University of Glasgow | School of Computing Science

Measuring Software Ticket Quality using Quantitative Data Analysis

Andrei-Mihai Nicolae (21473292)

April 18, 2018

ABSTRACT

Software tickets are of valuable importance to the whole computing science field - they guide engineers towards better planning, management and tracking of their progress throughout complex projects. However, there are few studies that investigate what makes for a high quality, valuable ticket. This can lead to multiple issues in a company, such as increased communication friction between developers and end users filing bug reports, as well as increased overall costs due to waste of development effort. In this research paper, we present our findings after investigating a large number of variables surrounding software tickets, such as whether the presence of stack traces influence the time to close for the ticket. Our results show that the presence and type of attachments, comments complexity (i.e. number of comments per ticket and total number of words), summary and description complexity, grammar correctness scores as well as the sentiment drawn from the comments can influence the quality of the ticket. We bring a couple of novel aspects to the research community including one of the largest dataset statistically analysed in the field, as well as state-of-the-art sentiment and grammar correctness analysis.

1. INTRODUCTION

In the past decade, technology has drastically increased its influence on virtually every aspect of our society. Therefore, software projects have inherently become more complex and require increasing number of developers in the team. Due to this, software engineers have created issue tracking systems, a means of planning, managing and tracking work products in the form of *software tickets* or *issues*.

There are multiple platforms for providing such issue tracking systems, among which the most popular are Jira [1] and Bugzilla [10]. For both platforms, the tickets are split into two main categories: feature requests (i.e. feature to be implemented into the system) and bug reports (i.e. issue encountered by an end user or discovered by a developer in the codebase). Regardless of the type of ticket, they possess various information that can be filled in by the reporter (i.e. person who created the ticket; can be both an end user or a developer in the team), providing the developers a detailed view of what is requested or what went wrong.

Even though tickets provide such comprehensive data regarding a specific task, studies have shown that fixing bugs is one of the most common tasks performed by developers [8]. One of the reasons for this is the communication friction between developers and end users [7] as developers might need clarification regarding what information the users have provided (e.g. cannot reproduce the bug, screenshot is unclear). Another main reason for this waste of effort on solving tickets, according to Just et. al [6] and Zimmermann et. al [13], is the generally poor design of issue tracking systems. This can lead to various issues, including increased costs for the company, wasted development effort, decreased customer satisfaction and overall poor performance.

Therefore, there is a need in the community to find the answer to what makes for a high quality, valuable software ticket that would improve the overall performance of the development team and, inherently, the company. As there are many fields in a ticket, there are numerous unanswered questions, such as whether stack traces have an influence on the quality of a ticket?

In this research paper, we present and discuss our findings after running a quantitative data analysis on over 3200,000 tickets taken from more than 15 open source projects. We have implemented a Go application with multiple commands (i.e. store, analyze, plot, statistics) that can automatically fetch any number of tickets from a Jira instance, analyze them, generate plots and run statistical tests.

During the analysis part, we investigate several variables in correlation with *Time-To-Close*, which we define as the *metric of quality*. *Time-To-Close* represents the period of time between the creation and the closing of a ticket; more specifically, the creation of the ticket is marked when the status of the ticket is set to *Open* and the closing of a ticket is considered when the ticket status is set to *Closed* (or some similar status, such as *Fixed*, *Resolved*). *Time-To-Close* is our dependent variable, and as independent variables (i.e. variables controlled in order to test the effects on the dependent variable) we have set a number of ticket fields.

This study brings several contributions to the research community:

- it is one of the very few research projects that investigates such a large number of tickets (over 300,000) spanning across 38 different projects;
- it is one of the few studies in the field that performs a quantitative analysis rather than a qualitative one;
- it is, to our knowledge, the first study to conduct sentiment and grammar correctness analyses on software tickets.

In Section 2 we iterate over state-of-the art studies in the field and then continue with Section 3 where we discuss how the ticket data set was collected and analysed. We continue with describing the data set (Section 4) where we provide insights into various aspects of the data (e.g. size of database, number of tickets with attachments) and then discuss about correlations between the variables (Section 5). Finally, we provide future research directions in Section 6 and present our conclusions in Section 7.

2. RELATED WORK

3. BUILDING THE DATA SET

The first step towards building the data set was to create a tool that was able to execute all the commands that we required: fetching the tickets from any Jira instance, storing them into some form of database, analyze the variables of interest, automatically plot the correlations between them and run statistical analysis on the data. After careful consideration, we decided that Go was the best way to go for various reasons:

- it is designed with simplicity in min, thus making it easier for others who might join the project to read and understand the codebase;
- it is compiled, statically typed, which implies that it is a much faster candidate than other interpreted languages such as Ruby or Python;
- is is designed with concurrency in mind, thus it helps reduce times of execution and computing power considerably;

Then, we split our tool into four main commands: *store*, *analyse*, *plot* and *stats*. They all follow the flow shown in figure 1 and complete the whole application cycle, in the end producing a database of analysed tickets, as well as plots and statistics for investigating correlations, saved on the filesystem.

In the following two subsections we describe how we designed and implemented the commands mentioned above.

3.1 Fetch and Store

This is the first command implemented in our tool - it is desgined to fetch, from any valid Jira URL, any number of tickets and then store everything inside an instance of BoltDB [5], which is a very simple yet powerful key-value pair database. The whole process is parallelized with the possibility of scaling even up to hundreds of goroutines (i.e. lightweight threads) running concurrently.

We have interacted with the Jira Server REST API which has very good documentation written by the Atlassian team. After getting familiar with all the features the REST API offers, we have opted for getting paginated issues from the server, which means slicing the whole set of tickets into multiple smaller arrays to improve performance and reduce the load on the server. We also specified to Jira exactly what set of issue fields to return and, eventually, while responses were retrieved from the instance, we began storing them into our Bolt DB instance. The way we chose to store them was to set the issue key (i.e. unique identifier that differentiates a ticket from all others across all projects inside a Jira instance) as the key of the pair and the value is the whole representation of an issue (e.g. attachments, story points, priority) as a JSON encoded value. This helped us reduce the size of the database once we reached hundreds of thousands of tickets.

3.2 Analyse

Once the Bolt DB instance is completely populated with all the tickets we were interested in, the analyze command first fetches all the issues in the database. Then, it runs in parallel the *seven types of analysis*:

- *attachments* - checks whether attachments influence Time-To-Close and also look at what types of attachments (e.g. screenshots, archive, code snippets) influence quality the most;
- *steps to reproduce* - performs a complex regex to detect steps to reproduce and then checks whether their presence influence the quality of a ticket;
- *stack traces* - runs complex regex for detecting stack traces in either summary, description or comments and verifies whether their presence influence Time-To-Close for the ticket;
- *comments complexity* - loops through all comments and counts all words; then, the tool verifies whether having many words or comments decreases or increases Time-To-Close for the ticket;
- *fields complexity* - same analysis as comments complexity, but only for summary and description;
- *grammar correctness* - it uses Azure Cloud Bing Spell Check API [9] to perform analysis on summary, description and comments; after concatenating everything and making it compatible with the API allowed formats, the tool receives back in the JSON payload not only the number of flagged tokens (i.e. grammar errors), but also their types (e.g. unknown token, misspell);
- *sentiment* - uses Google Cloud Platform's Natural Language Processing API [4] to retrieve the sentiment score for summary, description and comments; it first concatenates everything and makes it conform to Google's API and then sends the request, receiving a score from -1 to 1 inclusive (-1 is completely negative, 1 is completely positive).

Implementation wise, the application first checks Bolt for tickets and gets either all of them in memory (easier to parse, but heavy on resource utilization) or slices them into smaller arrays to get processed afterwards (harder to parse as it eventually requires re-creation of the whole array of tickets before inserting back into the database, but it is much lighter on computing power). After having the tickets available, we start looping through them and perform all analysis types mentioned above. Once some sort of value is computed (e.g. grammar correctness score), it gets set in its corresponding field in the ticket struct and subsequently is stored back in the database. We chose this approach because we do not want to run the analysis every time we want to plot correlations between variables, but rather have the data already available there. Moreover, for external analysis such as hitting Google Cloud Platform APIs, there are costs for running the analysis, thus in this way we drastically reduce overall costs for the project.

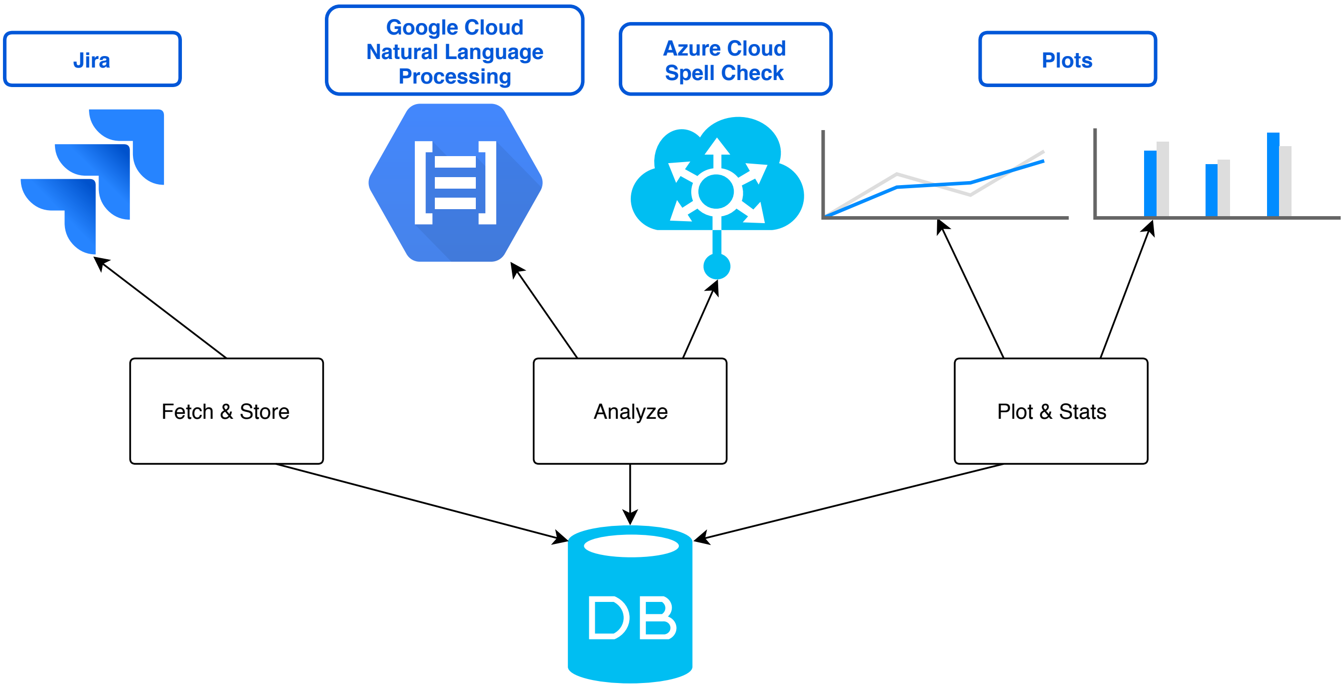

Figure 1: Application flow of the Go tool.

3.3 Plot and Statistical Tests

Afterwards, we have created two extra commands for automatically generating plots (i.e. scatter plots and bar charts) and running statistical tests on the collected data.

For plotting the data, we first connect again to the Bolt database instance and then filter out only the issues that have the variable *Time-To-Close* set (i.e. tickets marked as closed when they were fetched from the Jira instance). Afterwards, we either save scatter plots or bar charts to disk.

Also, in order to validate our findings, the statistics command fetches all the data from the database and runs two types of tests: Welch's T Test [12] for analysing categorical data (e.g. has/does not have attachments) and Spearman's rank correlation coefficient [11] for investigating continuous data (e.g. how do different grammar correctness scores change *Time-To-Close*).

4. CHARACTERISING THE DATA SET

The data that we collected is stored, as previously mentioned, in a Bolt database instance which is around 6 GB in size. All tickets are stored as key-value pairs, where the key is the ticket's unique ID (e.g. KAFKA-100) and the value is the JSON encoded representation of the ticket. The tickets stored have the following fields available for investigation:

- attachments - files or images attached to tickets;
- summary - short description of the ticket;
- description - more detailed information regarding what is requested or what bug was encountered;
- time estimate - estimated number of hours to complete the task (set by triagers or developers);
- time spent - time period between opening the ticket and a specific point in time;
- created timestamp;
- issue status - jira specific statuses, including Open, Closed, Awaiting Review, Patch Submitted;
- due date - optional deadline for when the ticket should get closed;
- comments - discussion around the ticket conducted by developers, end users, triagers;
- priority - it can range from low priority (minor) to critical/blocker;
- issue type - specific Jira field that specifies whether the ticket is either a bug, a feature request or a general task.

There are other fields that can be configured inside Jira, including custom fields, but we did not collect them as they would not have been helpful in conducting the analysis. However, in addition to the fields we saved, we also store grammar correctness scores (i.e. number of grammar mistakes identified in summary, description and comments), sentiment scores (i.e. sentiment drawn from summary and description), whether they have steps to reproduce, if stack traces are present and number of words in comments, summary and description. Even though all of them, apart from grammar and sentiment scores, can be computed locally, we store them because of performance issues due to the very large size of the database.

In total, we have collected 303,138 tickets spanned across 38 projects from the Apache Software Foundation: Impala, Eagle, Groovy, Lucene, Hadoop, Kafka, Apache Infrastructure project, Tika, Solr, ActiveMQ, ZooKeeper, Velocity, Tez, Storm, Stratos, CouchDB, Cassandra, Beam, Aurora, Bigtop, Camel, CarbonData, Cloudstack, Flex, Flink, Ignite, HBase, Mesos, Ambari, Cordova, Avalon, Atlas, Cactus, Flume, Felix, Geode, Ivy and Phoenix. These projects are using a large varieties of programming languages, ranging from Java, C, Go to Python and Ruby [3]. Moreover, in terms of contributors, the projects we selected range from small teams of people such as Tika to large numbers of developers spread across the globe, such as the people working on Kafka. We came to this final number and range of tickets and projects because of a couple of reasons:

- the study needed as much diversity as possible in order to correctly analyse and validate the data;
- these are the most important, up-to-date and contributed to Apache projects by the open source world;
- Apache is one of the few companies/foundations that use Jira exclusively for their projects and it is public (they do provide a Bugzilla alternative, but it is rarely updated compared to Jira).

More specifically, we computed the following numbers for the tickets we collected:

- out of the 303,138 tickets, 236,383 tickets have been closed (i.e. marked Closed, Resolved, Done or Completed) by the time we fetched them from the Jira instance;
- 287,120 tickets are of high priority (i.e. marked as Blocker, Critical, Major or High);
- 103,397 tickets have attachments which we split in the following categories:
  - 4,082 have code attachments (e.g. Go, Java, Python);
  - 7,171 have image attachments (e.g. png, jpg/jpeg, gif);
  - 164 have video attachments (e.g. mkv, avi, mp4);
  - 13,990 have text attachments (e.g. txt, md, doc(x));
  - 918 have config attachments (e.g. json, xml, yml);
  - 2,491 have archive attachments (e.g. zip, tar.gz, rar, 7z);
  - 280 have spreadsheet attachments (e.g. csv, xlsx);
  - 80,015 have other attachments (i.e. any file extension not pre-defined by us in one of the above categories).
- 270,907 tickets have comments;
- 39,988 tickets have steps to reproduce (i.e. sequence of steps specified by the bug reporter that would reproduce a certain bug);
- 1,942 tickets have Java stack traces - we have applied the technique described by Bettenburg et. al [2] for extracting structured data from bug reports and even though the method proved to be efficient, the total number of stack traces is so small due to the fact that many projects are actually written in languages other than Java;

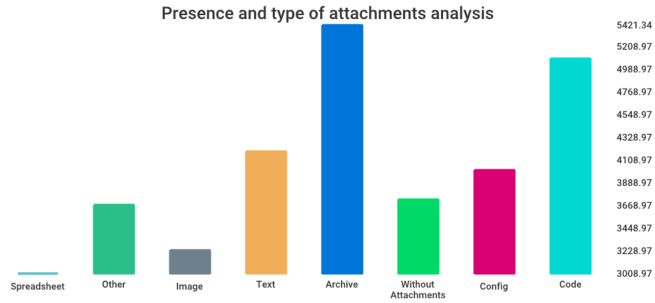

Figure 2: Application flow of the Go tool.

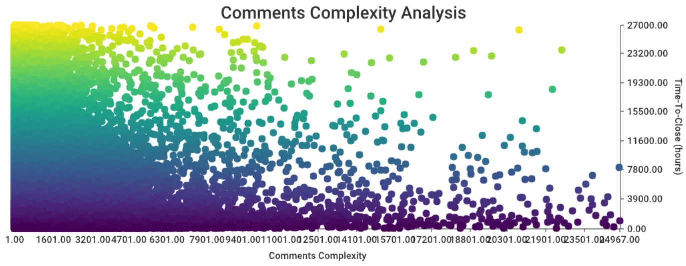

Figure 3: Application flow of the Go tool.

- 133,689 tickets have grammar correctness scores (i.e. number of grammar mistakes inside summary, description and comments) - they have been collected using Microsoft Azure Bing Spell Check API which is one of the best grammar checking tool in the industry;
- 157,047 tickets have sentiment scores (i.e. positive/negative sentiment on a scale from -1 to 1 drawn from summary, description and comments) - they have been collected using Google Cloud Platform Natural Language Processing APIs;

5. CORRELATIONS

5.1 Attachments

5.2 Comments

5.3 Summary and Description

5.4 Grammar Correctness

5.5 Sentiment Scores

5.6 Steps to Reproduce

5.7 Stack Traces

6. FUTURE WORK

7. CONCLUSIONS

Acknowledgments. Tim, my parents, Corina.

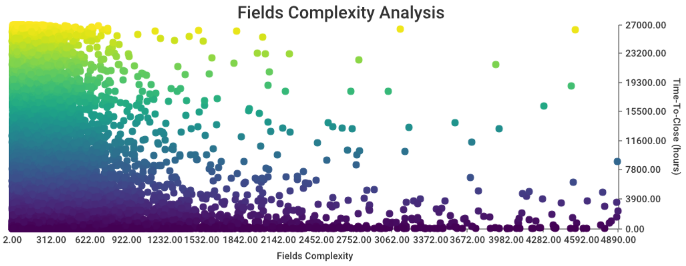

Figure 4: Application flow of the Go tool.

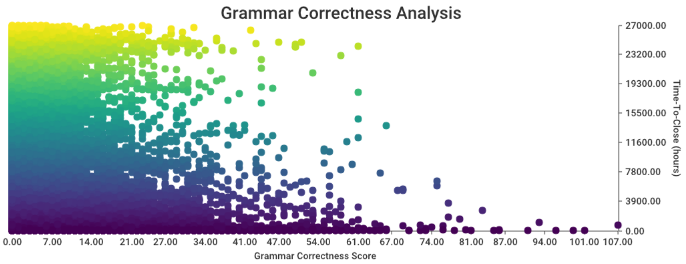

Figure 5: Application flow of the Go tool.

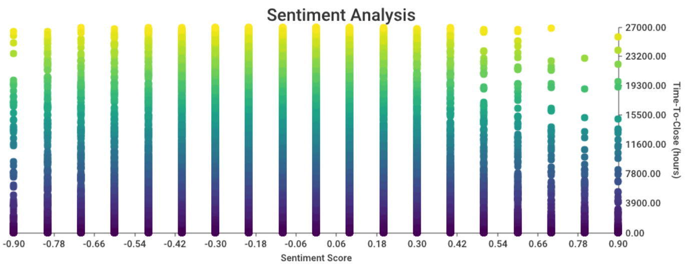

Figure 6: Application flow of the Go tool.

Figure 7: Application flow of the Go tool.

8. REFERENCES

[1] Atlassian. Jira. `https://www.atlassian.com/software/jira`. [Online; accessed 02-April-2018].

Figure 8: Application flow of the Go tool.

[2] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Extracting structural information from bug reports. pages 27–30, 2008.

[3] A. S. Foundation. Apache Projects. `https://projects.apache.org/projects.html?language`. [Online; accessed 12-March-2018].

[4] Google. GCP Natural Language Processing API. `https://cloud.google.com/natural-language`. [Online; accessed 05-March-2018].

[5] B. Johnson. Bolt. `https://github.com/boltdb/bolt`. [Online; accessed 19-February-2018].

[6] S. Just, R. Premraj, and T. Zimmermann. Towards the next generation of bug tracking systems. pages 82–85, 2008.

[7] M. Korkala and F. Maurer. Waste identification as the means for improving communication in globally distributed agile software development. *The Journal of Systems and Software*, 95:122–140, 2014.

[8] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501. ACM, 2006.

[9] Microsoft. Bing Spell Check API. `https://azure.microsoft.com/services/cognitive-services/spell-check`. [Online; accessed 05-March-2018].

[10] Mozilla. Bugzilla. `https://www.bugzilla.org/`. [Online; accessed 02-April-2018].

[11] C. Spearman. The proof and measurement of association between two things. *The American journal of psychology*, 15(1):72–101, 1904.

[12] B. L. Welch. The generalization ofstudent's' problem when several different population variances are involved. *Biometrika*, 34(1/2):28–35, 1947.

[13] T. Zimmermann, R. Premraj, J. Sillito, and S. Breu. Improving bug tracking systems. In *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 247–250. IEEE, 2009.