Predicting Acceptance of GitHub Pull Requests

Nikhil Khadke, Ming Han Teh, Minghan Shen
{nkhadke, minghan, sminghan}@cs.stanford.edu
Fall 2012, CS 229

1 Introduction

Social coding tools such as GitHub [1] have transformed the way software gets developed collaboratively and openly on the World Wide Web. GitHub has various avenues and features for collaborative development, but perhaps the most fundamental feature that allows non-listed collaborators to contribute to a repository is a pull request. A *pull request* [2] is a formal request from a potential contributor to a project owner to have his/her code improvements incorporated into the codebase. Because pull requests are made from external, unlisted contributors, there is an element of uncertainty in whether a pull request will be accepted. While there are certain guidelines and factors that may approximately indicate a successful pull request, the overall result is not clear and deterministic. We aim to use machine learning techniques to glean insights into what contributes to a successful pull request. We believe that if we can model pull request prediction accurately, we may be able to concretely understand the mechanisms that motivate successful pull requests, and use this to improve the collaborative software engineering process.

2 Preliminaries

2.1 Terminology

Borrowing GitHub terminology, we define the three states of a pull request at any point in time as:

i. **open**: pull request issue is awaiting decision on whether it will be accepted or rejected
ii. **rejected:** pull request was rejected
iii. **accepted**: pull request was merged into the code branch.

Typically the contributor and owner discuss the pull request via comments till a decision is reached. Additional revisions may also be required on the committed code. In practice, it is possible for a pull request to get accepted even though it had been rejected initially. Additionally, some pull requests are not explicitly rejected, but remain open for an indefinite amount of time. Pull requests sometimes remain open because the project owner lacked the time to act on the pull request or simply because the owner did not want to discourage the contributor by explicitly rejecting the pull request.

2.2 Problem Statement

We define the *pull request prediction problem* as follows: Given a pull request created at time t, decide if the pull request will be eventually accepted or rejected within the interval from time t to a future time t'. If the pull request remains open after time t', we assume the pull request has been rejected.

3 Data Collection and Processing

3.1 Dataset Description

Our primary data source is the GitHub Archive [3] data that is available for querying via Google's BigQuery. Because all data is stored in a columnar large table, any explicit relational information is lost, and we had to spend time understanding what the columns corresponded to. We used Google BigQuery to extract relevant data by filtering data on related event types.

3.2 Data Processing

Once a resulting data has been generated using Google BigQuery, we export the data over to Google Cloud Storage in Comma Separated Values (CSV) format. We then download the CSV file, do some pre-processing, and finally import the data into Sqlite3 tables for easy querying.

3.3 Data Crawling and Feature Extraction

In order to create our training and testing data, we extracted 3000 pull requests events that were made in the window from 04/01/2012 to 04/14/2012. After data sanitization and filtering, this resulted in 2734 usable data points. For some features such as contrib_prob and repo_prob (as reflected in Table 1), we ran queries on GitHub Archive using Google BigQuery to get information such as the number of successful pull requests and total number of pull requests made for each contributor and repository.

In search of predictive features, we realized our data set did not have an explicit representation for some of the features we were interested in. For example, the number of collaborators for a given repository was not directly provided, and we had to use the Github API to extract this information. Because data fetching via the Github API is slow and we are only limited to only 5000 API calls an hour, we wrote a web cache to cache all pages that we fetched.

Feature	Description of Feature	Justification
changed_lines	Number of line changes in patch.	The longer the change, the more scrutiny it will face and consequently affect its pull request acceptance.
changed_files	Number of file changes in patch.	If a few files are changed, the change may be simple and this could affect its pull request acceptance.
commits	Number of commits in patch.	The number of commits in a patch may indicate the complexity of a patch, and this could affect its pull request acceptance.
open_issues_count	Number of open issues on the head repository.	A repository with many open issues may be more willing to accept pull requests to fix these issues.
watchers_count	Number of watches on the head repository.	We suspect that if a repository is highly watched, it may be popular and more active in accepting pull requests.
forks_count	Number of forks on the head repository.	The number of forks is a loose indicator of the developer interest in the repository, which may affect pull request acceptance.
pushed_delta	Time difference in seconds between last push activity on the head repository.	This quantifies loosely the activity of the repository, which may affect the likelihood of pull request acceptance.
repo_collaborators_count	No. of collaborators in project.	We suspect that large repositories may be more willing to accept pull requests.
contrib_prob	The pull request success rate for the contributor.	We assume that a contributor high pull request acceptance rate has a higher chance of getting an accept.
repo_prob	The pull request acceptance rate for the repository.	We assume that a repository with a high pull request acceptance rate has a higher chance of granting an accept.
ngram_title	The probability of the pull request title text n-gram vector, given that the pull request is accepted.	We try to identify word n-grams in the title text which are more persuasive and lead to a better chance of acceptance.
text_sim	The text similarity between lines added and deleted. We used the text similarity function available in Python's difflib.SequenceMatcher [5].	We realized that features such as changes lines and changed files do not accurately measure the engineering complexity of changes made. For instance, a pull request that alpha-renames variables names would have a higher chance of being accepted compared to one that implements a new library. As such, text similarity aims to measure the complexity of the change involved.

Table 1: Features with their corresponding description and justification

4 Methodology

As described above, we wrote a lightweight Python framework, on top of scikit-learn [4], a Python machine learning library, that allowed us to fetch relevant GitHub data to be used as our input features. This pipe and filter framework is summarized below:

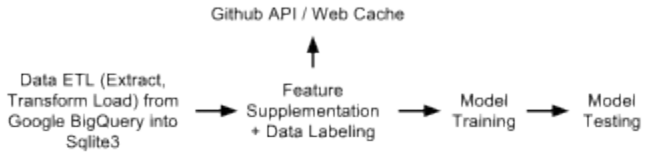

Figure 1: Pipe and filter framework used to train and test various models.

For our evaluation, we used a value of 31 days for t - t', because it represented an identifiable time period of 1 month. Additionally, we scaled all features by using standardization, such that each feature is scaled to zero-mean and unit-variance. To select our features, we used a logistic regression model and performed a combination of forward and backward search on the feature set to determine the predictive set of features for our dataset. We experimented with using multiple models to find a model that best predicts our data. For any given model, we used stratified k-fold cross validation (k=5) with 20% test and 80% train on a dataset of 2734 pull requests. We calculated the F1 scores and accuracy of each classifier to compare and contrast their performance. For every model, we tried to improve the model's predictivity by using techniques such as feature selection and parameter tuning via grid search maximizing the classifier's objective function.

5 Results

We begin with describing our feature choice and the process of using different machine learning techniques to arrive at reasonable models to solve the *pull request prediction problem.* When comparing accuracies, we note

that baseline probability of pull request being accepted is 0.5801.

5.1 Feature Choice

We summarize the description and intuition on our feature choice in Table 1. The head repository refers to the repository being asked to accept a pull request.

5.2 Logistic Regression

We use logistic regression as an initial method to build a model for the *pull request prediction problem* because of its ease of implementation and interpretation.

5.2.1 Logistic Regression and Regularization

Using the features described above, we achieved the following results:

Regularization Metric	Mean Accuracy	Std Dev	Mean F1	Std Dev
L1	0.6300	0.0172	0.6177	0.0159
L2	0.6455	0.0196	0.6306	0.0205

Table 2: Results of Logistic Regression on initial feature set

We observed that L2 regularization performed slightly better than L1 regularization.

5.3 Feature Refinement

As seen in Table 2, both L1 and L2 regularization were not able to reasonably improve our accuracy by implicitly penalizing features. This encouraged us to refine our features explicitly. When inspecting the coefficients for each feature in standard regression, we noted that ngram_title had an absolute coefficient of 6.213, while the other features had absolute coefficients in the range of [0.0011, 0.7011], indicating that ngram_title severely overfit the training data, reducing the overall accuracy. As a result, we decided to eliminate this feature. To further improve our feature set, we ran a backward search and isolated text_sim as an undesirable feature and eliminated it. Lastly, we inspected our training data, and realized that there were significant outliers, and to reduce their effect on the predictivity of our model, we used log transforms on features that took up large values. We applied a log base 2 transform on the following features: changed_lines, watchers_count, forks_count.

Logistic regression on the new feature set yielded the following results:

Regularization Metric	Mean Accuracy	Std Dev	Mean F1	Std Dev
L1	0.6602	0.0034	0.6432	0.0041
L2	0.6602	0.0034	0.6433	0.0045

Table 3: Results of Logistic Regression after feature refinement.

Feature	Coefficient	Std Dev
changed_files	0.1028	0.1945
log_2 (changed_lines+1)	-0.1117	0.0332
commits	-0.4774	0.1557
open_issues_count	0.0364	0.0200
log_2 (watchers_count+1)	-0.1034	0.0833
log_2 (forks_count+1)	-0.0947	0.1023
pushed_delta	-0.2065	0.0247
repo_collaborators_count	0.0239	0.0140
contrib_prob	0.1952	0.0271
repo_prob	0.5507	0.0160

Table 4: Weights for different features in L2 regularized Logistic Regression.

From Table 4, we notice that the most predictive features are repo_prob, commits, pushed_delta and contrib_prob. Because contrib_prob and repo_prob are based on the recent history of pull request acceptance, it makes sense for these values to be very predictive of pull request acceptance. We notice that repo_prob is more than twice as predictive as contrib_prob and pushed_delta was more predictive than contrib_prob and most other contributor features. This highlights that perhaps repository based features may be more useful in predicting the acceptance of a given pull request. We also see that commits are the second most predictive feature, and it intuitively follows that the more commits there are in a pull request, the more complex it is, reducing its chance of acceptance. What was surprising to note was the very low importance of open_issues_count, since we would expect that the more open issues a repository had, the higher opportunity of using and accepting potential pull request contributions. This may indicate that most of these repositories had issues that were only for the jurisdiction of formal listed contributors.

Grid Search over Parameters

To further improve this accuracy, we performed a grid search over the main parameters for logistic regression (L2), C and tolerance. The new parameters C=0.01 and tol=0.1 yielded a meager improvement in accuracy from 0.660 to 0.668.

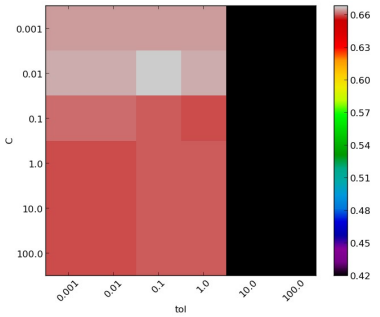

Figure 2: Logarithmic Grid of parameters C (penalty margin) and tol (tolerance) for L2 Logistic Regression

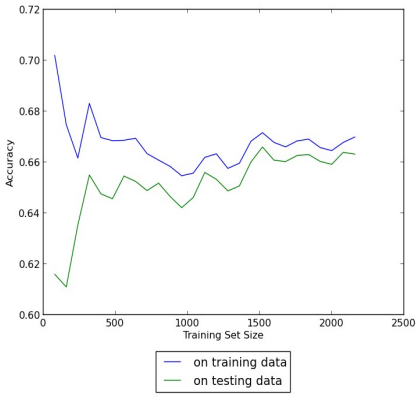

Figure 3: L2 Logistic Regression Learning Curve

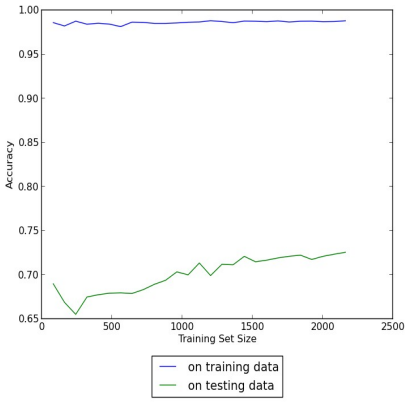

Figure 4: Random Forest Classifier Learning Curve

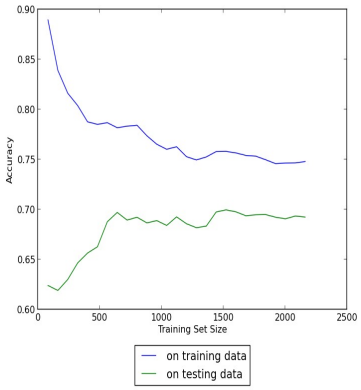

Figure 6: Tuned SVM (RBF Kernel) Learning Curve

Looking at the learning curve for L2 logistic regression (Fig. 3), we observe that there is high bias in the model, because error is high and there is only a small gap between accuracies for data set size greater than 1000 points. The accuracy graph is still increasing, which implies that a larger training set may improve performance.

5.4 Decision Trees

In an attempt to improve the accuracy of our model, we used decision trees and its ensemble variant Random Forests. Using default parameters we obtained the following results:

Method	Mean Accuracy	Std Dev	Mean F1	Std Dev
Decision Trees	0.6722	0.0161	0.6726	0.0158
Random Forest	0.7220	0.0164	0.7218	0.0163

Table 5: Results of Decision Tree Methods.

We observe that the accuracy for decision tree is lower than random forest, probably due to the decision tree overfitting the training data. In practice, random forest classifiers are more robust to outliers, and less likely to overfit the data compared to the simple decision tree [6]. However, the results for the random forest classifier still look suspicious. Looking at the learning curve for random forest in Figure 4, we see that training accuracy is significantly lower than testing accuracy, confirming our suspicion that the random forest classifier is a high-variance estimator which over-fits the data.

5.5 Support Vector Machine

As it was not possible to reduce the high-variance in the decision tree models after parameters tuning, we attempted to address this issue by using a support vector machine which is generally effective with large features. We used the refined features reflected in Table 4.

Parameter Tuning

We realized that for the Radial Basis Function (RBF) kernel SVM, setting correct hyperparameters are essential to achieving good results. We used a grid search over the main parameters to the RBF kernel SVM to arrive at $\gamma = 0.1$ and $C = 10.0$. This improved our accuracy marginally from 0.6847 to 0.7015, but with a slightly higher variance. We summarize our results and logarithmic plot in Figure 5 and Table 5.

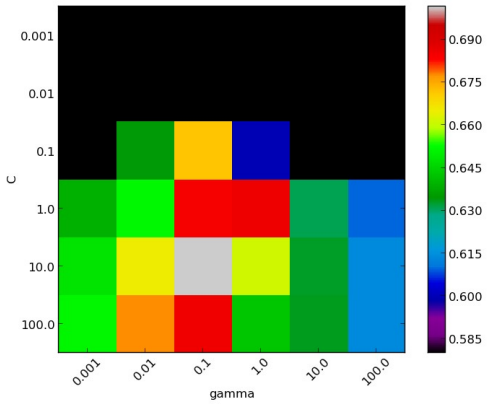

Figure 5: Logarithmic Grid of parameters C (penalty of error term) and gamma (kernel coefficient for RBF) for SVM

SVM Type	Mean Accuracy	Std Dev	Mean F1	Std Dev
RBF	0.6847	0.0039	0.6640	0.0048
Parameter Tuning (RBF)	0.7015	0.0090	0.6869	0.0111

Table 6: Results of Parameter Tuning the RBF Kernel SVM on the refined feature set.

The learning curve in Figure 6 for the tuned SVM shows a quantitative view of our results, informing us that accuracy reaches a plateau as training size increases. As compared to the learning curve for our other models, our tuned SVM model shows a better balance between bias and variance.

6 Discussion

Comparing results across Table 3, 5, 6, we see that while the random forest classifier has the highest accuracy and F1 score, its learning curve suffers from overfitting with high-variance. On the other hand, the logistic regression classifier suffers from high bias. With a good accuracy rate and tolerable deviation, it appears that SVM (giving 0.7015 accuracy against the baseline of 0.5801), is the most satisfying model we have tried.

Curious as to how our problem definition of using fixed time window restricted our experiments, we decided to evaluate our models for values of t-t' for step intervals of 5 days. We excluded the random forest classifier from this analysis as it had too high variance to be useful for this portion of the analysis. From Figure 7, we see a general trend of accuracy and F1 scores increasing as the time window increases. The metrics peak at around a time window of 80 days (~3 months), before decreasing slightly. This shows that the fate of the majority of pull requests get "decided" within 3 months. This window might seem long, but is supported in practice as decisions on pull requests are affected by project activity and management, and the persuasiveness of the contributor to get his code accepted in back-and-forth comments.

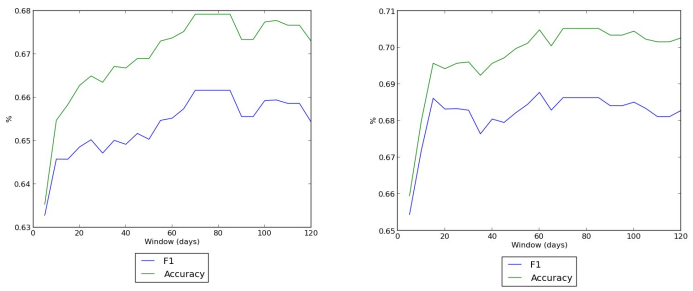

Figure 7: Plot of t-t' window in terms of days against F1 Score and Accuracy for various models. L2 Logistic Regression (left) and SVM (right)

By looking at the feature coefficients assigned to each feature (Table 4), we gleaned some insight into pull request acceptance:

- Changes to code should be succinct; having too many lines changed reduces the chance of acceptance.
- If possible, the contributor should not put off the pull request until the branch has multiple commits different from the master code. Because pull requests are made by people not directly involved in the planning of the project, a high complexity change often implies that the contributor's intention differs from the author's original intent of the code. Multiple commits imply a complex change and reduces the chance of acceptance.
- The track record of the contributor plays a role in pull request acceptance, since it is an indicator of the quality of their work.

7 Conclusion and Further Work

Building a highly predictive model for pull request acceptance is a difficult problem as pull requests represent informal patch-based contribution and subsequently are highly variable. Using the methods outlined above, we have presented reasonably predictive models such as the parameter tuned SVM and Random Forests that are able to predict a pull request's acceptance with an accuracy of 0.7015 and 0.7220 respectively, much better than the 0.5801 baseline.

In the future, we would like to spend more time crawling a richer dataset from GitHub and use this to build a richer feature set. We also plan to reevaluate our prediction based on each new event for a particular pull request. Additionally, given more time we want to enrich our model with features based on sentiment analysis on pull request comment logs.

8 References

[1] GitHub: Social Coding. [Online] Available: http://github.com/
[2] Using Pull Requests [Online] Available: https://help.github.com/articles/using-pull-requests
[3] GitHub Archive [Online] Available: http://www.githubarchive.org/
[4] Scikit-Learn [Online] Available: http://scikit-learn.org/
[5] difflib — Helpers for computing deltas [Online] Available: http://docs.python.org/2/library/difflib.html
[6] L. Breiman, "Random forests." J. Machine learning 45.1 pp. 5-32. Oct 2001