Classification of Github Repository Issues: Reducing the Noise for Maintainers of Open-Source Python Projects

William Fry
University of Pennsylvania
1 May 2016

I. INTRODUCTION

Software and the Internet it supports are largely products of open-source projects. From languages such as Java and Python to software such as Apache Hadoop and Linux, open-source projects ensure that the source code is maintained at the highest quality, while allowing anyone to leverage the software to make the world a better place.

Many of the top open-source projects are hosted on Github, a repository hosting service. Github allows anyone to upload software projects, helping them manage the different versions and revisions. Fittingly, Github is the primary hosting service among open-source projects. Among its most popular features, Github allows users to submit issues to projects. Users often do this when they have a question, find an error within the code, or have a feature they would like to request. Those who help fix and improve the open-source projects are often referred to as the project's maintainers or collaborators.

As some open-source projects on Github have as many as 20,000 users following their activity, it becomes quite a job in-and-of-itself just to prioritize and assign the issues to collaborators. As the popularity of an open-source project rises, it becomes increasingly important to fix 'critical' bugs right away as many developers and businesses rely on the source code for day-to-day operations. The collaborators of a project will often use labels to triage issues. In an attempt to allow open-source project maintainers to focus on the important submitted issues, I have developed a model in R using text mining and logistic regression methods to classify an issue as 'critical' or not (or rather to apply a label of 'critical'). To simplify this undertaking, this paper focuses only on projects written largely in Python.

The following paper will examine the data collection and cleaning process as well as the various methods used. The results will be presented and critiqued. Lastly, opportunity for further statistical work will be presented.

II. DATA COLLECTION

Luckily, Github provides an API which allows developers to pull information about public repositories. Since different languages have significantly different keywords, error messages, and syntax, I decided to focus on projects written in Python, which is the third most popular repository language on Github[1]. To this end, I wrote a script using the API to pull the top 1000 open-source projects written in Python. I decided to pull the projects in a manner sorted by popularity because this correlates with number of issues as well as quality of follow-up, comments, and activity. This gives me a fuller picture with which to train my model.

For each repository, I stored its id, url, name, description, language, number of open issues, number of stars, number of people watching the repository, and number of times the repository was forked. For the non-technical reader, two things most be clarified.

[1] Zapponi, Carlo. "Githut: A Small Place to Discover Languages in Github." *GitHut – Programming Languages and Github*. 2014. Web. 29 Apr. 2016.

First, an issue can either be opened or closed. An opened issue means it has not yet been resolved (this is what the resulting model of this undertaking should take in as an input and output a label). A closed issue has been resolved and has no more activity. Second, the number of stars, people watching and times a repository was forked are all measures of popularity and use by the community. The exact meaning of the three metrics is unimportant.

Moving on, using the ids of the stored repositories I searched for their closed issues which were labeled. As labeling is optional, it was important to filter for closed issues with labels; otherwise, the model would have no response variable to train on. As the script would have taken 24 hours to run due to rate-limiting on the API, I stopped the script after it had collected over 25,000 labeled and closed issues. These issues came from 600 of the most popular repositories. The features of issues which I collected included its id, title, body, issue number, number of comments and labels.

III. DATA CLEANING

Before analyzing features of the issues and the repositories to which they belong, I cleaned both sets of data.

For the repositories, I checked to make sure there weren't any gaps in the data. I then remove any repositories which weren't written python (the API returned one which was not). I then removed one project which was actually just a compilation of resources (awesome-python). As the API only returns a repository's count of open issues, I created variables for the count of closed issues and the count of total issues.

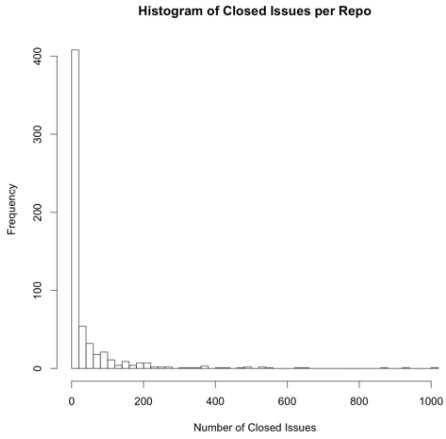
For issues, I double checked that each issue belongs to a repository which I had collected. I then checked again to see if there were any gaps in the data – there weren't. Next, I set the number of comments as an integer. I then created a variable for whether the issue body includes code snippets, which are delineated in the body by triple tick marks (```). I made this a factor variable. I then created a variable for the size of the body in characters. Next I pulled attributes of each issue's repository as a variable. This included the parent repository's open issues count, closed issues count, and total issues count. As Github promotes open-source behavior, it allows repository owners to create custom labels. Because of this, there were around 900 unique labels for the more than 25,000 issues. In a rather pain-staking fashion, I categorized each label as either being 'critical', meaning there was a bug or issue that needed to get fixed, or not 'critical'. I called this variable status, and it was a factor variable with either '1' for 'critical' or '0' for not 'critical'.

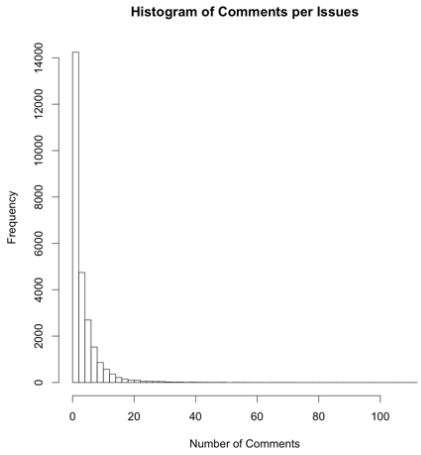
IV. DATA EXPLORATION

After cleaning and preparing the data, I was left with 26,316 labeled and closed issues belonging to 600 repositories. I began by exploring characteristics of repositories. The first step was exploring popularity metrics for repositories. In terms of stars, the average number of stars for the top 600 repos was 1,043, with the most popular repository being the requests library by Kenneth Reitz with 18,710 stars. Moving on from popularity to issues, the average number of total issues was 106.8, where the maximum of 2,835 issues belonged to the sympy library for symbolic mathematics (not necessarily a good sign!).

Moving closer to the dataset used for this paper, the average number of closed issues was 43.15, while the minimum was 0 and the maximum was 1019 closed issues. The maximum belonged to Bokeh, a Python interactive visualization library. In terms of the issues themselves, 10,905 of the 26,316 were labeled as dealing with a 'critical' bug. 6,844 issues included a code snippet in the body of the submission. The average closed issue had 3.7 comments, with the maximum having 111 comments – this belonged to an issue that noticed a certification issue that led to security problems.

The following are some of the relevant histograms:





V. FINDINGS

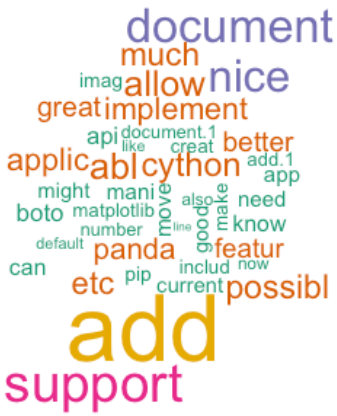
Among the important results of this project were the outcomes of LASSO analysis in selecting important variables as well as the production of two word clouds which give a visual of what words allow us to determine whether or not the issue refers to a 'critical' bug. The important finding was that the use of LASSO combined with logistic regression gives us a model with a 31% misclassification error, which – while not amazing – is good enough for maintainers to leverage to reduce the noise of issue submissions. The misclassification error seems to dramatically drop when random forest technique is used in RTextTools – all the way to roughly 1.5%, which motivated further research into this algorithm.

In terms of the variables which LASSO analysis selected, the table below shows the attributes of issues outside of text. As one can see, the presence of a code block in the issue is positively correlated with the issue referring to a 'critical' bug. Moreover, having a high number of closed issues slightly correlates with 'critical' issues; whereas, the opposite is true for the number of open issues:

Variable	**Coefficient**
Includes Code Block	.460196
Open Issues Count	-.0003
Closed Issues Count	.0015

Moving onto text mining, the following word cloud includes the words that are positively correlated with an issue referencing a 'critical' bug. The size of the word is based on the size of the word's coefficient in the logistic regression.

This second word cloud is the opposite visualization. It shows the words that are negatively correlated with an issue referencing a 'critical' bug, or rather it shows the words that are positively correlated with an issue not being a bug. Hence, it makes since why words such as 'add' and 'possible' are large in this cloud: these are words that could point to a feature request rather than something that's not working or broken:



This initial combination of LASSO variable selection and logistic regression on the document term matrix and issue attributes yielded a misclassification error of 31% and the following confusion matrix:

	Not Critical	Critical
0	2392	1021
1	582	1183

This led to the proposition of building a classifier on top of the DTM using a method other than glm. Using RTextTools, I found that the random forest method yielded a vastly improved misclassification error, which would supposedly follow from its ability to handle large numbers of parameters. The RF model from RTextTools weighs much more importance, as measured by the Mean Gini Decrease, on the number of issue comments and total issue count of the corresponding repository. Thus, while the logistic regression is easier to interpret and offers a platform for the conclusion that statistics can be used to allow open-source project maintainers to see through the noise, the random forest method as judged by initial results should strengthen this conclusion and make the day-to-day process even more accurate for project maintainers.

VI. DETAILED ANALYSIS

The statistical methods used in this project can be broken down into text mining, logistic regression with LASSO variable selection, and random forest classification.

A. Text Mining

After cleaning the data and creating the additional variables, it was important to leverage the body and title of the issues, as the hypothesis for this paper was that certain words allow us to assign a probability to an issue as referring to a 'critical' bug. I was able to use the body and title of the issues via the text mining library, tm.

For both the title and the body, I first created a volatile corpus (meaning it's stored as an R object in memory). I then followed similar sequential steps as those in standard text mining which use tm_map() to apply a transformation function to all of the single text documents – titles and bodies of issues – of the respective corpuses. The first transformation made all characters lower-case (although Python is case-sensitive, keywords are still reserved). The latter transformations included removing punctuation, numbers and stop words. Note that this model is supposed to predict whether an issue is referring to a 'critical' bug, not whether any code snippets in the body of the issue are erroneous (if this were the goal, transformations would take an entirely different form).

I next set a threshold of 2%, which means that words are dropped if they do not appear in at least 2% of the total documents in the corpus. This makes computation much easier, as it removes pure noise, or rather words that would likely be immediately driven to zero by LASSO selection. I used this frequency to find terms and add them to the control list.

"add"
"error"
"python"
"support"
"use"

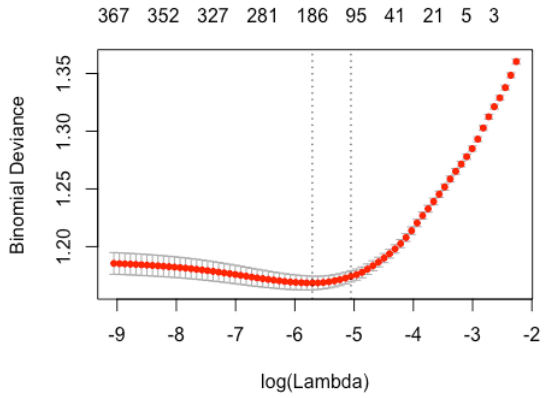
"file"
"import"
"line"
"python"
"use"

Top 5 Words in Titles (L) and Bodies (R)

After following these steps for both titles and bodies, I created a data frame including the matrices as well as the features pulled and created at the beginning of the project. From this data frame of 25,890 rows, I reserved 20% as testing data and used the remaining 80% for training using methods described in the following sections.

B. *LASSO variable selection*

After forming the data frame which included the sparse matrices, I had 384 variables. I needed to perform variable selection so that I could run reasonable regressions. I used LASSO selection as it imposes sparsity and makes the model easily interpretable. Through cross-validation, I selected lambda.1se as it left me with 126 non-zero coefficients in comparison with lambda.min and its 199 non-zero coefficients.



Besides certain words from the DTM, the attributes which were driven to zero by LASSO included number of comments, body size, and total issue count of the issue's repository. The open and closed issue counts remained, although their respective coefficients were very small. Not to my surprise, the factor variable of the issue containing a code snippet was left in and with a quite substantial coefficient (0.460196). Nevertheless, I wanted to derive the actual coefficients using classic logistic regression after using LASSO to get the right variables.

C. *Logistic Regression*

Using the 126 variables selected via LASSO, I used glm to run a logistic regression on the training data. This gave me optimal coefficients for each word, enabling the production of the word clouds displayed earlier in this paper. The following two tables show the top eight positive and negative coefficients, respectively:

"origin"	.732
"reproduce"	.668
"fail"	.641
"work"	.56
"bokeh"	.531
"report"	.489
"error"	.458
"fix"	.454

"support"	-.757
"nice"	-.548
"document"	-.547
"allow"	-.405
"able"	-.392
"cython"	-.364
"much"	-.339
"possible"	-.336

As can be concluded from the tables, words on the left align much more with bug or error reports. "Origin" and "reproduce" referring to suggestions or thoughts about causes. In contrast, words on the right seem to be from sentences requesting features, better documentation, support for a use case, or asking questions.

After analyzing the word clouds, I then used the model to predict classifications of issues. From there, I calculated the mean misclassification error, 31%. While the misclassification error doesn't represent 50% of pure guesswork, I believed that it could be improved. I decided to quickly test this hypothesis out with the help of RTextTools.

D. *Alternate Methods*

Using the RTextTools library, I created a container with the testSize set to the same index as used before. I then made trained and classified models using the glmnet, random forest, boosting, and support vector machine algorithms. The MCE for each algorithm is in the following table:

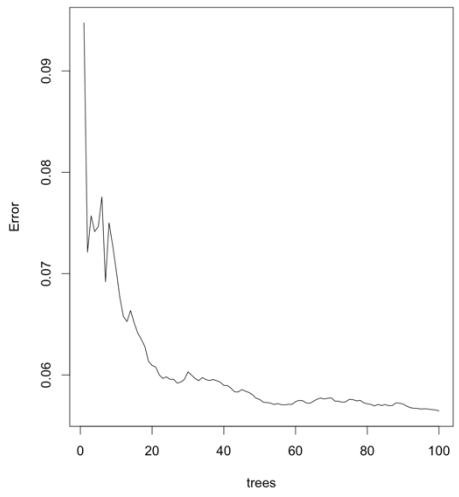
"GLMNET"	"RF"	"BOOSTING"	"SVM
.307	.016	.347	.005

The table shows that boosting and glmnet result in similar misclassification errors, while random forest and support vector machine provide clear improvements when using MCE as a metric. Due to my having more experience with random forest methodology than SVM methodology as well as the similarity in MCE between the two, I elected to further explore using random forest to build a classifier on top of the features and matrices I had already designed.

E. Random Forest Classification

Random forests are helpful in that the method maximizes the ability of standard decision trees to avoid overfitting the data while minimizing the bias and variance associated with normal decision trees. To enable greater scrutiny of results, I used the RandomForests library for classification and regression.

I grew 100 trees, and used the same testing data as before. With 100 independent trees, the resulting mean square error of regression was roughly 6%, appearing as an improvement from that of glmnet. The falling error rate that comes with generating a greater number of trees is clearly depicted in the following graph:



In terms of the difference in the model, it is useful to examine the MeanDecreaseGini. This measure builds on top of the Gini impurity index, which is use for calculating split, and illustrates the importance of individual variables. The MeanDecreaseGini in this case exhibits differences from the model suggested by LASSO variable selection. The top 10 variables as judged by this measure are displayed below:

"exit"	49.469
"api"	43.855
"file"	37.594
"master"	26.096
body_size	24.403
"response"	23.471
"method"	22.093
"return"	21.952
repo_open_issues_count	20.909
"label"	20.077

The distinct changes from the variables selected by LASSO are the inclusions of the variables body_size and issue_comments. In the LASSO model, these variables had coefficients of zero.

After switching from regression to classification using the randomForest library, it unfortunately became clear that the misclassification error, 33.9%, was not an improvement from that of glm. Thus, at the end, there was no substantial improvement found by using random forest technique, although it included variables that LASSO removed.

VII. OPPORTUNITY FOR FURTHER RESEARCH

While the misclassification errors from both logistic regression and random forest are rather high for the model to be used out-of-the-box, the principle of applying statistics to analyze issues on open-source repositories opens a door to an exciting future. The first step in working towards a better model would

be in depth-analysis of labels. It might be a worthwhile exercise for the maintainers to all use a standard set of labels (of which they can have customizable sub-labels), rather than the statistician inferring what labels should be categorized as referring to a 'critical' bug.

Additionally, mining the text from comments on submitted issues would likely lead to a stronger model and one that improves with each additional comment. As the data collection was a very slow process in this project, pulling and analyzing comments was beyond the scope.

In a more technical vein, there have been attempts of late to interpret both questions and code – some quite significant ones by Microsoft Research[2]. The idea can be posited that through a reverse search of code snippets, previous questions on sites like StackOverflow could provide additional metadata to use in model building.

Lastly, the transformation functions used on the body corpus can be dramatically improved to specialize in transforming code snippets. Some of the words that appeared to influence prediction were keywords and names of Python modules. A method of parsing code snippets from text would give a cleaner body to mine, likely leading to better results as R libraries such as tm are built for natural, not machine, language.

VIII. CONCLUSION

In terms of closing remarks, the process of combining logistic regression and text mining on issues did not yield as quality of a model as I initially hoped. Nonetheless, I think it is much more a factor of tm not being suited for machine language than inherent weaknesses in the statistical methods used in this paper.

The exercise of running LASSO variable selection and then logistic regression to build a word cloud definitely confirmed the hypothesis that certain words are occur more frequently in the bodies of issues referring to 'critical' errors than in the bodies of issues that discuss feature requests, questions, or additional documentation. While I must admit that the classification process yielded a rather high error, the predictions generated from the models could still be used for a scalar rating of issues in importance, with 1 being critically important.

Nevertheless, as open-source software projects continue to support a growing percentage of the world that we know, it will become increasingly important to react in a timely manner to any flaws that may exist in the source code. In order to allow those who maintain the software to focus on the signals and not the noise, statistics must be used to point them in the right direction. While this attempt is by no means a solution for today, it should open the door to thinking about how we can apply statistics to code maintenance and explore how we can tweak existing libraries and methods to suit the languages the power our technology.

[2] Raghothaman, Mukund, et al. "SWIM: Synthesizing What I Mean. Code Search and Idiomatic Snippet Synthesis." 13 Feb. 2016. Web. 29 Apr. 2016.