CS212

Final Project

Team Member: Yuming Chen 320180939611

Huiyi Liu 320180940030

Shijie Ma 320180940121

Qiaoyuan Yang 20180940450

Jiyang Xin 320180940391

Date: 25th June, 2020

1. Hypothesis and the mapping to data

1.1 Hypothesis

1.1.1 The more complex the code is, the harder it is to discover its bug(s).

1.1.2 The more complex the code is, the harder it is to fix its bug(s).

1.1.3 The bugs in the code from experienced developers are hard to discover.

1.1.4 The bugs in the code from experienced developers are easy to fix.

1.2 Data

1.2.1 Code level

1.2.1.1 The indicator of code complexity

1) the nloc (lines of code without comments),
2) CCN (cyclomatic complexity number),
3) token count of functions.
4) parameter count of functions.

1.2.2 Commit level

1.2.2.1 The indicators reflecting whether the bug is easy to be found

1) The time from the developer's commit to the "First Find Bug" commit pointed to it

1.2.2.2 The indicators reflecting whether the bug is easy to be fixed

1) The fix distances

The fix distance is the number of commits between bug commit and fix commit, which can reflect how difficult it is to fix a discovered bug in some degree.

2) The time difference between Bug commit and Fix commit

1.2.3 Developers level

1.2.3.1 The indicators of developers' experience

1) The numbers of developers' commit

1.3 Mapping

1.3.1 Hypothesis: "The more complex the code is, the harder it is to discover its bug(s)."

Data 1.2.2.1: The indicators reflecting whether the bug is easy to be found

Data 1.2.1.1: The indicator of code complexity

1.3.2 Hypothesis: "The more complex the code is, the harder it is to fix its bug(s)."

Data 1.2.2.2: The indicators reflecting whether the bug is easy to be fixed

Data 1.2.1.1: The indicator of code complexity

1.3.3 Hypothesis: **“**The bugs in the code from experienced developers are hard to discover.”

Data 1.2.2.1: The indicators reflecting whether the bug is easy to be found
Data 1.2.1.1: The indicator of code complexity
Data 1.2.3.1: The indicators of developers' experience

1.3.4 Hypothesis: “The bugs in the code from experienced developers are easy to fix.”

Data 1.2.2.2: The indicators reflecting whether the bug is easy to be fixed
Data 1.2.1.1: The indicator of code complexity
Data 1.2.3.1: The indicators of developers' experience

2. Requirements

2.1 Functional requirements

Rq1: The data stated in 1.2 shall be extracted from linux-stable kernel git.

Rq1.1: The indicators that measure code complexity can be calculated using third-party package.

Rq1.2: The raw data in commit level and developer level shall be collected from git log to ensure its reliability.

Rq2: The data collected shall be cleaned.

Rq2.1: The outlier can be considered.

Rq2.2: The names of developers can be cleaned by normalizing the character sets.

Rq3: The data shall be analyzed to justify the hypothesis.

Rq3.1: The appropriate technology shall be chosen to use when analyzing.

2.2 Non-functional requirements

Rq4: The results shall be interpreted.

Rq5: The technology selection and evaluation shall be reported.

Rq6: The concept shall be documented.

2.3 Technical requirements

Rq7: Python3 shall be used.

Rq8: The program shall work well in Linux system.

3.Design

3.1 Data Extract

```
---- extractor
        ---- prepare_data ---------------------- preparation for data extracting
                |--> get_all_author.py ---------------------- gets all authors and their commits' hash ID
                |--> get_all_bug_commits.py ---------------------- gets all fix-bug commits' hash ID
        |--> author_data.py ---------------------- extracts author level raw data
        |--> commit_data.py ---------------------- extracts commit level and code level raw data
        |--> repo.py ---------------------- uses to run git command
        |--> tools.py ---------------------- contains tools used in data extracting
```

3.2.1 class

3.2.1.1 Repo

3.2.1.2 CommitsFeatureExtractor

3.2.2 Function

3.2.2.1 get_all_fix_bug_commits: Get all the hash ID of fix-bug commits

3.2.2.2 get_all_author: Get all of the authors in linx-stable and the hash ID of their commits

3.2.2.3 error_log(message:str): Record error message when data extracting

3.2 Data Storage

```
---- data
        ---- rdata ---------------------- raw data
                ---- prepare_data
                        |--> all_author@< time >.json
                        |--> all_fix_bug_commit@<time>. json
                ---- author
                        |--> sample_name@<sample version>.csv
                        |--> author@<sample version>.csv
                ---- code&commit
                        ---- code_content_<random seed>
                        |--> code_<random seed>.csv
                        |--> commit_<random seed>.csv
        ---- pdata ---------------------- data after preprocess
                        |--> author_stat@<sample version>.csv
                        |--> commit_stat_<random seed>.csv
```

1) < time >:
The linux-stable is dynamic. It is use to record the time when data collection is finished.

2) <sample version>:
The sample selection of the author level data is manual processing. The version value is to record the sample.

3) <random seed>:
The sample selection of the commit level data is using "random" package. The random seed is to record the sample

3.3 Data Preprocess

|--> preprocess.ipynb

3.4 Data Analyze

|--> analyze.ipynb

4.Implemention and concept

4.1 Data Collecting

4.1.1 Data Preparation

4.1.1.1 Get all the authors' name and hash IDs of their commits

Program	Store
get_all_author.py	all_author@<date>.json

- **Reason**

Make a more scientific sample selecting according to the population.

4.1.1.2 Get all the hash ID of Fix commits and the commits pointed by them

Program	Store
get_all_bug_commits.py	all_fix_bug_commit@<date>. json

- **Reason**

∵ Fix commit --> Bug commit

∵ The commit we get is bug commit, we need to find the fix commit

∴ The time complexity is high when we use travel

∴ we decide to sacrifice space to store all of the "Fix commit --> Bug commit" into json(dict) format which we can use the key(bug commit) to find the value(fix commit) directly

4.1.2 Main Process (with data cleaning)

4.1.2.1 Get the commit level raw data

In the data preparation step, we have extracted all the Bug-commits and fix-commits pointed to them. We randomly select the sample with a size of 1000 from all the fix-bug commits. In order to translate the attributes into numeric data, we calculate the fix distance, time to find the bug (the time difference between the first commit and the Bug commit) and time to fix the bug (the time difference between Bug commit and Fix commit).

Program	Store
commit_data.py	commit_<random seed>.csv

4.1.2.2 Get the author level raw data

All the name of authors and the hash ID of their commits have been extracted in the data preparation step. The names are cleaned by normalizing the character sets in order to avoid names in messy format. The data is used in last two hypothesis, therefore we select sample authors and calculate their numbers of commit in order to measure their experience working in Linux.

Program	Store
author_data.py	athour@<sample version>.csv

4.1.2.3 Get the raw code of bug commits

In order to get the code of bug commits, we analyzed the commit format and extract the lines of code stated in the bug commits.

Program	Store
commit_data.py	code_content_<random seed>/ (dir)

1) commit_<random seed>.csv

Column Name	**bug**	**fix**	**fix distance**	**find bug time**(Second)	**fix bug time**(Second)
Type	string	string	float64	float65	float66

2) athour@<sample version>.csv

Column Name	**author**	**commits number**	**bug commits number**	**total fix distance**	**total find bug time**(Second)	**total fix bug time**(Second)
Type	string	float64	float64	float64	float64	float64

4.2 Data Clean & Preprocess

4.2.1 Commit Level Data

We have noticed that there is an outlier whose fix distance is larger than others in a large degree (it is 10 while most others are lower than 4). But for the reason that it is a legitimate member of the population we want to study, which are bugs, this outlier is not a bug we can remove. Then we go back to the data collecting step and extract the code of selected bugs.
In order to reduce the scope of time to make it easier to observe as a scatter plot, we translate the unit of time finding and fixing bugs from seconds to days. Besides, we normalize the data using StandardScaler provided in sklearn package, improving the precision of data.

4.2.2 Code Level Data

We have collected raw data of code. As an attribute, the complexity of code needs to be translated into numeric data. We use lizard package to analyze data automatically, getting the number of lines of code without comments, cyclomatic complexity number, token count of functions and parameter count of functions. The four arguments can reflect the complexity of code in some degree.

Program	**Store**
lizard package	code_<random seed>.csv

4.2.3 Author Level Data

In order to reduce the scope of time to make it easier to observe as a scatter plot, we translate the unit of time finding and fixing bugs from seconds to days. Besides, we normalize the data using StandardScaler provided in sklearn package, improving the precision of data. The names are cleaned by normalizing the character sets in order to avoid names in messy format. Regarding the population to be study, the outliers which have extreme values but is legitimate are not removed.

Column Name	**author**	**commits number**	**bug commits number**	**total fix distance**	**total find bug time**(Second)	**total fix bug time**(Second)
Type	string	int64	int64	float64	float64	float64

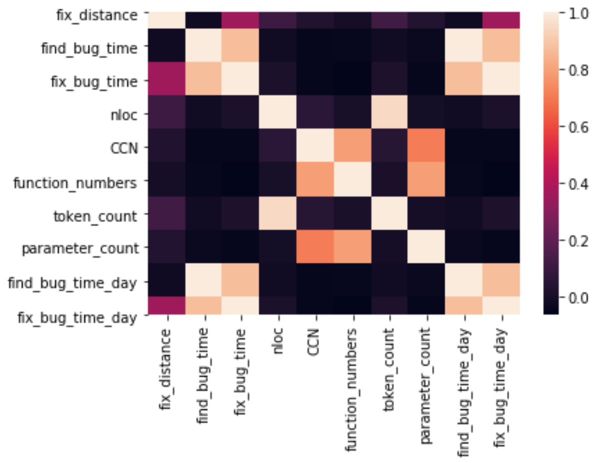
Column Name	**bug_ratio**	**average fix distance**	**average find bug time**(Second)	**average fix bug time**(Second)
Type	float64	float64	float64	float64

4.3 Data Analysis

4.3.1 Visualization

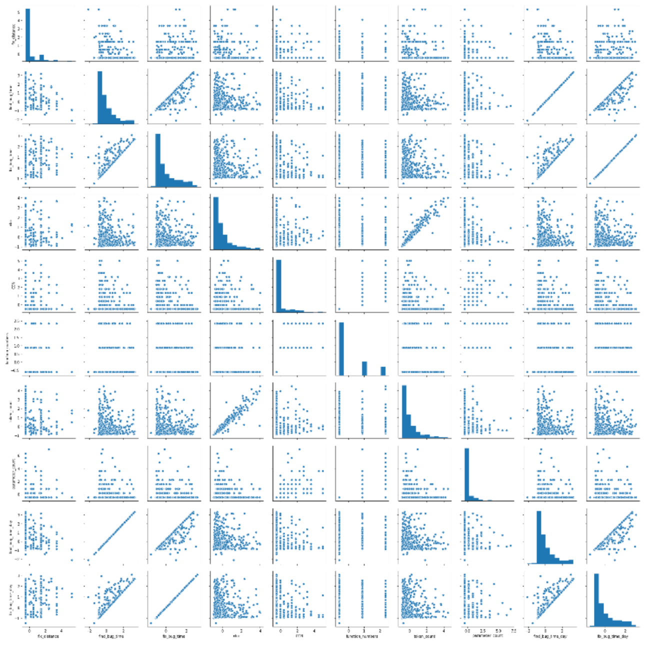
4.3.1.1 Code and Commit Level

1) heatmap



Correlation coefficients are used to measure the strength of the relationship between two variables. In the heatmap diagram, the larger the absolute value of correlation coefficients are, the shallower are the colors, which reflects the stronger the relations are. We can conclude from the diagram that most variables do not have strong relations with each other except some relations such as find_bug_time and fix_bug_time – just because they are logically strong related – developers can only fix a bug after it is found.

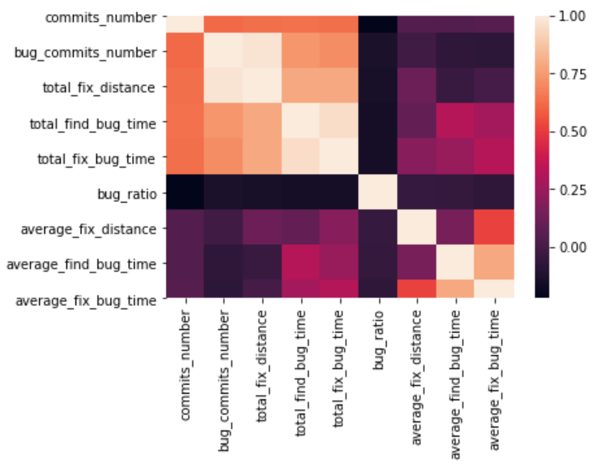
2) Scatter Diagram



The scatter diagram provides a preview of the relations between variables. The relations are not obvious in most diagrams. Most of the relatively strong relations seem to have linear relations.

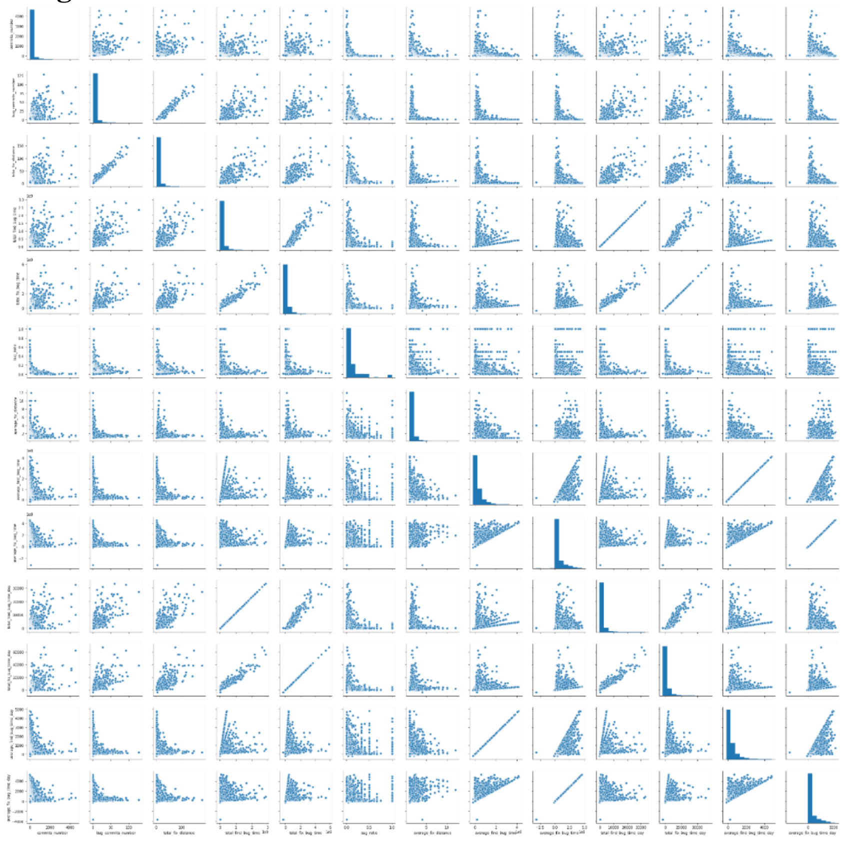
4.3.1.2 Author and Commit Level

1) heatmap



In the heatmap diagram, the larger the absolute value of correlation coefficients are, the shallower are the colors, which reflects the stronger the relations are. It reflects that the bug ratio have extremely weak relations with other variables. Only some logically related variables have relatively strong relations such as total fix distances of an author and the number of his/her commits—the calculate of fix distances relies on the number of commits in a large degree.

2) Scatter Diagram



4.3.2 Multiple Linear Regression

For the reason that there are several explanatory variables for each hypothesis, we use multiple linear regression.

4.3.2.1 Hypothesis 1

The more complex the code is, the harder it is to discover its bug(s).

x	nloc	CCN	function numbers	token count	parameter count
y	find bug time (day)				
Intercept	-0.0102				
Coef	-0.0773	-0.1226	0.0243	0.0370	-0.0049
Score			-0.0668		

4.3.2.2 Hypothesis 2

The more complex the code is, the harder it is to fix its bug(s).

x	nloc	CCN	function numbers	token count	parameter count
y	fix_bug_time_day (day)				
Intercept	-0.0225				
Coef	-0.0581	-0.0194	-0.0087	0.0541	-0.0389
Score			-0.0133		

x	nloc	CCN	function numbers	token count	parameter count
y	fix_distance				
Intercept	-0.0004				
Coef	-0.0564	0.0606	-0.1407	0.1643	0.0867
Score			0.0174		

4.3.2.3 Hypothesis 3

The bugs in the code from experienced developers are hard to discover.

x	average find bug time day	bug commits number
y	commits_number	
Intercept	11.6574	
Coef	0.0401	22.4961
Score		0.3954

4.3.2.4 Hypothesis 4

The bugs in the code from experienced developers are easy to fix.

x	average find fix time day	average fix distance
y	commits_number	
Intercept	103.8421	
Coef	1.3712	0.0000
Score		-0.0005

5. Conclusion

From the analysis above, we cannot conclude any of the hypothesis reliably. There are not obvious relations between the factors.