Predicting GitHub Pull Request Acceptance and Latency

Yifan Shao

I. Introduction

The popular open projects on GitHub receives a lot of pull requests every year. Repository maintainers are overwhelmed by the amount of review tasks. Jiang, Adams and German [1] studied patch acceptance in Linux Kernel repository, It will be be interesting to apply the similar methods to GHarchive dataset. This can potentially be used by managers of large public source software repository to sort review tasks. This project focuses on predict GitHub pull request acceptance and latency.

II. Data

A. Data Collection

The dataset is from GHarchive year 2017 and 2018. The dataset records events happened in GitHub. Following steps are used to get dataset from BigQuery:

1. Reduce size of the set by only choosing pull request that come from top 3000 most popular repository. Popularity is defined as number of watch event a repository gets in two years. A watch event is generated once a user start starring a project.
2. Further reduce the dataset by choosing repositories with no less than 50 pull requests opened and no less than 10 merged.
3. At last, pick pull requests from those repositories that are closed. Because when a pull requests just opened, its attributes are not fully developed. For example, when a pull request is opened, there can't be any review comments yet. On the other hand, closed pull requests allow me to observe the final state of a pull request.
4. After those steps, there are 1,130,186 rows of pull requests. The data is downloaded from BigQuery and read by the final notebook.

B. Data Quality

This part corresponds to Section 3 in notebook.

After inspecting the data, there are 8,063 rows where body is null value, 83 rows with null *closed_time*, 292,826 rows where *merged_time* is null. Null *body* value means that the original pull request message is empty. When being extracted by JSON_EXTRACT_SCALAR on BigQuery, an empty string will return null value. In this case, replace the null value with 0 value. If a *closed_time* is null, it means that the payload does not contain such field. JSON_EXTRACT_SCALAR will return null if no such field exist. It is not clear why some rows don't have a closed time when their actions are closed. This is treated as missing data and remove those rows from dataset. *merged_time* is expected to be null if a pull request is not merged. It is ignored because it won't interfere with later analysis. For latency analysis, only merged pull requests are used anyway.

Table I
Downloaded Dataset Field and Meaning

name	Meaning
rp_id	base repository unique id
action	action of pull request event (opened, closed, reopened)
additions	number of lines added
body	pull request message byte length
created_time	pull request created at time
closed_time	pull request closed at time
comments	number of comments of pull request
commits	number of commits in a pull request
files_change	number of files changed by pull request
deletions	number of lines deleted
pr_id	pull request unique id
merged	if pull request is merged
merged_time	pull request merged at time
review_comments	number of review comments pull request receive
title	tile of the pull request
author_id	user who started the pull request unique id

C. Data Exploration

This part corresponds to part 4 in notebook.

Each row in dataset represents a pull request and its attributes. Table I shows the name and meaning of those attributes.

Fig.1. is pull request distribution from each quarter. There are

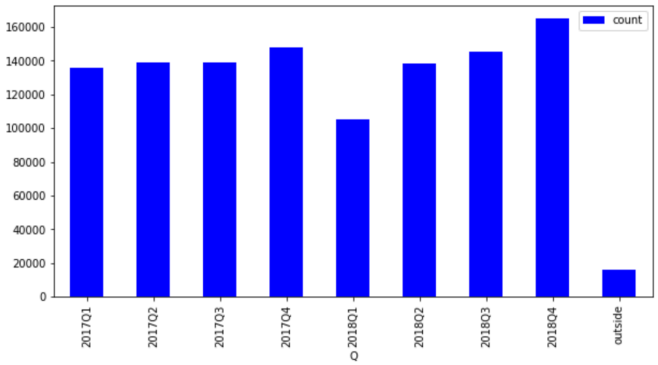

Fig. 1. Pull request distribution by quarters

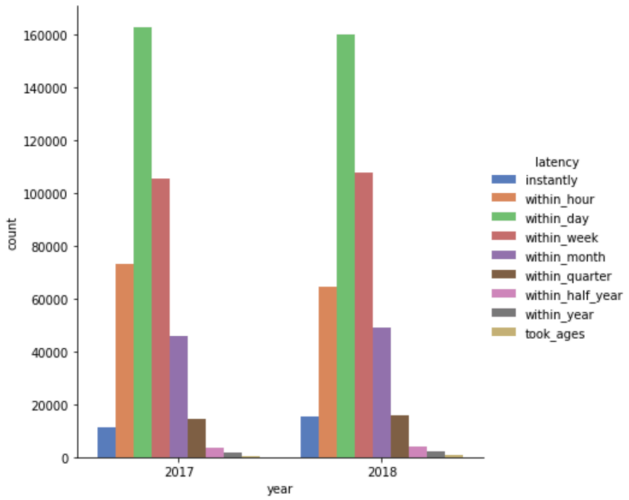

Fig. 2. Pull request latency distribution

pull requests outside 2017-2018. The earliest pull request is on 2017-01-01, and the latest pull request is on 2019-01-05. The pull request in the "outside" group are from 2019-01-01 to 2019-01-05. Because those pull requests are not a big fraction of the dataset, and they are not closed too long after 2018 Q4. I will not remove those repositories and consider them as in 2018 Q4.

The latency classes are generated with same method as Jiang, Adams and German [1]. The time difference between pull request opened time and merged time is divided into 9 classes: "instantly", "within_hour", "within_day", "within_week", "within_quarter", "within_half_year", "within_year "and "took_ages". The distribution is plotted into Fig 2. Significant percentage of pull requests are in "within_hour", within_day, within_week and within_month class.

Fig. 3. show the acceptance rate distribution by repositories. The acceptance rates are divided into bins every 10%. The horizontal represents the number of repositories in that bin. There is a tail in the distribution.

D. Data Selection

This corresponds to section 4 in notebook.

Because the pull requests are from repositories with different repository characteristics, the dataset should be filtered to only contain pull requests that come from repositories with similar

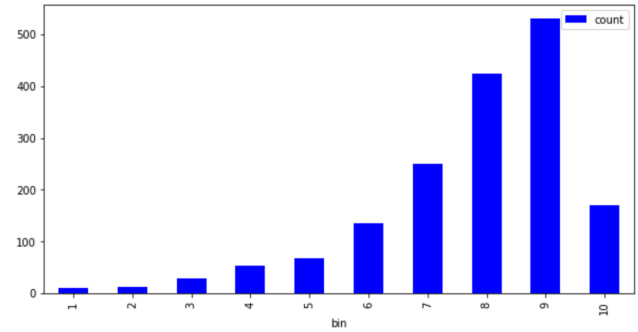

Fig. 3. Repository acceptance rate distribution

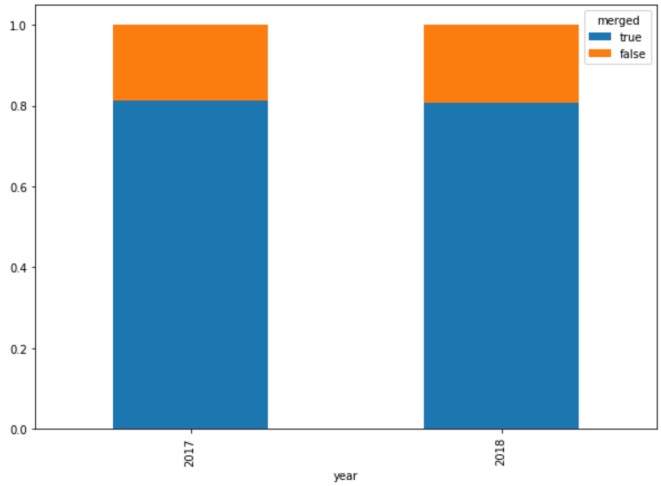

Fig. 4. Pull request acceptance rate after data selection

characteristics. Repositories which have acceptance rate lower than bin 4 in Fig 3 are filtered out. From what's left, repositories with over 90% pull requests accepted within hour, day, week and month are picked. The pull requests are selected from remaining repositories. After selection, there are 774,502 pull requests.

The acceptance rate of each year is around 80%, as shown in Fig 4.

In Fig 5, The latency of merged pull requests is concentrated in *within_hour*, *within_day*, *within_week* and *within_month* classes. Each year shows similar distribution pattern.

E. Define Pull Request Metrics

Jiang, Adams and German [1] defined 35 metrics from their dataset. There are 29 metrics used as features for decision tree model. Because of the nature of two different dataset, some metrics cannot be mapped. Table II shows the metrics defined for GitHub dataset.

Some of the metrics are straightforward to understand. Due to time limit and complexity, not all metrics are used as features in notebook code. The used metrics are: *pr_exp*, *commit_exp*, *create_year/month/week/day*, *body*, *reviews*, *commits*, *bug*, *size*, *files_change*, *repo_popularity*, *merged* and *latency.*

requested_reviewers: When a pull request is created, the author can choose to request other users to review the code and give suggestions.

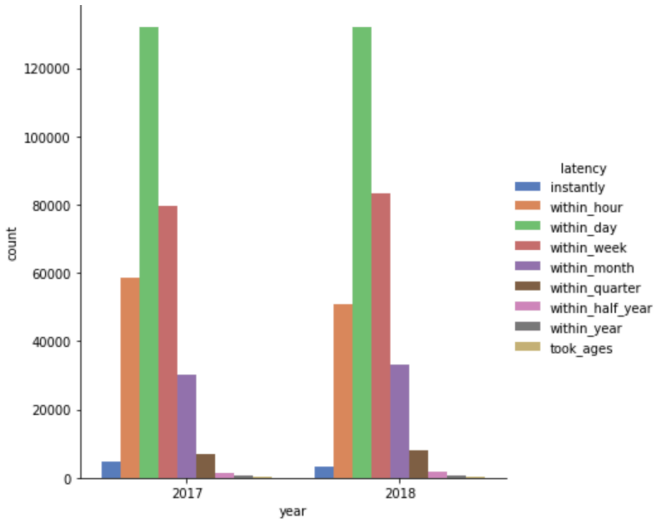

Fig. 5. Pull request latency distribution after data selection

first_try: after the contributor get the feedbacks from the review comments, changes can be made to the code. If contributor has pushed a newer version to his branch, this is no longer *first_try*.

thr_volume: The number of review comments between pull requests first opened and when the pull requested is updated if the author decides to update the pull request by pushing updates to his branch.

thr_part: The number of people joined in the discussion and review before the author pushed new updates to his branch. This is achievable by identifying unique users involved in review comments related to a certain pull request

reviewers: The number of users involved in pull request review. This can be done by finding unique users in pull request review comments related to the same pull request. I consider people who left comments as reviewers.

review_comments: The number of people involved in reviewing process. In GitHub, a reviewer can comment on the pull request as a general comment of the pull request. A user can also comment on specific part of the code in the pull request to offer opinions. In my project, there are attributes called comments and reviews in a pull request payload. I add them up to get the number of review comments.

bug: whether the pull request is a bug fix. I used similar method as Jiang, Adams and German [1]. The authors scanned the body of the email to search for key word "bug" or "fix". In my project, I chose to scan the title of the pull request instead of the body for word "bug" or "fix". The dataset with body comments included is very large, makes it hard to download and process on Jupyter notebook. I decided to use the title to replace the body comments. I will assume the submitters put the key information in titles as well. This gives smaller size dataset.

nth_try: what version is the pull request on. When a contributor incorporates the feedback from pull request review, he will make changes to his local repository and push the changes to head branch of the pull request. I consider when the submitter pushed the head branch of the pull request, it is an update of version. This can be done by finding push event happened to a pull request head branch. Count the number of push events happened between the pull request created time and closed time.

Table II
Defined Metrics, Type and Description

Metric	Type	Description
pr_exp	numeric	Number of pull request submitted by same author regardless in entire dataset
commit_exp	numeric	Number of merged pull request by same author in entire dataset
create_year /month/week/day	numeric	pull request created time
request_reviewers	numeric	no. of requested reviewers
body	numeric	byte length of pull request submission body
first_try	boolean	Whether this pull request is first version, check if branch is updated
thr_volume	numeric	number of review comments until the pull request is updated
thr_part	numeric	number of people participating in discussion before pull request is updated
thr_time	numeric	discussion time in seconds between first start of pull request time and pull request update time
reviewers	numeric	no of people involved in pull request review
reviews	numeric	no of review comments in a pull request
commits	numeric	no of commits of pull request
bug	boolean	whether this pull request is bug fix
size	numeric	sum pull request additions and deletions
nth_try	numeric	what version is the pull request on
files_change	numeric	no of file changed by pull request
repo_popularity	numeric	no of accepted pull request of repository so far
merged	boolean	whether the pull request got merged
latency	numeric	merged time - created time, later converted to 9 classes

F. Data Analysis Method in Paper

Jiang, Adams and German [1] used decision tree to predict the outcome of a patch submission with the metrics they defined. They split the data every 3 month and train decision tree models for each 3-month dataset. Then they performed 10-fold cross validation on the dataset to better train the model. This step generated 10 trees every 3-month interval. They

selected features appear in top 2 level of the 10 decision trees every 3 month and compared them based on their occurrence frequency.

They plotted the occurrence frequency into heatmaps graphs to compare how important features evolve through time. Y axis represents each feature and X axis represents each quarter from 2005 -2012. The darker the cell color the higher occurrence frequency a feature has in that quarter.

The authors evaluated their acceptance prediction with precision and recall metrics and evaluated latency prediction with accuracy.

III. METHOD

The two-year dataset is split every 3 months, resulting 8 subsets.

In project, random forest replaces decision tree in selected paper. Random forest randomly chooses subset of feature set. If a feature is important, it will show up in many trees in forest. This is similar to what Jiang, Adam and German [1] did. Both methods achieve similar goal: find the most frequent features appearing in multiple trees.

To evaluate pull request acceptance prediction, area under Precision and Recall is used. For latency prediction, time difference is discretized into 9 classes in data exploration step. Evaluating metric is accuracy.

In project *featureImportances* is used to find important features. This spark build-in function returns numeric values on each feature. Higher values mean higher importance. In random forest, it is calculated by averaging importance of the feature across all trees.

Each feature in each quarter is ranked based on its feature importance value provided by *featureImportance* function. Average rank of a feature is calculated from ranks from 8 periods.

The feature importance scores are plotted in heatmap to study how feature importance evolves through time. The darker the cell color, the higher importance a feature. Each cell represents the feature in a particular quarter.

To validate the model, 10-fold cross validation with grid search to do hyperparameter tuning is used. The grid evaluates 10, 100 and 200 trees random forest model.

Cross Validator tries the model through different subsets of the training data. Best model is selected from cross validation. In next section the changes in evaluation results and feature importance caused by cross validation are studied.

IV. RESULT

A. Pull Request Acceptance

This part corresponds to Section 6 in notebook.

The model after tuning has minimal gain in area under precision and recall from basic model with 100 trees. The top 3 important features by average rank remained unchanged. After cross validation, the model has about 0.92 average area under precision and recall from 8 quarters. The *commit_exp*, *create_week* and *pr_exp* are in top 3 of most important features.

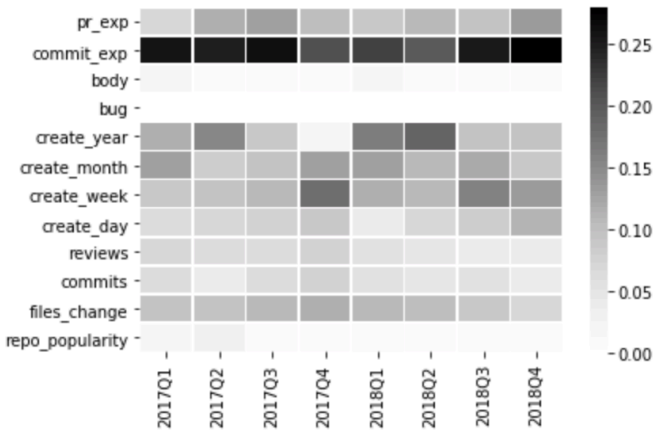

Fig. 6. Acceptance Prediction feature importance by each quarter for model before cross validation, darker color means higher importance value

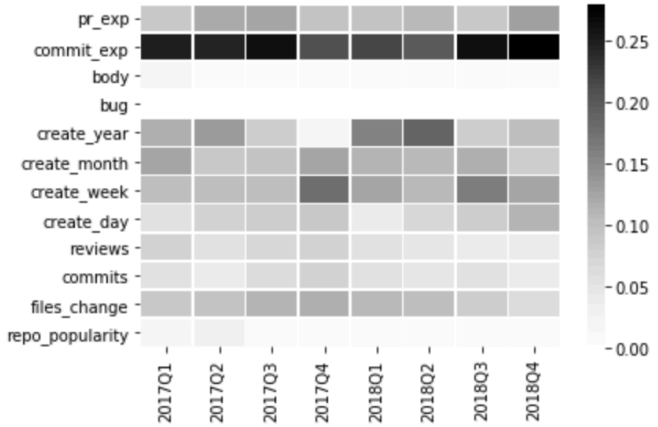

Fig. 6. Acceptance Prediction feature importance by each quarter for model after cross validation, darker color means higher importance value

Table III
Top Features After Cross Validation in Acceptance Prediction

feature_name	averge_rank
commit_exp	1.0
create_week	3.25
pr_exp	3.75

commit_exp and *pr_exp* are related to experience of the contributor. From the average rank, *commit_exp* remained to be the top most important feature throughout 8 quarters, in basic model and in model after cross validation. Also, *pr_exp* remained to be in top 3 most important feature after cross validation. This makes sense. If a contributor has a lot of accepted pull requests, the person usually has strong programming skill and knowledge. A pull request submitted by an experienced contributor is more likely to be accepted.

create_week is the week in a year when the pull request is opened. This cannot be justified yet.

B. Accepted pull Request Latency

This part corresponds to Section 7 in notebook.

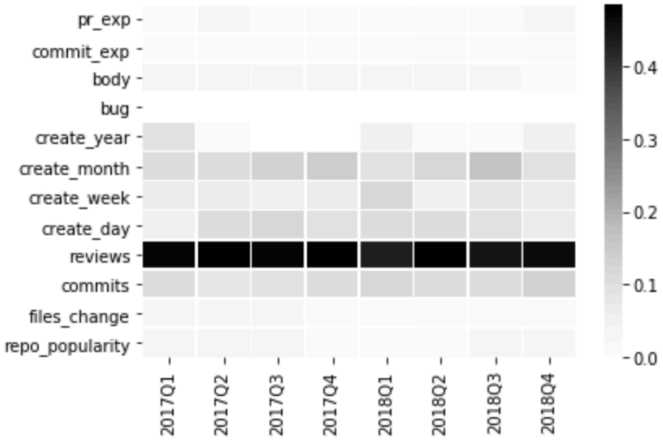

Fig. 7. Acceptance Prediction feature importance by each quarter for model before cross validation, darker color means higher importance value

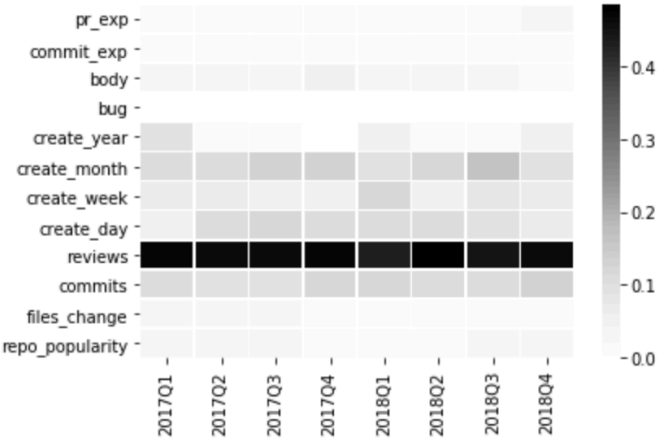

Fig. 7. Acceptance Prediction feature importance by each quarter for model after cross validation, darker color means higher importance value

The baseline model is a model that predicts every label as "within_day". I choose "within_day" because this class has the most number among 9 classes. I can calculate baseline accuracy by dividing number of "within_day" against total number of pull requests in dataset. The baseline model accuracy is about 0.42.

The average accuracy of basic model with 100 trees is about 0.49. After cross validation, the accuracy remained at about 0.49, with very small increase. In the 8 quarters, the worst model has about 0.48 accuracy and the best model has about 0.5 accuracy. The accuracy is still higher than baseline model, but not by much. I don't think the models I trained are good models. This may also be explained by different repository characteristics. Maybe the steps I used in data selection was not enough to filter out repositories with different characteristics. Pull requests come from different repositories may inherently have different acceptance latency.

reviews, *commits*, *create_month* and *create_day* are the top 4 most important features in the models. After cross validation, *create_month* gained a higher rank, commits and *create_day* get lower rank.

reviews remained to be top 1 most important feature throughout the 8 quarters. The explanation is when a pull request has a lot of reviews, the contributor have to go back to work on improving the pull request, or spend time discussing with reviewers. Both actions will delay the pull request merge.

Table IV
Top Features After Cross Validation in Latency Prediction

feature_name	average_rank
reviews	1.0
create_month	2.625
commits	3.0
create_day	3.875

commits is number of commits in the pull request. An assumption is that when a pull requests has large number of commits, the pull request will alter many parts of project. This may result long reviewing time because a repository maintainer has to take a long time to check many parts in project.

create_month and *create_day* are the month and day in a year when the pull request is created. There is no solid explanation to justify why these two metrics are important.

REFERENCES

[1] Y. Jiang, B. Adams, and D. M. German, "Will My Patch Make It ? And How Fast ?: Case Study on the Linux Kernel," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, 2013, pp. 101–110.