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. In this paper, we describe our solution approach to finding the regression bug inducing change, based on mining ITS and VCS data. We build a recommendation engine Sarathi 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

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 research’s attention [2].

In many scenarios, where regression testing after every change is not applied, regression bugs are injected into the system due to buggy changes which are later reported by testers or users. 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]. 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. 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, diffusions 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

Lal 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. 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.

3. EXPERIMENTAL DATASET

We conduct our study on large real-world publically 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 GitHub1 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.

1https://github.com/ashishsureka/sarathi
We download bug reports from the Google Chromium Issue Tracking System\(^2\) using the feed\(^3\) 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\(^4\), 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\(^5\). 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 a 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.

---

\(^2\)https://code.google.com/p/chromium/issues/
\(^3\)https://code.google.com/feeds/issues/p/chromium/issues/full/
\(^4\)http://src.chromium.org/viewvc/chrome?revision=<revisionID>&view=revision
\(^5\)http://www.chromium.org/for-testers/bug-reporting-guidelines/chromium-bug-labels
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.

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

Table 2: The Five-Number Data Summary and Mean Value for the Boxplot in Figure 3

<table>
<thead>
<tr>
<th>Type</th>
<th>Min.</th>
<th>Q1</th>
<th>Median</th>
<th>Mean</th>
<th>Q3</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash</td>
<td>0.00</td>
<td>3.00</td>
<td>12.00</td>
<td>44.51</td>
<td>39</td>
<td>1294</td>
</tr>
<tr>
<td>Perf.</td>
<td>0.00</td>
<td>2.00</td>
<td>12.00</td>
<td>54.31</td>
<td>52</td>
<td>1685</td>
</tr>
<tr>
<td>Regr.</td>
<td>0.00</td>
<td>2.00</td>
<td>8.00</td>
<td>26.84</td>
<td>23</td>
<td>1272</td>
</tr>
<tr>
<td>Security</td>
<td>0.00</td>
<td>2.00</td>
<td>8.00</td>
<td>42.84</td>
<td>35</td>
<td>1420</td>
</tr>
</tbody>
</table>

Table 3: The Five-Number Data Summary and Mean Value for Stars for the Four Types of Bugs

<table>
<thead>
<tr>
<th>Type</th>
<th>Min.</th>
<th>Q1</th>
<th>Median</th>
<th>Mean</th>
<th>Q3</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crash</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2.93</td>
<td>3</td>
<td>224</td>
</tr>
<tr>
<td>Perf.</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3.11</td>
<td>3</td>
<td>185</td>
</tr>
<tr>
<td>Regr.</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3.956</td>
<td>4</td>
<td>703</td>
</tr>
<tr>
<td>Security</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1.335</td>
<td>1</td>
<td>78</td>
</tr>
</tbody>
</table>
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.

\[
BSR(T) = \frac{\Delta_{Secr} + \Delta_{Perf} + \Delta_{Crsh}}{\Delta_{Regr}}
\]  

(1)

\[
BSS(T) = \frac{\Delta_{Regr} + \Delta_{Perf} + \Delta_{Crsh}}{\Delta_{Secr}}
\]

(2)

Equation 1 represents the Bubble Size for Regression Bugs at a Time \(T\) \([BSR(T)]\). \(\Delta_{Secr}\) at a time \(T\) represents the number of bugs which are open at that instant of time. Similarly, \(\Delta_{Regr}\), \(\Delta_{Perf}\) and \(\Delta_{Crsh}\) 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>
<thead>
<tr>
<th>Time</th>
<th>RBBS</th>
<th>SBBS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mid 2009-10</td>
<td>0.254</td>
<td>1.36</td>
</tr>
<tr>
<td>2011</td>
<td>0.79</td>
<td>1.06</td>
</tr>
<tr>
<td>Mid 2012-13</td>
<td>1.01</td>
<td>0.61</td>
</tr>
<tr>
<td>2014</td>
<td>0.246</td>
<td>2.8</td>
</tr>
</tbody>
</table>

5. PREDICTIVE MODEL

In this Section, we describe the entire process of building our predictive model called Sarathi, (a recommendation system 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

<table>
<thead>
<tr>
<th>Issue ID</th>
<th>Developer Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>308367</td>
<td>Based on the image the following revisions seem relevant: I don’t see how r228618 could be the culprit</td>
</tr>
<tr>
<td>308346</td>
<td>Suspecting r158839? Would you mind taking a look at the above issue &amp; see if this is related. Pardon me, if that’s not the case.</td>
</tr>
<tr>
<td>178769</td>
<td>I suspect your r146539 causing this issue on M25 branch</td>
</tr>
<tr>
<td>158552</td>
<td>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)</td>
</tr>
<tr>
<td>257984</td>
<td>You are probably looking for a change made after 209908 (known good), but no later than 209952 (first known bad).</td>
</tr>
<tr>
<td>270025</td>
<td>I could not reproduce this after reverting my bugfixes 214446 and 209891.</td>
</tr>
<tr>
<td>79143</td>
<td>The WebKit revision range is 81970-82392. I guess <a href="http://trac.webkit.org/changeset/82376">http://trac.webkit.org/changeset/82376</a> is the culprit.</td>
</tr>
<tr>
<td>88434</td>
<td>It looks like it’s r89641 (that rolled webkit from 89145 to 89228). r89640 is ok and r89645 is bad I’m now looking atwebkit changes between webkit r89145-r89228</td>
</tr>
</tbody>
</table>

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

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 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. 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 example, 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 heuristics for identifying revisions that are suspected of having regressed the bug. We see 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. Based on this heuristic 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 certain 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. 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.
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.
5.2.1 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 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 ($\mu$), variance ($\sigma^2$) and standard deviation ($\sigma$) 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 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.

### Table 6: The Five-Number Data Summary and Mean Value for Difference in Days

<table>
<thead>
<tr>
<th>Min.</th>
<th>Q1</th>
<th>Median</th>
<th>Mean</th>
<th>Q3</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2.00</td>
<td>5.00</td>
<td>26.91</td>
<td>16.00</td>
<td>1393.00</td>
</tr>
</tbody>
</table>

5.2.2 Similarity Features

Next, 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 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.

Table 7: Illustrative Examples of Stop Words, Phrases and Sentences Removed During Pre-Processing Stage

<table>
<thead>
<tr>
<th>S.No</th>
<th>Stop words/Phrase/Sentences removed</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>“Report ID”, “Cumulative Uptime”, “Other browsers tested”, “Meta information:”, “Thank you”</td>
</tr>
<tr>
<td>3</td>
<td>“What steps will reproduce the problem?”, “What is the expected output?”, “What do you see instead?”, “Kindly refer the screencast for reference”</td>
</tr>
</tbody>
</table>

Table 8: Illustrative Examples of Similarity Component and File Path and Title Description and Log Message showing Advantages of Character N-Gram Based Approach over Word Based Approach

<table>
<thead>
<tr>
<th>S.No</th>
<th>Issue ID</th>
<th>Component/Area</th>
<th>Revision ID</th>
<th>File Path Fragment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9505</td>
<td>Internals-Installer</td>
<td>10921</td>
<td>installer</td>
</tr>
<tr>
<td>2</td>
<td>209106</td>
<td>Platform-Apps-MediaPlayer</td>
<td>102183</td>
<td>media_media.gyp</td>
</tr>
<tr>
<td>3</td>
<td>263160</td>
<td>Internals-Network-Cache</td>
<td>201943</td>
<td>net_disk_cache</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S.No</th>
<th>Issue ID</th>
<th>Title</th>
<th>Revision ID</th>
<th>Log Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>161246</td>
<td>Fullscreen disabled</td>
<td>167006</td>
<td>Constrained Windows Disable fullscreen</td>
</tr>
<tr>
<td>2</td>
<td>128690</td>
<td>Importing bookmarks fails</td>
<td>112400</td>
<td>TEST=BookmarkModelTest, AddURLWithWhitespaceTitle</td>
</tr>
<tr>
<td>3</td>
<td>213026</td>
<td>Failed to switch the IME with “shift+Alt” on FindinPage</td>
<td>135791</td>
<td>Shift+Alt+ET_KEY_RELEASED accelerator for Ash</td>
</tr>
<tr>
<td>4</td>
<td>158995</td>
<td>rebuilds of remotingsresources string_resources.grd</td>
<td>165041</td>
<td>Chromoting strings to string_resources.grd.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S.No</th>
<th>Issue ID</th>
<th>Description</th>
<th>Revision ID</th>
<th>Log Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>117018</td>
<td>judging from about:sync, Signed in state</td>
<td>125111</td>
<td>ProfileSyncService, SignInTracker</td>
</tr>
<tr>
<td>2</td>
<td>137134</td>
<td>Go to chrome://chrome/settings/languages full width space</td>
<td>134134</td>
<td>TEST=browser_tests-gtest_filter=-TestSettingsLanguageOptionsPage</td>
</tr>
<tr>
<td>3</td>
<td>1286890</td>
<td>print a label with PayPal/Ebay, visible page is printed</td>
<td>112400</td>
<td>AddURLWithWhitespaceTitle, extra whitespace</td>
</tr>
<tr>
<td>4</td>
<td>18749</td>
<td>print a label with PayPal/Ebay, visible page is printed</td>
<td>20876</td>
<td>This also fixes printing issue with print selection</td>
</tr>
<tr>
<td>5</td>
<td>40272</td>
<td>Browser action icons should be displayed browser action extensions</td>
<td>43044</td>
<td>name for BrowserActionsContainer Set ExtensionShell view visibility</td>
</tr>
<tr>
<td>6</td>
<td>80106</td>
<td>page with an auto-filled password</td>
<td>75992</td>
<td>login autofills default password</td>
</tr>
</tbody>
</table>
Figure 11: Box Plot showing the textual similarity values for 4 features

\[ SIM(U, V) = \frac{\sum_{u \in U} \sum_{v \in V} \text{Match}(u, v) \times \text{Length}(u)}{|U| \times |V|} \]  \hspace{1cm} (3)

\[ \text{Match}(u, v) = \begin{cases} 1 & \text{if } u = v \\ 0 & \text{Otherwise} \end{cases} \]  \hspace{1cm} (4)

\[ |U| = \sqrt{f_{a_1}^2 + f_{a_2}^2 + \ldots + f_{a_n}^2} \]  \hspace{1cm} (5)

\[ \text{SIMSCORE} = W_1 \ast \text{SIM(Feature1)} + W_2 \ast \text{SIM(Feature2)} + W_1 \ast \text{SIM(Feature3)} + W_4 \ast \text{SIM(Feature4)} \]  \hspace{1cm} (6)

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. Also most of the bugs have high similarity values for maximum of 2 out of 3 features.

5.3 Experimental Evaluation and Validation

On studying the test data set we find out that 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 30 days back is almost 80%. Based on this result, for a given regression bug we decide to test our predictive model for three different time durations: 15 days, 20 days and 30 days.

Firstly, we find all the revisions that were committed to the VCS 15 days, 20 days and 30 days prior to the opening of the bugs in the test dataset. Now, for each of the revisions obtained in the previous step for a particular bug, we calculate the similarity value for all the four features using Equation 3. 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. A feature can be given importance over other features by increasing its weight as compared to others. The overall similarity value for each revision is expressed as a weighted sum of similarities of each feature as given in Equation 6. We tried different combinations of weights such as giving equal weights to all features or assigning weight as 1 for a particular feature and 0 for all others to find the importance of a feature as compared to others. From our experiments, Title-Changed file paths turns out to be most important feature. Based on the features, the higher the similarity value of a revision, the more likely it is to cause that regression bug. Thus, for a given bug we produce 3 ranked lists of all revisions committed 15, 20 and 30 days before the bug report timestamp.

The system then recommends top K bugs to the bug fixer for analysis. These top K bugs (in descending order of similarity value) are the top suspects of having caused the bug. We validate the model on the 50 bugs in our test dataset. We find that the bug inducing revision was a part of the top 30 list for 24 out of the 50 bugs (48%). If the list was expanded to the top 50 then the figure improved to 52% (26 bugs). Surprisingly, we find that increasing the number of days to 30, brings the percentage of hits down by 4. This clearly shows that the more number of revisions we analyse,
the higher is the chance of introducing false positives in our ranked list. This may have something to do with the fact that when some functionality is being worked upon by developers, generally a lot of commits which are textually similar to each other (same files, similar terms in log messages) are made. Lowering the number of days from 20 to 15, also produces a 2% drop in the results, showing that 15 days is too optimistic a duration for the purpose of our system.

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.

We find that features like the time difference, between the opening of the bug report and the commit that caused the bug, to be less than 20 days for about 78% of the bugs. We use other features like similarity between title, description of bug report and log message of VCS commit, Component and Area label and changed paths also help in identifying the bug inducing change. In our work, we assign all these features equal weightage and validate the model on the Google Chromium Project. We find that our recommendation engine Sarathi, returns the correct bug inducing revision in the top 30 list in 48% of the cases.

7. REFERENCES


