Automatic Labeling of Issues on Github – A Machine learning Approach

Arun Kalyanasundaram

December 15, 2014

ABSTRACT

Companies spend hundreds of billions in software maintenance every year. Managing and resolving bugs efficiently can reduce the cost of software maintenance. One of the key steps in the bug resolution process involves grouping bugs into different categories. However, this is done manually by assigning labels to each bug. Therefore, our goal is to automate this step by using machine-learning techniques to predict labels assigned to bugs, from a given set of predefined labels. Our dataset contains a set of bug reports (or issues) from the BootStrap project on Github. Our analysis showed that different classifiers performed better on different subsets of features. This suggested that we use Stacking. Therefore, our final model included two Naïve Bayes classifiers and one SMO classifier combined using Stacking with a JRip meta classifier. The performance of our final model was estimated with 2500 instances using a 10-fold cross validation to have an accuracy of 92% and a kappa of 0.85, which was about 15% higher in accuracy and 44% higher in kappa than the baseline performance. The results on our final test data, which consists 1000 instances were comparable with an accuracy of 85% and a kappa of 0.70.

1. INTRODUCTION

Bugs in software are inevitable. Bugs are expensive too, costing companies billions of dollars each year. Therefore, being able to accurately detect bugs and effectively resolve them can significantly reduce the cost of software development. There has been a growing interest in applying machine learning techniques to identify and resolve bugs.

Software defect prediction is the problem of identifying whether a piece of software (or parts of it) is buggy or not. There has not been a significant success in using machine learning techniques in addressing this problem. Challagulla et. al [5] performed an empirical assessment of several machine learning techniques and found that the maximum accuracy they could achieve was about 51.56%. On the other hand, machine-learning techniques have been successfully applied to facilitate the process of resolving a software bug. This process involves the several stages that a defect goes through before it is resolved or fixed. This paper aims to facilitate one stage of this defect resolution process using machine-learning.

A software defect is often represented in the form of a bug report. Almost all bug reports contain a description of the, who reported and when. One of the stages of bug resolution process is assigning the bug to an appropriate developer. This is a challenging problem and popularly known as bug triaging. Several approaches have been proposed in the literature, some of these include a supervised Bayesian learning based text categorization [8], using a latent semantic indexing method with a support vector machine [1], and using an SVM with a combination of text and other features with a set of classifiers [4].

Assigning one or more labels to categorize bugs is one of the ways in which the process of bug allocation can be facilitated. Some of these categories include, the severity of the bug, modules that might need change, technical knowledge and effort required. However, this is also an equally challenging problem. The simplest form of this problem is to classify whether a given bug report is actually a defect in the software that requires corrective maintenance or requires other kinds of activities such as adding a new feature, change documentation, or an invalid use case. In this regard, an accuracy of between 77% and 82% was achieved with simple decision trees, naïve Bayes classifiers and logistic regression for several open source software projects – Mozilla, Eclipse and JBoss [3].

Another form of bug classification is determining the severity of the bug. This is extremely useful in bug triaging as it helps in both prioritizing and allocating the bug. An array of machine-learning techniques was used to predict severity of bugs from NASA's PITS projects, with a maximum prediction accuracy of about 96.7% [6]. They also showed that the type of classification algorithm depends on the project.

Github[1] is a popular platform for hosting and managing open source software projects. Github provides a bug-tracking tool, which each project can use to create and manage bug reports. These bug reports are technically called 'Issues'. The tool allows users to assign one or more custom labels to each issue. These labels are useful in prioritizing and categorizing issues. However, assigning these labels requires considerable manual effort. Therefore, the goal of my project is to facilitate this process by using machine learning techniques to predict the set of labels that can be assigned to an issue.

The rest of the paper is organized as follows. Section 2 explains the goals and problem statement. Section 3 gives a literature review of the current state of art papers relevant to this problem. Section 4 talks about the dataset used for the project and techniques for data collection. Section 5 explains on how we came up with our features. Section 6 gives a high level approach of our model. Section 7 shows our initial baseline performance and our approach to improve it. Section 8 and 9 provide details on error analysis and optimizing the individual models used in Stacking. The implementation and evaluation of the final model is presented in Section 10. Section 11 shows the results from evaluating the final model on our final test data. Section 12 concludes with scope for future work.

[1] https://github.com/

2. PROBLEM STATEMENT

Given a set of issues (bug reports) of a project on Github and a set of predefined labels, the goal is to assign one or more labels to an issue. It is fair to assume that the labels are independent and therefore, the problem reduces to a binary classifier for each label. In other words, the goal is to predict whether a given label can be assigned (TRUE) or not (FALSE) to a given issue. Repeating this prediction for all labels, will give us a set of labels that can be applied to an issue. However, each label could require a different model for prediction. Therefore, for a deeper analysis of particular technique, we restrict the scope of this paper to the prediction of only one such label for one of the projects on Github.

3. RELATED WORK

It is important to review literature that addresses both bug identification and bug resolution, since they share the same domain and to a large extent, operate on the same data.

Identifying whether a given code is buggy or not is a challenging problem. An automated analysis not only involves looking at the source code but also other factors such as who modified it, how many times was it modified, and so on [9]. An empirical assessment of several machine-learning based approaches on bug identification showed that a maximum accuracy achievable was not more than 51.56% [5].

On the other hand, there has been reasonable success in categorizing bugs. Most of the approaches in prior literature have used text based categorization [6] [3]. A simplest form of categorization is to determine if a given bug requires just corrective maintenance or could require adding a new feature [3]. A common form of bug categorization is assigning a severity label to a bug [6]. These are typically a set of five ordered labels, where the lowest number would indicate least severe and highest indicates most severe bug.

Our problem is different from previous bug categorization problems in two ways. First, the set of categories or labels is unrestricted since it can be any user-defined text. In other words, each label has a meaning that is defined by the user and two or more labels may or may not be related. Second, we are using data from a social coding platform called Github that no prior work on this problem has used. Since Github allows users to collaborate in interesting ways, our approach to design new features is also a novel contribution to this problem.

4. DATASET

I used a subset of issues from a well known project on Github called "BootStrap"[2]. The project has over ten thousand issues and about thousand contributors. As of December 2014 the project has over ten thousand issues and about a thousand contributors. It uses a set of 12 user-defined labels to categorize issues. For example, a label called "CSS" is used to tag an issue to mark it as an issue related to the CSS technology used in web development. Another example is a label called "docs", which could indicate that the issue might need a change in the project documentation. These labels help users to quickly browse issues of relevance to them.

For this paper, I chose the do a binary classification to predict whether a given issue has the CSS label or not. Therefore, the class variable is a nominal variable with values TRUE or FALSE. I chose a random subset of 3900 labeled issues from the Bootstrap project such that fifty percent of these issues had the CSS label. Hence, the class value is TRUE in fifty percent of instances and FALSE in the other fifty percent. Ideally, I should have chosen the distribution of class values that reflects the real data. However, since multiple labels can be assigned to a single issue, using the distribution of a class label in a given project is highly unreliable. Therefore, I chose a uniform distribution of the class values in my dataset. I divided my dataset for development, model building and testing as shown following table.

TABLE 1

Dataset	**#Instances**
Complete Data	3900
Development Data	400
Cross Validation Data	2500
Final Test Data	1000

5. FEATURE SPACE DESIGN

Since this is not a readily available dataset, I designed a set of features based on our domain knowledge and improved them iteratively. I used the Github API to extract the identified set of issues in the Bootstrap project. The source code used to mine data is available online[3].

Each issue has a title and a body (description of the bug). It was obvious to use them as the set of text features. The issues also have other attributes (meta-data or column) such as who created it, and the creation time. Since I only considered closed (or resolved) issues in my dataset, I could also use the closing time and the number of comments in each issue as my features. Table 2 lists the first iteration of column features in my dataset.

TABLE 2

[2] https://github.com/twbs/bootstrap/

[3] https://github.com/arunk054/machine-learning-algos/tree/master/github-issue-labeling/data-collection

Meta Data Feature	Type
Created By	Nominal
Creation Time	Both
#Comments	Numeric
Closing Time	Both
Time taken to close in hours	Numeric

Throughout this paper I will refer to the meta-data features as column features. The feature 'Created By' is the username of the user who reported the issue. This feature had 1881 unique values out of 2500 instances in the cross validation data. Clearly, this feature will have a poor predictive power. Therefore, we decided to extract certain characteristics about each user. Table 3 shows the features extracted for each user.

TABLE 3: Extracted Features of "Created By"

Feature	Type
#Followers	Numeric
#Following this user	Numeric
#Repositories Starred	Numeric
#Repositories Watched	Numeric
#Repositories Owned	Numeric
Programming Language of repositories watched or starred	Both
Programming Language of repositories owned or contributed to.	Both

Each user has a set of repositories that they own, watch or star. The programming languages that a user knows or is interested in can be obtained by identifying the programming language of each such repository. There were a total of about 75 unique programming languages in all repositories. These features were named with a prefix Repo_ or Watch_ followed by the programing language to indicate whether they owned the repository or watched / starred the repository respectively. Further, these can be treated as nominal (whether atleast one repository had the programming language or not) or numeric (the actual number of repositories).

The creation time contained the date, month, year and time when an issue was created. They were spread between years 2011 to 2014. Looking closely at the development data, there was some correlation between the month and class value. So we decided to only use the 'month' of the creation time and closing time. Also the time taken to resolve an issue looked promising. Hence, we added the difference between the closing time and creation time in hours as another feature. This process resulted in about 129 column features.

6. HIGH LEVEL APPROACH

In this section, we will summarize our final model. The subsequent sections will talk about how the model was iterated using error analysis.

We found that a naive bayes classifier performed better than an SMO when only text features were used. On the other hand, an SMO performed better than naïve Bayes when only column features were used. When the entire set of features was used, SMO showed a slight increase in performance, however, NB showed a significant decrease in performance. We found that NB with text features gave the best performance among the above six models. So, on one hand, we only want to use NB with text features, while on other hand we also want to leverage the predictive power of column features. Therefore, this suggested that we use NB with text features and SMO with both text and column features, and combine both models using Stacking with JRip as a meta classifier. We found that model built using the stacked classifier performed significantly better than any of the above four models individually.

7. BASELINE PERFORMANCE

Prior literature suggests that both weight based and probabilistic models are reasonable choices for the problem of bug categorization. Therefore, we used an SMO and a Naïve Bayes (NB) classifier over the entire set of features on the cross validation data, and performance values are shown in Table 4.

TABLE 4: Performance on entire set of features

Classifier	Accuracy	Kappa
NB	0.79	0.56
SMO	0.80	0.59

Further, previous approaches in the literature showed that NB performed significantly better in text based categorization of bugs. Therefore, we ran both NB and SMO selecting only text features (unigrams) and then with only the column features. The performance values are shown in TABLE 5.

TABLE 5:

Classifier	Features	Accuracy	Kappa
NB	Text	0.84	0.68
NB	Column	0.58	0.1
SMO	Text	0.79	0.57
SMO	Column	0.62	0.18

Let us consider SMO on the entire set of features as our baseline model, we have a baseline performance of – **accuracy: 0.80 and kappa: 0.59**. From Tables 4 and 5, we see that NB with text features outperforms the baseline performance. Since we want to use the better performing NB classifier with text features and the predictive power of column features, we combine both these models using stacking. The assumption is that the final model will perform significantly better than either of the models.

Therefore, our goal is to optimize the following two models before combining them with stacking.

1. NB with text features
2. SMO with text and column features

8. OPTIMIZE NB WITH TEXT FEATURES

We used Light Side to do our error analysis. Using an NB classifier with unigram text features on our development resulted in 47 false positives and 33 false negatives. We first looked at features with highest horizontal difference in the misclassified cells of the confusion matrix. We found that the word ‘click’ is often used when there is an issue with the user interface. We also noticed that ‘click’ was used interchangeably with clicking, clicked, clicks and clickable in the same context. Therefore we decided combine these terms with a regular expression – cilck|click(s|ed|ing|able).

Looking at some of the misclassified instances we found that a few of them used the phrase ‘vertically aligned’ to refer to some issue with the alignment on the user interface. This suggested we create a bi-gram feature with the same phrase.

We found that the modified feature space gave a performance of accuracy: 0.84, kappa: 0.67, which is slightly lower than the performance of the unmodified feature space. Therefore, we decided not to use the features identified during our error analysis

9. OPTIMIZE SMO WITH BOTH TEXT AND COLUMN FEATURES

Since we already looked at improving the text features in our model with NB classifier, we wanted to understand how the column features behaved in our model. I extracted all the column features in my dataset and used Weka's AttributeSelectedClassifier with a ChiSquare evaluator and a Ranker. The purpose of doing this was to identify the features with high predictive power. The usual approach would be to build a model with a subset of features that have a high predictiveness score and compare its performance with baseline. I found that selecting any subset of features never gave better performance than selecting all features in my SMO based model. Also, I realized that narrowing down on features based on the ranking might over-fit the model to the training data.

However, looking at the ranked features gave me a useful insight. I found that some features that were ranked higher by the feature selector had very low weights in the SMO classifier. This indicates that these features had a good correlation with the class value but for some reason the SMO assigned low weights to these features. I thought these features could be good candidates to do error analysis. This approach when combined with the approach of looking at horizontal (or vertical) differences is more powerful.

I found that the highest ranked feature - "number of hours to close" had a very skewed distribution. This suggested that I do a log transformation of the values. To avoid undefined values with logarithm of zero, I added one to each value before doing the log transformation. I built an SMO classifier after adding the newly added feature to my entire feature set (both text and column) and got a slightly lower than baseline performance of accuracy: 0.79 and kappa: 0.59. However, when the newly added feature was used with only the column features, I got a statistically significant improvement in accuracy: 0.63 kappa: 0.21. Since my final SMO model will use both text and column features, I decided not to use this feature in my final model.

I found that a number of programming language related features (explained in Section 5) had a high rank but had low weight in the SMO classifier. After looking closely into the feature values in the development data, it appeared that almost all such features were sparse. In other words, the features had more than fifty percent of instances with zero values. This suggested that a boolean transformation will be appropriate. There were 120 programming related features and so, I realized that doing the boolean transformation to only a subset of these features might over fit the model. Therefore, I converted all of these features to boolean values, and the got a lower than baseline performance of accuracy: 0.77 and kappa: 0.54 using an SMO classifier. Interestingly, the NB classifier gave a higher than baseline performance with this boolean transformation, accuracy: 0.83 and kappa: 0.65. Using the experimenter we found that this was statistically significant than the baseline performance. Although it is surprising that the boolean transformation significantly increased the performance of NB, a possible explanation is that because the transformation made the feature values normally distributed for each class value.

Prior literature [2] in bug categorization has showed that subset of features can be selected based on certain semantic similarities. Using this idea, I was able to identify two groups of features from among the set of column features in my dataset. The first is the content specific features. These features are the attributes related to content in the issue, such as creation time, closing time and the number of hours to close. The second set of features is the user specific features. These are the attributes related to the user who created the issue, such as the number of followers the user has, number of repositories contributed, the programming languages the user knows, etc. The performance on the first set of features was – accuracy: 0.59, kappa: 0.15 and second set of features was – accuracy: 0.6, kappa: 0.15. This showed that both the sets of features had almost similar predictive power, but neither is better than the performance of the combined set of features.

The error analysis techniques so far allow us to identify features that we might want to tweak or add. However, having identified a feature, a more formal approach to do error analysis on meta-data attributes (column features) is to compare the distribution of feature values between misclassified instances and correctly classified instances. Unfortunately, I could not find a tool that readily does this. LightSide only shows the values of text features. However, Lightside allows us to export a set of instances, either correctly classified or misclassified. Although this only gives the IDs of the instances, one could use a script to extract the corresponding instances from these IDs.

Also looking at low ranked but with high weight features might also be interesting. Although this is not a common scenario, however, if it does occur these could be features that might have been over-fitted to the model. So we have to be careful if such a scenario occurs. However, this was not observed in this dataset.

I tuned my SMO classifier with c = 2.0 and c = 3.0, to account for any non-linear relations. I found that when the SMO classifier is used with only the column features, there is an increase in performance with c = 2.0, however, when both text and column features are used, there is no increase in performance compared to the baseline. Therefore, I decided to use the baseline SMO classifier setting with c = 1.0 to build my final model.

From all the above error analysis, we found that the NB classifier with all features and a boolean transformation of the programming language specific features performed best. Therefore we decide to use this to include as one of our stacked classifiers.

10. FINAL MODEL – STACKING

When we evaluated our baseline performance we found that NB performs best with text features whereas SMO performs best with column features. The analysis in Section 7 showed that combining an NB classifier with text features and an SMO classifier on the entire feature set. In our error analysis we found that an NB Classifier with the entire feature set gave a significantly higher performance when a subset of features were transformed from count to boolean values. So we decided to include that model in our stacked classifier.

Therefore, to summarize the stacked classifier contains the following three models,

1. NB classifier with only unigram text features.
2. SMO classifier with the entire set of features.
3. NB classifier with the entire set of features but a subset of features transformed to boolean values. This subset is the set of programming language features.

The reason to include the SMO classifier in stacking was because prior literature [7] shows that combining a weight based and probabilistic model in stacking is an effective approach.

The meta classifier used in Stacking is JRip. I could have used any other meta-classifier since almost all classifiers gave a similar performance. However, a rule or a tree based classifier is usually an appropriate meta classifier in stacking.

10.1 MODIFYING WEKA'S STACKING CLASSIFIER

One major limitation with Weka's Stacking classifier is that it requires all stacked classifier to use the same set of features. Even the Weka Java API only allows one set of instances for all the stacked classifiers. Therefore, I created a new Java class which extends the Stacking class in Weka. In this new class, I modified the methods : *buildClassifier*, *generateMetaLevel* and *metaInstance* to work with an array of instance sets instead of a single instance set. I also added corresponding methods to perform cross validation and to evaluate a test set. The source code is available online[4].

10.2 EVALUATING THE PERFORMANCE OF THE FINAL MODEL

We could not use the Experimenter to compare the performance of the final model with the baseline. This is because we had to use a custom Stacking classifier as shown above. Therefore we used a statistical tool (R Studio) to compare the performance of each fold of cross validation for both models. The following table shows the performance of models on each fold.

Fold	Accuracy -Baseline	Accuracy -Final	Kappa - Baseline	Kappa -Final
1	0.8	0.93	0.59	0.86
2	0.82	0.94	0.63	0.87
3	0.8	0.91	0.6	0.82
4	0.8	0.93	0.59	0.86
5	0.79	0.9	0.58	0.81
6	0.84	0.93	0.67	0.87
7	0.76	0.94	0.52	0.88
8	0.8	0.91	0.61	0.82
9	0.79	0.91	0.58	0.82
10	0.77	0.94	0.53	0.87
Ave	0.80	**0.92**	0.59	**0.85**

A T-test to compare the final and baseline model gave a p-value of $1*10^{-10}$ for accuracy and a p-value of $8*10^{-11}$ for kappa showing that the performance gain is statistically significant for both accuracy and kappa. The estimated performance of the final model is accuracy: 0.92 and kappa: 0.85. The net increase in accuracy is about 15% and kappa is about 44%.

11. RESULTS - FINAL TEST SET

Since my final model had a statistically significant performance improvement over the baseline, I decided to use the final model to evaluate its performance on the unseen final test data. My final test set had 1000 instances. The performance on my final test set is **accuracy: 0.85** and **kappa: 0.70**. The performance although lower, is comparable to the performance of the final model.

The source code to build, evaluate and test the classifier, and the complete dataset used is available online[5].

12. CONCLUSION AND FUTURE WORK

We looked at the problem of bug categorization in software. We showed novel approaches to design and extract features for issues (bug reports) from the Github platform. Our analysis showed that Stacking multiple classifiers on subset of features will give the optimal performance. Our final model consisted two Naïve Bayes classifiers and one SMO classifier with Stacking using a JRip meta classifier.

One major limitation of our work is that we reduced the problem of multi-label classification to a binary classifier. This is based on the assumption that class labels are independent and do not interact. However, this may not always be true in all projects. Therefore, extending our work to a multi-label classification is useful. Applying our approach to other datasets and other projects on Github is a reasonable next step.

[4] https://github.com/arunk054/machine-learning-algos/tree/master/Extended_Weka_Classifiers/Stacking

[5] https://github.com/arunk054/machine-learning-algos/tree/master/github-issue-labeling/building-classifiers

REFERENCES

1. Ahsan, S.N., Ferzund, J., and Wotawa, F. Automatic Software Bug Triage System (BTS) Based on Latent Semantic Indexing and Support Vector Machine. *2009 Fourth International Conference on Software Engineering Advances*, (2009), 216–221.
2. Ahsan, S.N., Ferzund, J., and Wotawa, F. Automatic Classification of Software Change Request Using Multi-label Machine Learning Methods. *2009 33rd Annual IEEE Software Engineering Workshop*, (2009), 79–86.
3. Antoniol, G., Ayari, K., Penta, M. Di, and Khomh, F. Is it a Bug or an Enhancement ? A Text-based Approach to Classify Change Requests. .
4. Anvik, J., Hiew, L., and Murphy, G.C. Who Should Fix This Bug ? 361–370.
5. Challagulla, V.U.B., Bastani, F.B., Yen, I., and Paul, R.A. Empirical Assessment of Machine Learning based Software Defect Prediction Techniques. (2005).
6. Chaturvedi, K.K. and Singh, V.B. Determining Bug Severity using Machine Learning Techniques. .
7. Giraud-carrier, C., Vilalta, R., and Brazdil, P. Is Combining Classifiers with Stacking Better than Selecting the Best One ? (2004), 255–273.
8. Murphy, G.C. Automatic bug triage using text categorization. (2004), 1–6.
9. Pan, K., Kim, S., and Whitehead, E.J. Bug Classification Using Program Slicing Metrics. (2006).