SARATHI: Characterization Study on Regression Bugs and Identification of Regression Bug Inducing Changes: A Case-Study on Google Chromium Project

Student Name: MANISHA KHATTAR

IIIT-D-MTech-CS-DE-13-042
Oct 13, 2014

Indraprastha Institute of Information Technology
New Delhi

Thesis Committee
Prof. Ashish Sureka (Chair)
Prof. Rahul Purandre
Dr. Atul Kumar

Submitted in partial fulfillment of the requirements
for the Degree of M.Tech. in Computer Science,
with specialization in Data Engineering

Keywords: Mining Software Repositories, Automated Software Engineering, Information Retrieval,Recommendation Engine

Certificate

This is to certify that the thesis titled **"SARATHI: Characterization Study on Regression Bugs and Identification of Regression Bug Inducing Changes: A Case-Study on Google Chromium Project"** submitted by **Manisha Khattar** for the partial fulfillment of the requirements for the degree of *Master of Technology* in *Computer Science & Engineering* is a record of the bonafide work carried out by her under my guidance and supervision in the Data Engineering group at Indraprastha Institute of Information Technology, Delhi. This work has not been submitted anywhere else for the reward of any other degree.

Prof. Ashish Sureka
Indraprastha Institute of Information Technology, New Delhi

Abstract

Software regression bugs are defined as defects which occur, when a previously working software feature or functionality stops behaving as intended. One of the reasons for regression bugs is code changes or system patching which leads to unexpected side effects. Running a test suite to validate the new features getting added and faults introduced in previously working code, after every change is impractical. As a result, by the time an issue is identified and reported a lot of changes are made to the source code, which makes it very difficult for the developers to find the regression bug inducing change. A bug fixer has to go through several suspected revisions before being able to locate actual regression causing revision. Thus finding regression bug inducing change is a non trivial and challenging problem.

We first conduct an in-depth characterization study of regression bugs by mining issue tracking system dataset belonging to a large and complex software system i.e. Google Chromium Project showing: priority, number of comments, closure-time distribution and opening and closing trend analysis and quality of bug fixing process for regression bugs in comparison to crash, performance and security bugs. We also define a metric which computes the quality of bug fixing process for one type of bug report in comparison to the quality of bug fixing process for other types of bug reports. We present our results thus obtained using several visualisations. We then describe our character n-gram based solution approach for finding the regression bug inducing change. We mine existing issue reports and log messages of regression bugs for establishing ground truth dataset for our model using several heuristics. We extract several features of regression causing revisions from training dataset that then are used in building the model for predicting the regression inducing revision. 78% of regression issues are reported within 20 days after the revision causing them was committed and almost 3073 revisions are made in around 20 days before the reporting timestamp of the issue. So a bug fixer will have to look back and analyze several hundreds or thousands of revisions for finding the culprit. Our aproach provides a bug fixer with Top K revisions(in decreasind order of similarity score) that are suspected of having regressed an issue. We implemented the proposed IR model and evaluated its performance on our test dataset.

We demonstrate the effectiveness of character n-gram based textual features for identifying regression causing revision as there is high textual similarity between title,description of issue reports and log message of regression inducing revision and tite,component of regressed issue and paths of modified files its corresponding regression inducing revision. We conduct a series of experiments to validate and demonstrate effectiveness of our proposed approach. We find that for 60% of issues the actual regression causing revision was part of Top K(K=75) recommended revisions.

Acknowledgments

I take this opportunity to express my gratitude and deep regards to my guide Prof. Ashish Sureka for his exemplary guidance, monitoring and constant encouragement throughout the course of this thesis. His guidance helped me in all the time of research and writing of this thesis. Without his guidance and persistent help this thesis would not have been possible. I would also like to thank my fellow mate Yash Lamba for his insightful comments, suggestions and encouragement and helping me with technical stuff. Last but not the least I would like to thank my parents and lovely brother for their constant support and encouragement and trust in me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research Motivation and Aim

Software regression bugs are defined as defects which occur, when a previously working software feature or functionality stops behaving as intended. One of the reasons for regression bugs is code changes or system patching which leads to unexpected side effects. Consider a source code repository S consisting of several files. Let Fp be a software functionality of S which is working correctly. A developer D enhances S to S0 to implement another feature Fq by making a code change P (patch). A change can have side effects and it is possible that P breaks Fp. An issue ticket reporting a defect in Fp (in S0) which is a feature that was working earlier, is called as a Regression Bug. Regression testing is a technique consisting of creating a test suite to validate both the new features getting added as the system evolves, and to detect if any faults are introduced in previously working and tested code as a result of a source code change. Conducting regression testing after every change or some changes to the code is a solution to immediately detect source code changes with side effects and thus prevent regression bugs from creeping into the system. In small systems with less number of changes being done to the system with time, it is still feasible to perform regression testing frequently and thus prevent regression bugs. However, regression testing is an expensive process because it becomes very time consuming to run a large number of test-cases or an entire test-suite, covering all the functionalities of a large and complex software. The number of test-suites grows as the system evolves and grows, due to which it becomes impractical to create test-cases for every functionality, and execute it after a change is made to the source-code. Regression testing minimization, selection and prioritization is an area that has attracted several researcher's attention [1].

Figure 1.1 indicates the fact that number of features to be tested during regression testing increases significantly as more features are added into the system. New feature testing involves testing only the current feature fixed or added whereas regression testing involves testing all the previously working features. Thus the time and cost involved for regression testing increases significantly in large systems like Google chromium, Mozilla, Apache etc which have several millions lines of code and changes to such large software systems are made frequently.

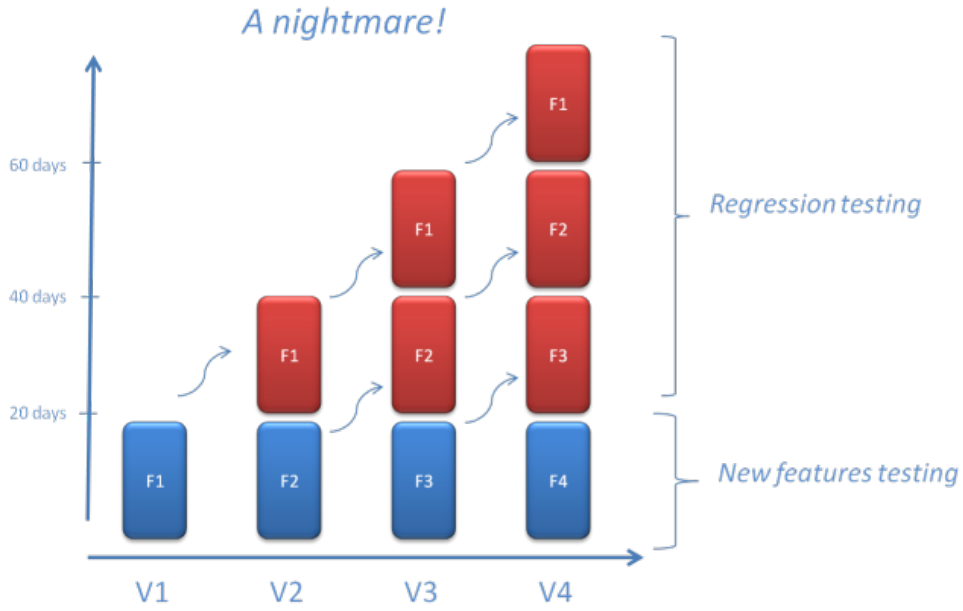


Figure 1.1: Figure indicating increase in complexity of regression testing with increase in size of software

In many scenarios, where regression testing after every change is not applied like the ones described above, regression bugs are injected into the system due to buggy changes which are later reported by testers or users. Regression bugs are considered to be inevitable and truism in large and complex software systems [2]. Identification of the source code change which caused the regression bug is a fundamental problem faced by a bug fixer or owner who is assigned a regression bug. Locating the source code change which caused the regression bug is a non-trivial problem [2] [3] [4]. Manual regression hunting process can be extremely time consuming and error prone. Hence this calls for a need to automate process of identifying regression bug inducing change. Automating identification of regression bug inducing change will help developersbug fixers *Issue Tracking System (ITS)* are applications to track and manage issue reports and to archive bug or feature enhancement requests. *Version Control System (VCS)* are source code control systems recording the author, timestamp, commit message and modified files. The broad research motivation of the study presented in this work is to investigate mining issue tracking system and version control system based solutions to develop a recommendation engine (called as *Sarathi*) to assist a bug fixer in locating a regression bug inducing change. Following are the specific research aims of the work presented here :

1. To conduct an in-depth characterization study of regression bugs by mining issue tracking system dataset belonging to a large and complex software system.

2. To investigate bug report and source-code commit meta-data and content based mining solution, to develop a predictive model for identifying a regression bug inducing change. To validate the proposed model and demonstrate effectiveness of the proposed approach on real-world dataset belonging to a large and complex software system.

Our proposed predictive model is more applicable and beneficial for large systems like Mozilla,

Apache, Eclipse, linux with several millions lines of code and large user base. These large software systems have several millions lines of code and frequent updates and revisions are being made to such systems. Thus the probability of an update or fix having regressed a bug are large. Also it is difficult to perform regression testing frequently in such systems. Thus bugs do creep in there is frequent bug reporting. Developers or bug fixers do frequently come across regression bug reports and need to fix them soon. As customers are generally not ok if a previously working functionality stops working as intended. Therefore our approach provides bug fixers with Top K suspected revisions which might have caused the bug. Bug fixers can then go through only the top K suggested bug reports to find the culprit revision instead of going through several thousands of revisions. Therefore in the world of Software as Service where updates to the software are continuous and not just release based, such an approach is useful.

1.2 Related Work & Contributions

In this Section, we discuss closely related work (to the research presented here) and present the novel research contribution our work in context to the existing work. We organize closely related work into following three lines of research:

1.2.1 Regression Bug Hunting and Location

Bowen et al. present a method to automate the process of identifying which code addition or patch created the regression (called as regression hunting) [2]. They implement a solution in Python to test the Linux Kernel using the Linux Test Project [2]. Johnson et al. describe their experiences in automating regression hunts for the GCC and Linux kernel projects [3]. They provide a solution for automated regression hunts for the Linux kernel based on patch sets rather than dates [3]. Yorav et al. present a tool called as CodePsychologist which assists a programmer to locate source code segments that caused a given regression bug [4]. They define several heuristics based on textual similarity analysis between text from the test-cases and code-lines and check-in comments to select the lines most likely to be the cause of the error [4].

1.2.2 Regression Bug Prediction

Tarvo et al. propose a statistical model for predicting software regressions. They investigate the applicability of software metrics such as type and size of the change, number of affected components, dependency metrics, developer's experience and code metrics of the affected components to predict risk of regression for a code change [5]. In another study, Tarvo et al. present a tool called as Binary Change Tracer (BCT) which collects data on software projects and helps predict regressions. They conduct a study on Microsoft Windows operating system dataset and build a statistical model (based on fix and code metrics) that predicts the risk of regression for

each fix [6]. Mockus et al. develop a predictive model to predict the probability that a change to software will cause a failure. The model uses predictors (such as size in lines of code added, deleted, and unmodified, diffusion of the change and its component sub-changes as well as measures based on developer experience) based on the properties of a change itself [7]. Shihab et al. conduct an industrial study on the risk of software changes [8]. Sunghun et al. present an approach to classify a software change into clean or buggy [9]. Marco DAmbros, Michele Lanza and Romain Robbes [10] provides extensive comparison of several bug prediction approaches. They described different categories of bug prediction approaches.Bernstein Ekanayake and Pinzger [11] emphasize use of temporal features and non-linear model to uncover some of the hidden interrelationships between the features and the defects.

1.2.3 Characterization Study on Bug Types

Lal et al. present a study of mining issue tracking system to compare and contrast seven different types of bug reports: crash, regression, security, clean up, polish, performance and usability [12]. They compare different bug report types based on statistics such as close-time, number of stars, number of comments, discriminatory and frequent words for each class, entropy across reporters, entropy across component, opening and closing trend, continuity and debugging efficiency performance characteristics [12]. Zaman et al. conduct a case-study on Firefox project and study two different types of bugs: performance and security [13]. Zaman et al. conduct a qualitative study on performance bugs [14]. Gegick et al. perform an industrial study on identification of security bug reports via text mining [15]. Khomh et al. study crash bug types [16] and Twidale at al. study usability issues [17].

In context to existing work, the study presented in this paper makes the following novel contributions:

1. An in-depth characterization study of regression bugs on Google Chromium dataset showing: priority, number of comments and closure-time distribution for regression bugs in comparison to crash, performance and security bugs. The characterization study also includes opening and closing trend analysis and quality of bug fixing process for regression bugs in comparison to other types of bugs.

2. A character n-gram based information retrieval model for predicting a bug inducing change for given regression bug report. The predictive model is based on four features: character n-gram based textual similarity between bug report title and revision log message, bug report description and revision log message, bug report title and file path names in the revision and time difference between the bug report and source-code change transaction.

Table 1.1: Experimental Dataset details

S.No	**Issue Tracking System**	**Chromium**
1	Start Issue ID	2
2	End Issue ID	388954
3	Reporting Date Start Issue	8/30/2008 4:00:21 PM
4	Reporting Date End Issue	6/26/2014 12:16:25 AM
5	Number of Issue Reports Downloaded	295202
6	Number of Issue Reports unable to Download	93571
7	Number of Labelled Bug Reports	39658
S.No	**Version Control System**	**Chromium**
1	Start Revision ID	0
2	End Revision ID	279885
3	Reporting Date Start Revision	7/25/2008 7:49:01 PM
4	Reporting Date End Revision	6/26/2014 1:27:51 AM

1.3 Experimental Dataset

We conduct our study and experiments on large real-world publicly available dataset of *Google Chromium Project*.Table 1.1 gives details about our experimental dataset. We download bug reports from the Google Chromium Issue Tracking System[1] using the feed[2] provided by the Chromium project. We download a total of 295202 issue reports from Issue ID 2 (8/30/2008 4:00:21 PM) to 388954 (6/26/2014 12:16:25 AM). In addition to *ITS* data, we also download the data for all versions of the chromium project from the Version Control System[3], by scrapping the web page for each revision, right from the start to revision 279885 (6/26/2014 1:27:51 AM).

Futher work in this report is divided in six chapters. Next chapter i.e chapter 2 describes the characterization study that we conducted on regression bugs in Google Chromium Project. In Chapter 3 we describe the procedure followed for ground truth dataset establishment and the heuristics used. In chapter 4 we describe our proposed approach for regression bug prediction. In chapter 5 we discuss experimental evaluation and Validation followed by limitations and future work discussed in chapter 6. In the last chapter we conclude our work.

[1] https://code.google.com/p/chromium/issues/

[2] https://code.google.com/feeds/issues/p/chromium/issues/full/

[3] http://src.chromium.org/viewvc/chrome?revision=¡revisionID¿ &view=revision

Chapter 2

Characterization Study

In this chapter analysis on the Google Chromium Issue Tracking System and comparison/contrast among the various different types of bugs is presented to try to understand how regression bugs differentiate themselves from the other bug types.
Out of 295202 issues downloaded from Chromium ITS system 39658 i.e. 13:43% of the bug reports have at least one of the eight bug-type labels (crash, clean-up, performance, polish, regression, usability, security and stability).These 39658 bugs include both open and closed bugs. Table 2.1 shows the percentage distribution of the eight different types of bug reports. Table 2.1 reveals that 51.09% of the labelled bug reports are regression bugs (the focus of this paper). Table 2.1 indicates that the number of labelled Stability and Usability issue reports are less than 110. This is a clear cut indication of the regularity of regression bugs. Among the bugs that are labelled, regression bugs clearly have the majority.

Table 2.1: Number and Percentage of 8 Bug Report Types in the Experimental Dataset (Labelled Bug Reports)

	Bug Type	**Num Bugs**	**Percentage**		**Bug Type**	**Num Bugs**	**Percentage**
1	Crash	11281	28.4%	5	Regression	20263	51.09%
2	Clean Up	459	01.16%	6	Usability	56	00%
3	Perf.	3444	08.68%	7	Security	5920	14.9%
4	Polish	877	02.21%	8	Stability	109	00%

2.1 Bug Priority

Whenever a bug is reported, a developer assigns a priority to the bug. The priority of the bug is a good reflection of the importance of fixing the bug. There are 4 priority levels in Google Chromium project[1]. An issue can only have one priority value: 0 is most urgent and 3 is least urgent. Figure 2.1 shows a pie-of-pie chart which consists of a main pie-chart and a sub pie-

[1]http://www.chromium.org/for-testers/bug-reporting-guidelines/chromium-bug-labels

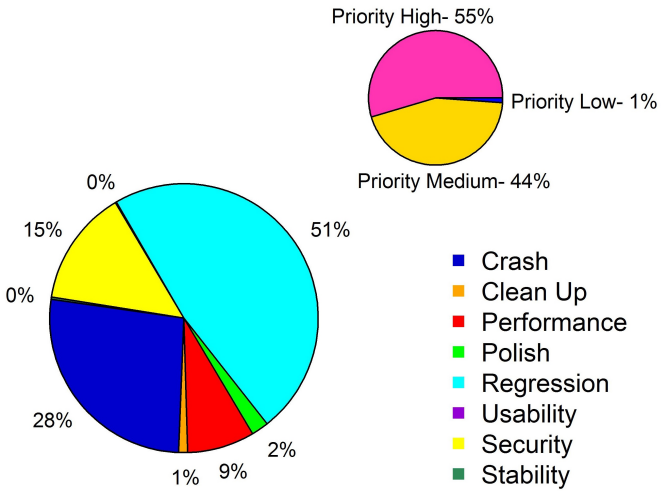


Figure 2.1: *Large Pie* (distribution of 8 bug report types in the experimental dataset), *Small Pie* (distribution of regression bugs across 4 priority types)

chart. The sub pie-chart separates the regression bug slice from the main pie-chart and displays the issue report priority distribution as an additional pie-chart. Figure 2.1 reveals that 55% of the regression bugs are high priority (0 [2.8%] and 1 [51.7%]) bugs. Figure 2.1 is also indicative of the importance of regression bugs. Although, all types of bugs are unwanted, regression bugs in particular are a real nightmare for any developer, since it leads to the undoing of an correctly functioning feature. Hence, unsurprisingly regression bugs lie at the top of the priority list of the developers and bug fixers.

2.2 Number of Comments

Each bug report allows developers to comment on the issue report. The developers post comments on the threaded discussion forum of the ITS to discuss the bug and the possible reasons for the bug. The discussions help the developers to arrive at a solution to the bug. Figure 2.2 displays a violin plot (which is a combination of a box plot and a kernel density plot) of number of comments for four types of bug reports. We consider only fixed (closed) bug reports as the number of comments for open bug reports may increase after our snapshot of the dataset. The number of comments posted in response to a bug report serves as a proxy for popularity, user interest and amount of clarification and discussion [18]. The vertical axis of the violin plot reveals that the number of comments has a range of 0 to 876. The median values of number of comments for crash, performance, regression and security bugs are 12, 9, 13 and 20 respectively (security bugs have the highest median). The minimum, Q1, Q3 and maximum value of number of comments for regression bugs are 0, 8, 19 and 876 respectively (the thick black bar represents the interquartile range). The violin plot provides a comparison of the distribution of number of comments for the four type of bug reports. Figure 2.2 reveals that broadly the shape of the distribution is the same for the four types of bug reports. The violin is most thicker

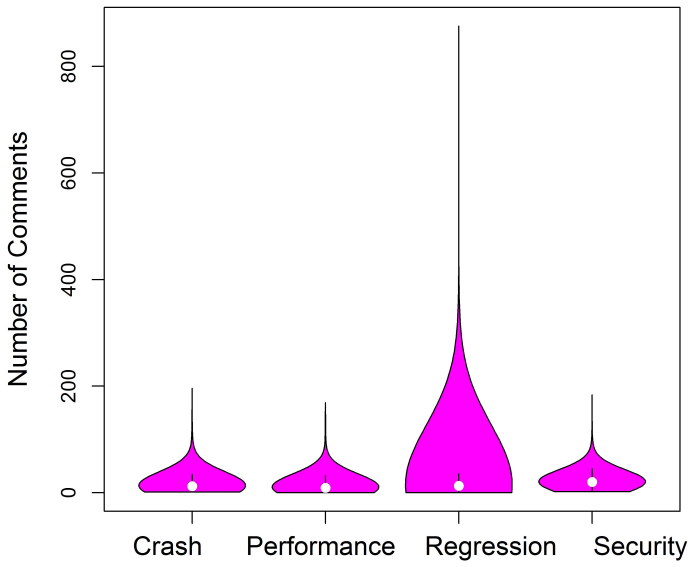


Figure 2.2: Violin plot of number of comments for Crash, Performance, Regression and Security bugs

Table 2.2: The Five-Number Data Summary and Mean Value for the Boxplot in Figure 2.3

Type	**Min.**	**Q1**	**Median**	**Mean**	**Q3**	**Max.**
Crash	0.00	3.00	12.00	44.51	39	1294
Perf.	0.00	2.00	12.00	54.31	52	1685
Regr.	0.00	2.00	8.00	26.84	23	1272
Security	0.00	2.00	8.00	42.84	35	1420

(highest probability) at 9, 3, 8 and 18 for the crash, performance, regression and security bugs respectively.

2.3 Closure Time

Closure Time is the time taken to close an opened issue in the issue tracking system. Our objective is to understand the spread of closure time of various types of bug reports and to get an indication of the data's symmetry and skewness. Figure 2.3 shows four boxplots displaying the five-number data summary (median, upper and lower quartiles, minimum and maximum data values) for the closure time of four different types of bug reports (crash, performance, regression and security). Table 2.2 shows the exact values plotted in Figure 2.3. While normally the boxplot show outliers, we mention the maximum value in Table 2.2 and not in Figure 2.3 due to wide variability between the minimum and maximum value. As shown in Table 2.2, 50% of the crash and performance bugs are closed within 12 days and 50% of the regression and security bugs are closed within 8 days. The mean is greater than the median for all four type of bug

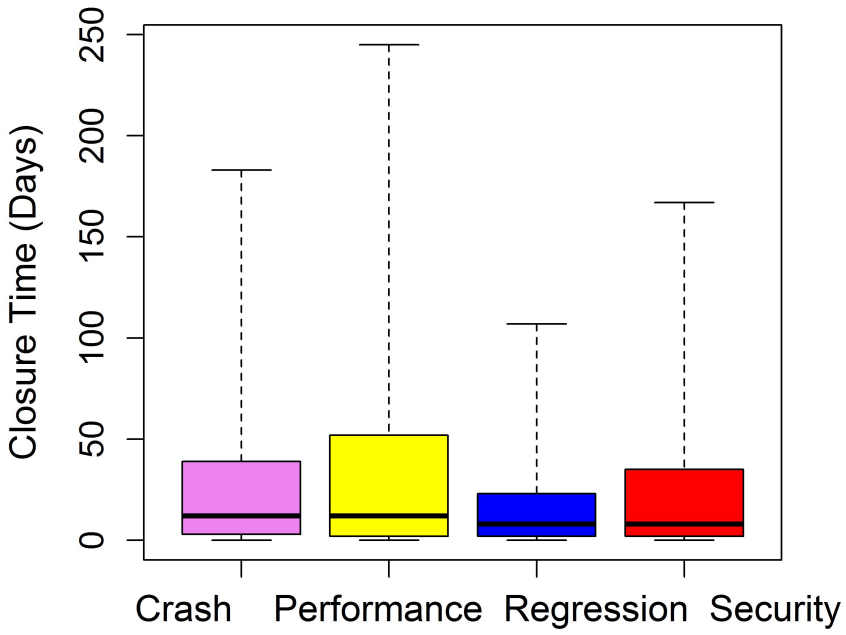


Figure 2.3: Boxplot for the closure time (days) for crash, performance, regression and security bugs

Table 2.3: The Five-Number Data Summary and Mean Value for Stars for the Four Types of Bugs

Type	Min.	Q1	Median	Mean	Q3	Max.
Crash	0	1	2	2.93	3	224
Perf.	0	1	1	3.112	3	185
Regr.	0	1	2	3.956	4	703
Security	0	1	1	1.335	1	78

reports and hence the distribution is shifted towards the right. Since the distribution is skewed towards the right (which is apparent from the visual inspection of the boxplot), most of the values (50%) are small but there are a few exceptionally large ones. The few large ones impact the mean and pull it to the right as a result of which the mean is greater than the median.

2.4 Number of Stars

For any bug report, interested users are allowed to star an issue. A star is similar to a bookmark. A user who stars an issue will be informed about the progress made in the issue. The number of stars on a bug report indicate the number of people who are interested in that issue. Table 2.3 shows that the median values of number of stars for crash, performance, regression and security bugs are 2.93, 3.11, 3.95 and 1.33 respectively. Experimental results reveal that regression bugs have the highest median.

2.5 Bug Opening and Closing Trends

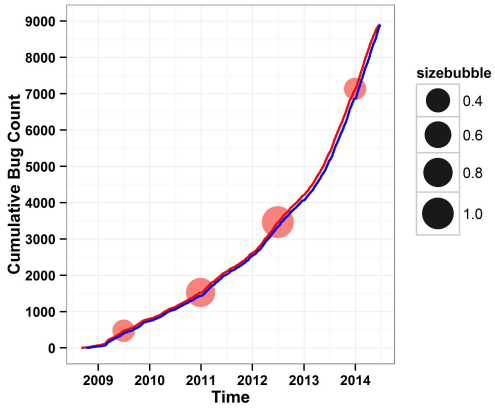


Figure 2.4: Line and bubble chart showing the regression bug opening and closing trend

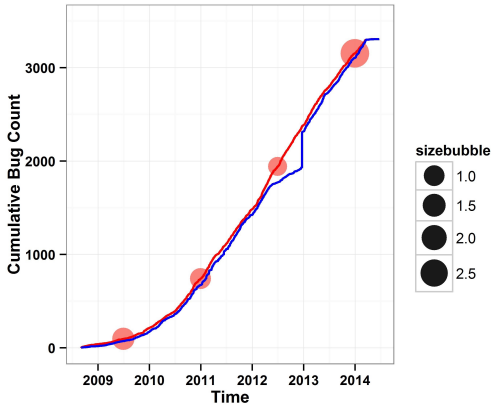


Figure 2.5: Line and bubble chart showing the security bug opening and closing trend

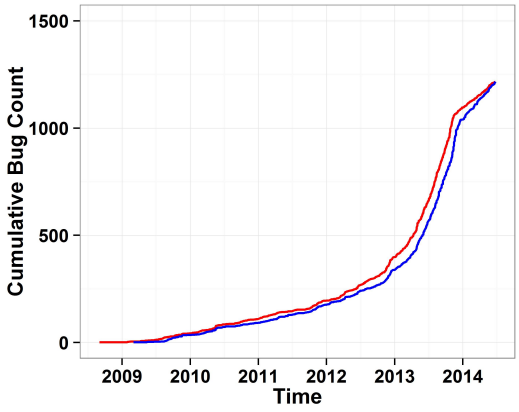


Figure 2.6: Line chart showing the performance bug opening and closing trend

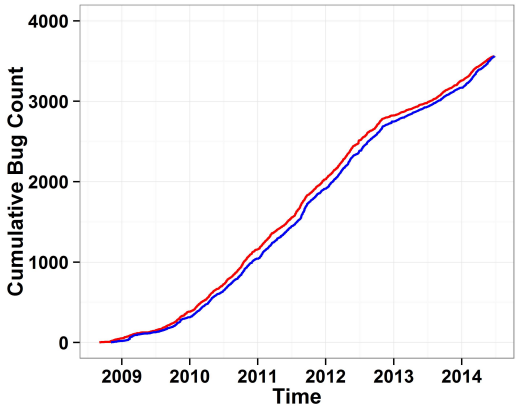


Figure 2.7: Line chart showing the crash bug opening and closing trend

Francalanci et al. [19] define bug opening and closing trend as performance indicators reflecting the characteristics and quality of defect fixing process. They define continuity and efficiency as performance characteristics of the bug fixing process. We apply the concepts presented by Francalanci et al. [19] in our research consisting of characterizing regression bugs and comparing it with other types of bugs. In their paper, cumulated number of opened and verified bugs over time is referred to as the bug opening trend. Similarly, closing trend is defined as the cumulated number of bugs that are resolved and closed over time.

Figure 2.4 shows the opening and closing trend for regression bugs. At any instant of time, the difference between the two curves (interval) can be computed to identify the number of bugs which are open at that instant of time. The debugging process will be of high quality if there is no uncontrolled growth of unresolved bugs (the curve for the closing trend grows nearly as fast or has the same slope as the curve for the opening trend). We plot all opening and closing trend graphs on the same scale and hence the differences between their characteristics

are visible. Figure 2.4 reveals that the debugging and bug fixing process for regression bugs is of high quality as there is no uncontrolled growth of unresolved bugs (the curve for the closing trend grows nearly as fast or has the same slope as the curve for the opening trend) across all bug types. Figure 2.5, 2.6 and 2.7 shows the opening and closing trend for security, performance and crash bugs respectively. We see a noticeable and visible difference between the trends for performance bug reports in comparison to other types. As shown in Figure 2.6, the slope for the performance bugs becomes relatively steep after the year 2012 indicating an increase in the number of performance issues or bugs.

We define a metric which computes the quality of bug fixing process for one type of bug report in comparison to the quality of bug fixing process for other types of bug reports.

$$BSR(T) = \frac{\Delta_{Secr} + \Delta_{Perf} + \Delta_{Crsh}}{\Delta_{Regr}} \tag{2.1}$$

$$BSS(T) = \frac{\Delta_{Regr} + \Delta_{Perf} + \Delta_{Crsh}}{\Delta_{Secr}} \tag{2.2}$$

Equation 2.1 represents the Bubble Size for Regression Bugs at a Time T $[BSR(T)]$. Δ_{Secr} at a time T represents the number of bugs which are open at that instant of time. Similarly, Δ_{Regr}, Δ_{Perf} and Δ_{Crsh} at a time T represents the number of regression, performance and crash bugs which are open at that instant of time. The numerator in Equation 2.1 represents the total number security, crash and performance bugs open at time T. If $BSR(T)$ is large then it means that the bug fixing quality of regression bug is much better than the average bug fixing quality of other types of bugs. Equation 2.2 represents the Bubble Size for Security Bugs at a Time T $[BSS(T)]$. As shown in Figure 2.4 and 2.5, we plot the BSR and BSS values for four time instants. Figure 2.4 reveals that the bug fixing quality of regression bugs (in comparison to the other three types of bugs) was the best during the second half of the year 2012. Table 2.4 shows the exact values for the metrics in Equation 2.1 and 2.2.

Table 2.4: Size of Bubble for Regression Bugs (RBBS) and Security Bugs (SBBS) in Figure 2.4 and Figure 2.5 respectively

Time	**RBBS**	**SBBS**
Mid 2009-10	0.254	1.36
2011	0.79	1.06
Mid 2012-13	1.01	0.61
2014	0.246	2.8

Chapter 3

Ground Truth Data Establishment and Feature Extraction

3.1 Ground Truth Dataset Establishment

In this Section, we describe the process followed for establishing ground truth data, which is the most important step towards building our model. Establishing ground truth for our work involves identifying a pair of bug id and a revision id that caused the bug. Since, there is no official public record of such mapping for fixed bugs, we mine the bug reports and the developer comments on the issue reports to identify the revision id that caused the respective issue.

Identification of bug causing revision is a non-trivial problem. Developers keep trying to identify the bug causing revision till the time the problem is not fixed. If a developer suspects an issue or a particular range of issues, he generally mentions it in the comments so as to narrow down the range of suspected revisions. This discussion goes on, till the person who is assigned the bug is able to identify the issue and fix it.

However, it is not necessary that the bug inducing revision is always mentioned in a comment. Largely we see that developer mention a range of suspects or the last known good revision. Table 3.1 shows developer comments on some issue ids. In many comments, we find that the developer has mentioned a range or multiple suspect issue ids. In this case as well, the developers are not sure, and are making educated guesses, based on the least known working revision. We also observe that very rarely developers mention the exact bug inducing revision. Even if they do, some other developer is quick to refute the claim. Moreover, we find that after fixing a bug, a majority of the developers, mention the revision in which the bug is fixed, but they do not mention the revision in which the bug regressed. Hence, establishing ground truth for the purpose of our work is very challenging.We use a combination of manual inspection and heuristics to identify the regression inducing revision for fixed and verified bugs. Figure 3.1 shows the steps involved in the process of establishing ground truth dataset. All these steps involved are discussed in detail below.

Table 3.1: Developer Comments from Google Chromium Issue Tracking Discussion Forums indicating Challenges in Identification of Regression Bug Inducing Changes

Issue ID	Developer Comment
308367	Based on the image the following revisions seem relevant: **I don't see how** r228618 could be the culprit
308336	**Suspecting** r158839? Would you mind taking a look at the above issue & see if this is related. Pardon me, if that's not the case.
178769	I **suspect** your r184639 causing this issue on M25 branch
158542	Nothing login-related **seems to have changed between** 23.0.1271.56 and 23.0.1271.59: Seems to be related to revision=164748 (crbug.com/148878)
257984	You are **probably looking for a change made after** 209908 (known good), but no later than 209952 (first known bad).
270025	I **could not reproduce** this after reverting my bugfixes 214446 and 209891.
79143	The WebKit **revision range is** 81970:82392. I guess http://trac.webkit.org/changeset/82376 is the culprit.
88434	It **looks like it's** r89641 (that rolled webkit from 89145 to 89228). r89640 is ok and r89645 is bad I'm now looking at webkit changes between webkit r89145-r89228

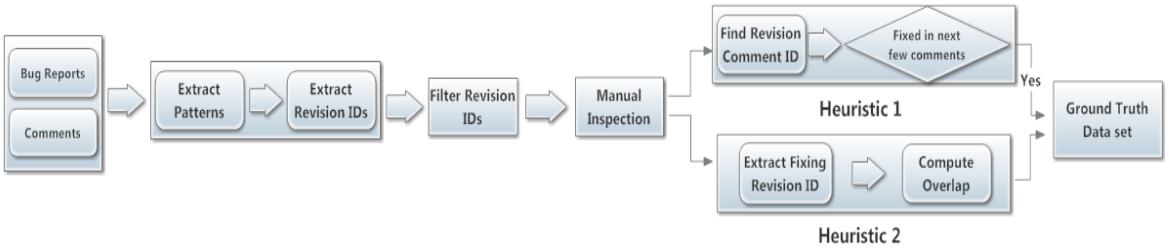


Figure 3.1: Steps for Ground Truth Dataset Establishment

3.1.1 Revision ID Extraction

From the regression issue reports in the ITS data, we consider only fixed and verified issue reports, which are 8903 in number.We then mine these issue reports and their comments to extract revision ids for each issue from them. We find that a reference to a revision id generally follows some fixed patterns. For e.g.- *r12845* , *revision 128696*, *revision=181939*, *suspecting - 179554*, *range=r25689*. We extract all revision IDs mentioned in the report and comments with the help of these patterns.

3.1.2 Filter Revision IDs

We filter the revision IDs, thus, obtained in previous step for each issue using the commit timestamp of the revisions mentioned in the VCS. Any revision id mentioned in a comment, that is committed after the bug was opened is either a revision where the developers have tried to fix the bug or have merged it. Hence, such revisions are filtered out. We consider only those revisions that were committed before the bug was opened. We further narrow down the dataset and consider only those bugs whose bug reports and comments mention only one revision that

was committed before the bug was opened.

3.1.3 Heuristics

We then manually inspect several bug reports and come up with two heuristics for identifying revisions that are suspected of having regressed the bug. *First heuristic* we used is that if a revision id is mentioned in the comments and the bug's status is changed to fixed within the next 8 comments and the comments on the issue ends up in next 3-4 comments, then the revision id mentioned has actually regressed the bug, because bug was closed soon after it was found.
Second heuristic that we come up with is overlap between paths of files modified in bug fixing and bug causing revision.For this we extract bug fixing revision ids from comments of an issue and then find changed paths overlap between revision id having timestamp before that of the bug and bug fixing revision. If the overlap value for the issue is high(i.e ≥ 0.4), then the issue id and its corresponding regression causing revision is added to the ground truth dataset. For example for issue ID 5511 there is 100% textual overlap between regression causing revision i.e. revision 6437 and regression fixing revision i.e. revision 11016 as shown in Figure 3.2.

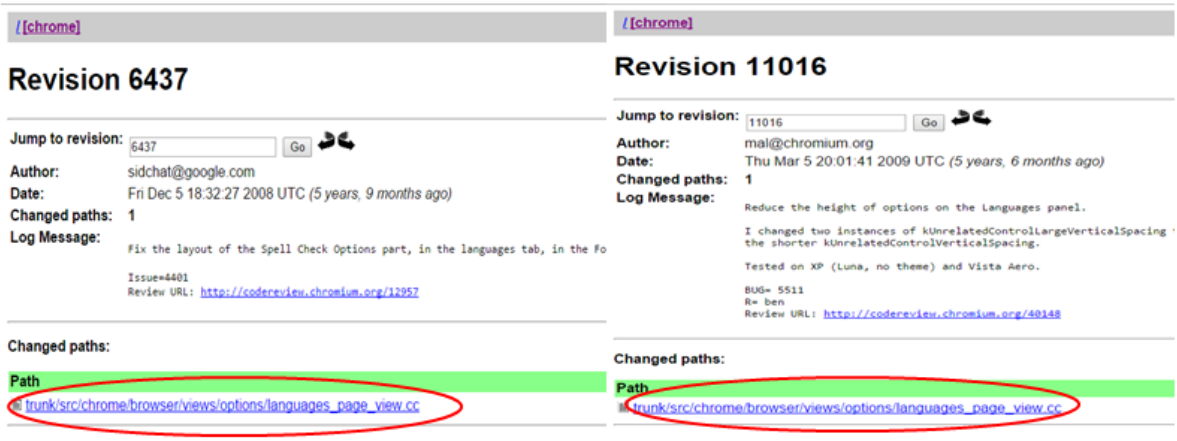


Figure 3.2: Full Textual Overlap between Changed Paths in regression causing revision ID 6437 and regression fixing revision ID 11016 for issue ID 5511

3.1.4 Ground Truth Dataset

Based on these heuristics and manual inspection, we come up with a dataset of 350 issues and the corresponding bug inducing revisions.We divide the into training dataset(300 issues) and test dataset(50 issues)

3.1.5 Challenges in Ground Truth Dataset Creation

Since there are no records of revisions causing a particular regression bug, ground truth dataset creation was a non-trivial task.When a regression bug is fixed bug fixers majorly don't specify

the revision which caused the bug. In heuristic 1 error may creep in due to the fact bug fixers may not always be right in suspecting a revision ID of having caused the bug.

3.2 Feature Extraction

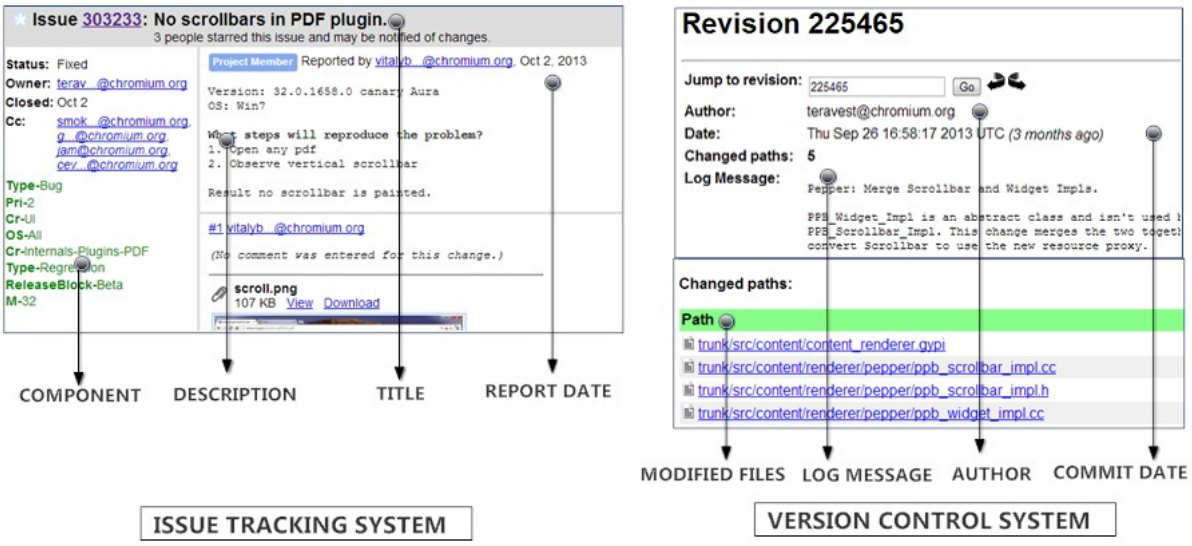


Figure 3.3: Snapshot of a Google Chromium Bug Report in Issue Tracking System and a Commit Transaction in Version Control System

We divide our dataset of 350 regression bugs and their corresponding bug inducing change into training dataset and test dataset (300 training and 50 test). Using the training data, we identify certain features that can be used to identify potential revisions that may have regressed the bug. Figure 3.3 is a snapshot of the Google Chromium Issue Tracking System and the Version Control System. It points to the location of the data, we find to be useful when identifying a bug-inducing revision. The features are as follows:

1. Time Difference between the date of report of the issue and the commit date of the bug inducing revision.
2. Similarity between Title of a bug report and the Log Message of the revision.
3. Similarity between the Description of a bug report and the Log Message of the revision.
4. Similarity between the Cr and Area labels of the issue and the top levels of the Changed Paths in the VCS.
5. Similarity between the Title of the issue and the Changed Paths in the VCS.

3.2.1 Time Difference

We first identify the statistical and density distribution of the time difference between bug inducing change and regression bug report dataset and check if it has a Gaussian or normal

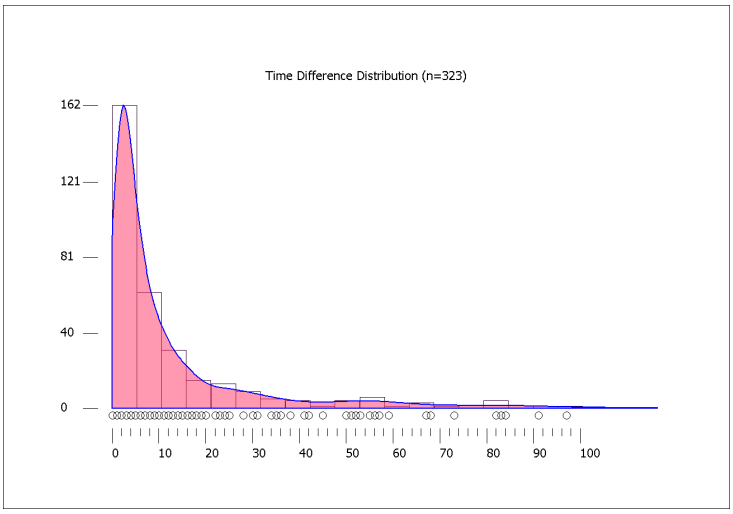


Figure 3.4: Kernal Density Estimate showing the distribution of the time difference between bug inducing change and regression bug report dataset

distribution. Figure 3.4 shows the histogram plot dividing the horizontal axis into sub-intervals or bins covering the range of the data from a minimum of 0 days to a maximum of 100 days. The size of the data sample for the histogram and density distribution is not the entire population and consists of dataset with time difference less than 100 days (93.33%). The solid blue curve is the kernel density estimate which is a generalization over the histogram. We use kernel density estimation to estimate the probability density function of the time difference variable with the aim of investigating the time difference variable as a predictor of bug inducing change for a given bug report.

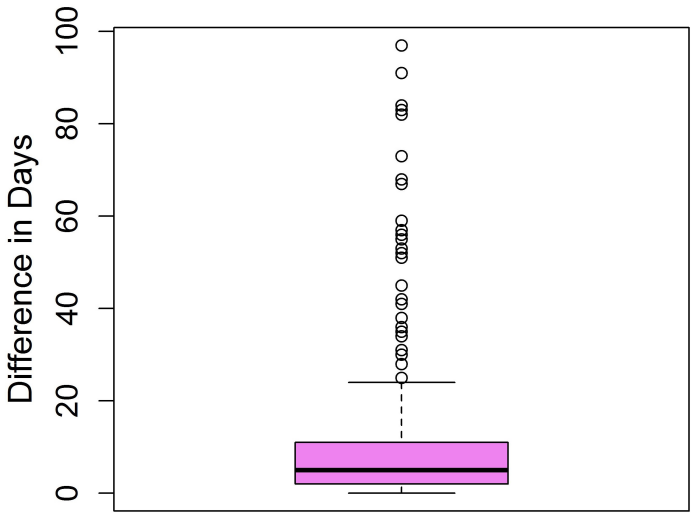


Figure 3.5: Box plot showing difference between the reporting date of the issue and the commit date of the bug inducing change

In Figure 3.4, the data points are represented by small circles on the x-axis. We observe that the data has a Gaussian distribution. The smoothing parameter (bandwidth) for the kernel density

estimate in Figure is 4. The mean (μ), variance (σ^2) and standard deviation (σ) for the data is 11.97, 294.17 and 17.17 respectively. We observe that the probability distribution is asymmetric and has a positive skewness or right skewed (the right tail is longer and the mean is greater than the mode). Figure 3.5 shows a box plot for the difference in days for 93.33% of total dataset. The plot refers to the data in Table 3.2. The mean difference in days is about 26 days with the max difference being 1393 days (almost 4 years). We also observe, that 78% of the bugs are reported within 20 days of the induction of the bug.

Table 3.2: The Five-Number Data Summary and Mean Value for Difference in Days

Min.	**Q1**	**Median**	**Mean**	**Q3**	**Max.**
0	2.00	5.00	26.91	16.00	1393.00

3.2.2 Character N-gram Approach

Before moving to textual similarity features, we will first discuss the approach used for finding textual similarity i.e. Character N-gram approach. N-gram is a continuous subsequence of n items from a given sequence of items. Word N-grams are a continuous subsequence of words whereas character N-grams are a continuous subsequence of characters. For example word bi-grams(N=2) and tri-grams(N=3) for phrase *"Regression Bug Prediction"* are *Regression Bug*, *Bug Prediction* and *Regression Bug Prediction* respectively whereas character bi-grams(N=2) and tri-grams(N=3) for word *"Regression"* are *Re,eg,gr,re,es,ss,si,io,on* and *Reg,egr,gre,res,ess,ssi,sio,ion* respectively.
For finding textual similarity character n-gram matching is chosen over whole word matching because of it provides some unique advantages in context of information retrieval [20] [21] [22].For example whole word matching will require stemming to match component "Network" with corresponding directory "net".Also word based matching will require tokenization based on "+" in order to match "shift+Alt" and "Shift+Alt"+ET_KEY_RELEASED" .

Following are some advantages of character n-gram approach over word based approach supported by real world examples.

1. Ability to Match Concepts: Bug reports, comments and log messages frequently consists of source code segments and system error messages which are not natural language text.Consider title of issue ID 128690, it contains word *"bookmarks"* and Log message of its corresponding regression causing Revision ID 112400 which contains class name *BookmarkModelTest*. Character N-gram based approach performs matching at character sequence level and hence indicates high similarity between two whereas a word level matching technique can not match directly concept present in these two strings.

2. Match Term Variation to Common Root: For example, description of issue ID 18749 contains words *"print,printed"* and its corresponding regression causing Revision ID 20876 also contains words *"printing", "print"*. Word level matching will require stemming to

Table 3.3: Illustrative Examples of Similarity between Component and File Path and Title Description and Log Message showing Advantages of Character N-Gram Based Approach over Word Based Approach

S.No	Issue ID	Component/Area	Revision ID	File Path Fragment
1	8505	Internals-**Installer**	10921	**installer**
2	204106	Platform-Apps-**Media**Player	102183	**media,media.gyp**
3	263160	Internals-**Network-Cache**	201943	**net**,disk_**cache**
S.No	**Issue ID**	**Title**	**Revision ID**	**Log Message**
1	161246	**Fullscreen disabled**	167006	Constrained Window Cocoa **Disable fullscreen**
2	128690	Importing **bookmarks** fails	112400	TEST=**BookmarkModelTest**. AddURLWithWhitespaceTitle
3	213026	Failed to switch the IME with **"shift+Alt"** on FindinPage	135791	**Shift+Alt**+ET_KEY_RELEASED accelerator for Ash
4	158995	rebuilds of remotingresources **string_resources.grd**	165041	Chromoting strings to string_resources.grd.
S.No	**Issue ID**	**Description**	**Revision ID**	**Log Message**
1	117018	judging from **about:sync**, **Signed in** state	125111	**ProfileSyncService**, **SigninTracker**
2	125323	Go to chrome://chrome/**settings/languages**	134134	TEST=browser_tests–gtest_filter=*. **TestSettingsLanguageOptionsPage**
3	1286890	**full width space**	112400	AddURLWith**WhitespaceTitle**, extra **whitespace**
4	18749	**print a label** with PayPal/Ebay,visible page is **printed**	20876	This also fixes **printing** issue with **print selection**
5	40272	**Browser action icons** should be displayed; **browser action extensions**	43044	name for **BrowserActionsContainer**; **Set ExtensionShelf** vew visibility
6	80106	page with an **auto-filled password**	75992	login **autofills** default **password**

convert all these words into their common root form whereas character level matching can match the concepts without stemmer.

3. Ability to Match Words and their Short-Forms:Bug Reporters and bug fixers frequently use short forms and abbreviation in the bug reports and directory names. For example component of issue ID 263160 is *"Internals-Network-Cache"* and files modified in its corresponding regression causing Revision ID 201493 are in directory *"net"*.

4. Ability to Match Words Joined Using Special Characters: Often terms in bug reports and file paths contain special characters. For example description of issue ID 125323 contains url chrome://chrome/*settings/languages* and log message of its corresponding regression causing Revision ID 134134 contains "TEST=browser_tests–gtest_filter=*. *TestSettingsLanguageOptionsPage*". Word level matching will require pre knowledge of such tokens to remove them in pre-processing stage whereas character level are able to detect the concept. Table 3.3 contains more examples.

3.2.3 Similarity Features

We observe that there is high textual similarity between the title and description of issue reports and log message and paths of files modified in the corresponding regression causing revision. Therefore, we extract similarity values between these four features from our training dataset. First, we pre-process the bug report title,description and log message of bug inducing change and find the textual similarity using character n-gram based approach (explained in previous section) between Title of regression bug report and Log Message of bug inducing revision, Description of bug report and Log Message of bug inducing revision and Component and Area labels of regression bug and paths changed by bug inducing change.

Pre-processing includes removal of stop words, phrases and sentences that are common in almost all bug reports and log messages and hence are irrelevant and redundant.Table 3.4 provides list of few such stop words and phrases removed during pre-processing stage.

Table 3.4: Illustrative Examples of Stop Words, Phrases and Sentences Removed During Pre-Processing Stage

S.No	Stop words/Phrase/Sentences removed
1	“hard”, “what”, “instead”, “being”, “do”, “you”, “will”, “well”, “reproduce”, “something”, “properly”, “getting”, “basically”
2	“Report ID”,“Cumulative Uptime”,“Other browsers tested”,“Meta information:”,“Thank you”
3	“What steps will reproduce the problem”,“What is the expected output”,“What do you see instead”,“Kindly refer the screencast for reference”

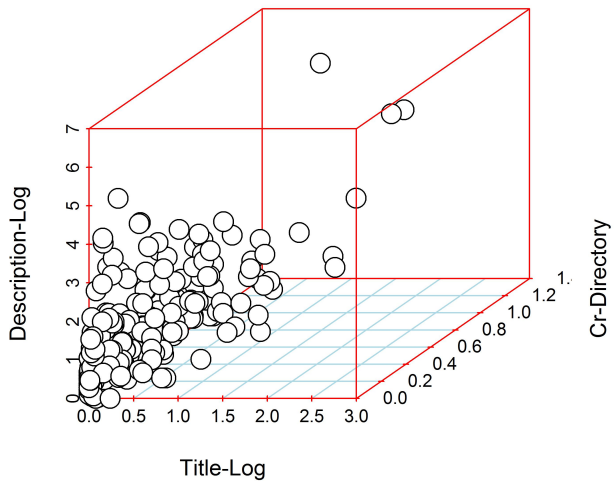


Figure 3.6: Scatter Plot showing similarity values of 3 features i.e. Cr-Directory,Title-Log, Description-Log

We use the similarity function suggested by [20] for computing the textual similarity or related-

ness between a bug report (query represented as a bag of character n-grams) and a source-code file (document in a document collection represented as a bag of character n-gram). Equation 3.1 is the formula for computing the similarity between two documents in the proposed character n-gram based IR model [20].

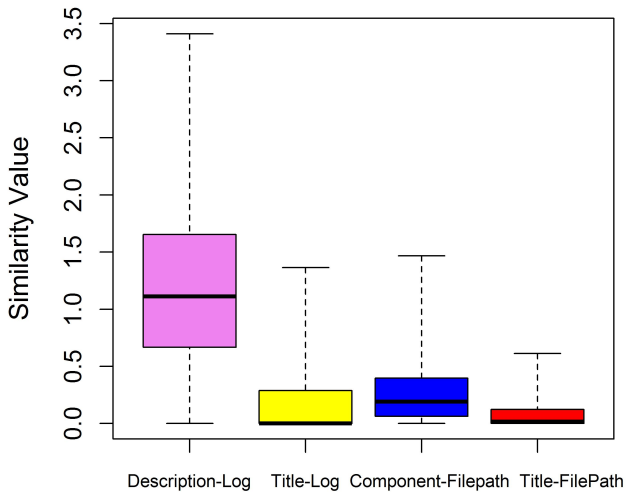


Figure 3.7: Box Plot showing the textual similarity values for 4 features

$$SIM(U,V) = \frac{\sum_{u \epsilon U} \sum_{v \epsilon V} Match(u,v) \times Length(u)}{|U| \times |V|} \tag{3.1}$$

$$Match(u,v) = \begin{cases} 1 & if: u = v \\ 0 & Otherwise \end{cases} \tag{3.2}$$

$$|U| = \sqrt[2]{f_{u_1}^2 + f_{u_2}^2 + .. + f_{u_n}^2} \tag{3.3}$$

Let U and V be two vectors of character n-grams. In the context of our work, U can be a bag of character n-gram derived from the title of the bug report and V can represent a bag of character n-gram derived from the log message of the committed revision. U and V can also represent character n-grams from bug report description and log Message of the revision respectively or cr and area labels from issue reports and paths changed in a revision respectively. The numerator of $SIM(U,V)$ compares every element in U with every element in V and counts the number of matches. A match is defined as the case where the two character n-grams are exactly equal. The value of n is added to the total sum in case of a match. The idea being that the higher the number of matches, the higher is the similarity score.

Also, the formula ensures longer strings that match contribute more towards the sum. The

denominator of $SIM(U, V)$ acts as the length normalizing factor. It removes any bias related to the length of the title and description in bug reports as well as the for log messages and changed paths in the commit data. The final similarity score is computed as a weighted average of the similarity between title and description of the bug report with the log message and component label of the bug report with the changed paths of the source-code modifying revision.

Figure 3.7 shows boxplots of similarity results found for each of the three features. The mean values for Description-Log,Title-Log Cr-Directory and Title-Directory are 1.4, 0.25, 0.32 and 0.12 respectively. The textual similarity overlap as indicated by the boxplot is found maximum in bug report description and log message of regression causing bug.

Figure 3.6 shows scatter plot for similarity values of 3 features i.e Cr-Directory,Title-Log,Description-Log for each bug in the training dataset. Scatter plot is more inclined towards x-z axis which indicates that majority of bugs have high similarity value for Title-Log and Description-Log with few outliers having similarity values for each of the three features. Also most of the bugs have high similarity values for maximum of 2 out of 3 features.

Chapter 4

Proposed Approach

In this chapter we will discuss the proposed predictive model for predicting top K regression causing bugs. Figure 4.1 shows the architecture diagram of the proposed model. The model has 4 components - 1. Revision IDs Extractor 2.Feature Extractor 3.Similarity Calculator 4. Rank generator. Each of the 4 components are explained below.

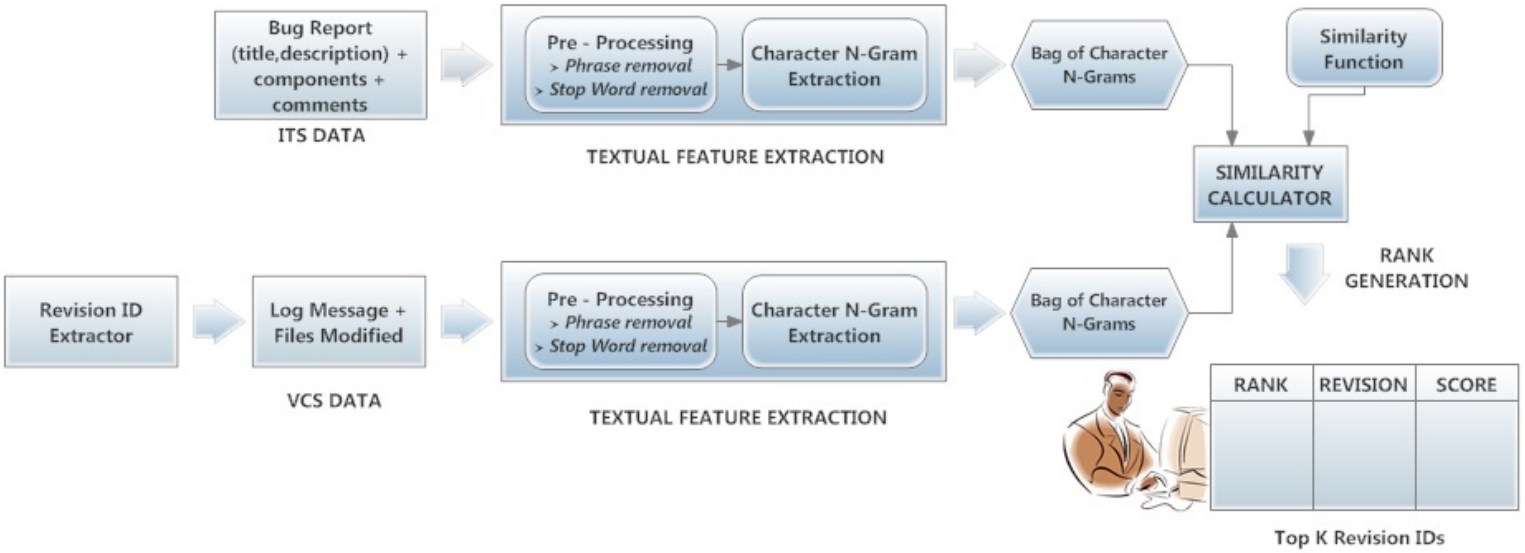


Figure 4.1: High level model architecture diagram

4.1 Revision IDs Extractor

This module extracts all revision IDs that were committed maximum "N" days before the reporting timestamp of given issue. About 78% of regression bugs are reported within 20 days after the revision inducing them was committed. Therefore, for a given regression bug the probability that it is caused by a revision made maximum 20 days back is almost 78%. Hence, this component extracts all the revision ids from the VCS system that were committed to the

VCS N=20 days prior to the opening of the bugs in the test dataset.We conducted experiments with different values of N to find out the appropriate value of N and found N=20 to be most appropriate value amongst others.

4.2 Feature Extractor

For each revision, this component extracts required features i.e title,description from bug reports and bug component/area from ITS data and log message, modified filepaths from VCS Data. The module the pre-process (i.e remove phrases and stop words and converts into lower case)the features and converts them into bag of character n-grams(of length 3 to 7). The module then returns bags of n-grams for each of these features for each revision for use by similarity calculator.

4.3 Similarity Calculator

For each revision ID, this module computes similarity between following 4 features :

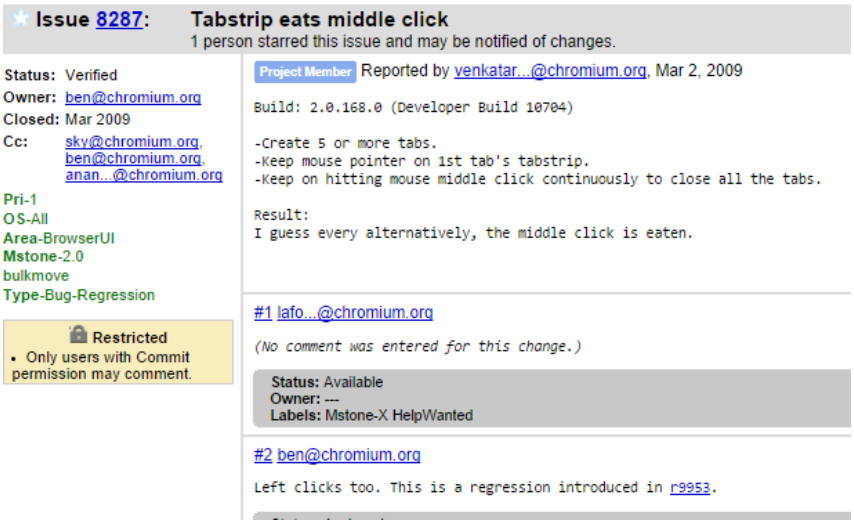
1. Title of a bug report and the Log Message of the revision.
2. Description of a bug report and the Log Message of the revision.
3. Cr and Area labels of the issue and the top levels of the Changed Paths in the VCS.
4. Title of the issue and the Changed Paths in the VCS.

The similarity between each of these features is computed using equation 3.1 as described in section 3. The output of this module is the similarity scores of each of the 4 features for each revision IDs given by Revision ID Extractor.

4.4 Rank Generator

This module computes the overall similarity value for each revision expressed as a weighted sum of similarities of each feature as given in Equation 4.1 [20]. W1, W2, W3 and W4 are weights to increase or decrease importance of one feature as compared to others.The weights are chosen such that their sum is always equal to 1. Rank for a revision ID is decided by its overall similarity score. The higher the similarity score, higher is the rank of the revision.

$$SIM_{SCORE} = W_1 * SIM(Feature1) + W_2 * SIM(Feature2) + W_3 * SIM(Feature3) + W_4 * SIM(Feature4)$$



Issue 8287: Tabstrip eats middle click

1 person starred this issue and may be notified of changes.

Status: Verified
Owner: ben@chromium.org
Closed: Mar 2009
Cc: sky@chromium.org, ben@chromium.org, anan...@chromium.org
Pri-1
OS-All
Area-BrowserUI
Mstone-2.0
bulkmove
Type-Bug-Regression

Restricted
- Only users with Commit permission may comment.

Project Member Reported by venkatar...@chromium.org, Mar 2, 2009

```
Build: 2.0.168.0 (Developer Build 10704)

-Create 5 or more tabs.
-Keep mouse pointer on 1st tab's tabstrip.
-Keep on hitting mouse middle click continuously to close all the tabs.

Result:
I guess every alternatively, the middle click is eaten.
```

#1 lafo...@chromium.org

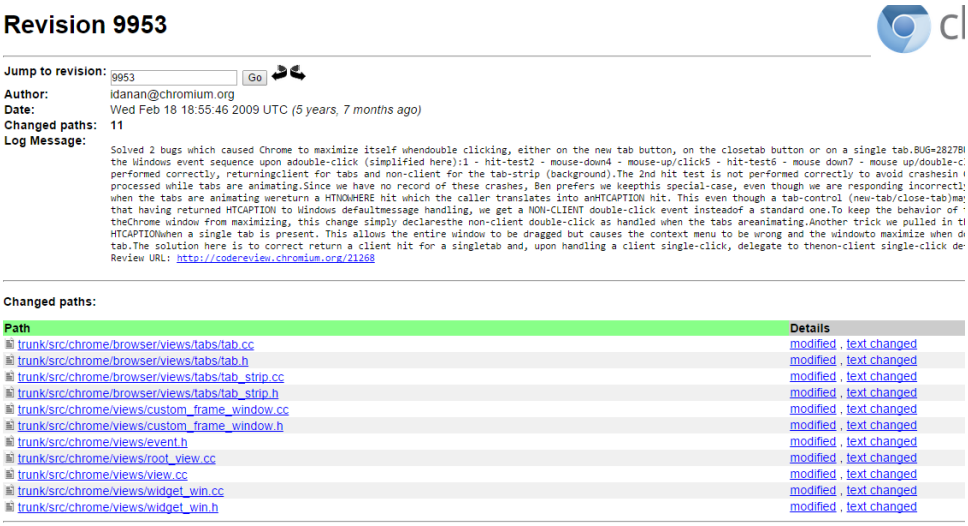
(No comment was entered for this change.)

Status: Available
Owner: ---
Labels: Mstone-X HelpWanted

#2 ben@chromium.org

Left clicks too. This is a regression introduced in r9953.

Figure 4.2: Screenshot of issue 8287 and its comments



Revision 9953

Jump to revision: 9953 Go
Author: idanan@chromium.org
Date: Wed Feb 18 18:55:46 2009 UTC *(5 years, 7 months ago)*
Changed paths: 11
Log Message:

Solved 2 bugs which caused Chrome to maximize itself whendouble clicking, either on the new tab button, on the closetab button or on a single tab.BUG=2827BI
the Windows event sequence upon adouble-click (simplified here):1 - hit-test2 - mouse-down4 - mouse-up/click5 - hit-test6 - mouse down7 - mouse up/double-c
performed correctly, returningclient for tabs and non-client for the tab-strip (background).The 2nd hit test is not performed correctly to avoid crashesin
processed while tabs are animating.Since we have no record of these crashes, Ben prefers we keepthis special-case, even though we are responding incorrectl
when the tabs are animating wereturn a HTNOWHERE hit which the caller translates into anHTCAPTION hit. This even though a tab-control (new-tab/close-tab)ma
that having returned HTCAPTION to Windows defaultmessage handling, we get a NON-CLIENT double-click event insteadof a standard one.To keep the behavior of
theChrome window from maximizing, this change simply declaresthe non-client double-click as handled when the tabs areanimating.Another trick we pulled in t
HTCAPTIONwhen a single tab is present. This allows the entire window to be dragged but causes the context menu to be wrong and the windowto maximize when d
tab.The solution here is to correct return a client hit for a singletab and, upon handling a client single-click, delegate to thenon-client single-click de
Review URL: http://codereview.chromium.org/21268

Changed paths:

Path	Details
trunk/src/chrome/browser/views/tabs/tab.cc	modified , text changed
trunk/src/chrome/browser/views/tabs/tab.h	modified , text changed
trunk/src/chrome/browser/views/tabs/tab_strip.cc	modified , text changed
trunk/src/chrome/browser/views/tabs/tab_strip.h	modified , text changed
trunk/src/chrome/views/custom_frame_window.cc	modified , text changed
trunk/src/chrome/views/custom_frame_window.h	modified , text changed
trunk/src/chrome/views/event.h	modified , text changed
trunk/src/chrome/views/root_view.cc	modified , text changed
trunk/src/chrome/views/view.cc	modified , text changed
trunk/src/chrome/views/widget_win.cc	modified , text changed
trunk/src/chrome/views/widget_win.h	modified , text changed

Figure 4.3: Screenshot of culprit revision 9953

(4.1)

Top k revision IDs are then recommended to the bug fixer as the most suspected revisions of having caused the bug. The value of K that we experimented with are 20, 25, 30 and 50. As the value of K increases the accuracy improves. We experimented with different values of and the weights to learn the impact of various variables used for computing textual similarity and also to identify configuration which gives best results. The results are discussed in next section.

Consider the Query : Provide Top 10 revision IDs suspected of having caused issue 8287.
Result Set : Top 20 revisions as returned by our model are - 9518, 10338, 10024, 9982, **9953**, 10347, 9808, 9707, 9899, 9914. Table 4.1 shows these top 10 suspected revisions and their similarity scores.

Table 4.1: Table showing Top 10 suspected revisions for issue 8287 and their overall similarity score

Rank	**Revision**	**Similarity Score**
1	9518	0.59
2	10338	0.58
3	10024	0.57
4	9982	0.56
5	9953	0.55
6	10347	0.55
7	9808	0.55
8	9707	0.54
9	9899	0.53
10	9914	0.53

The actual culprit i.e revision 9953 as mentioned in comments of issue 8287 4.2 comes at rank 5. As seen in 4.3 there is high textual similarity between title,description of issue 8287 and log message of revision 9953 and component of issue 8286 and modified file paths of revision 9953, which brings it at rank 5. Keywords *tabstrip, click* in title and description of issue 8287 occur frequently in the log message of revision 9953. ALso component of issue 8287 is *Browser-UI* which is level 4 directory in many file paths modified in revision 9953. This shows the extent of textualsimilarity between an issue and its corressponding regression inducing revision and effectiveness of using textual similarity features.

Chapter 5

Experimental Evaluation and Validation

Multiple experiments are done to measure effectiveness of proposed model for predicting regression causing revision and to get more insights. Proposed solution ranks the revisions committed "N" days before the reporting timestamp of the issue under consideration. These ranked revisions are then compared with test dataset to evaluate the effectiveness of the proposed approach. Apart from measuring effectiveness of the proposed solution we did experiments to identify the most appropriate value of "N" and also to find out which one out of four textual similarity features is most effective for predicting regression causing revisions. In the following sections we will discuss details of our experiments and results obtained.

5.1 Appropriate value of N

As discussed in Section 3.2.1, we find that 78% of bugs are reported within 20 days after the revision causing the bug was committed, and 93.3% bugs are reported within 30 days after the revision causing them was committed. Figure 5.1 shows different values of N and corresponding number and percentage of issue which were regressed by revisions committed at most N days before issue reporting timestamp.So if value of N is 20 then there are 78% chances that it the bug under consideration is regressed by a revision made at most 20 days before the bug. Choosing appropriate value of N is of prime importance as if N is small then we might end up lowering accuracy and if N is large then it might increase the chances of including false positives. Table 5.1 shows average number of revisions committed maximum N days before the issue for different values of N. If a bug is induced because of a revision committed in last 10 days then the average number of revisions that a bug fixer might need to check is 1543 as given in figure 5.1. So the deeper the bug fixer goes to search the regression causing, larger is the number of revisions he/she has to scan through.

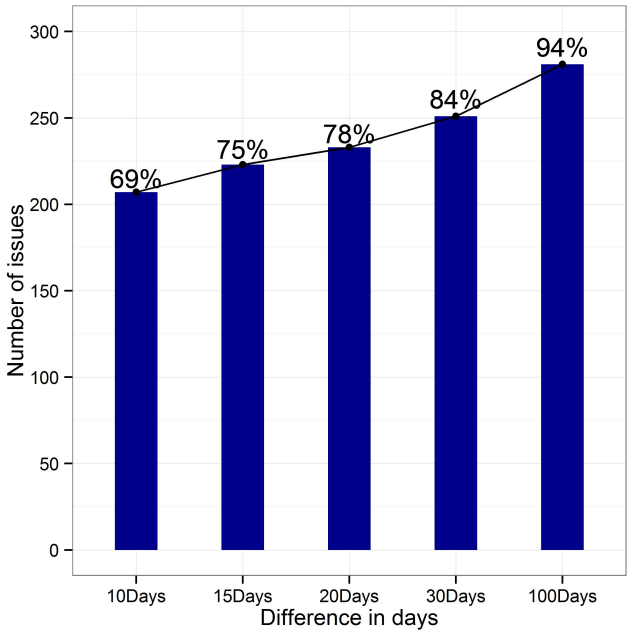


Figure 5.1: Bar Plot showing showing different values of N and corresponding number and percentage of issue which were regressed by revisions committed at most N days before issue reporting timestamp

Table 5.1: Table Showing Average Number of Revisions Committed Maximum 'N' Days before opening of Issue ID in Test Dataset

S.N0	N	Avg. Number of Revisions
1	N=10	1543
2	N=15	2340
3	N=20	3073
4	N=30	4515

5.2 Predictive Power of Each Feature

We experimented with five different weight (tuning) configurations to throw light on the predictive power of the four textual similarity features (title and description in bug report with log message in version control system and title and component of bug with changed file paths).Weights W(T-L), W(D-L), W(C-P) and W(T-P) present in Table 5.2 denotes the weights W1;W2;W3 and W4 present in equation 4.1 respectively. SIM TL represents weight configuration giving full weightage to feature T-L to find its importance. Similarly SIM DL,SIM CP and SIM TP are calculated to importance of features D-L,C-P and T-P.As indicated by the accuracy results in table 5.2, we find that feature D-L is the most important feature amongst all 4 textual features. Based on the accuracy results of SIM DL,SIM CP and SIM TP and using weight shuffling algorithm we found that best accuracy results are obtained for weight configuration SIM UNEQ.

Table 5.2: Five Different Weight Configurations to Study Predictive Power of Four Textual Similarity features

Label	W(T-L)	W(D-L)	W(C-P)	W(T-P)
SIM TL	1.0	0	0	0
SIM DL	0	1.0	0	0
SIM CP	0	0	1.0	0
SIM TP	0	0	0	1.0
SIM UNEQ	0.25	0.3	0.2	0.25

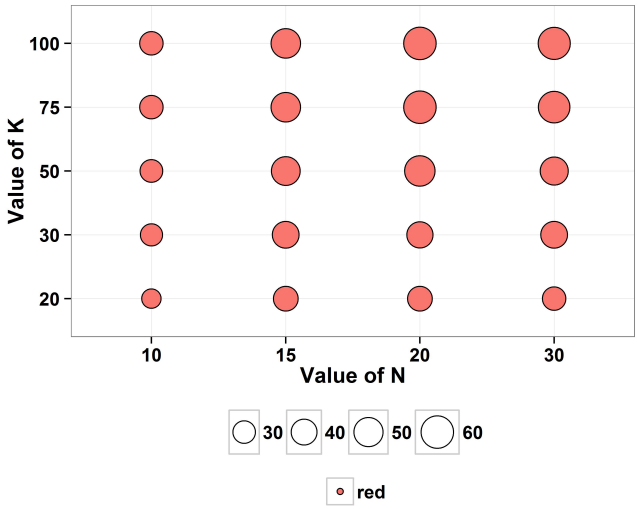


Figure 5.2: Bubble Plot showing accuracy(percentage) results for different values of N and K

5.3 Results

Table 5.3: Table Showing Accuracy in Percentage for Top K revision IDs

N	Top 20	Top 30	Top 50	Top 75	Top 100
10	21	29	30.61	33	33
15	36.73	42.85	49	51.02	51.02
20	36.73	41	54	60	60
30	32.65	42.86	46.94	57.14	59.18

Further we calculate the Top K accuracy values for different N values, with SIM UNEQ weight configuration. From Table 5.3 We find that for N=20 the bug inducing revision was a part of the top 30 list for 21 out of the 50 bugs (48%). If the list was expanded to the top 50 then the figure improved to 54% (27 bugs). Surprisingly, we find that increasing the number of days to 30, brings the percentage of hits down by 4 for Top 50. This clearly shows that the more number of revisions we analyse,the higher is the chance of introducing false positives in our ranked list. Lowering the number of days from 20 to 15, also produces a 4% drop in the results of Top 50, showing that 15 days is quite optimistic a duration for the purpose of our system.

The system achieves its best accuracy of 60% for N=20 and K=75 for weights configuration SIM UNEQ. The average number of suspected files that a bug fixer might need to go through for finding out correct regression causing revision is around 3073 as indicated in table 5.1.
Our system has 60% accuracy, which means that in 60 out of 100 cases a bug fixer will have to go through only 75 out of 3073 for finding out regression inducing change. Figure5.2 shows the bubble plot for accuracy values corresponding to N and K values. Larger bubble size indicates greater accuracy.Bubble size is almost similar for K=75 and K=100 which indicates that K=75 results in equally good accuracy as k=100.

Chapter 6

Limitations and Future Work

Our system is able to provide accuracy upto 60%. However, there are a few limitations to our work. The limitations are as follows :

1. Small size of training and test dataset - The size of our dataset is very small, primarily due to the fact that there is no record of revisions that regress a bug. Most of the developers do comment on bug reports while trying to identify the bug inducing change, but rarely do they clearly mention that exact revision id. Mostly the developers simply fix the bug and do not mention anything about the revision that caused the bug, though, at times developers do mention the id of the bug-fixing revision in the comments.

2. There can be false positives in the ground truth data. The regression causing revisions have been identified from the list of ids mentioned in the comments by the developers. If the ids mentioned in the comments were incorrect, then it is possible that the incorrectness crept into our ground truth data.

6.1 Discussions and Future Work

We demonstrated our recommendation engine Sarathi over Google Chromium dataset and results show the effectiveness of our approach with few limitations.There isn't much consistency in the language used by developers and bug reporters - Bug report titles, descriptions and log messages are written in natural language, hence, the same message can be written in different ways. Also the amount of detail provided regarding the issue and revision is dependent on the bug reporters and fixers, and varies from person to person. Hence, it is difficult to find patterns that caters to all the reports. Therefore there is a scope for improvement in the pre-processing stage by enriching the lexicons to be removed. The predictive model can further be improved by increasing the size of training dataset. More patterns for extracting mentions of revision IDs can be identified by analysing the training dataset in detail.

Chapter 7

Conclusion

From our categorization study, we find that more than 50% of the regression bugs are assigned a high priority. Security bug reports receive maximum number of comments followed by regression bugs. We observe that 50% of the regression bugs are closed within 8 days. The median value for the number of stars for regression bug is 3.95 which is the highest in comparison to other types of bugs. There is a noticeable difference between the bug opening and closing trend of various types of bugs. Our experiments indicate that the debugging and bug fixing process for regression bugs is of high quality. We also observe that the quality of bug fixing process of regression bugs in comparison to other types of bugs varies over the years and shows the best value for second half of year 2012.

Our experimental results revealed that degree of textual similarity between title, description of bug reports and log message and title,component of issue and paths of modified files can be used for identifying suspected regression causing revisions.Character n-gram based four textual features identified are effective and have several advantages over word based features. Also we find that feature of time difference between the opening of the bug report and the commit that caused the bug, is less than 20(N=20) days for about 78% of the bugs. We find accuracy results for different N values. N=20 turns out to give best accuracy results with less false positives than N=30 and better accuracy results as compared to N=10,15. In our work, we experimented with different weight configurations and found that similarity between description and log message is most important feature. We validate the model on the Google Chromium Project. We find that our recommendation engine Sarathi, returns the correct bug inducing revision in the top 50 list in 54% of the cases and around 60% in top 75 list.

Bibliography

[1] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab. '12*, pages 67–120.

[2] Aaron Bowen, Paul Fox, James M Kenefick Jr, Ashton Romney, Jason Ruesch, Jeremy Wilde, and Justin Wilson. Automated regression hunting. In *Linux Symposium '06*, pages 27–35.

[3] Janis Johnson, James Kenefick, and Paul Larson. Hunting regressions in gcc and the linux kernel. 2004.

[4] Dor Nir, Shmuel Tyszberowicz, and Amiram Yehudai. Locating regression bugs. In *Hardware and Software: Verification and Testing*, Lecture Notes in Computer Science '08, pages 218–234. Springer Berlin Heidelberg.

[5] Alexander Tarvo. Using statistical models to predict software regressions. In *ISSRE '08*, pages 259–264. IEEE.

[6] Alexander Tarvo. Mining software history to improve software maintenance quality: A case study. *IEEE Software '09*, pages 34–40.

[7] Audris Mockus and David M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal, Special Issue: Software*, pages 169–180, 2000.

[8] Emad Shihab, Ahmed E. Hassan, Bram Adams, and Zhen Ming Jiang. An industrial study on the risk of software changes. FSE '12, pages 62:1–62:11. ACM.

[9] Sunghun Kim, E. James Whitehead, Jr., and Yi Zhang. Classifying software changes: Clean or buggy? *IEEE Trans. Softw. Eng.*, pages 181–196, 2008.

[10] Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 31–41. IEEE, 2010.

[11] Abraham Bernstein, Jayalath Ekanayake, and Martin Pinzger. Improving defect prediction using temporal features and non linear models. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, pages 11–18. ACM, 2007.

[12] S. Lal and A. Sureka. Comparison of seven bug report types: A case-study of google chrome browser project. In *APSEC '12*, pages 517–526, 2012.

[13] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. Security versus performance bugs: A case study on firefox. MSR '11, pages 93–102. ACM.

[14] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. A qualitative study on performance bugs. In *MSR '12*, pages 199–208. IEEE.

[15] M. Gegick, P. Rotella, and Tao Xie. Identifying security bug reports via text mining: An industrial case study. In *MSR '10*, pages 11–20.

[16] F. Khomh, B. Chan, Ying Zou, and A.E. Hassan. An entropy evaluation approach for triaging field crashes: A case study of mozilla firefox. In *WCRE '11*, pages 261–270.

[17] M.B. Twidale and D.M. Nichols. Exploring usability discussions in open source development. In *HICSS '05*, pages 198c–198c.

[18] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. ASE '07, pages 34–43.

[19] Chiara Francalanci and Francesco Merlo. Empirical analysis of the bug fixing process in open source projects. pages 187–196.

[20] S. Lal and A. Sureka. A static technique for fault localization using character n-gram based information retrieval model. In *India Software Engineering Conference (ISEC)*, volume 1, pages 109–118, 2012.

[21] Pankaj Jalote Ashish Sureka. Detecting duplicate bug report using character n-gram-based features. In *Software Engineering Conference (APSEC), 2010 17th Asia-Pacific*, volume 1, pages 366 – 374, 2010.

[22] P Majumder, M Mitra, and BB Chaudhuri. N-gram: a language independent approach to ir and nlp. In *International conference on universal knowledge and language*, 2002.