ROSAEC center
Research On Software Analysis for Error-free Computing
소프트웨어 무결점 연구센터 KOSEF ERC

ROSAEC MEMO
2009-006
September 13, 2009

Improving Code Review by Predicting Reviewers and Acceptance of Patches

Gaeul Jeong
Seoul National University
gejeong@ropas.snu.ac.kr

Sunghun Kim
Hong Kong University of Science and Technology
hunkim@cse.ust.hk

Thomas Zimmermann
Microsoft Research
tzimmer@microsoft.com

Kwangkeun Yi
Seoul National University
kwang@ropas.snu.ac.kr

September 13, 2009

Abstract

Code reviews are an important part of software development because they help to increase code quality and reliability. For this paper, we observed the review processes of two open-source projects, Firefox and Mozilla Core. We noticed that code reviews are mostly organized manually. In particular, finding appropriate reviewers is a complex and time-consuming task and, surprisingly, impacts the review outcome: review requests without an initial reviewer assignment have lower chances to be accepted (and take longer).

Based on our observations we propose two improvements: (1) predict whether a given patch is acceptable and (2) suggest reviewers for a patch. We implemented and tested both approaches for the Firefox and Mozilla Core projects. In our experiments, the prediction accuracy was 73% for the the review outcome and 51–80% for the reviewer recommendation. The values for accuracy are higher than those of comparable approaches and are high enough to be useful and applicable in practice.

1 Introduction

Code reviews are an important part of software development. They can help to improve the quality of software by finding and fixing bugs [1, 2]. In addition, they can help to train developers on the code being reviewed. In open-source projects, code reviews are a popular measure to ensure the quality of external contributions [38].

In this paper we present an empirical study of the review process in the Firefox and the Mozilla Core projects. Both projects encode the code reviews in the Bugzilla system as attachments (patches) and activities (creating flags that a patch has been reviewed). In total, we analyzed over 56,000 code reviews for this study.

We found that the number of review requests has increased over the past years, which indicates the increased popularity and importance of code review in open source. On average a review takes 1.5 days and between 39% and 45% of patches are accepted. We also observed that the majority of the code review process is organized manually; in particular, finding appropriate reviewers is a complex and time-consuming task. This has also an impact on the review outcome: review requests without initial reviewer assignment take longer and have lower chances to be accepted. These findings contribute towards the empirical body of validated knowledge in the field of code reviews.

Furthermore, our results inform the design of tools for code review. To reduce the organizational overhead of code reviews, we propose the following two enhancements, which we both implemented and tested on the Firefox and Mozilla Core projects.

- *Predicting the review outcome.* This can help to reduce the review interval; for example, patches which are predicted as accepted could be auto-accepted in case no review is provided within a certain time period. In addition, it helps patch writers to check their patches and to get early feedback on how to improve patches.

  In our experiments, we could **predict the review outcome with an accuracy of 73%**. In comparison, the accuracy is 60% for change classification approaches [20], which address a similar, but different problem.

- *Recommending reviewers.* This can help patch writers to direct the patches to the most appropriate reviewers and reduce the time between the submission of the patch and the start of the actual code review. In addition, it helps reviewers because they do not need to monitor bug reports for unassigned review requests anymore.

  In our experiments, the **top five recommendations predicted the actual reviewer in 51–80% of all cases**. In comparison, the accuracy is 55% for approaches that assign developers to bug reports [5, 8], which is a similar, but different problem.

With such tools, the time spent on organizing code reviews can be reduced, which leaves more time for the actual code reviews.

1.1 Contributions

This paper makes the following contributions.

- *Empirical analysis of the code review process.* We characterize the code review process of Firefox and Mozilla Core, two large open-source projects. We found that code reviews are mostly organized manually and that missing reviewer assignments significantly decreases the chances of patch acceptance.

- *Predicting review outcomes.* We developed the first prediction model for the outcome of a code review. In our experiments, the accuracy of the model was 73%, which is higher than related work [20]. This prediction model gives early feedback to patch writers and helps developers to prioritize code reviews.

- *Recommending patch reviewers.* We developed the first recommender system for suggesting who should review a patch. The top five recommendations predicted the actual reviewer in 51–80% of all cases, which is higher than related work [5, 8]. This helps patch writers to direct patches to the most appropriate reviewers and relieves developers from monitoring bug reports for unassigned review requests.

1.2 Paper Outline

In Section 2 we explain the code review process of two open-source projects (Firefox and Mozilla Core) and provide statistics on review time, outcome, and frequency. Based on two observed challenges, the need for (semi-)automated tool support and the difficulty of finding reviewers, we propose novel prediction models in Section 3 for patch outcome and patch reviewer. We then describe the experimental setup to evaluate our results in Section 4, and present the results in Section 5. We close the paper with a discussion of threats to validity, related work and consequences.

2 Review Process

In this section, we observe the detailed code review process of two projects, Firefox and Mozilla Core. Their code review process is recorded in bug reports in Bugzilla as attachment and activity information [11].

Table 1: Example of Firefox code review activities.

who	when	what	action
d1	2003-08-11	Attach#129636	review? (d6)
d1	2003-10-13	Attach#129636	review? (d7)
d2	2003-11-30	Attach#129636	review? (d8)
d3	2003-12-12	Attach#129636	review? (d4)
d4	2004-02-17	Attach#129636	review+ (accepted)
d5	2004-04-25	Status	RESOLVED
		Resolution	FIXED

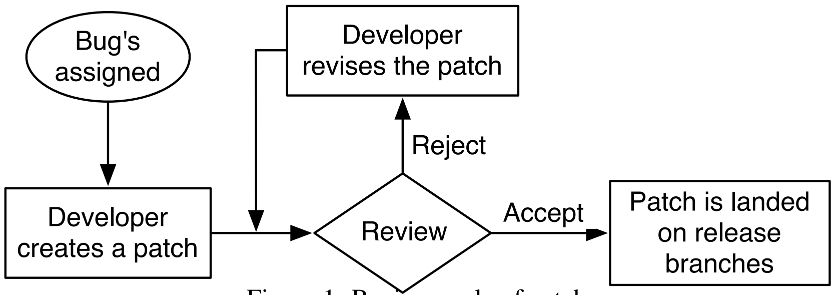
Bug's assigned
Developer creates a patch
Developer revises the patch
Reject
Review
Accept
Patch is landed on release branches

Figure 1: Review cycle of patch.

Table 1 shows how the Firefox code review process is recorded in Bugzilla through flags and activities. First, *d1* writes a patch and posts it with a review request to *d6*. For some reason, *d6* is unable to carry out the review, so it is reassigned to *d7*. Finally after a few more review reassignments, *d4* reviews the patch, and it is accepted (*review+*).

To understand the code review process and identify its challenges, we extracted and analyzed 12,503 review activities from Firefox and 45,997 from Mozilla Core. Based on our analysis, we define a review cycle in Section 2.1, measure the review time in Section 2.2, and observe the review outcomes in Section 2.3. We also measure the review requests per year in Section 2.4. In Section 2.5, we observe how the reviewer assignment affects the outcome of a review, and finally, we discuss the review process challenges based on our observations in Section 2.6.

2.1 Review Cycle

From Firefox and Mozilla Core review activities, we identify a typical review cycle as shown in Figure 1. Usually, a review cycle starts with a patch posted by a developer. When posting a patch, a developer usually recommends a suitable reviewer for the patch, then the patch will be reviewed by the recommended reviewer or other available reviewers. The review outcome is typically either *accept* or *reject*. Accepted patches will be committed in a release branch. Usually reviewers provide comments and suggestions for rejected patches, and developers revise and resubmit them. In addition to *accept*/*reject*, another possible review outcome is *open*, no review response.

2.2 Review Time

How long does it take to review a patch? We measure the review time for 56,083 patches from two projects. Since we can only measure the review time of either accepted or rejected patches, we exclude all *open* reviews for this observation.

Figure 2 shows the review time and corresponding number of patches. The y-axis indicates corresponding patch numbers and the x-axis indicates review time in the log-scale. It is noticeable that most patch reviews take a relatively short time. For example, about 2,042 Mozilla Core patches took 4 hours to be reviewed, and 506 Firefox patches took 4 hours. Similarly, 1,876 Mozilla Core patches took 1.5

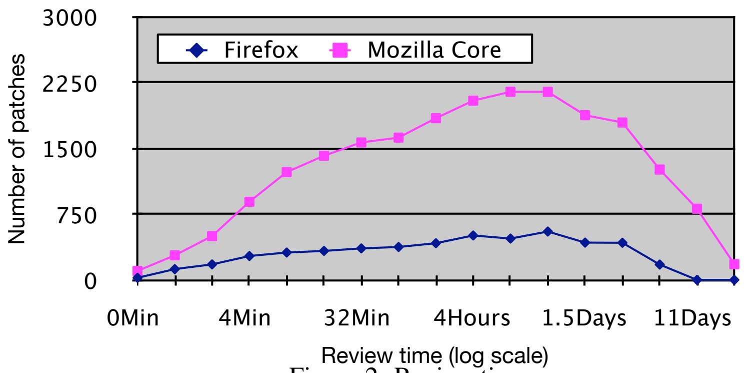

Figure 2: Review time.

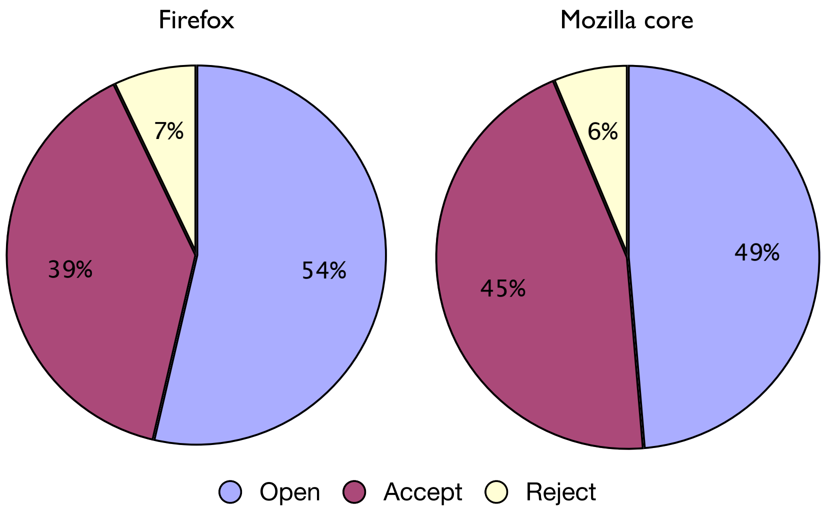

Figure 3: Review outcomes.

days to be reviewed.

The two graphs are shaped like normal distributions. The peak of the graph roughly indicates the median review time. For example, for both projects, the median review time is around 1.5 days.

The information in Figure 2 can be used to remind reviewers to reduce their review time. For example, if a patch is not reviewed within 1.5 days, which is the median review time, we can send an automatic reminder to reviewers.

2.3 Review Outcomes

In this section, we observe the ratio of review outcomes, *accept*, *reject*, or *open*.

The review outcomes for Firefox and Mozilla Core are shown in Figure 3. For Firefox, 39% of the patches are accepted while 7% are rejected. Similarly, for Mozilla Core 45% of the patches are accepted while only 6% is rejected. Overall these review outcomes look very generous.

However, a large number of patches were not reviewed. The *open* review outcome is either a gentle way of rejecting patches, or it indicates reviewers are not interested in the proposed patches. This is confirmed by two Mozilla developers, Gary Kwong and Channy Yun.

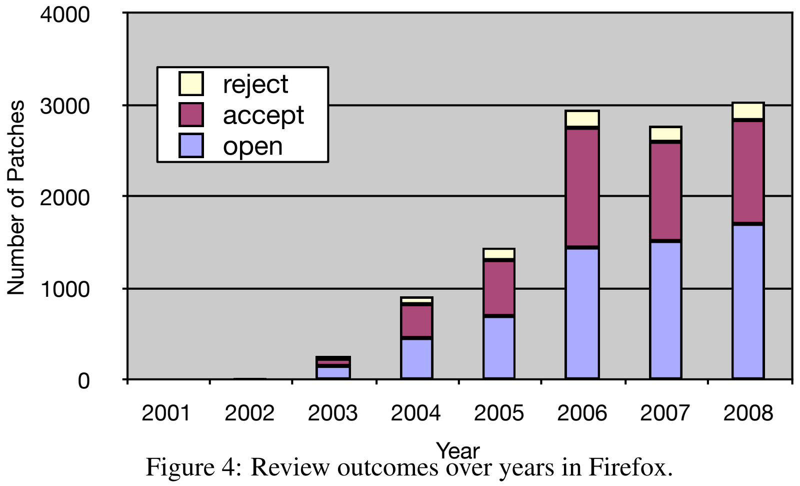

Figure 4: Review outcomes over years in Firefox.

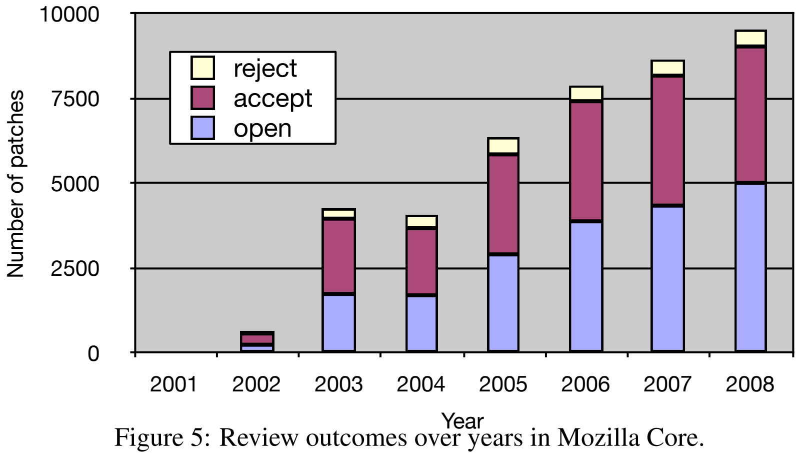

Figure 5: Review outcomes over years in Mozilla Core.

2.4 Reviews per Year

Next, we observe review request numbers and review outcomes over years (from 2001 to 2008).

We measure the review request numbers and outcomes up to the year 2008 since getting the review outcomes takes time. In other words, if we were to collect review outcomes up to time of writing, we might have too many open reviews for the year 2009.

Figure 4 and Figure 5 show the number of review requests and their outcomes per year. The review requests increase over time, which means that more human effort will be required to review proposed patches. Note that the rate of open reviewed patches slightly increases over the years. This is perhaps due to the increase in the number of review requests or too many review requests per year.

2.5 Reviewer Requesting

When a developer requests a patch review, it is possible to recommend a desirable reviewer. However, as shown in Figure 6, about 10% of the Firefox patches (more than 1,200 patches) are posted without reviewer requests, since some developers do not know any suitable reviewers for their patch. (This is also confirmed by the two Mozilla developers.)

In both projects, there are consequences for patches *without* a recommended reviewer. Surprisingly,

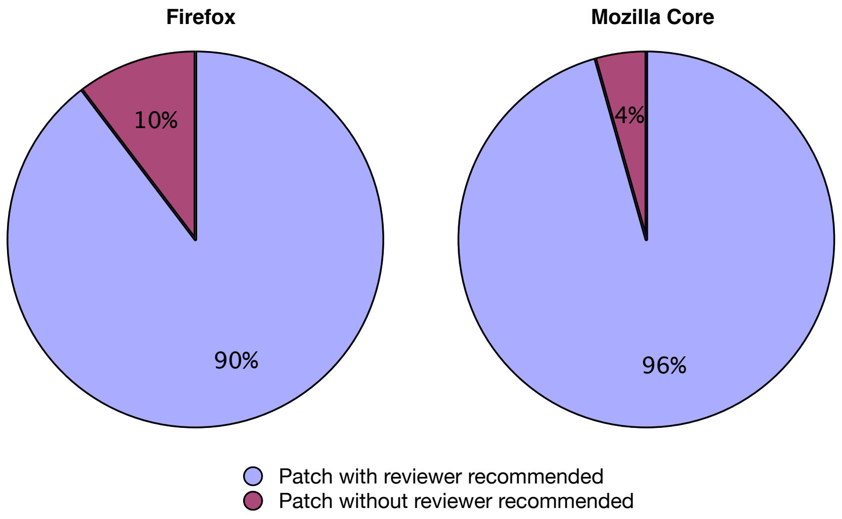

Figure 6: Percentages of patches with/without reviewer recommended.

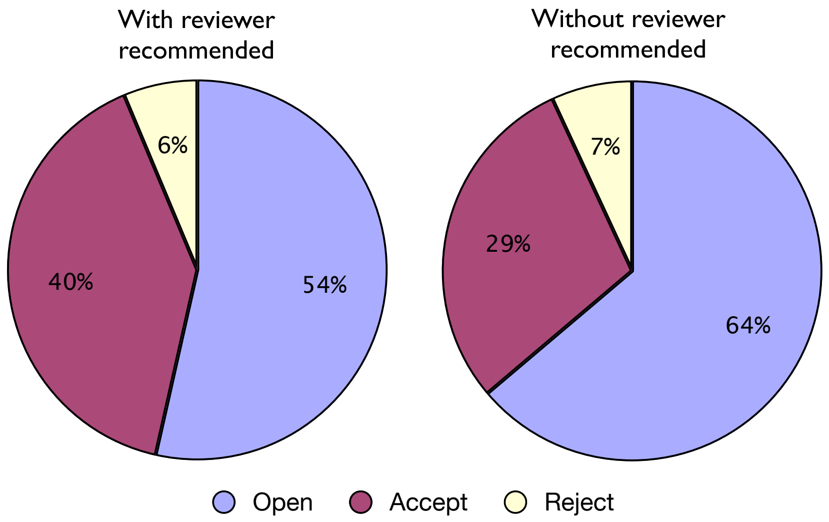

Figure 7: Review outcomes with/without reviewer recommended in Firefox.

patches without an initial reviewer have lower acceptance rates. For example, as shown in Figure 7 and Figure 8, in Firefox 40% of patches *with* reviewer recommended are accepted while only 29% of patches *without* reviewer recommended are accepted. Similarly, in Mozilla Core 45% of patches *with* reviewer recommended are accepted while only 32% of patches *without* reviewer recommended are accepted. These differences are statistically significant at $p \ll 0.001$ [45].

This interesting observation suggests that it is important to recommend reviewers for patches. Patches *with* reviewer recommended would have a higher likelihood of being reviewed and accepted.

2.6 Challenges

In this section, we discuss potential challenges in the review process based on our observations.

First, we learn that the number of review requests is consistently increasing as shown in Figures 4 and 5. However, all reviews are organized manually without any tool or prediction support and developers have to manually identify suitable reviewers for their patches. As reviewers manually inspect

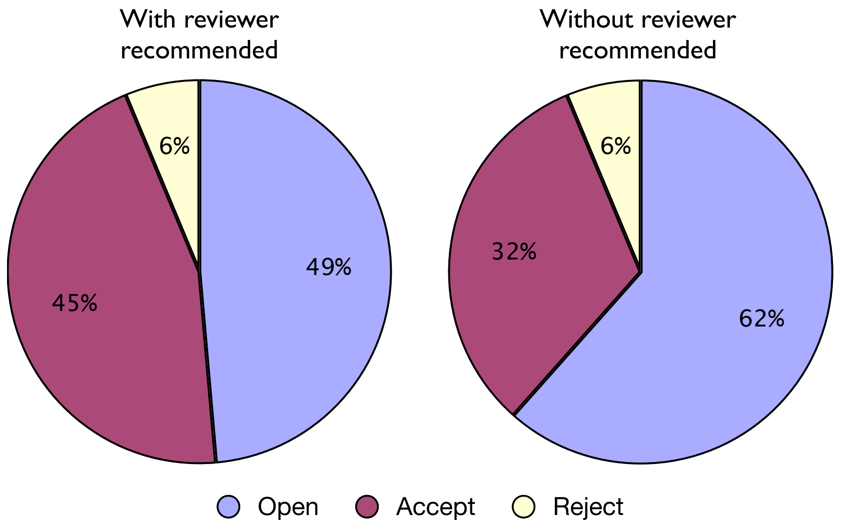

Figure 8: Review outcomes with/without reviewer recommended in Mozilla Core.

the proposed patches without any additional information about the patches, it would be desirable to automate or semi-automate review processes to reduce the manual effort.

Second, some patches are posted without a recommended reviewer. We show that such patches have a lower chance of being accepted in Section 2.5. Moreover, as shown in Table 1, review assignment activities, like bug tossing [17], may slow down the review process. It would be useful to predict suitable reviewers automatically for developers who cannot find reviewers for their patches.

3 Our Approach

Based on the identified challenges in Section 2, we propose two prediction models to assist the code review process: *reviewer prediction* and *patch acceptance prediction* using machine learning techniques [3]. To build the prediction models, it is necessary to characterize each patch by extracting features. The following subsections describe our prediction models and feature extraction techniques.

3.1 Prediction Models

We noticed that some developers do not know suitable reviewers to review their patches. To assist these developers, we propose a *reviewer prediction* model based on previous assignment history, inspired by Anvik et al. [5].

In addition, we propose a *patch acceptance prediction* model to predict review outcomes for a given patch using previous review outcomes. Similar to the change classification approach [20], this prediction model assists developers (patch writers) or reviewers by predicting review outcomes. Since code review requires a huge amount of manual effort, the automatic prediction helps developer to quickly decide the patch quality. As a result, developers may receive the review outcome early, which reduces the review and bug fix process time.

Leveraging the two prediction models, the current review cycle can be revised as follows: (1) A developer writes a patch. (2) The developer optionally runs our *patch acceptance prediction* model and revises the patch if the prediction result is a *rejection*. (3) The developer posts the patch with the assistance of our *reviewer prediction* model to identify suitable reviewers. (4) When a reviewer investigates the patch, our *patch acceptance prediction* model can again be used to predict the patch quality.

Table 2: Features from patch meta-data.

Patch meta-data	
Feature	*Example*
File Name	toolkit/mozapps/plugins/service/attic/ pluginfinderservice.java
Module Name	toolkit
Patch Attacher	user@domain.com
Number of characters	[0-9]+
Lines of codes	deleted [0-9]+, added [0-9]+, others [0-9]+
Number of '{', '}'	[0-9]+

Additionally, our prediction model can be used to reduce the number of *open* reviews. If a review takes more than a couple of days, we can either send a reminder or predict other suitable reviewers for the review using our *reviewer prediction* model. This approach would also reduce the review process time.

Overall, we believe our proposed prediction models assist developers and reviewers, reduce human effort, and yield effective review processes.

3.2 Feature Extraction

To characterize patches and build prediction models, we extract features from patches. We consider three possible feature sources: patch meta-data, patch content, and bug report information.

Patch meta-data (Table 2): Patch meta-data includes patch information such as patch writer, patch file names, and the number of patch files. These data represent the characteristic of patches, and we believe they are good features for our prediction models.

Like BugCache [21], the file location is a good feature candidate, and we use the file and module names as features. For reviewer prediction, the file location, the names of folders, can be especially important features since developers have code ownership [28], and most likely the owner of the corresponding patch file would review the patch [30].

The size of a patch and the number of files are good feature candidates, since *line of codes* (LOC) is a strong indicator for bug prediction models [21].

Usually, the patch quality largely depends on the developers' skill. We believe the patch writer would be a good feature. Similarly, the author information has been identified as an important feature to classify changes [20].

Patch content (Table 3): The quality of patch content plays a major role in deciding the review outcome. Correct, good, and simple patches will be accepted [30]. However, extracting patch quality as features is a challenging task.

Some syntactic and semantic information such as Abstract Syntax Trees (AST) or Control Flow Graphs (CFG) would be good features [7, 18]. However, patch files contain only a partial source code, and it is hard to build AST or CFG from a partial code.

Instead of syntactic and semantic information, we extract keyword vectors from patch files and use the vectors as features inspired by the change classification approach [20].

Bug report information (Table 4): We also consider bug report information as features. The bug reports and patches included in the bug reports are closely related. For example, if a bug in a report is not a real bug or a trivial feature enhancement request, then patches of the bug may not be interested either. For this reason, we extract features from bug report information such as *bug priority* and *severity*.

Table 3: Selected features from patch contents.

Selected patch contents (The number of keywords)	
Feature	*Example*
boolean, break, catch, char, class, continue, do, double, extends, final, finally, float, for, if, else, implements, interface, new, package, private, protected, public, return, static, super, synchronized, this, throw, try, while, null, get, set	[0-9]+, ..[0-9]+

Table 4: Features from bug report information.

Bug report information	
Feature	*Example*
Time after open	[0-9]+ days
Bug severity	null, blocker, critical, enhancement, major, minor, normal, trivial
Bug priority	null, –, P1, P2, P3, P4, P5
Bug reporter	user@domain.com

We also use the *time after open* as a feature. It indicates how long it took to write a patch after the bug was reported. If a patch was quickly written and posted, the corresponding bugs might be important ones. The *time after open* indirectly indicates the importance of a corresponding bug and patches.

Similar to using the patch writer information as a feature, we use bug reporter information as a feature.

4 Experimental Setup

We experimentally evaluate our prediction models and this section explains our experimental setup.

4.1 Subject Systems

The review process data of Firefox and Mozilla Core are used for our experiments. As we observed in Section 2, review process information including the review outcomes and activities are collected from Bugzilla [11].

Applying feature extraction techniques described in Section 3, we create a corpus for *patch acceptance prediction* and *reviewer prediction* models. The corpus for the two prediction models shares the same features.

For the patch acceptance prediction, the Firefox corpus includes 26,191 instances and Mozilla Core corpus contains 60,000 instances. Due to the large number of features and computing resource limitations (such as memory), we limit the instance number for Mozilla Core by selecting the last 60,000 instances. (We use the entire instances for Firefox.)

We label patch acceptance prediction instances as *accepted* or *rejected*. As confirmed by two Mozilla Core developers, *open* review is a way to not accept a given patch, or it indicates reviewers are not interested in the proposed patches. We label open reviews as *rejected*. About 39% of instances are labeled as *accepted* for Firefox and 45% are labeled as *accepted* for Mozilla Core. Since we limit the instance numbers, the accepted and rejected patch rates are slightly different than the rates in Figure 3.

For reviewer prediction, we can only use reviewed patches. The reviewer prediction corpus includes 10,124 instances for Firefox, and 57,540 for Mozilla Core. We use real reviewers as the instance labels. Note that Firefox has more than 1,641 reviewers and Mozilla Core has more than 1,804 reviewers.

Tables 5 and 6 show detailed information about the corpus created and used for our experiment.

Table 5: Corpus instances information for predicting acceptances.

Subject	**Period**	**# of accepted (%)**	**# of rejected (%)**	**# of features**
Firefox	Sep 1999 - Apr 2009	8,209(31%)	17,982(69%)	1,481
Mozilla Core	Feb 2006 - May 2009	21,056(35%)	38,946(65%)	1,481

Table 6: Corpus instances information for recommending reviewers.

Subject	**Period**	**# of reviewers**	**# of features**
Firefox	Sep 1999 - Apr 2009	1,641	1,483
Mozilla Core	Oct 1998 - May 2009	1,804	1,483

4.2 Evaluation Measures

Applying our **patch acceptance prediction** model, the prediction can result in four possible outputs: (1) predicting an accepted patch as accepted ($a \to a$); (2) predicting an accepted patch as rejected ($a \to r$); (3) predicting a rejected patch as rejected ($r \to r$); and (4) predicting a rejected patch as accepted ($r \to a$).

We use the number of above outputs to evaluate our models using the standard measures: accuracy, precision, recall, and F-measures [3, 20, 40]. Some of these measures, we can compute for predicting (a) patch acceptance and (b) patch rejection.

Accuracy (patch outcome): The number of correctly predicted patches divided by the total number of patches. This is a good overall measure for the prediction performance.

N_x indicates the number of corresponding outputs x.

$$Accuracy = \frac{N_{a\to a} + N_{r\to r}}{N_{a\to a} + N_{a\to r} + N_{r\to a} + N_{r\to r}}$$

Accuracy is the same for patch acceptance and rejection.

Precision: For patch acceptance, the number of patches correctly predicted as accepted patch $N_{a\to a}$ over the number of all patches predicted as accept.

$$Precision\ for\ patch\ acceptance \quad P(a) = \frac{N_{a\to a}}{N_{a\to a} + N_{r\to a}}$$

For patch rejection, the number of correctly rejected patches $N_{r\to r}$ over the number of all patches predicted as reject.

$$Precision\ for\ patch\ rejection \quad P(a) = \frac{N_{r\to r}}{N_{a\to r} + N_{r\to r}}$$

Recall: For patch acceptance, the number of patches correctly predicted as accept $N_{a\to a}$ over the number of accepted patches.

$$Recall\ for\ patch\ acceptance \quad R(a) = \frac{N_{a\to a}}{N_{a\to a} + N_{a\to r}}$$

For patch rejection, the number of patches correctly predicted as reject $N_{r\to r}$ over the number of rejected patches.

$$Recall\ for\ patch\ rejection \quad R(a) = \frac{N_{r\to r}}{N_{r\to a} + N_{r\to r}}$$

F-measure: A composite measure of precision P and recall R.

$$F\text{-}measure \quad F(x) = \frac{2 * P(x) * R(x)}{P(x) + R(x)}$$

For patch acceptance, x is a; and for patch rejection x is r.

Since **reviewer prediction** is a multi-level prediction model [3], we use accuracy to evaluate a model similar to related work [5, 8]. Our reviewer prediction model predicts the top N suitable reviewers. We consider the top N prediction to be correct if the real reviewer is included in the predicted top N reviewer list.

Accuracy (reviewer prediction): The percentage of predictions with at least one correct reviewer in the top N.

$$Accuracy\ (top\ N) = \frac{Number\ of\ correct\ top\ N\ predictions}{Number\ of\ total\ predictions}$$

5 Evaluation Results

5.1 Predicting Patch Acceptance

To evaluate the patch acceptance prediction model, we use 10-fold cross validation [3], since accepting or rejecting patches are non-time and non-order dependent activities. The corpus shown in Table 5 is used. For the machine learning algorithm, we used the Bayesian Network [3] (the Weka implementation [47]).

Table 7 summarizes the prediction accuracy, precision, and recall of the two projects. The accuracy is around 73% for Firefox and 72% in Mozilla Core. In Mozilla Core, the precision and recall values when prediction patch rejection are more than 78%. When predicting acceptance of patches, the precision and recall values are slightly lower (around 60%), but still substantially higher than random models (which would have precision and recall of 32–35%)

Overall, our patch acceptance prediction model yields a reasonably good accuracy, which is comparable and in some cases higher than previous results on change classification which had average precision/recall of around 60% [20]. Our model particularly predicts *rejectable patches* very accurately (more than 82% of recall and precision for Mozilla Core). Our prediction model does not use historical data such as the number of changes and number of previous bugs, which are often considered the top predictors for fault localization [41]. Shin et al. [41] showed that without historical data, fault prediction models usually yields low accuracy. Considering that we do not use historical data, our prediction results are even more impressive.

It is often possible to improve recall by reducing precision and vice versa, since there are tradeoffs between precision and recall [3]. Figure 9 shows the precision-recall graph for the *rejectable patch* prediction. As shown in Figure 9 (a), we can easily reach to the 100% recall by predicting all instances as *reject*. Then the precision will be around 68%, the rate of *reject* labeled instances in Table 5. Similarly, we can achieve very high precision, 98% as shown in Figure 9 (b) by sacrificing the recall to 20% for Mozilla Core. This precision-recall graph shows that if a developer or reviewer needs a highly precise prediction model, our model can achieve high precision by lowering recall. Such a model can effectively be used to filter patches and reduce the workload of reviewers, only 1 in 50 patches is accidentally rejected.

5.2 Recommending Reviewers

In this section, we evaluate our second model, reviewer prediction. The corpus shown in Table 6 is used. Similar to the patch acceptance prediction model, we use the Bayesian Network for the reviewer prediction model.

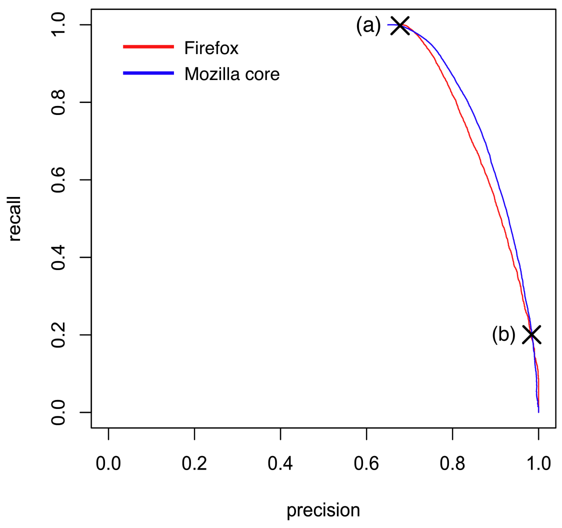

Figure 9: Precision-recall graph.

Table 7: The patch acceptance prediction results with the 10-fold cross validation model.

Subject	**Accuracy**	**Reject**			**Accept**		
		Precision	**Recall**	**F-Score**	**Precision**	**Recall**	**F-Score**
Firefox	73.49	78.7	84.1	81.3	59.1	50.2	54.3
Mozilla Core	72.19	78.1	79.4	78.8	60.7	58.8	59.8

Unlike patch acceptance prediction, review assignment activities are time and order-dependent. Suppose a reviewer, 'Kim' joined the Firefox team in 2008, then he will only appear in the reviewer list after 2008, not before. For this reason, we cannot randomly mix the instances for the reviewer prediction model or use the 10-fold cross validation.

Inspired by [8], we use 10-fold on-line validation. As depicted in Figure 10, we divide instances into 10-folds chronologically. Like the on-line learning approach [3], we train a model using instances in fold 1 (training set) and apply the model to fold 2 (testing set). Then we train a model using instances in fold 1 and 2 (training sets), and apply the model to fold 3 (testing set). We continue this process until we train a model using instances in fold 1 to fold 9 (training sets), and apply the model to fold 10 (testing set).

Table 8 shows the accuracy of each run of our evaluation. For reviewer prediction, we select the top *N* reviewers and measure the prediction accuracy. Overall, the accuracy for the top 1 is around 34%, and the top 5 is around 70% in Firefox. For Mozilla Core, top 1 is around 30% and top 5 around 66%.

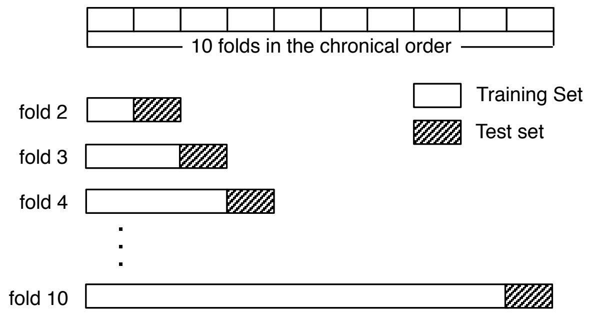

Figure 10: 10 folds on-line validation model inspired by [8].

Table 8: Predicting reviewers.

Subject	**Testing (training on)**	**Accuracy**				
		Top1	**Top2**	**Top3**	**Top4**	**Top5**
Firefox	fold 2 (training on fold 1)	30.79	44.05	52.24	55.94	59.55
	fold 3 (training on fold 1-2)	30.89	39.57	47.27	53.60	57.11
	fold 4 (training on fold 1-3)	36.06	51.16	61.20	67.73	70.85
	fold 5 (training on fold 1-4)	30.89	44.54	50.38	54.38	60.42
	fold 6 (training on fold 1-5)	31.38	46.19	55.75	62.37	67.93
	fold 7 (training on fold 1-6)	32.94	50.68	62.96	72.51	76.31
	fold 8 (training on fold 1-7)	27.38	50.38	68.03	76.60	80.70
	fold 9 (training on fold 1-8)	43.17	65.20	73.29	76.41	77.97
	fold 10 (training on fold 1-9)	42.98	65.05	73.28	76.37	77.92
	average	34.05	50.75	60.48	66.21	69.86
Mozilla Core	fold 2 (training on fold 1)	24.70	36.00	43.37	48.99	51.97
	fold 3 (training on fold 1-2)	23.57	36.28	45.24	52.18	58.00
	fold 4 (training on fold 1-3)	25.41	40.88	54.26	62.65	68.68
	fold 5 (training on fold 1-4)	27.96	44.07	54.69	59.89	63.59
	fold 6 (training on fold 1-5)	26.45	42.05	52.60	59.06	62.86
	fold 7 (training on fold 1-6)	36.97	54.07	64.58	71.77	75.71
	fold 8 (training on fold 1-7)	33.28	51.19	61.04	67.25	71.82
	fold 9 (training on fold 1-8)	34.89	49.98	60.51	67.53	70.92
	fold 10 (training on fold 1-9)	34.32	49.68	60.00	67.25	70.37
	average	29.72	44.91	55.14	61.84	65.99

These results, between 66% and 70% accuracy for the top 5, are superior to the results reported by Bettenburg et al. [8] (55% accuracy for the top 5). Note that there are around one thousand reviewers in Firefox and Mozilla Core, and predicting the top 5 among all reviewers is nontrivial. Considering the above, our prediction accuracy is very high.

One possible explanation of the high accuracy is that we leverage information of the patch file location as features. The developers own a couple of modules and the module owners usually review the corresponding patches [28, 30]. The file location and reviewers are highly correlated. We discuss this more in Section 5.3

Our prediction model guides developers to find suitable reviewers with high accuracy. As Figure 7 and Figure 8 show, the patches with reviewer recommended have a better chance to be accepted than patches without reviewer recommended. It is important to recommend a reviewer for a patch and developers can leverage our tool to find suitable reviewers.

5.3 Feature Sensitivity

We showed that our prediction models yield a reasonably good accuracy. Then questions arose: how does the model work and which features are important? To identify informative features, we performed feature sensitivity analysis [3] for features used in the patch acceptance prediction model.

Using the chi-square measure, which is commonly used for feature selection [3], each feature is ranked to determine how correlated it is to the accepted and not-accepted patches. Then we normalized the chi-square values of each features to select and compare the top 20 features as shown in Figures 11 and 12.

This provided insight into which features are the most informative for the patch acceptance prediction model.

It is noticeable that the selected top features of two projects, Firefox and Mozilla Core are almost the same. The top 5 features of both projects are exactly the same even though their orders are slightly

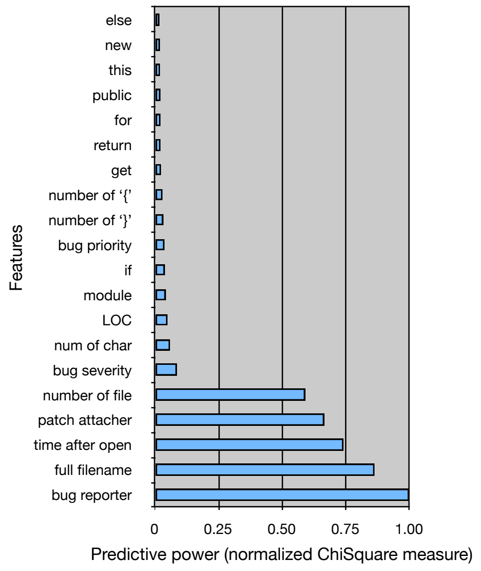

Figure 11: Feature sensitivity of Firefox.

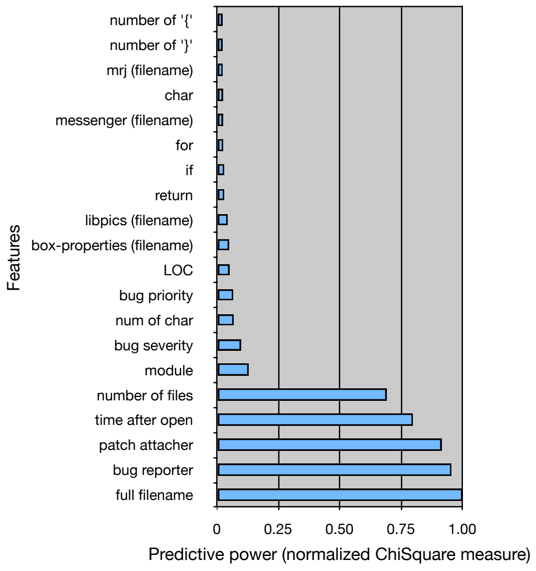

Figure 12: Feature sensitivity of Mozilla Core.

different.

Developers including *reporter* (who reported the bug) and *patch writer* are ranked as top features. This result is consistent with other research results [20, 48], which indicates that developers or having a developer relation is an important predictor for fault localization.

The *time after open* is also an important feature. Intuitively, if a patch is rapidly written and posted for a bug, probably the bug is important, and the importance may affect the patch review outcomes. Similarly, the *bug severity* also affects the review outcomes.

The traditional basic complexity metrics such as the *number of characters*, the *number of files*, the *mumber of blocks ({)* and the *line of codes* (LOC) are ranked top.

Interestingly, the program control related keywords, *if (keyword)* and *for (keyword)* are identified as important features. Obviously adding these features add more complexity and risk into the source code [20, 48].

Some location related features such as *full filename*, *libpics (filename)* and *mrj (filename)* are ranked as top features. This also confirms previous research results [21], which shows the location is an important feature for the localization.

However, this analysis is based on the individual feature correlation. When Bayesian Network uses many features all together for training and prediction, the important features in the model may differ.

6 Threats to Validity

We identify the following threats to the validity.

Systems examined might not be representative. Review process of two systems were examined in this paper. Since we intentionally chose systems for which we could extract high quality review process information, we might have a project selection bias.

Systems are all open source projects. All systems examined in this paper are developed as open source projects. Usually commercial software developers have quality assurance (QA) team supports. The QA team classifies bug reports, inspects patches, assigns reviewers. They might have a small set of reviewers and assign patches quickly for reviews. However, our patch acceptance prediction model is useful for both, open source and commercial software developers.

Reviewer retirement information is hidden. It is possible that some of the reviewers are retired and do not review patches any more. However, our reviewer prediction corpus may include activities of retired reviewers. This could affect reviewer prediction prediction accuracy.

7 Related Work

We can distinguish between two types of code reviews: For *formal* code reviews, developers and reviewers typically follow a carefully designed process, which involves several phases and many meetings [14]. In contrast, *lightweight* code reviews involve less overhead and are typically conducted as a part of the normal development process [12]. The focus of this paper is on lightweight code reviews, more specifically on patch reviews in open-source projects.

Previous work on patch reviews in open source by Rigby et al. [38] compared two peer review techniques, review-then-commit and commit-then-review, used in the Apache project. Motivated by earlier research of Porter et al. [33], the study by Rigby et al. focused on frequency of reviews, level of participation, size of patches, and the review interval. They concluded that early and frequent reviews of small and complete contributions, conducted by a small core group of self-selected experts leads to an efficient and effective peer review technique. Nurolahzade et al. [31] conducted a qualitative study of 310 patches from 66 bug reports of the Mozilla and Firefox projects. They identified several review patterns such as patchy patcher, new comer, merciful reviewer, and doubtful reviewers. Weissgerber et al. [46] observed that small patches are more likely to be accepted than longer patches. Our study complements these works with a qualitative study of the patch process in Mozilla Core. Unlike Rigby et al. [38] and Weissgerber et al. [46] who mined email discussions with techniques introduced by Bird et al. [9], our study is based on flags in bug reports. This has the advantages that patches and their review outcomes can be identified more reliably. In addition, we propose and evaluate tools to reduce the overhead when doing code reviews.

Rigby and German [37] described and compared the reviewing process of GCC, Linux, Mozilla, and Apache. They observed several commonalities: every project followed a coding standard and required patches to be independent, complete, and small. The review process of Apache and Mozilla was also described in a study by Mockus et al. [29], which was replicated by Dinh-Trong and Bieman [13] for the FreeBSD project. Several papers analyzed the level of participation of open-source developers in code review [6, 19, 24] or shared their experiences of participating in open-source code review [25, 36]. Compared to these works, our study quantifies the reviewing process in Mozilla, characterizes which patches get accepted, and proposes two novel prediction models to reduce the overhead of code review.

Several papers recommended and built better tool support for software inspections, but mostly focused at the actual code review and the inspection meeting. Improvements include annotations of source code and cooperative discussion of comments [10], a distribute collaborative environment for multiple process models [15], and tool support for asynchronous code inspections [27, 42], organization of inspection meetings [16], and analysis of the semantics of the program being analyzed [4]. Sutherland and Venolia observed that information created during code reviews is often not retained and called for tools to archive and build on this [43]. To the best of our knowledge, we are the first to propose tools that recommend reviewers and acceptance of patches, which helps to further reduce the overhead of code reviews and inspections. As pointed out by Votta et al. [44] 20% of the time is often wasted before it comes to the actual code review. Our work helps to reduce the time that it takes to find the appropriate expert to review a code change.

Work on code inspections in general includes the following. Ackerman and colleagues [1, 2] and Sauer et al. [39] showed that software inspections are an effective way to reduce defects and increase software quality. Perry et al. [32] showed evidence that manual inspections without meetings can be as effective as inspections with meetings, which helps to reduce the inspection interval. Other studies by Porter and colleagues aimed at understanding how developer activities [34] and the number of reviewers and teams [35] can affect the effectiveness of code inspections. Mantyla and Lassenius [26] analyzed what type of defects are discovered during code review. They found that 75% of defects did not have any effect on the visible functionality of the software, they rather improved code quality. This might explain why our patch acceptance prediction outperforms existing approaches for change classification [20]. For a comprehensive and complete overview on the research on code reviews and code inspections we refer to a survey by Laitenberger [22].

8 CONCLUSIONS

Code review is a powerful technique to decrease the number of defects and to increase the overall code quality of a software [1, 2]. However, these benefits come at a cost. Organizing code reviews is a non-trivial and time-consuming task. For code inspections, 20% of the review interval is spent on scheduling meetings [44]. While in open source, patches are reviewed asynchronously without any meetings, we still found that a large amount of time is spent on finding the appropriate person to review a patch. Review requests that do not mention a specific reviewer are less likely to be accepted.

To reduce the cost of code reviews, we proposed two novel recommendation systems in this paper.

1. *Predicting the review outcome.* This recommendation helps code reviewers to prioritize patches and can provide patch writers with early feedback on their code.

2. *Recommending patch reviewers.* This helps patch writer to direct patches to the right people and relieves code reviewers from monitoring unassigned patches.

We expect such tools to reduce the time spent on organizing and finding reviewers, which will leave developers with more time to conduct actual code reviews. While we tested our prediction models only on open source, we are confident that they are helpful in an industrial context as well. As pointed out by LaToza et al. [23] “finding the right person to review a change” is a problem in industry as well. In future work, we will replicate our study in industry and run user studies to further demonstrate and improve on the usefulness of our technique.

9 Acknowledgments

Our thanks to the two Mozilla Developers, Channy Yun and Gary Kong for their discussion and feedback on this research.

References

[1] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski. Software inspections: An effective verification process. *IEEE Softw.*, 6(3):31–36, 1989.

[2] A. F. Ackerman, P. J. Fowler, and R. G. Ebenau. Software inspections and the industrial production of software. In *Proc. of a symposium on Software validation: inspection-testing-verification-alternatives*, pages 13–40, 1984.

[3] E. Alpaydin. *Introduction to Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2004.

[4] P. Anderson, T. Reps, T. Teitelbaum, and M. Zarins. Tool support for fine-grained software inspection. *IEEE Softw.*, 20(4):42–50, 2003.

[5] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE ’06: Proceedings of the 28th international conference on Software engineering*, pages 361–370, 2006.

[6] J. Asundi and R. Jayant. Patch review processes in open source software development communities: A comparative case study. In *HICSS ’07: Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, page 166c, 2007.

[7] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Using findbugs on production software. In *OOPSLA ’07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 805–806, 2007.

[8] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Duplicate bug reports considered harmful... really? In *Proceedings of the 24th IEEE International Conference on Software Maintenance*, September 2008.

[9] C. Bird, A. Gourley, and P. Devanbu. Detecting patch submission and acceptance in oss projects. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 26, 2007.

[10] L. Brothers, V. Sembugamoorthy, and M. Muller. Icicle: groupware for code inspection. In *CSCW '90: Proceedings of the 1990 ACM conference on Computer-supported cooperative work*, pages 169–181, 1990.

[11] Bugzilla bug tracking system. `http://www.bugzilla.org/`. Last accessed 2009-09-02.

[12] J. Cohen, editor. *Best Kept Secrets of Peer Code Review*. Smart Bear Inc., Austin, TX, 2006.

[13] T. T. Dinh-Trong and J. M. Bieman. The freebsd project: A replication case study of open source development. *IEEE Trans. Softw. Eng.*, 31(6):481–494, 2005.

[14] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 38(2-3):258–287, 1999.

[15] J. Gintell, J. Arnold, M. Houde, J. Kruszelnicki, R. McKenney, and G. Memmi. Scrutiny: A collaborative inspection and review system. In *ESEC '93: Proceedings of the 4th European Software Engineering Conference on Software Engineering*, pages 344–360, 1993.

[16] P. Grünbacher, M. Halling, and S. Biffl. An empirical study on groupware support for software inspection meetings. In *ASE '03: Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pages 4–11, 2003.

[17] G. Jeong, S. Kim, and T. Zimmermann. Improving bug triage with bug tossing graphs. In *ESEC/FSE '09: Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 111–120, 2009.

[18] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 96–105, 2007.

[19] J. P. Johnson. Collaboration, peer review and open source software. *Information Economics and Policy*, 18(4):477–497, November 2006.

[20] S. Kim, E. J. W. Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Software Eng.*, 34(2):181–196, 2008.

[21] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 489–498, Washington, DC, USA, 2007.

[22] O. Laitenberger. A survey of software inspection technologies. In *Handbook on Software Engineering and Knowledge Engineering*, volume 2, pages 517–555. 2002.

[23] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 492–501, New York, NY, USA, 2006. ACM.

[24] G. K. Lee and R. E. Cole. From a firm-based to a community-based model of knowledge creation: The case of the linux kernel development. *Organization Science*, 14(6):633–649, 2003.

[25] S. Lussier. New tricks: How open source changed the way my team works. *IEEE Softw.*, 21(1):68–72, 2004.

[26] M. V. Mantyla and C. Lassenius. What types of defects are really discovered in code reviews? *IEEE Trans. Softw. Eng.*, 35(3):430–448, 2009.

[27] V. Mashayekhi, C. Feulner, and J. Riedl. Cais: collaborative asynchronous inspection of software. In *SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 21–34, 1994.

[28] A. Mockus. Succession: Measuring transfer of code and developer productivity. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 67–77, 2009.

[29] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, 2002.

[30] Mozilla developer guide. `https://developer.mozilla.org/En/Developer_Guide/`. Last accessed 2009-09-02.

[31] M. Nurolahzade, S. M. Nasehi, S. H. Khandkar, and S. Rawal. The role of patch review in software evolution: an analysis of the mozilla firefox. In *IWPSE-Evol '09: Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, pages 9–18, 2009.

[32] D. E. Perry, A. Porter, M. W. Wade, L. G. Votta, and J. Perpich. Reducing inspection interval in large-scale software development. *IEEE Trans. Softw. Eng.*, 28(7):695–705, 2002.

[33] A. Porter, H. Siy, A. Mockus, and L. Votta. Understanding the sources of variation in software inspections. *ACM Trans. Softw. Eng. Methodol.*, 7(1):41–79, 1998.

[34] A. A. Porter, H. P. Siy, C. A. Toman, and L. G. Votta. An experiment to assess the cost-benefits of code inspections in large scale software development. *IEEE Trans. Softw. Eng.*, 23(6):329–346, 1997.

[35] A. A. Porter, H. P. Siy, and L. G. Votta, Jr. Understanding the effects of developer activities on inspection interval. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 128–138, 1997.

[36] E. S. Raymond. *The Cathedral and the Bazaar*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1999.

[37] P. C. Rigby and D. M. German. A preliminary examination of code review processes in open source projects. Technical Report DCS-305-IR, University of Victoria, January 2006.

[38] P. C. Rigby, D. M. German, and M.-A. Storey. Open source software peer review practices: a case study of the apache server. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 541–550, 2008.

[39] C. Sauer, D. R. Jeffery, L. Land, and P. Yetton. The effectiveness of software development technical reviews: A behaviorally motivated program of research. *IEEE Trans. Softw. Eng.*, 26(1):1–14, 2000.

[40] S. Scott and S. Matwin. Feature engineering for text classification. In *Proc. 16th International Conf. on Machine Learning*, pages 379–388. Morgan Kaufmann, San Francisco, CA, 1999.

[41] Y. Shin, R. Bell, T. Ostrand, and E. Weyuker. Does calling structure information improve the accuracy of fault prediction? *Mining Software Repositories, International Workshop on*, 0:61–70, 2009.

[42] M. Stein, J. Riedl, S. J. Harner, and V. Mashayekhi. A case study of distributed, asynchronous software inspection. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 107–117, 1997.

[43] A. Sutherland and G. Venolia. Can peer code reviews be exploited for later information needs? In *ICSE '09: Companion Volume of the International Conference on Software Engineering*, pages 259–262, 2009.

[44] L. G. Votta, Jr. Does every inspection need a meeting? In *SIGSOFT '93: Proceedings of the 1st ACM SIGSOFT symposium on Foundations of software engineering*, pages 107–114, 1993.

[45] S. Weerahandi. *Exact Statistical Methods for Data Analysis (Springer Series in Statistics) (Paperback)*. Springer (October 17, 2003), 2003.

[46] P. Weißgerber, D. Neu, and S. Diehl. Small patches get in! In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 67–76, 2008.

[47] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 2nd edition, 2005.

[48] T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *ICSE '08: Proceedings of the International Conference on Software Engineering*, pages 531–540, 2008.