to the complexity of the these files (e.g., McIntosh et al. have demonstrated the complexity of build files [30]) where the impact of changes on the overall system are sometimes harder to assess than for source code. Code reviewing tools and practices also often emphasize the review of code, whereas review of configuration files and build files are given less attention.

VII. Threats To Validity

All projects selected for this study belong to the same organization practicing code reviews using the same tool. While the tool itself may be specific, prior work has shown that most reviewing performed today follows a similar workflow. Code reviews via CodeFlow are similar to the processes based on other popular tools such as Gerrit, ReviewBoard, GitHub pull requests, and Phabricator. Many companies and open source projects that practice review are using tools such as these rather than email. Like CodeFlow, these tools facilitate feedback from reviewers about the change, often allowing reviewers to indicate specific parts of the change [7], [31].

Most of the attributes calculated for this study can be also calculated for code reviews conducted with these other tools. Also, prior study results suggest that there are large similarities between the code review practices of different OSS and commercial projects [7]. We have attempted to mitigate threats to external validity by including projects in this study that represent diverse product domains and platforms. Nonetheless, some biases remain; all projects are large-scale, relatively mature, and come from the same company.

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, using k-fold cross validation, and comparing classification results with

ratings from the authors that received the comments. While the model achieves high levels of accuracy with precision and recall values between 85% and 90%, the mean classification error rate is around 15%. 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 in phase three of our study (if, for example, our model incorrectly classifies the usefulness of comments in large reviews far more than those with fewer files). We have no reason to believe such a relationship exists, but have no empirical evidence.

Lastly, a common misinterpretation of empirical studies is that nothing new is learned (e.g., "I already knew this result"). However, such wisdom has rarely been shown to be true and is often quoted without scientific evidence. This paper provides such evidence: Most common wisdom and intuition is confirmed (e.g., "prior experience in artifacts help useful reviews") while some is challenged (e.g., "reviewers from the same team are more useful").

VIII. Implications of the Results

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

Reviewer Selection: This study showed that experience with the code base is an important factor to increase the density of useful comments in code reviews. Therefore, we suggest that reviewers should be selected carefully. Automatic review suggestion systems can help to identify the right set of developers that should be involved in a review. Few studies [32], [33] have attempted to suggest reviewers for code changes, and in some popular code review tools such as Crucible and CodeFlow automatic reviewer suggestion features already exist.

However, such tools should be used with caution. As our results suggest that new hires and reviewers with limited experience provide feedback with limited utility, one might be tempted to exclude them from the reviewing process. Due to the knowledge dissemination aspects of code review [6], we recommend including inexperienced reviewers so that they can gain the knowledge and experience required to provide useful comments to change authors. Rather than excluding the inexperienced, authors of changes should include at least one or two experienced reviewers to ensure useful feedback.

Managing changesets: Our results suggest that review effectiveness decreases with the number of files in the change set. Therefore, we recommend that developers submit smaller and incremental changes whenever possible, in contrast to waiting for a large feature to be completed. Special care should be taken when non-code files, such as configuration or build files, are included in reviews, as these elicit less useful feedback. Reviewers should be encouraged to give them more attention and authors can be of help by providing more details of the changes to these files in the change description prior to sending the change out for review.

Identifying weak areas: As comment usefulness density can be calculated and compared along various dimensions such as types of files or particular modules of a system, this measure can be used by teams to identify areas where code reviews are less effective. Teams can also continuously monitor themselves and address issues when they arise. For example, one team that we talked to indicated that they have been manually reading through the comments in reviews each week to see if there are patterns that need to be addressed. Our classifier can help by automatically identifying the less useful comments which can speed up the process of determining areas of code experiencing problems or can reduce the number of comments that team members need to read. Project management can also identify weak reviewers and take necessary steps to help them become efficient.

IX. Related Work

Prior studies exist that have examined the effects of different factors on the effectiveness of traditional Fagan-inspection [2]. Porter et al. studied the effect of three types of factors (i.e. code unit, reviewer, and team) on inspection effectiveness. They found code unit factors (e.g., code size, and functionality) and reviewer factors (e.g., presence of certain reviewers) as the most influential factors [28]. Although tool-based contemporary code review practices differ a lot from the traditional inspection techniques [7], we also observed the effect of changeset size and reviewer experience in this study.

Most of the earlier studies examined traditional software inspection techniques, only a few recent studies have examined informal code review practices. Rigby has published a series of studies examining informal peer code review practices in OSS projects [34], [35], and comparing the review process between commercial and open source projects [7]. In the later study, Rigby and Bird found that despite differences between different projects, many of the characteristics of contemporary code review practices (i.e. review interval, number of comments, number of reviewers) were very similar [7]. Bacchelli and Bird investigated the purposes and outcomes (i.e., accept or reject) of modern code reviews and found that although finding defects is the primary motivation, only a fraction of review comments finds defects. On the other hand, code reviews provide additional benefits such as knowledge dissemination, team awareness, and identifying better solutions [6]. Baysal et al. found a variety of factors (such as review size, component, reviewer characteristics, or author experience) to have signification effects on code review response time and outcome [36].

While most of the studies on modern code reviews have examined the code review process and factors that affect review interval or outcome, we are not aware of any study that has explored the factors influencing the effectiveness of modern code review.

X. Conclusion

In this study, we have identified a set of factors affecting the usefulness of code reviews. We provided recommendations to both practitioners and researchers to improve code reviews. It is our hope that the insights discovered in this study will be helpful to improve code review process as well as to build better code review tools.

REFERENCES

[1] P. Rigby, B. Cleary, F. Painchaud, M. Storey, and D. German, "Contemporary peer review in action: Lessons from open source development," *Software, IEEE*, vol. 29, no. 6, pp. 56–61, Nov 2012.

[2] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182 –211, 1976.

[3] M. Fagan, "A history of software inspections," in *Software pioneers*. Springer, 2002, pp. 562–573.

[4] L. G. Votta Jr, "Does every inspection need a meeting?" in *ACM SIGSOFT Soft. Eng. Notes*, vol. 18. ACM, 1993, pp. 107–114.

[5] P. M. Johnson, "Reengineering inspection," *Comm. of the ACM*, vol. 41, no. 2, pp. 49–52, 1998.

[6] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 712–721.

[7] 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*, ser. ESEC/FSE 2013, 2013, pp. 202–212.

[8] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 192–201.

[9] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 931–940.

[10] 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*, 2013, pp. 133–142.

[11] "Gerrit code review." [Online]. Available: https://code.google.com/p/gerrit/

[12] "Phabricator." [Online]. Available: http://phabricator.org/

[13] "Review board." [Online]. Available: https://www.reviewboard.org/

[14] T. R. Lindlof and B. C. Taylor, *Qualitative communication research methods*. Sage, 2010.

[15] M. Mantyla and C. Lassenius, "What types of defects are really discovered in code reviews?" *Software Engineering, IEEE Transactions on*, vol. 35, no. 3, pp. 430–448, May 2009.

[16] J. Cohen, E. Brown, B. DuRette, and S. Teleki, *Best Kept Secrets of Peer Code Review*. Smart Bear, 2006.

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

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

[19] M. Sahami, S. Dumais, D. Heckerman, and E. Horvitz, "A bayesian approach to filtering junk e-mail," in *Learning for Text Categorization: Papers from the 1998 workshop*, vol. 62, 1998, pp. 98–105.

[20] A. McCallum, K. Nigam *et al.*, "A comparison of event models for naive bayes text classification," in *AAAI-98 workshop on learning for text categorization*, vol. 752. Citeseer, 1998, pp. 41–48.

[21] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE*

[21] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 970–983, 2002.

[22] M. F. Porter, "Snowball: A language for stemming algorithms," http://www.tartarus.org/\~{}martin/PorterStemmer, 2001.

[23] 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.

[24] C. Quirk, P. Choudhury, J. Gao, H. Suzuki, K. Toutanova, M. Gamon, W. tau Yih, L. Vanderwende, and C. Cherry, "Msr splat, a language analysis toolkit," in *Proceedings of NAACL-HLT 2012*. Association for Computational Linguistics, June 2012. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=173539

[25] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. CRC press, 1984.

[26] 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.

[27] J. Carver, "The impact of background and experience on software inspections," 2003.

[28] A. Porter, H. Siy, A. Mockus, and L. Votta, "Understanding the sources of variation in software inspections," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 7, no. 1, pp. 41–79, 1998.

[29] P. Rigby, B. Cleary, F. Painchaud, M.-A. Storey, and D. German, "Contemporary peer review in action: Lessons from open source development," *IEEE Software*, vol. 29, no. 6, pp. 56–61, 2012.

[30] S. McIntosh, M. Nagappan, B. Adams, A. Mockus, and A. E. Hassan, "A large-scale empirical study of the relationship between build technology and build maintenance," *Empirical Software Engineering*, pp. 1–47, 2014.

[31] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 345–355.

[32] J. B. Lee, A. Ihara, A. Monden, and K.-i. Matsumoto, "Patch reviewer recommendation in oss projects," in *Software Engineering Conference (APSEC, 2013 20th Asia-Pacific*. IEEE, 2013, pp. 1–6.

[33] G. Jeong, S. Kim, T. Zimmermann, and K. Yi, "Improving code review by predicting reviewers and acceptance of patches," *Research on Software Analysis for Error-free Computing Center Tech-Memo (ROSAEC MEMO 2009-006)*, 2009.

[34] 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*. ACM, 2008, pp. 541–550.

[35] P. C. Rigby and M.-A. Storey, "Understanding broadcast based peer review on open source software projects," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 541–550.

[36] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, "The influence of non-technical factors on code review," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*. IEEE, 2013, pp. 122–131.