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

Manisha Khattar Yash Lamba
Ashish Sureka
Indraprastha Institute of Information Technology, Delhi (IIITD)
New Delhi, India
{manisha1342, yash10097, ashish}@iiitd.ac.in

ABSTRACT

As a software system evolves, maintaining the system becomes increasingly difficult. A lot of times code changes or system patches cause an existing feature to misbehave or fail completely. An issue ticket reporting a defect in a feature that was working earlier, is known as a *Regression Bug*. 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.

Regression bugs are considered to be inevitable and truism in large and complex software systems [1]. *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. 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. We then describe our solution approach to find the regression bug inducing change, based on mining ITS and VCS data. We build a recommendation engine *Sarathi*[1] to assist a bug fixer in locating a regression bug inducing change and validate the system on real world Google Chromium project.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Debugging aids; H.2.8 [**Database Applications**]: Data mining; K.6.3 [**Software Management**]: Software Development

[1] *Sarathi* is a Hindi word which means "helper". Our recommendation engine *Sarathi* aims at helping bug fixers in the process of identifying regression bug inducing change.

ISEC '15, February 18 - 20, 2015, Bengalore, India

http://dx.doi.org/10.1145/2723742.2723747.

General Terms

Algorithms, Experimentation, Measurement

Keywords

Mining Software Repositories, Software Maintenance, Empirical Software Engineering and Measurements, Regression Bugs, Predictive Modeling, Issue Tracking System

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 F_p be a software functionality of S which is working correctly. A developer D enhances S to S' to implement another feature F_q by making a code change P (patch). A change can have side effects and it is possible that P breaks F_p. An issue ticket reporting a defect in F_p (in S') 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. 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 [2].

In many scenarios, especially in large and complex systems applying regression testing after every change is almost impractical, therefore 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 [1]. 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 [1][3][4]. Automating hunting of regression inducing change will help bug fixers in spending their time in fixing the bug rather than searching for its cause. This will in turn speed up the bug fixing process. The broad research motivation of the study presented in this paper 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 in this paper:

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 metadata 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.

2. RELATED WORK & CONTRIBUTIONS

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

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) [1]. They implement a solution in Python to test the Linux Kernel using the Linux Test Project [1]. 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].

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 each fix's risk of regression [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].

2.3 Characterization Study on Bug Types

et al. present a study of mining issue tracking system to compare and contrast seven different types of bug reports: crash, regression, security, cleanup, polish, performance and usability [10]. 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 [10]. Zaman et al. conduct a case-study on Firefox project and study two different types of bugs: performance and security [11]. Zaman et al. conduct a qualitative study on performance bugs [12]. Gegick et al. perform an industrial study on identification of security bug reports via text mining [13]. Khomh et al. study crash bug types [14] and Twidale at al. study usability issues [15].

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

1. We conduct 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. Although there has been work for identifying source code change that caused the work, we propose a unique *character n-gram based information retrieval model for predicting a bug inducing change for given regression bug report.*

3. EXPERIMENTAL DATASET

We conduct our study on large real-world publicly available dataset so that our experiments can be replicated. The work presented in this paper holds the required replication standards ensuring sufficient information for any third party to replicate the results without any additional information from us. As an academic, we believe and encourage academic code or software sharing in the interest of improving openness and research reproducibility. We release our code and dataset in public domain so that other researchers can validate our scientific claims and use our tool for comparison or benchmarking purposes (and also reusability and extension). Our code and dataset is hosted on GitHub[2]which is a popular web-based hosting service for software development projects. We select GPL license (restrictive license) so that our code can never be closed-sourced.

[2]https://github.com/ashishsureka/sarathi

Table 1: Number and Percentage of 8 Bug Report Types in the Experimental Dataset (Labeled 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%

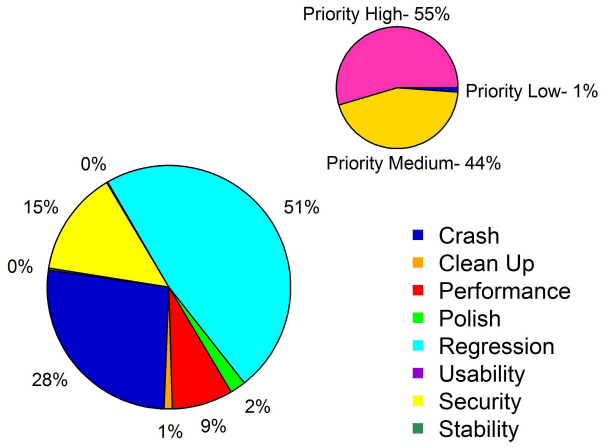

Figure 1: *Large Pie* (distribution of 8 bug report types in the experimental dataset), *Small Pie* (distribution of regression bugs across 4 priority types)

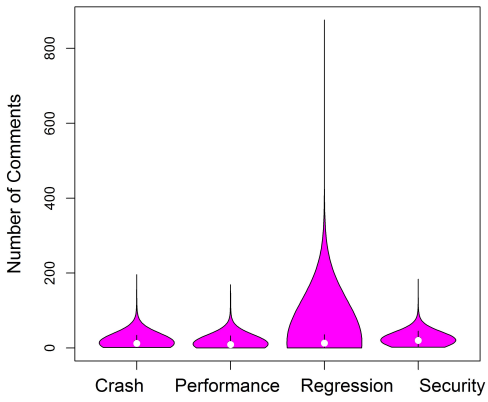

Figure 2: Violin plot of number of comments for Crash, Performance, Regression and Security bugs

We download bug reports from the Google Chromium Issue Tracking System[3] using the feed[4] 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[5], by scrapping the web page for each revision, right from the start to revision 279885 (6/26/2014 1:27:51 AM). In the ITS data, we observe that 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 1 shows the percentage distribution of the eight different types of bug reports. Table 1 reveals that 51.09% of the labeled bug reports are regression bugs (the focus of this paper). Table 1 indicates that the number of labeled 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.

4. CHARACTERIZATION STUDY

In this Section, we present our analysis on the Google Chromium Issue Tracking System and compare/contrast among the various different types of bugs and try to understand how regression bugs differentiate themselves from the other bug types.

4.1 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[6]. An issue can only have one priority value: 0 is most urgent and 3 is least urgent. Figure 1 shows a pie-of-pie chart which consists of a main pie-chart and a sub pie-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 1 reveals that 55% of the regression bugs are high priority (0 [2.8%] and 1 [51.7%]) bugs. Figure 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.

4.2 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 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

[3] https://code.google.com/p/chromium/issues/
[4] https://code.google.com/feeds/issues/p/chromium/issues/full/
[5] http://src.chromium.org/viewvc/chrome?revision=<revisionID>&view=revision
[6] http://www.chromium.org/for-testers/bug-reporting-guidelines/chromium-bug-labels

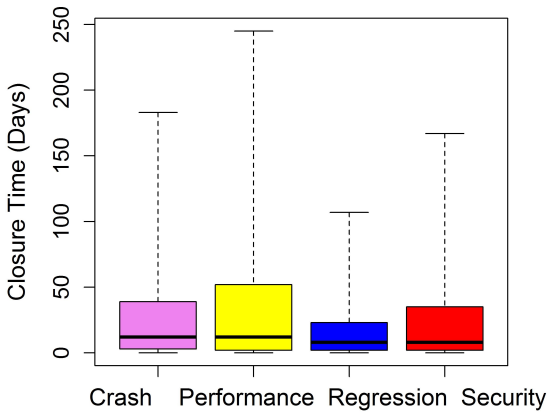

Figure 3: Boxplot for the closure time (days) for crash, performance, regression and security bugs

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 [16]. 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 reveals that broadly the shape of the distribution is the same for the four types of bug reports. The violin is most thicker (highest probability) at 9, 3, 8 and 18 for the crash, performance, regression and security bugs respectively.

4.3 Closure Time Box

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 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 shows the exact values plotted in Figure 3. While normally the boxplot show outliers, we mention the maximum value in Table 2 and not in Figure 3 due to wide variability between the minimum and maximum value. As shown in Table 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 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.

Table 2: The Five-Number Data Summary and Mean Value for the Boxplot in Figure 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

Table 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

4.4 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 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.

4.5 Bug Opening and Closing Trends

Francalanci et al. [17] 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. [17] 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 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 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 5, 6 and 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 6, the slope for the performance bugs

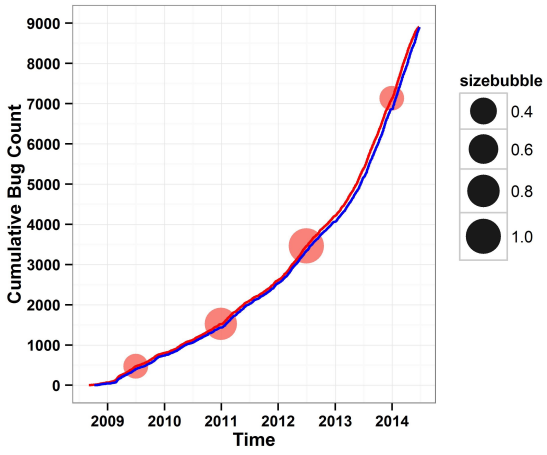

Figure 4: Line and bubble chart showing the regression bug opening and closing trend

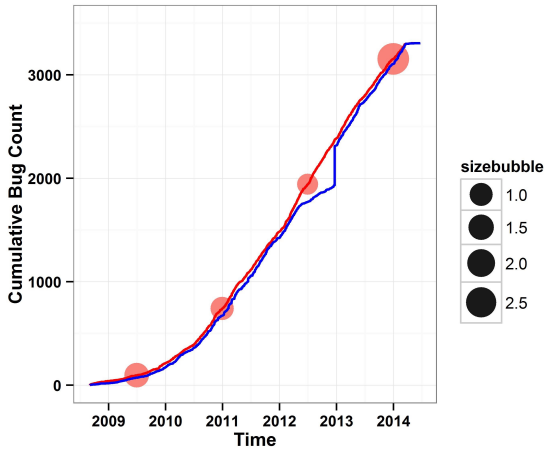

Figure 5: Line and bubble chart showing the security bug opening and closing trend

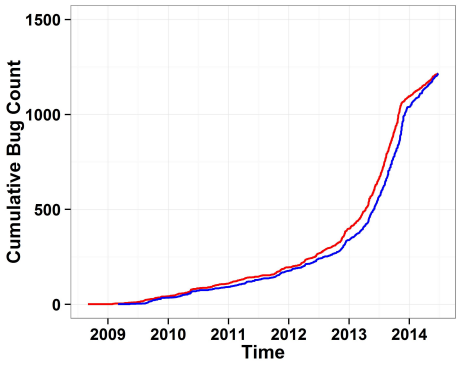

Figure 6: Line chart showing the performance bug opening and closing trend

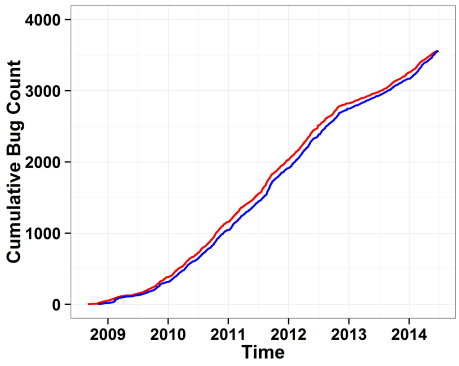

Figure 7: Line chart showing the crash bug opening and closing trend

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.

$$BSR(T) = \frac{\Delta_{Secr} + \Delta_{Perf} + \Delta_{Crsh}}{\Delta_{Regr}} \quad (1)$$

$$BSS(T) = \frac{\Delta_{Regr} + \Delta_{Perf} + \Delta_{Crsh}}{\Delta_{Secr}} \quad (2)$$

Equation 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 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 represents the Bubble Size for Security Bugs at a Time T $[BSS(T)]$. As shown in Figure 4 and 5, we plot the BSR and BSS values for four time instants. Figure 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 4 shows the exact values for the metrics in Equation 1 and 2.

Table 4: Size of Bubble for Regression Bugs (RBBS) and Security Bugs (SBBS) in Figure 4 and Figure 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

5. PROPOSED APPROACH

In this Section, we describe the entire process of building our predictive model called Sarathi, (a recommendation system for predicting top K revision-ids as potential bug inducing changes for a given regression bug report). The following 3 subsections explain the following:

Table 5: Developer Comments from Google Chromium Issue Tracking Discussion Forums indicating Challenges in Identification of Regression Bug Introducing 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

1. Ground Truth Data Collection
2. Feature Extraction from the Regression Bug and the Code Revision that Induced the Bug
3. Predictive Model

Each of these 3 steps are discussed in detail in the following subsections.

5.1 Ground Truth Data Collection

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. We mine only fixed (closed) bugs for the same. 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 5 shows developer comments on some issue ids. In 5 of the 8 comments, we find that the develop 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. So, we use a combination of manual inspection and heuristics to identify fixed regression bugs where the inducing revision is clearly mentioned.

We first mine the bug reports and all the comments of fixed and verified regression bugs. 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. We filter the revision IDs, thus, obtained 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 get 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. 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 10 comments then the revision id mentioned has actually regressed the bug because the bug was closed soon after its cause was found. Second heuristic used is overlap between paths of files modified in bug fixing and bug causing revision because often the source files in which regression is introduced are the ones being corrected(modified) in the bug fixing revision. Sp 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. Based on these heuristics and manual inspection, we come up with a dataset of 350 issues and the corresponding bug inducing revisions.

5.2 Feature Extraction

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 Temporal and Textual Similarity features that can be used to identify potential revisions that may have regressed the bug. Figure 8 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. Next two subsections discuss each of these features in detail.

5.2.1 Time Difference(N)

The temporal feature identified is the Time Difference(N) between the date of report of the issue and the commit date

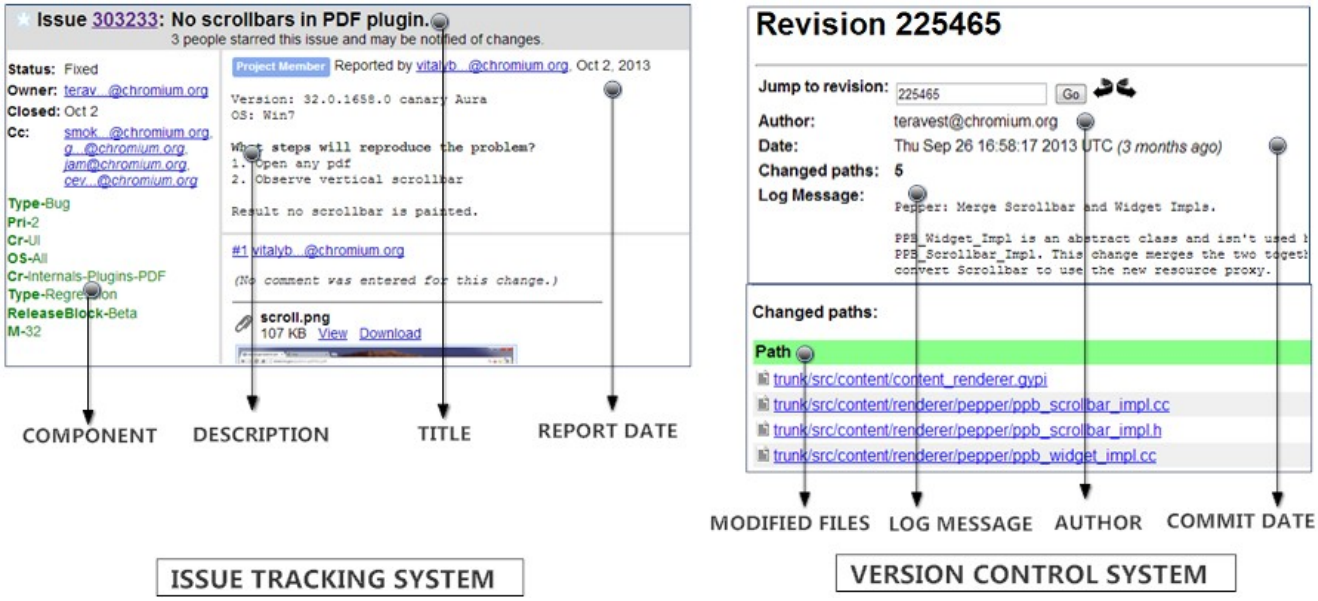

Figure 8: Snapshot of a Google Chromium Bug Report in Issue Tracking System and a Commit Transaction in Version Control System

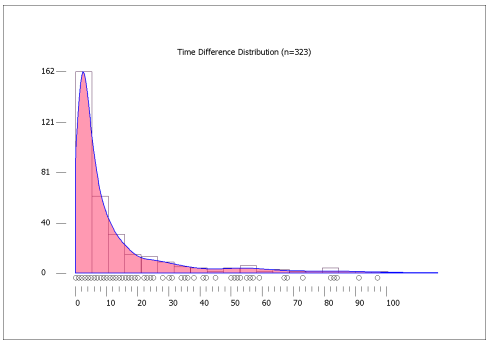

Figure 9: Kernal Density Estimate showing the distribution of the time difference between bug inducing change and regression bug report dataset

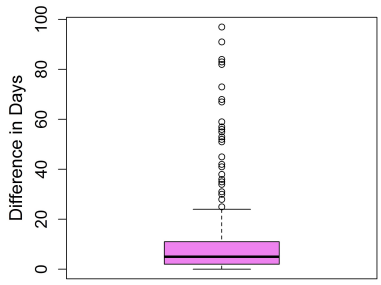

Figure 10: Box plot showing difference between the reporting date of the issue and the commit date of the bug inducing change

of the bug inducing revision.We first identify the statistical and density distribution of the temporal feature and check if it has a Gaussian or normal distribution. Figure 9 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 (93.08%). 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.

In Figure 9, 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

Table 6: 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

right skewed (the right tail is longer and the mean is greater than the mode). Figure 10 shows a box plot for the difference in days for 93.33% of total dataset. The plot refers to the data in Table 6. The mean difference in days is about 26 days with the max difference being 1393 days (almost 4 years). We also observe, that almost 80% of the bugs are reported within 20 days of the induction of the bug.

5.2.2 *Similarity Features*

The 4 textual similarity feature identified are :

1. Similarity between Title of a bug report and the Log Message of the revision.

2. Similarity between the Description of a bug report and the Log Message of the revision.

Table 7: 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”

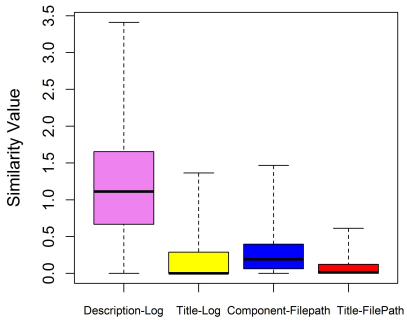

Figure 11: Box Plot showing the textual similarity values for 4 features

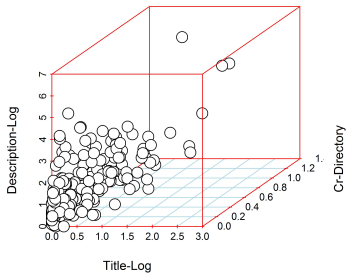

Figure 12: Scatter Plot showing similarity values of 3 features i.e Cr-Directory,Title-Log, Description-Log

3. Similarity between the Cr and Area labels of the issue and the top levels of the Changed Paths in the VCS.

4. Similarity between the Title of the issue and the Changed Paths in the VCS.

We first 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 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 7 provides list of few such stop words and phrases removed during pre-processing stage. For finding textual similarity character n-gram matching is chosen over whole word matching because of its advantages over the latter.For example whole word matching will require stemming to match component “Network” with corresponding directory ”net”. Also in word based matching will require tokenization based on “+” in order to match “shift+Alt” and “Shift+Alt”+ET_KEY_RELEASED” Table 8 shows further more examples clearly indicating advantages of character n-gram matching over whole word matching. We use the similarity function suggested by [10] for computing the textual similarity or relatedness 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 is the formula for computing the similarity between two documents in the proposed character n-gram based IR model [10].

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.

$$SIM(U,V) = \frac{\sum_{u \epsilon U} \sum_{v \epsilon V} Match(u,v) \times Length(u)}{|U| \times |V|} \quad (3)$$

$$Match(u,v) = \begin{cases} 1 & if: u = v \\ 0 & Otherwise \end{cases} \quad (4)$$

$$|U| = \sqrt[2]{f_{u_1}^2 + f_{u_2}^2 + .. + f_{u_n}^2} \quad (5)$$

$$SIM_{SCORE} = W_1 * SIM(Feature1) + W_2 * SIM(Feature2) + W_3 * SIM(Feature3) + W_4 * SIM(Feature4) \quad (6)$$

Table 8: 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**	13413	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**

Figure 11 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 12 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.

5.3 Predictive Model

Predictive model comprises of different components. *Revision IDs Extractor* extracts all revision IDs that were committed maximum "N" days before the reporting timestamp of given issue. *Textual Feature Extractor* 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 for each revision.It then pre-processes these features and extracts bags of character n-grams from each of these features which are then passed to Similarity Calculator for computing similarity values. *Similarity Calculator* computes similarity values for all 4 textual features using equation 3. *Rank Generator* computes the overall similarity value for each revision expressed as a weighted sum of similarities of each feature as given in Equation 6. W1,W2,W3 and W4 are weights to increase or decrease importance of one feature as compared to others.

5.4 Experimental Evaluation and Validation

On studying the test data set 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 13 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 the bug under consideration is regressed by a revision made at most 20 days before the bug.

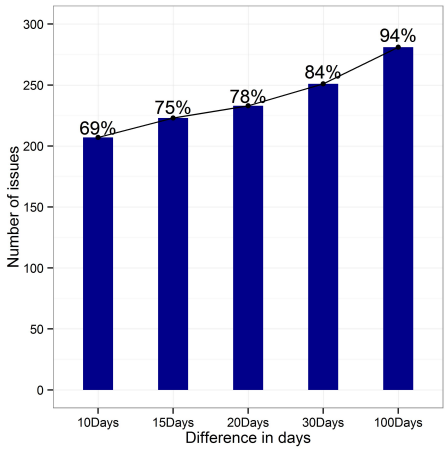

Figure 13: Bar Plot showing showing different values of N and corresponding number and percentage of issue which were regressed by revisions committed at most N days before issue reporting timestamp

Table 9: 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

Table 10: Table Showing Accuracy in Percentage for Top K revision IDs

N	Top 20	Top 30	Top 50	Top 75	Top 100
10	21	29	31	33	33
15	37	43	49	51	51
20	37	41	54	60	60
30	33	43	47	57	59

Table 9 shows average number of Revisions Committed Maximum 'N' Days before opening of Issue ID in Test Dataset. So on an average if an issue is regressed by revision made at most 20 days (N=20) before the issue was reported then the bug fixer will have to go through around 3073 revisions. 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 10 shows accuracy results for different values of N and K. The system achieves its best accuracy of 60% for N=20 and K=75. 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 9. 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.

6. CONCLUSION

Our experimental results reveal 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. 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.

7. REFERENCES

[1] Aaron Bowen, Paul Fox, James M Kenefick Jr, Ashton Romney, Jason Ruesch, Jeremy Wilde, and Justin Wilson. Automated regression hunting. In *Linux Symposium*.

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

[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] 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.

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

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

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

[14] 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.

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

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

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