

The Angry Coder

*A cursory examination of the correlation of commit message sentiment and bug-inducing commits*

Author: Brendan Robert

Class: CS515 Spring 2020

1. Research problem/question(s)

To err is human. To err severely during fits of rage is also very human. In layman’s terms it seems a simple question to affirm: Are strong emotions somehow accompanied by mistakes? Stated more succinctly, is it possible to show a correlation between developer moods and trends in observable code bugs? What kinds of metrics in MSR provide a possible statistical correlation? The goal of this project is to explore some possibilities, attempt an answer and more importantly pave the way for additional research to further explore this topic.

* 1. Motivation

If easily computable code metrics or user activity can be correlated to high likelihood of bug-inducing behavior, it might be also possible to offer developers suggestions at key moments to help avoid bugs, or to help affirm strong sentiments when developers fix bugs. If this is possible then we can detect these developers quickly and motivate them to contribute again in the future. This is important in projects where participation is generally voluntary and frequent contributors feel encouraged by the feeling that they are helping others – lack of acknowledgement or validation is a quick way to lose key contributors.

1. Methodology / Analysis approach
   1. High-level approach

The overall methodology ties together three different streams of data using three different tools (Perceval [Dueñas 2018], SentiStrength-SE [Islam 2018], and SZZUnleashed [Borg 2019]) and then ultimately combine the collected analysis into a single dataset subjected to linear regression for exploring the possibility of significant correlations. Though the hard work of the analysis is thanks in extremely large parts to those researchers who worked on those tools, it was also necessary to facilitate and convert data between the different programs and ultimately collate the results so that they were suitable for final analysis. That work is provided by a purpose-built tool named “Stanners” which is available in Github at the following URL: <https://github.com/badvision/Stanners>.

* 1. Project selection

The first goal was to identify a good target project suitable for study. For this I leveraged the GHTorrent [Gousios 2012] database, mirrored on Google Compute. This only required creating a Google Developer account and installing the Google cloud SDK, and then finally a project, albeit an empty one, so that I could set a default project for the bq tool to use locally. I was able to identify the most active Java projects on GitHub by determining which projects had the greatest number of pull requests, indicating projects which follow a stronger community model favoring communication and review as part of accepting code:

bq query --use\_legacy\_sql=false --format=prettyjson "select p.id as projectId, p.name, count (pr.id) prCount from ghtorrent-bq.ght.projects p LEFT JOIN ghtorrent-bq.ght.pull\_requests pr on (pr.base\_repo\_id = p.id) where p.language = 'Java' group by p.id, p.name order by prCount desc" > most\_base\_pr.json

The other point to consider was the number of commits total, which would offer more data points to analyze and hopefully eliminate some of the noise which would otherwise lead to low model cohesion.

bq query --use\_legacy\_sql=false --format=prettyjson "select p.id as projectId, p.name, count (c.id) commitCount from ghtorrent-bq.ght.projects p LEFT JOIN ghtorrent-bq.ght.commits c on (c.project\_id = p.id) where p.language = 'Java' group by p.id, p.name order by commitCount desc" > most\_commits.json

These data points were interesting but in JSON, I needed to quickly collate these in Excel so I converted both files to CSV:

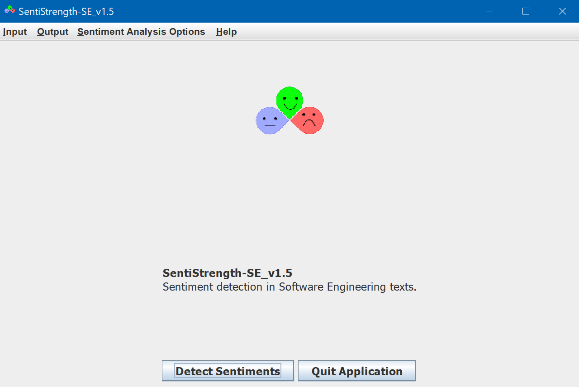
cat JSON\_FILENAME\_HERE | jq -r '(.[0] | keys\_unsorted) as $keys | ([$keys] + map([.[ $keys[] ]])) [] | @csv' > OUTPUT\_FILE\_HERE.CSV

|  |  |  |  |
| --- | --- | --- | --- |
| Project name | Pull Req | Commits | c/pr |
| hazelcast | 5160 | 15419 | 2.988178 |
| elasticsearch | 5444 | 16365 | 3.006062 |
| neo4j | 5916 | 18836 | 3.183908 |
| liferay-portal | 42033 | 140294 | 3.337711 |
| liferay-plugins | 5196 | 21025 | 4.046382 |
| openmicroscopy | 4743 | 25427 | 5.360953 |
| elasticsearch | 3468 | 19487 | 5.619089 |
| crate | 3121 | 17985 | 5.762576 |
| netty | 2655 | 15789 | 5.946893 |
| pentaho-kettle | 2737 | 16808 | 6.14103 |
| voltdb | 3806 | 26408 | 6.938518 |
| cdap | 5748 | 42859 | 7.456333 |
| jenkins | **2499** | **20514** | **8.208884** |
| WordPress-Android | 1962 | 16648 | 8.485219 |
| cloudstack | 1667 | 15323 | 9.191962 |
| mule | 3157 | 29589 | 9.372506 |
| repose | 1540 | 15089 | 9.798052 |
| Catroid | 1457 | 20608 | 14.14413 |
| JMRI | 1489 | 29577 | 19.86367 |

Figure - Most active Java projects on GitHub

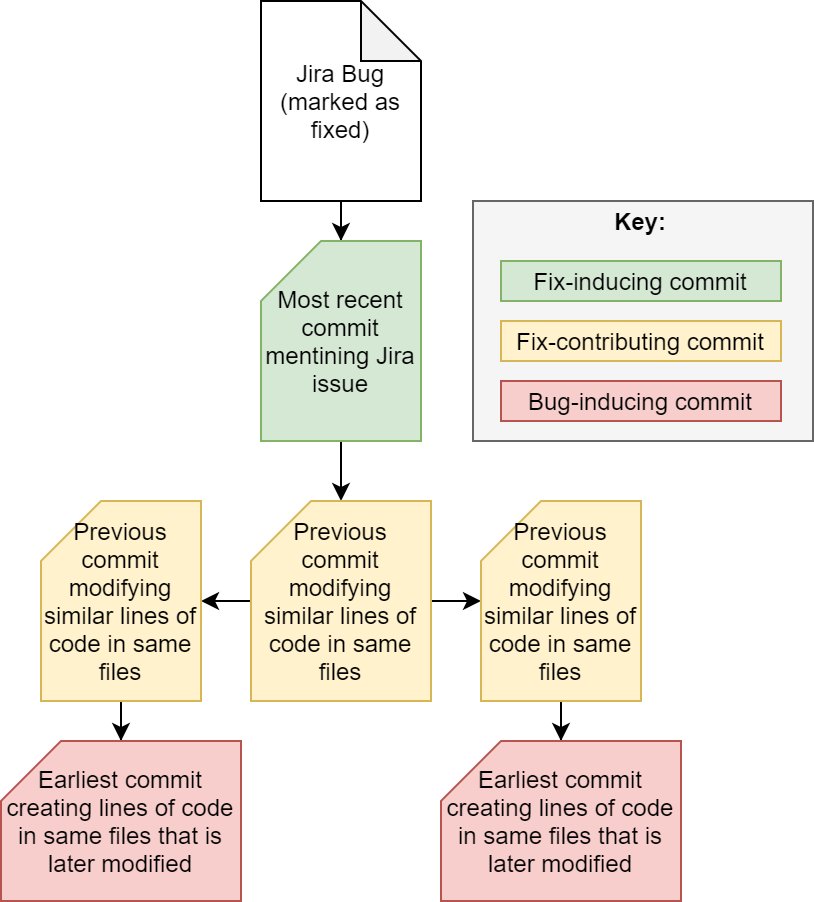
The projects closer to the overall average number of commits per pull request (**c/pr**) also seemed to have a high level of collaboration (based on number of contributors, not shown) and well-scoring sentiment values in the comments; particularly this is the case for Netty. Unfortunately, only Jira-based projects were ideal candidates for analysis, so I had to select a project with high commits and Jira usage, and primarily chose Liferay-Portal. This wound up being an overwhelming choice (explained in the conclusion) and I instead settled for Jenkins, largely because the SZZ Unleashed examples show Jenkins and I was forced to accept smaller data sizes due to constraints of SZZ Unleashed (again, discussed later in the closing remarks.)

Jenkins is a highly active project, so it was reasonable suitable for this analysis as well. Unfortunately, the ultimately much smaller data set allowed for quite a bit of noise because of the additional reduction of available data in the final analysis, explained later in the conclusions.

Novelli (et. all) [Novelli 2018] draw attention to the rising trend in sentiment analysis and highlight SentiStrength-SE [Islam 2018] as a contender for determining sentiment in technical texts. SentiStrength-SE [<https://laser.cs.uno.edu/Projects/Projects.html>] consumes a flat file consisting of text records in the format ‘{id}\t{text}’. The output is the same as the input with an additional two columns for positive sentiment on the scale of 1 (no effect) to a maximum of 5. The negative scale works the same way but in the negative range as well (so -1 = no effect, -5 is maximum negative score).

The only preprocessing required for SentiStrength-SE is to extract commit messages from the Git commit log and retain the git hash for merging with other data later. However, it seemed it could also be useful to collect other statistics for each commit such as total number of changed files and counts of lines added or removed. To simplify this, I used Perceval [<https://github.com/chaoss/grimoirelab-perceval> - Dueñas 2018] to process the Git log and produce a structured JSON representation of the Git commit history. Using a structured data feed also allowed pre-identifying which commits would be more relevant in the final data set. I filtered out commits which contained no java code or had more than 10 source files. This is because I wanted to avoid too much noise from large initial commits or project reorganization kinds of activities. This filtered set of commit messages were sent to SentiStrength-SE for processing and collated with the rest of the data during the final step.

Figure - SentiStrength-SE

The other factor to consider the tendency of bugs for each commit, such that the bug-inducing tendency can be weighed against the measured sentiment. My initial idea was to use a measure of code quality metrics, such as code smells and average coupling between each commit and its successor commit in the project history. Because this was difficult to automate, and even still much more difficult in terms of overall computational power required, I looked for other possible ways to infer bug proneness drawing inspiration and direction from Misirli, et. al [Misirli 2016] on identifying high-impact fix-inducing code changes. This directed me to discover more about the SZZ algorithm and I decided to explore that as an option.

Following the work of Śliwerski et al, it was matter of convenience to use an open interpretation of the SZZ algorithm [Śliwerski 2008] to measure the n umber of bug inducing commits, bug contributors (which are perhaps partial fixes in and of themselves), and finally bug fixes which are attributed to closed defects. This implementation was imperfectly provided by SZZ Unleashed [Borg 2019] which is found in Github at <https://github.com/wogscpar/SZZUnleashed>.

For each identified bug-fixing commit, SZZ Unleashed produces a record for each file in that commit, and a history of previous commits leading up to the fix, including multiple branches for different sections of code. Ultimately walking this graph reveals the initial bug-inducing commits (the leaves of the history graphs) as well as the contributing fixes in the code. In the final analysis step, the number of times a commit is identified in one of the three categories is tallied and applied a scoring rule to reduce the three values into one score for defect proneness, explained more in the next section.

Figure - Classification of commits using SSZUnleashed commit graphs

1. Mining Results

Data collection was very straight-forward, largely drawing on the Git to provide the raw data:

git log --raw --numstat --pretty=fuller --decorate=full --parents --reverse --topo-order -M -C -c --remotes=origin --all > jenkins\_gitlog.log

Perceval processed the git log into a more usable JSON format:

~/grimoirelab-perceval/bin/perceval git --git-log 'jenkins\_gitlog.log' .git > jenkins\_log.json

For SZZ, it was also necessary to mine the defects reported in Jira which have a fixed or closed status:

python3 fetch.py --issue-code JENKINS --jira-project issues.jenkins-ci.org

The other steps of running SZZ Unleashed follow the instructions provided in that project. It is worth noting that the java process required a lot of memory so adding an -Xmx parameter to raise the heap size was critical in a [mostly] successful execution, explained more in the conclusion.

After loading the Git commit log into Stanners, it was then possible to generate the needed data file for SentiStrength-SE to process. The output of this process was saved for final analysis.

The annotations generated by SZZUnlimited indicate for each commit matched to the closure of a bug (aka detected bug fixes) what other commits in the history were contributors to those bugs (in-between commits) or bug inducers (initial commits with no earlier history). The count of instances a commit was labeled for each of these three cases was collated with commit totals (number of files, number of lines added, number of lines removed) as well as the sentiment positive and negative scores.

The processed data was subjected to multiple linear regression using the bug proneness as the Y-value. Bug proneness is a combinatory score based on adding the follow three values:

* number of defects \* -10
* number of contributing \* -0.941
* number of fixes \* 10

The three factors were combined with weights such that adding all factors for all data points reaches a zero-sum, so as to achieve a balance between the negative and positive aspects of the data. This was mainly to help ensure that contributing commits are weighed against other activities in a way such that additional churn (bug-contributing) is considered slightly negative whereas bug-fixing is considered just as strong as bug-inducing in the scoring scale.

The regression analysis was provided by the stats.blue website calculator: <http://stats.blue/Stats_Suite/multiple_linear_regression_calculator.html>

Machine generated alternative text:
- 1.2558 • Pos - 1.2677 • Neg 
Model: 
Score 
0.3624 - 0.0321 • Files 
Predictor 
Constant 
Files 
Lines Added 
Lines Removed 
Pos 
Neg 
R-Squared: 
0.0027 • Lines Added 
— 0.0072 • 
Lines Removed 
Coefficient 
Estimate 
0.3624 
-0.0321 
-0.0027 
-0.0072 
-12558 
-1.2677 
Standard Error 
1258 
0.0178 
0.0005 
0.0015 
1.1027 
0.5949 
t-statistic 
0.2881 
-18054 
-5.563 
-4.6645 
-1.1388 
-2.131 
p-value 
0.7733 
0.071 
0.2548 
0.0331 
Adjusted R-Squared: 
Residual Standard Error: 
Overall F-statistic: 
Overall p-value: 
Source 
Regression 
Residual Error 
Total 
Summary of Overall Fit 
2 
adj 
22.4024 on 17364 degrees of freedom, 
218223 on 5 and 17364 degrees of freedom, 
Analysis of Variance Table 
df 
5 
17364 
17369 
54759.5967 
8714452.5052 
8769212.1019 
MS 
10951.9193 
5011869 
504.8772 
F-statistic 
2118223 
p-value 

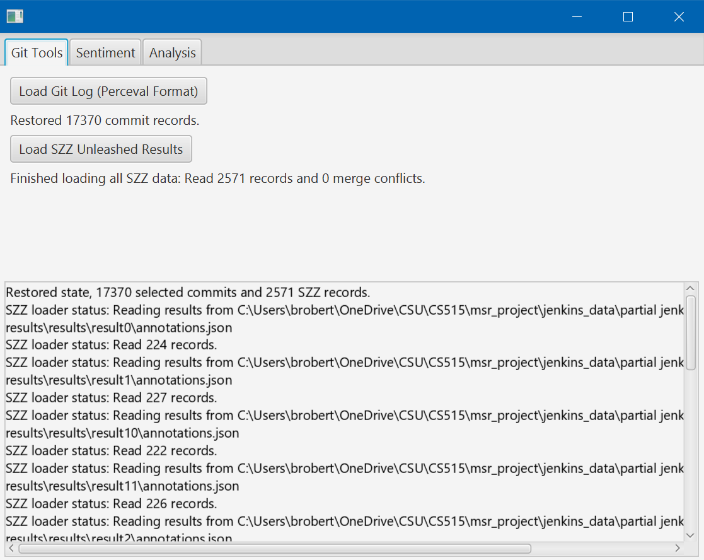
Figure - Results of multiple linear regression

Lines added and Lines removed both show a high significance (p=0) to the measure of code bug proneness, with line removal being a stronger indicator than line addition by over 2.6 times. However, given the coefficients of both, the overall impact is relatively low. The number of files in a commit have a low significance in overall effect of bug inducing behavior (p = 0.07).

1. Answer to research questions

The hypothesis variables are measured in the Positive and Negative sentiment values, which are measured separately because as SentiStrength explains these concepts can mutually co-exist. The initial hypothesis was to look for a trend in sentiment corelating with defect proneness (which to say a score below zero indicating trending towards bugs rather than fixes.) In this model, that correlation appears with some statistical significance with respect to negative sentiment (p=0.03) but unfortunately the model is not well-fit to explain much of the variance in the data; R2 is a mere 0.006. There is low statistical value in the effect of positive sentiment on defect proneness. (p=0.25)

1. Conclusions

There appears to be some measurable statistical significance in the correlation of negative sentiment in commit messages to the defect proneness of the sample commits studied, but ultimately a much larger and more complete data set would be required to improve the R2 value of the model and offer a more complete analysis.

The Stanners app is limited to the confines of its limited purpose-built existence, but it has a lot to offer for those looking to build a larger application which understands the various data formats employed by these other tools. As JSON is quite commonly used, I created a set of model beans which support direct GSON deserialization, making it easy to read the data in from the other programs and perform lightning-fast post processing. It also saves its own state regularly, allowing terminating and loading the program later on.

Figure - Stanners

1. Closing remarks
   1. Areas of improvement

There are areas of opportunity that could greatly improve the model used for this analysis:

* SZZ Unleashed needs a lot of optimization to analyze large code projects, this and other issues are explained in the last section.
* Alternatively, defect proneness is a rather subjective measure and instead a more computationally intensive, yet more consistent, metric would be static code analysis based approaches such as SonarQube.
* The Jenkins project only reports a 17.5% match of issues to bug-fixes, which means a lot of commits were not factored well into the resulting data. Liferay Portal, on the other hand, was a staggering 83% match on initial preprocessing but later steps were unable to complete.
* More though and exploration for the code scoring metric could be improved, and ultimately the scoring rule used is arguably a zero-sum outcome that could be construed to mean “In the end we’re no better than where we started” which is arguably unfair. However, without a lot of context of the in-between contributing commits, their overall effect on the bug proneness scoring is 10 times lower than the bug inducers and bug fixes to minimize the overall impact on the defect proneness scoring altogether.
* Additional developer venting might be revealed in code comments, and a more complete analysis would also include located developer comments for each commit.
  1. Further exploration

If a suitable data set with high correlation is identified, then a future possibility is training a recognizer against smaller developer behaviors such as frequency of code edits and deletes within a file (e.g. within the IDE) and measure those activities against error proneness. With a trained model, it can then be used by IDE plugins to help raise developers’ awareness that they are exhibiting defect-inducing behavior by the nature of their work and offer meditative exercises or other diversions to break the tension and lower the likelihood of additional defect-prone inducing behavior.

Inversely, there also were some observed trends against a more incomplete set of data that seemed to indicate a possibility that negative sentiment is strongly present in bug fixes, such as developers comments such as “I finally fixed this \*expletive\* bug” which inevitably scores a high negative sentiment. In such cases, it might be possible to look for additional correlations of fix-confirming behavior by the strength of the language in the commit message. One useful outcome of this kind of detector is to help a community project boost the contributors helping maintain the source by fixing bugs, so such a detector can help the core maintainers become more aware of the contributions of others.

* 1. The unwritten fine print on SZZ Unleashed

SZZ Unleashed provided a valuable amount of analysis required for this project to succeed. However, it is very unoptimized and needs a lot of performance improvement.

* The pre-processing scripts are written in Python and are not very fast. Python only uses one CPU core. For Jenkins, the scripts took on average 20 minutes or so. However, for Liferay-Portal the pre-processing took closer to 22 hours, offering little to no feedback (other than a counting number with no indicator of when it would finish.)
* Many of the pre-processing steps have an O(C\*D) amount of processing (C being number of commits, D being number of defects.) In addition, for each of these, there are many of the same operation running repeatedly, adding a lot of time to the processing required.
* Additional pre-processing scripts should be provided to mine defects reported in other systems such as Github issues or Bugzilla. For example, Netty uses Github issues and has a significantly better-suited pattern of commit messages that lend themselves to a wider range of sentiment for study.
* The SZZ Unleashed implementation is extremely memory-hungry and likely has either a cubic order of magnitude (C\*D\*L where L is number of lines of code total) or possibly even parametric. For Liferay-Portal, the process consumed 38gb of ram right away and ran very slowly, eventually having to be aborted after 32 hours of not producing any viable output at all. For Jenkins, the program only consumed 6gb of ram, so was at least not memory-constrained – it did however hang after running for a few hours but fortunately produced a fair amount of data useful for completing this initial cursory analysis.
* SZZ Unleashed repeats the same data across threads. This makes post-processing more complex because additional work is required to interleave the different output files and clean up the mess. This also indicates a high likelihood of resources leaking across threads and possible thread safety issues. It could also be an indicator that the same work is being performed across multiple threads in duplicate causing a lot of extra re-work.
* Long-running programs should offer the option of logging activity semi-frequently to log files using timestamps to help users understand if programs are still actively running, and if they are getting any closer to completion.

1. References

[Gousios 2012]

Gousios, G. and Spinellis, D. “GHTorrent: GitHub's Data from a Firehose” MSR 2012: Proceedings of the 9th Working Conference on Mining Software Repositories. Jun 2012, pp. 12-21

[Novelli 2018]

N. Novielli, D. Girardi and F. Lanubile, "A Benchmark Study on Sentiment Analysis for Software Engineering Research," 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), Gothenburg, 2018, pp. 364-375.

[Islam 2018]

Md Rakibul Islam, Minhaz F. Zibran, "SentiStrength-SE: Exploiting domain specificity for improved sentiment analysis in software engineering text”, Journal of Systems and Software, Volume 145, 2018, PP 125-146

[Dueñas 2018]

Dueñas, S., Cosentino, V., Robles, G., & Gonzalez-Barahona, J. M. (2018, May). Perceval: software project data at your will. In Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (pp. 1-4). ACM.

[Misirli 2016]

Misirli, A.T., Shihab, E. & Kamei, Y. Studying high impact fix-inducing changes. Empir Software Eng 21, 605–641 (2016). <https://doi-org.ezproxy2.library.colostate.edu/10.1007/s10664-015-9370-z>

[Śliwerski 2008]

Śliwerski J, Zimmermann T, Zeller A (2005) “When do changes induce fixes?” In: Proceedings of Int’l Workshop on Mining Software Repositories (MSR), pp 1–5

[Borg 2019]

Markus Borg, Oscar Svensson, Kristian Berg, Daniel Hansson “SZZ Unleashed: An Open Implementation of the SZZ Algorithm -- Featuring Example Usage in a Study of Just-in-Time Bug Prediction for the Jenkins Project.” In: In Proc. of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE'19), pp. 7-12, August 27, 2019, Tallinn, Estonia