Sentiment Analysis of Commit Comments in GitHub: An Empirical Study

Team Composition

Andrew Berg
FSUID: aab13j

Raidel Hernandez
FSUID: rh13k

Tyler Kelly
FSUID: tck13

GitHub Repo

https://github.com/raidel123/GitSentiment

1. Introduction

In *Sentiment Analysis of Commit Comments in GitHub*, the researchers were conducting research by performing sentiment analysis on GitHub commit comments from the MSR 2014 Data Dump. The data that this dataset composes is the GitHub projects and all associated information with those. The point of interest for this research was to find correlation from the commit comment sentiment score calculated by SentiStrength and then various other data points selected from the dataset. The goal was to establish the relationship if it exists between the used programming language, the time and day of the week in which the collaboration occurred, team distribution and project approval. In our replication and addition of the additional research question, we attempted to follow the guidelines and example set in the paper as closely as possible. As a group, we did see some minor differences in some of the result but we suspect this due to slightly different datasets since we followed the research as closely as possible with the given information. In our analysis, we used the MSR 2014 MySQL torrent from http://ghtorrent.org/msr14.html as instructed but the project did not explicitly say what data they used in their evaluation. We used various tools in the analysis of the data which will be detailed in our methodology section. This project was very interesting and provided a great learning experience for us all for working with this type of data.

Research Questions:

1. Are emotions in commit comments related to the programming language in which a project is developed?
2. Are emotions in commit comments related to the day of the week or time in which the commits were written?
3. Are emotions in commit comments related to the team geographical distribution?
4. Are emotions in commit comments related to project approval?

5. Additional Research Question: What will a users sentiment on a git commit messages reveal about the quality of their personal open source projects?

2. Methodology

2.1 Data Retrieval and Manipulation

In regards to the given dataset and the established research questions, this research paper was very data centric. So where we started was with establishing a strong core data dump built from the MSR 2014 MySQL Data Dump. We also used the SentiStrength[1] Java client to do our sentiment analysis to get the closest results possible to the previously established results by the original researchers. This SentiStrength client was not the most simple application to use and required to request the creator of SentiStrength to approve usage for academic research. This was done by email so this is how the whole process started. In hindsight, there are more popular frameworks for sentiment analysis that could have possibly used but we restricted ourselves to SentiStrength due to previous researchers using this. Now we will detail the extensive process of gathering the data and storing this data into the table that we used to analyze the results along with the sentiment analysis portion. All of the programming for this portion was SQL and Python 2.7.14. The packages that we used were *MySQLdb*, *json*, *SQLite3*, and *sys*.

Initially, we installed a local MySQL server to MacOS. This is where we imported the MSR 2014 data dump into. This allowed us to freely query the database at will. One of the stranger parts of this project is that the MSR 2014 website that provided the download link to the data set and also a schema. This schema, however, was incorrect on a few accounts. When doing the extraction of the data points into our personal table, we found that the *users* table was supposed to have columns for *city, state,* and *country_code*. These were all important points of data for our analysis of one of our research questions but to our surprise, these columns were not even in the dataset. The column that did exist however was *location*. The column *location* seemed to be whatever the GitHub user decided they wanted their location to be, even if that is “InterWebs” which was one of the more odd examples. After we had imported this data into our personal MySQL server we began breaking down the points that we needed from the dataset.

[1] http://sentistrength.wlv.ac.uk/

After we established what information we needed from the database, we came up with a list of points, which came out to:

Field	Reasoning
commit_comment_id	Used as ID to identify the individual comment
commit_comment_body	Used for sentiment analysis with SentiStrength
created_at	Used for research question regarding time and day of the week
commit_sha	Used for gathering user repository data for given user for additional research question
project_language	Used for research question regarding project language and comment sentiment score correlation
commenter_email	Used as backup for identification of user
commenter_github_login	Used to find the user's repository data
commenter_location	Used to find continent of the user for research question regarding regions

After viewing the schema and breaking down our actual needs from the data dump, we built a single SQL statement that could retrieve all of this data. This SQL statement was run through the Python MySQL client and then we processed the returned rows with Python. This provided for small, single purpose scripts to achieve the process of capturing the data we needed from the large dump. The SQL statement that we ran against the MSR 2014 data dump is as follows:

```sql
SELECT commit_comments.id,
                    body,
                    commit_comments.created_at,
                    sha,
                    projects.name,
                    projects.language,
                    users.email,
                    users.login,
                    users.location
                       FROM commit_comments
          INNER JOIN commits on commit_comments.id = commits.id
          INNER JOIN projects on commits.project_id = projects.id
          INNER JOIN users on commit_comments.user_id = users.id
```

This SQL was lengthy but the basic idea is that it takes the commit_comments table as the starting point and then using *inner joins* we were able to combine all the data into one SQL statement instead of having to query the data more than once. This data was then fetched from the database in Python and then iterated over. We built a db_utils Python module that automatically connected to the MySQL database and ran the query passed as a parameter.

We then chose to use a SQLite database for ease of distribution between the group. We created a database file with this schema:

```python
import sqlite3 as sqlite

conn = sqlite.connect('./whole_database_new.db')
print("Opened database successfully!")

conn.execute("""CREATE TABLE
                commit_sentiments (commit_comment_id int,
                                   commit_comment_body int,
                                   created_at timestamp,
                                   commit_sha varchar(40),
                                   project_name varchar(255),
                                   project_language varchar(255),
                                   commenter_email varchar(255),
                                   commenter_login varchar(255),
                                   location varchar(255),
                                   sentiment_pos int,
                                   sentiment_neg int)""")

conn.execute("""CREATE TABLE IF NOT EXISTS
                commit_sentiments_store (commit_comment_id int,
                                         commit_comment_body varchar(255),
                                         sentiment_pos int,
                                         sentiment_neg int)""")

print("Table created successfully!")

conn.close()
```

As we can see above this includes all of the needed tables and columns in the respective table. With this we are able to collect all the data in the needed SQLite database: *whole_database_new.db*.

We will now detail the process of getting the sentiment based on the commit comments stored in MSR data set. First, we created a SQL statement as follows to retrieve the needed columns:

```
SELECT commit_comments.id, body FROM commit_comments
```

This SELECT statement queried our MSR database and returned the commit_comments id and body. With this we were able to start the process of analyzing the positive and negative sentiment, then storing this in our sentiment table in the SQLite database. The SentiStrength was run through the subprocess python library which allows us to run command line programs, which we ran the SentiStrength jar file provided by the earlier mentioned link. We then broke up the output of the sentiment analysis in the *senti_strength_module*. This returned a positive and negative score that we stored in the *commit_sentiments_store* table along with the *commit_comment_id*, *commit_comment_body*, *sentiment_pos*, and *sentiment_neg*. We stored the above mentioned information with the SQL command as follows:

```
cur.execute("""INSERT INTO commit_sentiments_store
    (commit_comment_id, commit_comment_body, sentiment_pos, sentiment_neg) VALUES (?, ?, ?, ?)""",
    (ccid, body, pos, neg))
```

On the system in which we ran this portion, this was a computationally intense process. The process took around ~12 hours to run the whole sentiment analysis on ~60k comments. The SentiStrength sentiment analysis takes about 1 second for the average commit comment.

Finally, we took this table and effectively merged it with SQL statement with the inner joins above to create the final database that was distributed to the group to perform the necessary analysis. The statement we used for this is as follows:

```
with sqlite.connect('./whole_database_new.db') as con:
    con.text_factory = str
    cur = con.cursor() # get the current spot for executing
    cur.execute("""INSERT INTO commit_sentiments
        (commit_comment_id,
         commit_comment_body,
         created_at,
         commit_sha,
         project_name,
         project_language,
         commenter_email,
         commenter_login,
         location,
         sentiment_pos,
         sentiment_neg)
         VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)""",
        (ccid, body, created_at, sha, project_name, language, email, login, loc, pos, neg)) # execute insert to add the score and name
    con.commit() # commit the query
```

Now that we have all of the data in one database, we are able to pass this data along to the rest of the group such that they do not have to run complex queries for any of their interaction with the actual data dump. This allowed for small queries and fast query times and the rest of the group not having to setup a local SQL server and mess with schema and complex table structure. It also confirmed the data that we actually had and what we would have to retrieve from other sources. We will see how we used the actual database in subsequent sections of our methodology.

2.2 Replicating the Results

To replicate the methodology used in the paper to obtain similar results, we used python, along with pandas for data querying and transformation. Transformations need to be applied to the queries once obtained to parse many fields such as *created_at,* to obtain the day of the week and the time of the day from the unicode string. The DateTime library was then used to obtain the properly format the data, and store it, as well as query it for information such as: the day of the week it was committed. Another very helpful and powerful library utilized was scipy, obtaining statistical information with scipy is very simple and intuitive. Lastly, once the data has been grouped into the proper subset, we use MatPlotLib to display the graphs produced in the outputs. The functionality for most graphs produced involved the same procedure described above, along querying the database of emotion values and obtaining the relevant fields required to answer our research question. An example query using Pandas can be found below:

df = pd.read_sql_query("SELECT project_name, sentiment_pos, sentiment_neg FROM commit_sentiments;", db_connection)

2.3 Additional Research Component

The additional research component used the aforementioned commenter sentiment data, in conjunction with the commenters average code quality. To do this,

we needed to analyze a user's GitHub account, and then explore their public repositories, analyzing code as we go. To access a user's programs, we needed some way of interacting with GitHub's data. While the MSR dumps are useful for getting top level metadata, we needed to examine all the public files a user has committed to their personal projects. In order to do this we need to use GitHub's Data API so that we can access this sort of information, and we found a python wrapper for this API called pyGithub[2] which allows us to succinctly access all public data about a GitHub user. In order to access the API, we need to provide either our own GitHub login credentials, or a personal access token which can be accessed from your profile page. This is needed to authenticate your request, and also for our purposes, is needed to set a rate limit which we will mention shortly. After the API request has been authenticated, we can then resume our work analyzing the user's data.

There exists a database in the repository which contains many information about a commenter. For the purpose of the additional research component, we only need the commenter_login, pos_sentiment, and neg_sentiment. So to obtain this information from the database, we use a database class wrapper[3] for SQLite to better implement our procedures. Modifications were made to better use in this project such as query parameterization, custom procedure when opening a database file, and implementing SQLite's row factory property for better table to object instantiation. After this Database class was finalized, we can go ahead and access this commenter data, where we simply use each row's data to instantiate a class of type User. This User class allows us to better work with the database data in our python code so that we can perform simple arithmetic, comparison, and updates/deletes in a temporary environment (preserving the integrity of the database). At this point we have a list of User objects representing all the rows in the database, we now need to access this User's GitHub information.

To do this we can perform a simple API request with pyGitHub to turn our User's username attribute into a full blown pyGitHub User class which allows us to access many facets of the user with simple member function calls. For example, we can access all of a User's repos with a call such as user.get_repos(), and then we can access all of the directory contents with user.get_repo().get_dir_contents('/'). The last statement has a bit of a problem however, as .get_dir_contents('/') will only access the top level directory. Any files within subdirectories are ignored. While some work was gone into traversing the entire repository, subdirectories and all, we realized that the computational power needed for this is beyond the scope of this project. Needing to

[2] http://pygithub.readthedocs.io/en/latest/
[3] https://gist.github.com/goldsborough/c973d934f620e16678bf

access every single file of every single repository of thousands of users is just not feasible. At this point, we can access a GitHub users public repositories and top level files, but we still need a way to analyze and evaluate the user's code.

Analyzing code quality is a nontrivial task. Some naive approaches were explored at first such as simply analyzing the ratio of comments to code, or the complexity of a Readme, but neither seemed universal enough to justify accurate measures in our experiment. After a bit more research, it was decided to use linters for our static analysis. Linters are programs which statically analyze code based on a language's style guidelines. These guidelines are not necessarily concrete, or accepted by everyone, but they provide a good indication as to whether or not code has been written well, which suits are needs. Linters are, obviously, unique to each language, there is no universal linter. So we need to find some quality linters which we can ideally run within our project. The most obvious starting point was to use Python's popular linter pylint[4], as it provides a python module which we can include in our script to analyze code. Pylint (which adheres to PEP8) also provides us with a quality score, which ranges from -50, to 10. After some more research, we found some command line linters which would make it so we could analyze languages within our program by accessing the subprocess module. Namely we used, cpplint[5], Excellent[6], and Standard[7] which work on C++, JavaScript, and Ruby respectively. Cpplint (adheres to Google's C++ style guidelines) is actually a 6200 line python script provided by Google which analyzes a given cpp file and outputs all the style guideline warnings. Standard (adheres to StandardJS) is a npm module which analyzes JavaScript files, and outputs all the style warnings, and Excellent is a ruby gem which outputs style warnings in some ruby code.

While pylint provides a score, the other three linters do not. So we decided to get the ratio of warnings to lines of code in a file. While there are possibly better metrics to use here, our approach is simple, and if all results are judged this way for a specific language, we should lose any ambiguity with enough results. At this point we have all of a user's publicly committed files that exist at the root directory, and we have a way to determine the quality of a given program in our set of languages, now we just need to combine the two.

GitHub provides a tag on most repositories detailing the most common language in that repository. For example, if you are building a Django web application, the GitHub

[4] https://www.pylint.org
[5] https://github.com/google/styleguide/tree/gh-pages/cpplint
[6] https://github.com/simplabs/excellent
[7] https://github.com/standard/standard

repository language tag for this application would be Python. You may have some bash scripts, or some yaml configuration files, but for the most part, you are dealing with a Python project. To reduce overhead, and to possibly provide better insight into a developer's skill, we strictly only analyze a user's most used language repositories. Meaning, if a user has five Python projects, and three JavaScript repositories, we simply only examine those five Python projects. By only examining these repositories we limit our use of API requests (which is limited to 6000 per hour), and we also only examine the code which is written in the language a particular user prefers, and as such they may be more familiar with the proper style for that language. We can now loop over all repositories belonging to a user, and if that repositories tagged language matches the most common language for that user, we analyze the top level contents of that repository, exclusively examining files which end in the proper file extension matching a particular language.

So we have an individual user, where we have analyzed their code and gotten a tangible score back which we average together for a particular user. So say Bob has 5 repositories tagged as Python, we examine these five repositories by extracting all files that end in '.py' from these repo's and analyze each one using the previously mentioned strategy, and average out this score for the user. We then add this user to some results SQLite database for permanence.

Before mentioning the final component to our additional research component, we do need to mention some limitations of the above procedure. Because we are adhering to GitHub's API guidelines, we cannot make more than 6000 request per hour, this is especially limiting because each user we examine has a fair amount of request made ranging anywhere from five to a few hundred (each file access is considered a request). Averaging out to about 35 users processed per hour before we need to wait and examine the next batch. Some amount of work was done to limit the amount of requests per user, and we also needed to cut down on computation time as the linters and subprocess calls got unwieldy when files were too large, so we restrict files that are deemed to be too large or repositories which consist of too many files.

Lastly, we need to analyze the result in our results database so that we can calculate some correlation. By first connecting to this results database mentioned previously using the database wrapper, and converting all rows to objects of class Users for easier manipulation, we have our starting data. Each User object in our list of users has a language attribute which is set to be the user's most used language. By

grouping our results based on language and using the pearson correlation coefficient[8] we can find the correlation between sentiment and code quality for a particular language as well as show the distribution for each with matplotlib[9].

3. Results

This section will focus on presenting your results and also on contrasting your results to the ones obtained in the paper. If your results are different, please write down any hypotheses you may have on why they may be different.

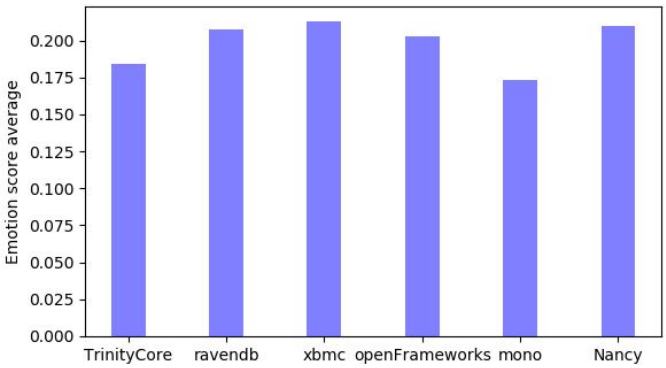
As we will see, the results we obtained are different then those from the paper assigned to us. In the assignment write up, we were assigned the MSR ‘14 data set, however we recently discovered the paper actually used seemed to use a different data set, possibly an earlier version of the GHTorrent MySQL database we used. At the point we found this inconsistency we had already build the emotion database, and subjected the commit messages to many hours of processing by sentistrength. Therefore, we decided to continue our study with the MSR ‘14 data set. Our results are obtained by applying similar techniques, discussed in the paper, for each section, unless otherwise stated.

This difference is clear to see in the results because the summation of a simple statistic from our data dump like the amount of commit_comment’s programming language is different. As follows this is the table from the research paper:

Language	Commits	Mean	Stand. Dev.
C	6257	0.023	1.716
C++	16930	0.017	1.725
Java	4713	-0.144	1.736
Python	2128	-0.018	1.711
Ruby	15257	0.002	1.714

[8] https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.stats.pearsonr.html
[9] https://matplotlib.org/api/_as_gen/matplotlib.pyplot.boxplot.html

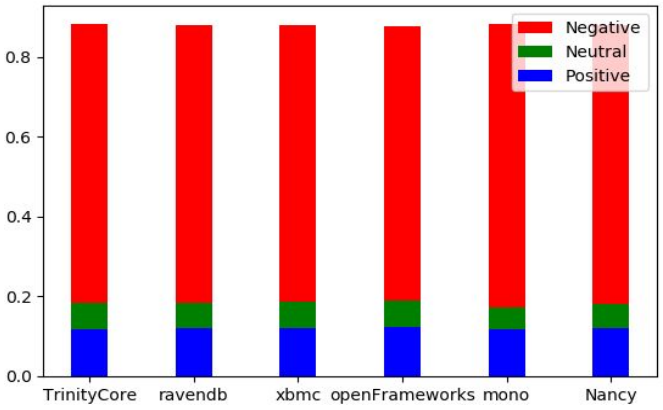
3.1 *Emotions in Commit Comments*



Project	Commits:	Mean	Stand. Dev.	p-value
TrinityCore	13179	0.184	1.099	< 0.002
ravendb	7790	0.208	1.085	< 0.002
xbmc	5796	0.213	1.089	< 0.002
openFrameworks	5374	0.203	1.133	< 0.002
mono	3873	0.173	1.066	< 0.002
Nancy	3227	0.209	1.070	< 0.002

Fig 1. Emotion Score Average Per Project

For this comparison we analyzed the emotions in commit comments for 48 separate projects. Much we like the results obtained by the original paper, the average emotion score for commit comment for each project tended to be neutral (between -1 and 1). As we can see in Fig .1, we grouped commits by project and the top 6 projects, in terms of overall total number of commits, and obtained a statistical analysis of the results. In Fig 2, we can see the emotion distribution , which allows us to compare the overall emotions present in commit comments. In most projects there tend to be a large number of negative commit comments, however the mean emotion is a weak positive value (very small positive value) for most projects.



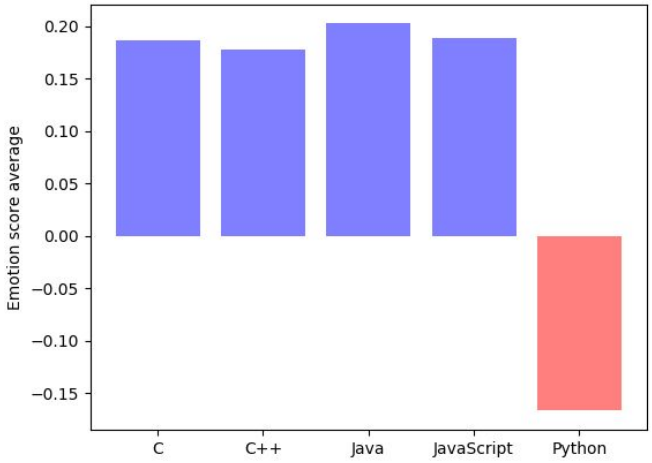
Project	Negative	Neutral	Positive
TrinityCore	0.701	0.182	0.117
ravendb	0.696	0.183	0.121
xbmc	0.693	0.188	0.120
openFrameworks	0.686	0.190	0.124
mono	0.712	0.171	0.116
Nancy	0.697	0.182	0.121

Fig 2. Proportion of average positive, neutral and negative commit comments per project

3.2 *Emotion and Programming Language*

Compared to the MSR ‘13 data set which contained 14 different programming languages among the projects analyzed, the data we used from MSR ‘14 contains projects in 11 different languages. To find the average emotion for each programming

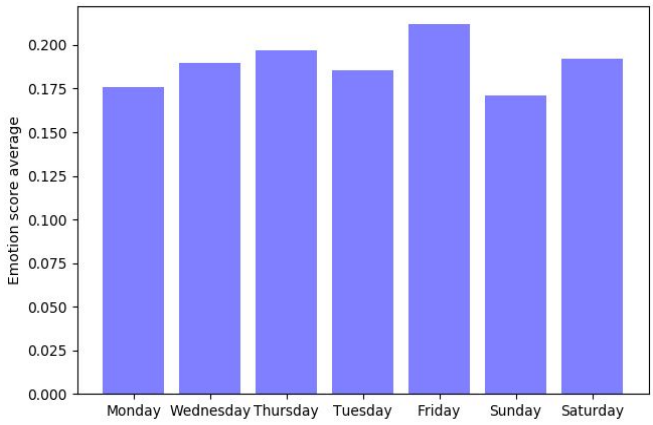
language, we grouped together all commits belonging to each language. Then we applied pythons scipy package, for statistical analysis of the data, as shown in Fig 3. In the paper, Java tended to be the most negative language, although not by much. In our findings, Java was actually the language with most positive commits commits on average. The languages shown in the graph were selected to be the same as the ones used in the original study, Ruby being the only exception, as it was not used in any of the project in the MSR '14 data set. Therefore, in place of ruby we decided to use another popular programming language, JavaScript. Overall, most commits have positive averages, except a few including python. We also used a Wilcoxon rank sum to confirm out results, the p-value obtained can be seen in Fig, 3 as well (p-value<=0.002).



Project	Commits:	Mean	Stand. Dev.	p-value
C	14868	0.186	1.107	< 0.002
C++	13592	0.178	1.100	< 0.002
Java	7188	0.203	1.056	< 0.002
JavaScript	924	0.188	1.234	< 0.002
Python	6	-0.167	0.753	0.564

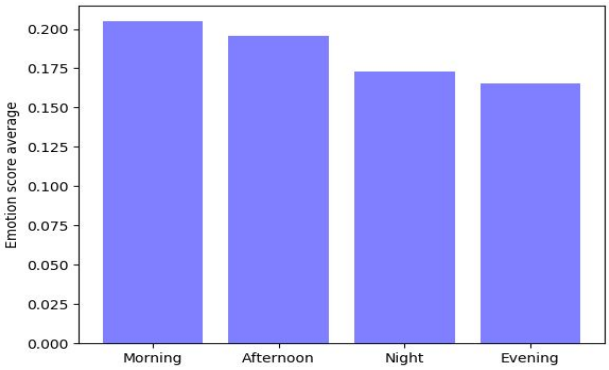
Fig 3. Emotion score average grouped by programming language

3.3 Emotions, Day and Time of the Week



Project	Commits:	Mean	Stand. Dev.	p-value
Monday	9426	0.176	1.101	< 0.002
Wednesday	9394	0.190	1.085	< 0.002
Thursday	9372	0.197	1.110	< 0.002
Tuesday	9248	0.186	1.085	< 0.002
Friday	8533	0.212	1.095	< 0.002
Sunday	6561	0.171	1.083	< 0.002
Saturday	6476	0.192	1.074	< 0.002

Fig 4. Emotion score average of commit comments grouped by weekday



Project	Commits:	Mean	Stand. Dev.	p-value
Morning	19284	0.205	1.090	< 0.002
Afternoon	18504	0.195	1.099	< 0.002
Night	13783	0.173	1.071	< 0.002
Evening	7439	0.165	1.117	< 0.002

Fig 5. Emotion score average of commit comments grouped by time of the day

One surprising result, was observed when we grouped commit comments by both day of the week and time of the day in which the commits took place. Both the paper assigned to us, and our results show that *Mondays* are the day of the week with the least average of positive emotions in commit comment, as shown in Fig. 4. It also shows that *Mornings* is the time of the day that contained the least average of positive emotions of all times of the day, as shown in Fig. 6. Similarly to the paper assigned, we separated the emotions of commit comment for time of day by assigning the following times: morning [6:00-12:00), afternoon [12:00-18:00), evening [18:00-23:00), night [23:00-6:00). Like the paper we also performed a Wilcoxon rank sum and found no significant differences between the times of the day commited. However, as previously stated we obtained similar results, where we both concur that Mornings are the best time of day to commit, and Evenings are the worst time of day.

3.4 Additional Research Component

For the additional research component, we got the following results on our dataset:

Language	**Sample Size**	**Pearson Correlation Coefficient**	**p-value**
Python	74	0.00410	0.972
C++	13	0.162	0.595
JavaScript	185	0.00599	0.935
Ruby	84	-0.0694	0.531

It should be noted that our dataset was just a subset of the MSR data. Here is the distribution information

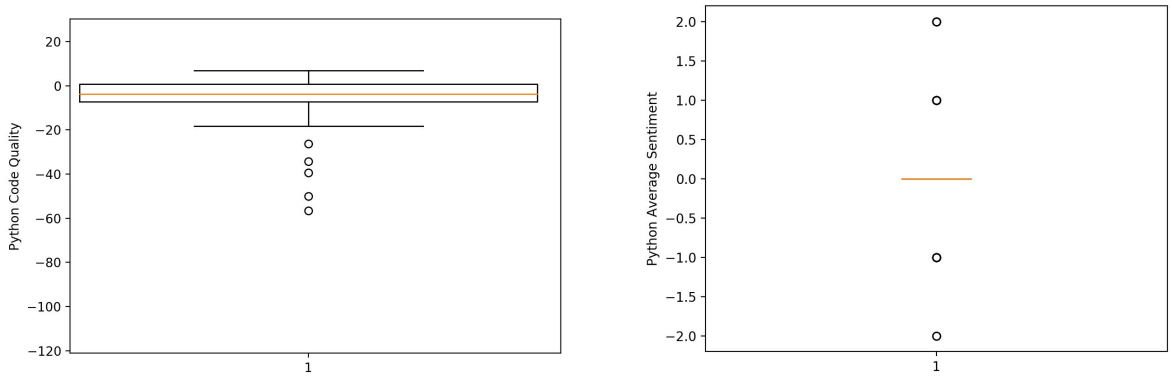


Fig 6. Distribution of Python code quality per user and average sentiment per user

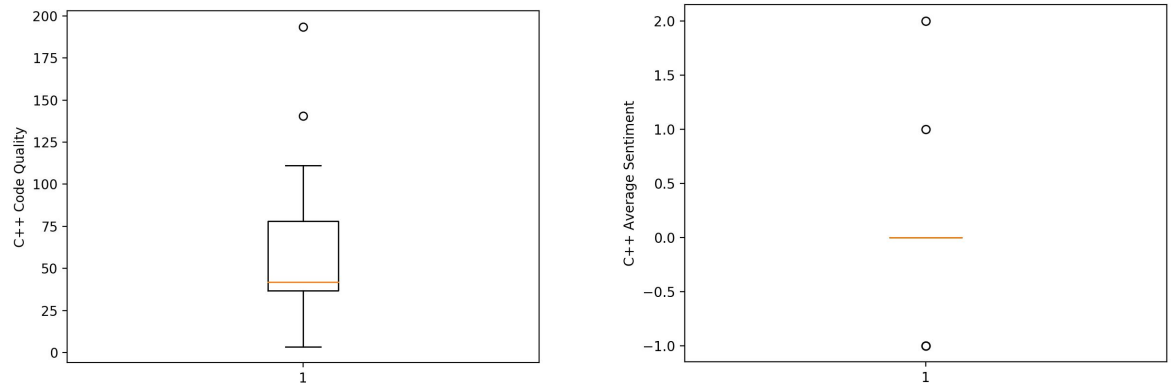


Fig 7. Distribution of C++ code quality per user and average sentiment per user

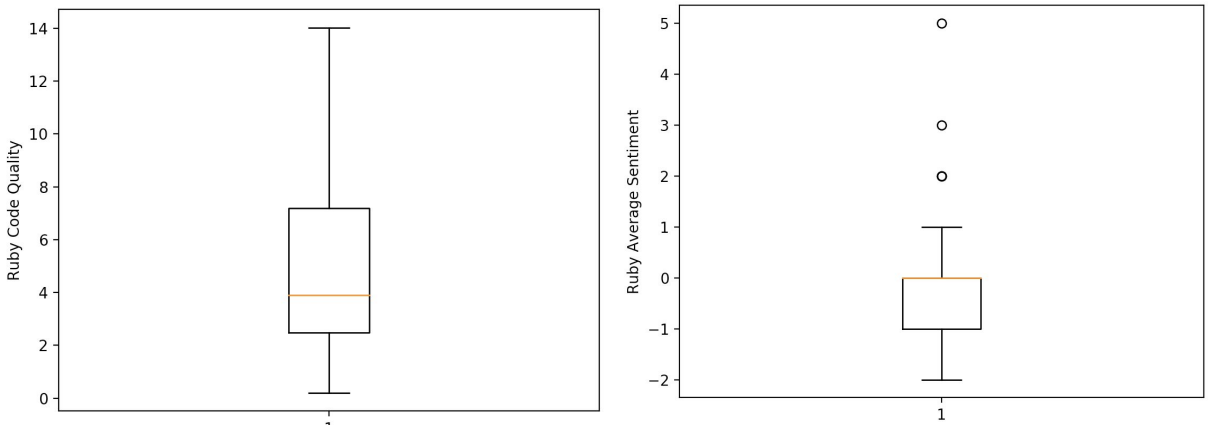


Fig 8. Distribution of Ruby code quality per user and average sentiment per user

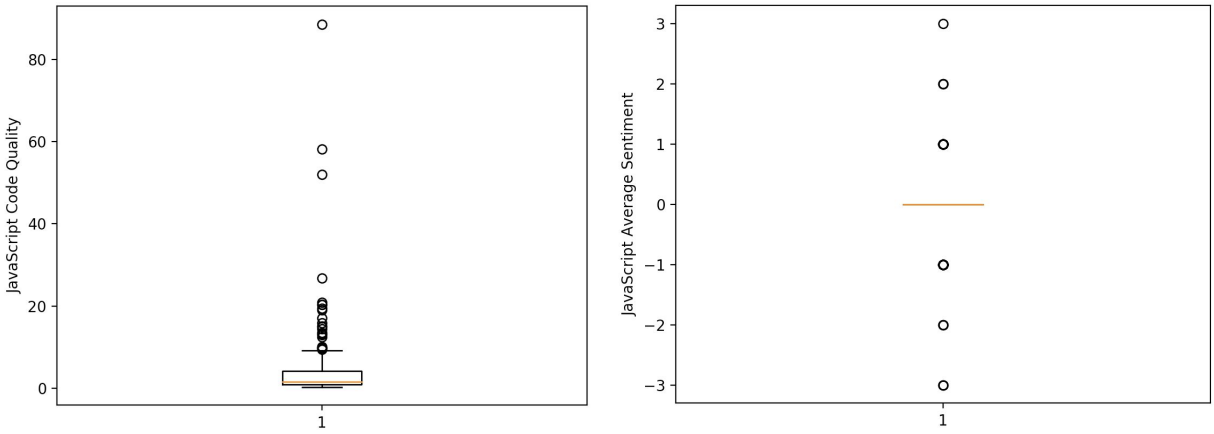


Fig 9. Distribution of JavaScript code quality per user and average sentiment per user

3.5 Stars and Average Sentiment Data

Repo Name	Average Sentiment	Stars
libgit2	0.134502924	0
bitcoin	0.1794871795	0
AutoMapper	0.1526845638	0
hiphop-php	0	0
facebook-android-sdk	0.1339449541	0
PR4TC	0	0
phantomjs	-0.3	0
ActionBarSherlock	0.2430414121	0
redis	0.6	0
openFrameworks	0.2030145143	0
RestSharp	0.2003710575	0
storm	0.2654824772	0
ravendb	0.2075738126	0
plupload	0.1645813282	0
mono-1	-0.375	0
Nancy	0.2094824915	0
SparkleShare	0.1725888325	0
mono	0.17325071	0

SignalR	0.2136469029	1
liboio	1	1
plupload-1	-1	1
AutoMapperThreadSafe	1	2
BizzyCore	1	3
VistaTC	0	4
ZoneEmuWotlk	0.04901960784	6
BlackSun	-1	7
RestSharp-.NET-2.0-Fork	-0.07142857143	7
ChaosCore	1	11
TrinityCore_MOP	-0.5	11
xbmca10	0.75	21
TrinityNya	2	23
uwom-server	-0.09126984127	23
TrinityCore_434	0.6	null
mongo	0	null
ServiceStack	0.1783264746	null
Kingswow	1	null
trinitycore	0	null
WoW4.3.4	0	null
XBMC	0.3684210526	null
contrib-libuv	0	null
ravendb-WithIndexWarmerPlugin	0.5	null
bitcoin-git	0	null
TrinityCore	0.1841566128	null
elasticsearch	0.1652046784	null
MiniProfiler	0.1587802313	null
clojure	0.129468599	null
libuv	0.05526315789	null
xbmc	0.2125603865	null

We calculated the average sentiment per project but did not see any correlation but posted the data so that we could prove that we at least tried to implement their last research question. You can see in the table above. We also had issues with getting star counts from the GitHub API. (Not supplied with the MSR 2014 data)

4. Conclusions

For the additional research component, we set out to figure out the relationship between a GitHub user's comment sentiment, and their overall code quality. Our results show that this relationship exists, and varies somewhat based on the language used. Also some languages do not lend them self well to our analyzation process. For example, many C++ files are on average much longer than that of other languages we found, as well as the repositories being much larger, as a result, many C++ repositories fell to our filter in order to limit computation time. Based on our results, we saw for Python, C++, and JavaScript, there was a slight positive correlation between a user's sentiment, and code quality, meaning that nicer developers who specialize in these languages, write on average higher quality code whereas rude developers write poorer quality code. On the converse, Ruby developers who are rude, tend to write higher quality code, whereas those which are nice, tend to write poorer quality code. The hypothesis going into this experiment was that developers who are nicer in their comments, will develop better code then their ruder counterparts, so it is a bit surprising to see Ruby developers have to opposite result.

Further work could be done to improve upon our system of analysis. The most important of which being to increase our dataset. Improvements could also be done to better determine the quality of code such as checking for things other than style guideline adherence such as determining the complexity of readme, structure of a repository, length of files, or dynamic analysis tools. Combining all above mentioned components would better reveal the quality of the code.

Team Member Contributions

In this section explain in detail what are the contributions of each member of the team to the various artifacts and parts of the project:what parts of the code/functionality, methodology implementation, tasks, README file, final report, etc.did each member contribute to. Be specific and thorough, as this will be used for grading, in addition to the contributions visible in GitHub

Tyler:

Implemented all code found in the repoAnalysis subdirectory except for cpplint.py which was provided by Google's team as well as the Database wrapper class, although it was modified from its original source. Implemented all functionality of the additional research component. Wrote the methodology portion about the additional research component as well as the results section 3.4

Raidel:

Implemented the code found in the emotionStat directory, as well as the utils directory. The emotionStat directory contains a python script that analyzes all the emotion scores produced by SentiStrength (done by Andrew), and produce results similar to those presented in the assigned paper. Wrote the result sections 3.1 - 3.3 and provided a comparative analysis between the results presented in both papers. There is one section I was not able to complete, emotion and team distribution by continent. The research paper analyzed the location of each commit comment and classified it into a continent manually. Due to the time constraints for this assignment, the immense quantity of commit comments in the repositories analyzed, and inconsistencies in location format, we leave this section as future work to be implemented.

Andrew:

Implemented the sentimentAnalysis subdirectory, which is described in detail in section 2.1 of the methodology of the project. Implemented the SentiStrength module to interface with the provided SentiStrength system and gathered sentiments of all commit comments through the SentiStrength module. Studied provided MySQL dataset to extract data that was needed to analyze for the results for rest of group. Provided SQLite database to be used by the rest of the group instead of using the MSR dump that had unnecessary data for our purposes. Attempted to recreate the stars research question but did not see any correlation between stars and average sentiment of repo, which lines up with the given results.