University of Glasgow | School of Computing Science

Measuring Software Ticket Quality using Quantitative Data Analysis

Andrei-Mihai Nicolae (2147392)

April 17, 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), 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 [8]. 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 provide various types of information that can be filled in by the reporter, including:

- summary - short description of the feature to be implemented/encountered bug;
- description - longer textual field which goes into more detail regarding the ticket; it can include various types of information, such as stack traces or steps to reproduce a bug;
- attachments - screenshots, snippets of code or configuration files that might help close the ticket faster;
- comments - each ticket can have any number of comments where the people interested in it can talk about what might have caused the bug, how to fix it etc.

Even though tickets provide such comprehensive information regarding a specific task, studies have shown that fixing bugs is one of the most common tasks performed by developers [6]. One of the reasons for this is the communication friction between developers and end users [5] 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 [4] and Zimmermann et. al [10], 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, the number of unanswered questions is rather large: do stack traces have an influence on the quality of a ticket? What about attachments and whether a screenshot is more helpful than a snippet of code? Does a negative sentiment drawn from comments increase or decrease the time taken to solve a bug? How does grammar correctness influence the communication friction?

In this research paper, we present and discuss our findings after running a quantitative data analysis on over 300,000 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-complete*, which we define as the *metric of quality* (*time-to-complete* represents the period of time between the creation and the closing of a ticket). There are multiple novel aspects brought by our study, among which is the fact that it is one of the very few studies that have tackled this issue using a quantitative rather than a qualitative approach (with a very large number of tickets analysed) and also that it uses state-of-the-art sentiment [9] as well as grammar correctness analysis techniques [9]. We believe that our results are valuable contributions to the research community and the database produced after the analysis can help further studies with analyzed and well-tested data.

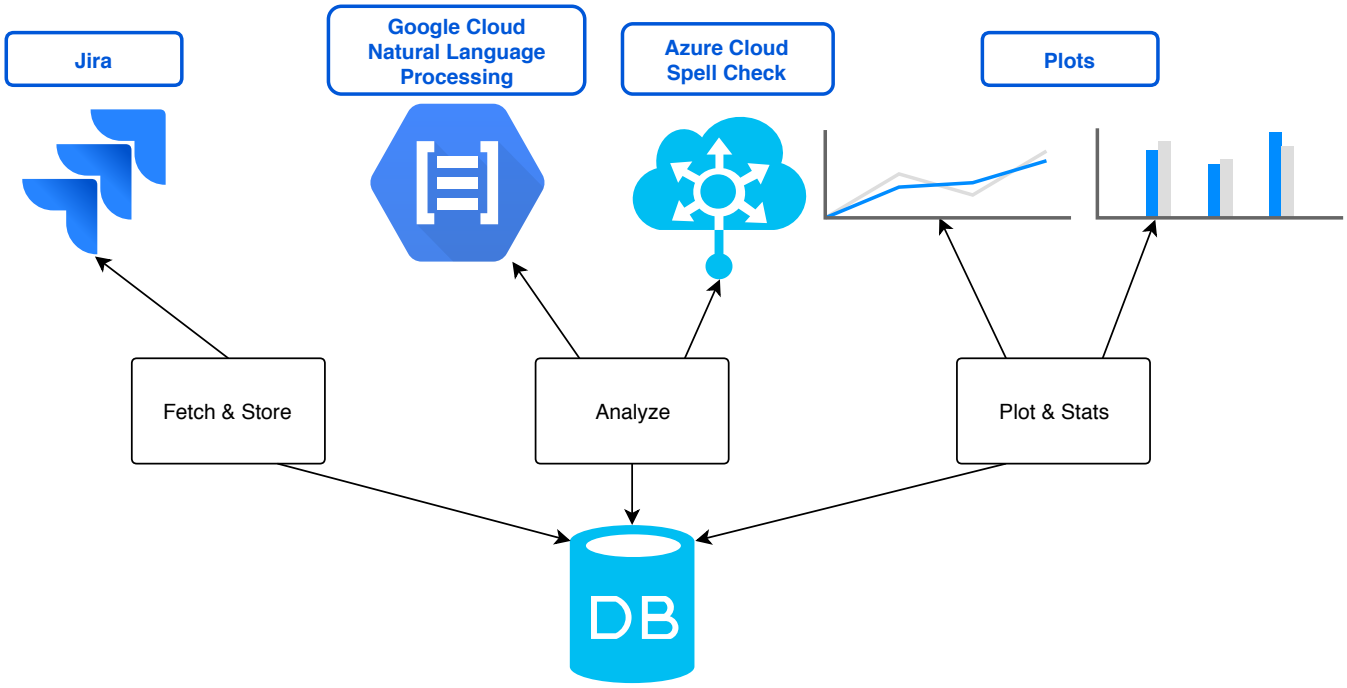

Figure 1: Application flow of the Go tool.

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 [3], 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 [7] 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 [2] 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 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.

3.3 Plot and Statistical Tests

4. CHARACTERISING THE DATA SET

5. CORRELATIONS

5.1 Attachments

5.2 Comments

5.3 Summary and Description

5.4 Grammar Correctness

5.5 Sentiment Analysis

5.6 Steps to Reproduce

5.7 Stack Traces

6. FUTURE WORK

7. CONCLUSIONS

Acknowledgments. Tim, my parents, Corina.

8. REFERENCES

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

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

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

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

[5] 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.

[6] 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.

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

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

[9] T. Wilson, J. Wiebe, and P. Hoffmann. Recognizing contextual polarity in phrase-level sentiment analysis. pages 347–354, 2005.

[10] 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.