Masaryk University
Faculty of Informatics

Computer-Aided Ticket Triage

Master's Thesis

Bc. Václav Dedík

Brno, Fall 2015

Replace this page with a copy of the official signed thesis assignment and the copy of the Statement of an Author.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Václav Dedík

Advisor: Bruno Rossi, PhD

Abstract

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Keywords

Bug, Machine Learning, Assignee

Contents

1 Introduction

The advancements in the 21st century brought many improvements in *machine learning* and *natural language processing* allowing us to predict and determine the correct action based solely on prior knowledge with almost no human interaction. The possible applications of these techniques are very broad ranging from medicine to transportation, engineering, research and many more. The domain of this thesis is to utilize machine learning and natural language processing algorithms in software engineering.

With the advent of computer software in our contemporary world, it was quickly discovered that software is almost never perfect and it needs to be continuously maintained as new issues (*bugs*) keep arising as long as the computer program is used. It is not unusual to discover thousands or even tens of thousands of bugs in a software application and therefore there are usually more than one developers working on these bugs trying to fix them. For convenience and to reduce the effort necessary to maintain the list of the issues, software development community came up with a web application designed specifically for keeping the track of bugs. This software is called an *issue tracker* and one entry of an issue is called a *bug report*. A bug report usually contains the summary, description and *assignee* of the discovered bug, as well as other fields (priority, status, comments etc.). Assignee is the developer who is assigned to investigate and possibly fix the bug. The necessity of determining the assignee raises an important question – who should fix the bug? The goal of this thesis is to find a way to answer this question in real time with the involvement of almost no human interaction.

1.1 Objectives

Primary objective of this thesis is to evaluate different machine learning models and to determine which one with what configuration is the best choice for this problem considering all its aspects. Most important aspect of all is the performance, i.e. the number of cases in which the computer-generated prediction is correct. Another important aspect is the time complexity of the model and the resources cost (hardware requirements). The existence of a great number of models with even

greater number of possible configurations makes the evaluation rather complex, which is the reason why not all models in all its possible configurations are evaluated. Instead, the more logical choices determined from related works are studied.

Another objective is to study related works to eventually compare our results with previous attempts. This will also allow us to determine the baseline for our own experiments and pick the best candidate models.

Using machine learning and natural language processing requires some sort of dataset to be passed to the learning algorithm. In this case, we are using old bug reports from various issue trackers. One of our main goals is to analyze these datasets to estimate their relative (di)similarity as well as optimize their attributes (e.g. size, date of creation) to aid the models to achieve the best possible qualities.

As we were able to retrieve data from open-source projects (accessible on the Internet) as well as from a proprietary project, our last objective is to compare open-source and proprietary data. We will focus not only on the evaluation of performance with these datasets, but also on the analysis.

1.2 Outline

First chapter is an introduction to the machine learning domain as wall as the problem domain of this thesis, summary of its objectives and the methodology used to fulfill them. In the second chapter, we will summarize some related works from various scientific journals. ... (1) The XXX chapter is the core of the thesis, it includes the evaluation of the models and analysis of the datasets. The last chapter concludes this study with a summary of our findings and an outline for possible future work.

2 Related Work

Over the years there have been many attempts to automate bug triage. First effort was made by Čubranić and Murphy [1] who used a text categorization approach with a Naive Bayes classifier on Eclipse data. Their feature vector included a bag of words constructed from the data with stop words and punctuation removed paving the way for subsequent attempts of automatic bug assignment. Number of reports in their dataset is 15,670 with 162 classes while the downloaded reports were created in a span of half a year. They achieved about 30% accuracy and also showed that stemming has no real effect on classification performance, which is a main reason why many future efforts do not use it. Another interesting conclusion is that removing words from feature vector occurring less than *x*-times seems to decrease the performance rather than increase it.

Anvik et al. [2] also chose a text categorization approach but instead of a Naive Bayes classifier, they used Support Vector Machine (SVM) on Eclipse, Firefox and GCC data. Similarly to [1], they used a bag of words with stop words removed. The amount of reports in the Eclipse dataset was 8655 and 9752 in the Firefox dataset. Reports from both datasets were created in a span of half a year. For Firefox data, if a report was resolved as FIXED, they labeled the data with the name of the developer who submitted the last patch, for Eclipse data, the name of the developer who marked the report as resolved was used. DUPLICATE reports were labeled by the names of those who resolved the original report, both for Eclipse and Firefox. WORKSFORME (only applicable to Firefox) reports were removed from the dataset. In total only 1% of Eclipse data was deemed unclassifiable and thus removed, in contrast 49% of Firefox data was removed. With this approach, precision of 64% and 58% was achieved on Firefox and Eclipse data respectively, however, only 6% on GCC data. As for recall, only 2%, 7% and 0.3% results were achieved on Firefox, Eclipse and GCC data respectively.

Ahsan et al. [3] used an SVM classifier on Mozilla data. Text terms were weighted using simple term frequency (TF) and term frequency inverse document frequency (TF-IDF). Furthermore, they reduced the number of features by applying feature selection (all terms that did not

appear in at least 3 documents were removed from the feature vector) and latent semantic indexing (LSI). Only resolved and fixed bug reports were considered, the rest such as duplicate were removed from the training dataset. In total, 792 bug reports were used for this classifier with 18 labels after removing reports that were fixed by developers that had not fixed at least 30 bugs in this dataset. The data was labeled by the name of the developer who had changed the status of the bug report to resolved. This approach has 44.4% classification accuracy, 30% precision and 28% recall using SVM with LSI.

All previous efforts used supervised learning approaches, Xuan et al. [4] used a semi-supervised text classification to avoid the deficiency of unlabelled bug reports in supervised approaches. Using Eclipse data, they trained their classifier in two steps. First, a basic classifier was built on labeled data, second, the classifier was improved by labeling the unlabelled dataset with current classifier and rebuilding the classifier on all data. The number of reports was 5050 with 60 labels after removing all reports that were fixed by developers that resolved less than 40 of them. As for the classifier, Naive Bayes with a weighted recommendation list (used to further improve the second step of training the classifier) was employed. With this setting, the classifier achieves a maximum of 48% accuracy for top-5 recommendations but only 21% for top-1 recommendation.

Shokripour et al. [5] chose an entirely different approach that I will mention only briefly as versioning data were needed for bug assignment. Instead of text classification used previously, Information Extraction methods on versioning repositories of Eclipse, Mozilla and Gnome projects were employed accomplishing recall values of 62%, 43% and 41% respectively for top-5 recommendations.

Quite an extensive study was done by Alenezi et al. [6], who used a Naive Bayes classifier on Eclipse-SWT, Eclipse-UI, NetBeans and Maemo. Bugs that were fixed by developers who resolved less than 25 bugs in the last year were removed from the dataset resulting in the size of the datasets equal to 6560, 6104, 9284 and 4659 reports for Eclipse-SWT, Eclipse-UI, NetBeans and Maemo respectively. What makes their research different from others is their feature selection. 5 different methods were used to reduce dimensionality of the feature vector, namely Log Odds Ratio (LOG), which measures the odds of a term occurring in a positive class normalized by the negative class, χ^2, which exam-

ines the independents of a term in a class, Term Frequency Relevance Frequency (TFRF), Mutual Information (MI) and Distinguishing Feature Selector (DFS). Best results were achieved using χ^2 resulting in precision values of 38%, 50%, 50% and 50% and recall values of 30%, 35%, 21% and 46% on Eclipse-SWI, Eclipse-UI, NetBeans and Maemo projects respectively.

Somasundaram and Murphy [7] used SVM model with Latent Dirichlet Allocation (LDA) feature selection and Kullback Leibler divergence (KL) with LDA. They tested their models on Eclipse, Mylyn and Bugzilla data and claim to be using recall metric for comparison, although it must be pointed out that the equation they used for the computation of recall matches equation for accuracy, not recall, this is probably an error either in the metric they actually used or in the paper. For Bugzilla data, the number of components (categories) is 26 and the amount of training data is 6832. With this setting, they achieved recall values of 77% and 82% on SVM-LDA and LDA-KL models respectively. In addition, the LDA-KL model seems to produce more consistent results when the number of categories changes.

Very interesting idea is to use Content-based Recommendation (CBR) or Collaborative Filtering (CF). One of the interesting attempts was conducted by Park et al. [8]. They employed a Content-boosted Collaborative Filtering (CBCF), which is the combination of the two mentioned recommender algorithms, with a cost-aware triage algorithm CosTriage that tries to prevent overloading developers. The data used for the training was downloaded from Apache, Eclipse, Linux kernel and Mozilla projects. From these datasets, they removed all bugs that were fixed by inactive developers (determined by interquartile range) resulting in datasets of 656 reports assigned to 10 developers for Apache, 47,862 reports and 100 developers for Eclipse, 968 reports and 28 developers for Linux kernel and 48,424 reports assigned to 117 developers for Mozilla. Reports from all repositories were created in a period of about 8 to 12 years. In this setting, accuracy equal to values 64%, 35%, 25% and 59% was achieved on Apache, Eclipse, Linux kernel and Mozilla projects respectively while taking into account cost of each developer recommendation.

Thung et al. [9] chose a semi-supervised learning algorithm for bug classification to 3 defect families and were able to achieve some very good results. This approach combines clustering, active-learning and

semi-supervised learning algorithms and the main feature is that it requires very few labeled samples to achieve good classification results. The used dataset consists of 500 bug reports from Mahout, Lucene and OpenNLP projects. This model is able to achieve a weighted precision, recall, F-measure and AUC of 65%, 67%, 62%, and 71% respectively.

Xia et al. [10] employed an interesting algorithm they named DevRec based on multi-label k-nearest neighbor classifier (ML-kNN) and topic modeling using Latent Dirichlet Allocation (LDA). The algorithm uses a composite of BR-based analysis and D-based analysis. BR-based (Bug report based) analysis computes a list of scores of developers that are potential resolvers of a new bug report. D-based (Developer based) analysis computes affinity of a bug to a developer based on their history (if they resolved some bugs in the past). The composite DevRec score is computed as a weighted sum of BR score and D score. The dataset used for the evaluation of this model is from GCC, OpenOffice, Mozilla, NetBeans and Eclipse consisting of about 6,000, 15,500, 26,000, 26,000 and 34,400 bug reports respectively. Reports assigned developers who appear less than 10 times were removed. For top-5 recommendation, the recall values are 56%, 48%, 56%, 71% and 80% and the precision values are 25%, 21%, 25%, 32% and 25% for GCC, OpenOffice, Mozilla, NetBeans and eclipse respectively. For top-10 recommendation, the recall values increase by about 10-15% while the precision values decrease by about 7-10%.

3 Methodology

This chapter describes the methodology of the evaluation as well as its application. It can be quite difficult to put together a good structure for a paper or a study that details some complicated research especially in a highly specialized field. With this in mind, we decided to opt for a framework proposed by Victor Basili of the University of Maryland, College Park and the Software Engineering Laboratory at the NASA Goddard Space Flight Center [(2)] called *Goal Question Metric* (or just GQM).

3.1 The Goal Question Metric Approach

The focus of the GQM approach is to define a good measurement mechanism mainly for engineering disciplines like software engineering, computer science and others. Application of this approach helps support project planning, determine pros and cons of the current project and process, provide an intuition about the impact of modifying or refining a technique as well as assess current progress and last but not least to write the final work in a comprehensible and structured way.

The first prominent step in terms of GQM is to define a set of goals. Goals are entities that are to be assessed and therefore must be defined in a way that allows their assessment. Second step is to define a set of questions, these characterize the way the assessment of a goal is going to be carried out. The last step at the very bottom is to define metrics that are used to answer the questions in a quantitative way. the usual workflow is to define goals and then refine each into several separate questions. The questions are then further refined into metrics where more than one question can have the same metric in common.

3.2 Application of GQM

In this section, we apply the GQM methodology to our project. We proceed from top to bottom (i.e. from goals to metrics) as defined by Basili.

3.2.1 Goals

First, we define goals. In our case, there is only one general goal that looks like this:

> Goal 1:
>
> Analyze **machine learning models** in order to **find the best possible approach** with respect to **assignee prediction** from the point of view of **project managers** in the context of **issue tracking systems**.

3.2.2 Questions

Next, we need to refine questions for our single goal. We define seven questions, each one addressing a different concern of our research.

> Question 1:
>
> Is the performance of the model better than performance of some baseline model?
>
> Question 2:
>
> Does the model predict the correct assignee often enough?
>
> Question 3:
>
> Does the model predict the correct assignee often enough even if the distribution of bug reports assigned to developers is imbalanced?
>
> Question 4:
>
> Is the model general enough to work with all projects (a project means a dataset of bug reports)?

Question 5:

Is there a difference between open source and proprietary data?

Question 6:

Does the size of a window (a time period from which the data is collected) affect performance of the model?

Question 7:

Does the number of predicted assignees affect performance?

3.2.3 Metrics

The last step is to define metrics used to answer questions above. We use five metrics:

Accuracy:

The amount of correctly predicted assignees over all tested predictions, formally:

$$Accuracy = \frac{tp + tn}{tp + tn + fp + fn}$$

Macro-Averaged Precision:

The amount of assignees correctly predicted as positive for given test sample over number of assignees either correctly or incorrectly predicted as positive, averaged over all classes[11], formally:

$$Precision_{macro} = \frac{1}{q} \sum_{\lambda=1}^{q} \frac{tp_\lambda}{tp_\lambda + fp_\lambda}$$

Macro-Averaged Recall:

The amount of assignees correctly predicted as positive for given test sample over number of assignees correctly predicted as positive or incorrectly predicted as negative, averaged over all classes[11], formally:

$$Recall_{macro} = \frac{1}{q} \sum_{\lambda=1}^{q} \frac{tp_\lambda}{tp_\lambda + fn_\lambda}$$

Chi-Squared Test:

Test to assert whether two samples are from the same distribution.

T-Test:

Test to assert whether two samples have the same population mean.

4 Evaluation

In this chapter, we present the results for various models on three datasets. First, we provide a description and origin of these datasets. Then we compare the performance of the models and the datasets. This includes comparison of open source data downloaded from the Internet with the proprietary data provided by the private company. In the third section, we compare the performance for different number of recommendations. (3)

4.1 Datasets

To evaluate performance of a model, a set of necessary data is necessary. We provide description of the three datasets that we chose for the evaluation of the models in this section.

4.1.1 Firefox Data

This dataset is downloaded from Mozilla repository[1] in product Firefox. We downloaded all bugs that are in status `RESOLVED` with resolution `FIXED` and were created in year 2010 or later. We also removed bugs with field `assigned_to` set to `nobody@mozilla.org` as those tickets were resolved and fixed but not assigned. This yields $9,141$ bugs in total. To get a better comparison with the other datasets, we only use $3,000$ datasets for training and cross-validation that were created between 2010-01-01 and 2012-07-10. This dataset contains 343 labels (developers). Finally, we remove developers who did not fix at least 30 bugs, yielding $1,810$ bugs with 20 developers. The histogram with frequencies of the developers and cumulative distribution is on figure 4.1.

4.1.2 Netbeans Data

Netbeans data were downloaded from Netbeans bug repository[2]. We considered latest $3,000$ bugs that are in status `RESOLVED` with resolution `FIXED`. We removed bugs with assignee `kenai_tester_git` as those

1. https://bugzilla.mozilla.org
2. https://netbeans.org/bugzilla/

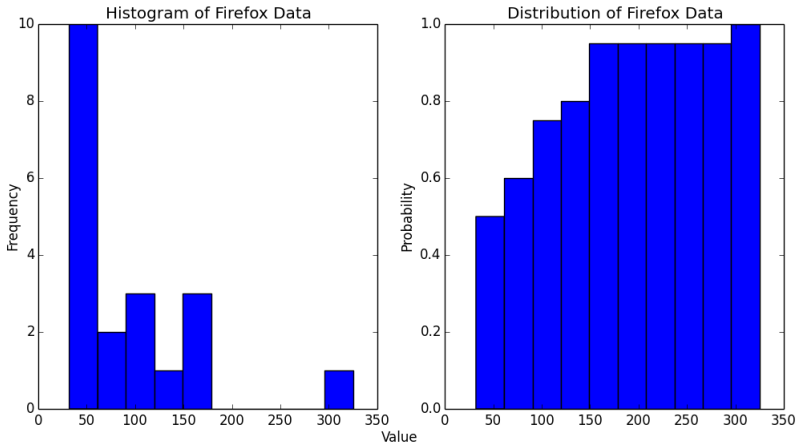

Figure 4.1: Histogram and distribution of the Firefox data.

were created automatically resulting in 2,924 bug reports. These bugs were created between 2014-06-14 and 2015-06-14 and they contain 92 developers. After removing developers who did not fix at least 30 bugs, the dataset contains 2,528 reports with 30 developers. Figure 4.2 is a histogram and distribution of this datasets.

4.1.3 Proprietary Data

The Proprietary dataset was provided by a company that wants to remain anonymous. The provided dataset contains 2,243 bug reports created between 2009-03-07 and 2013-08-23. Only bug reports that were resolved with assigned developer were considered. There are 141 developers in this dataset. Only 1,479 bugs assigned to 28 developers were retained after removal of developers with less than 30 fixed bugs. The histogram and probability distribution is pictured on figure 4.3.

4.1.4 Chi-Squared Test

In this section, we test the null hypothesis that two samples are from the same distribution. For this, we use the two-sided alternative of two-sample Chi-Squared test. We split the samples from each datasets into five bins and compute the *p-value* for three pairs of samples (Netbeans and Firefox, Firefox and Proprietary, Proprietary and Netbeans).

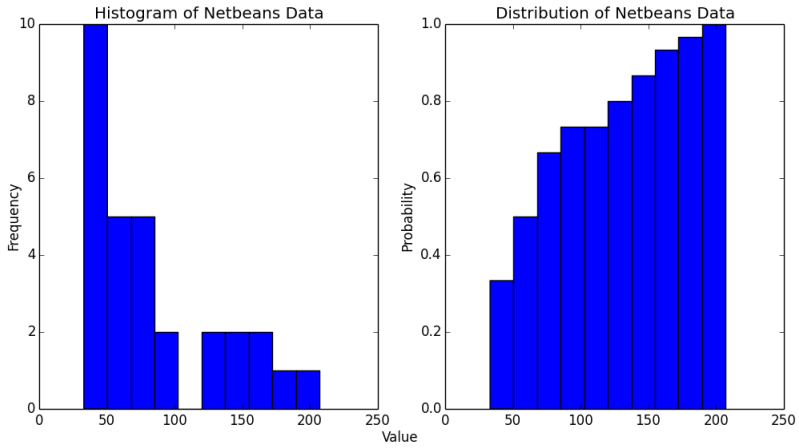

Figure 4.2: Histogram and distribution of the Netbeans data.

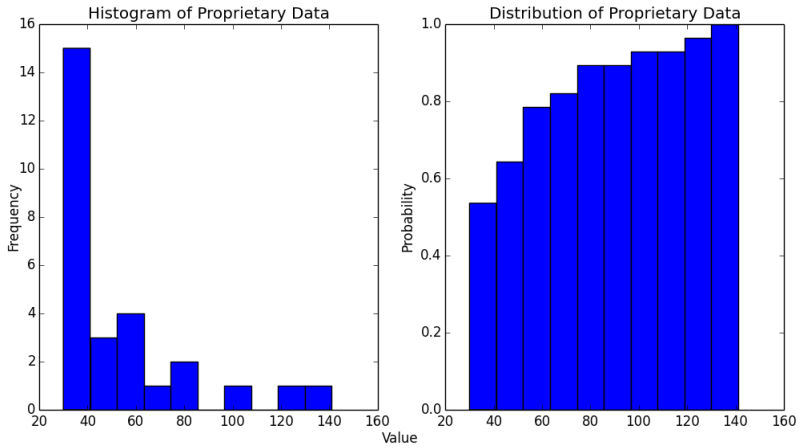

Figure 4.3: Histogram and distribution of the Proprietary data.

The Chi-Squared test with Netbeans and Firefox samples yields p-value of 0.4283, as p-value > 0.05 we fail to reject the null hypothesis that the two samples are from the same distribution with 5% level of significance.

The test with Firefox and Proprietary samples returns p-value equal to 0.8169 so we again fail to reject the null hypothesis with 5% level of significance. The last test with samples from Proprietary and Netbeans data results in p-value of 0.6735, which again means the null hypothesis cannot be rejected with 5% significance level.

4.1.5 T-Test

In the previous section, we tested the hypothesis that the datasets used in this thesis are from the same distribution. We failed to reject the hypothesis with all samples, in this section, we test the null hypothesis that the samples have the same population mean. For that, we have to test the hypothesis that the variance of the samples are the same.

We use the two-sided alternative of the standard independent two-sample t-test to test the null hypothesis that the samples have the same population mean, and the two-sided alternative of the Levene test to test the null hypothesis that the samples have equal population variance. If we reject the null hypothesis of the Levene test, we use the two-sided alternative of the Welch's t-test instead of the standard t-test.

First, we test the samples from Firefox and Proprietary datasets. The Levene test yields p-value equal to 0.0373, as p-value < 0.05, we reject the null hypothesis with 5% significance level. To test the population mean of the two samples, we therefore have to use the Welch's t-test. The Welch's t-test results in p-value of 0.0343, which means we have to reject the null hypothesis of equal population mean of the two samples with 5% significance.

Second, we test the samples from Firefox and Netbeans repositories. As the Levene tests returns p-value equal to 0.5166, which is greater than the α of 0.05, we fail to reject the null hypothesis of equal population variances with 5% level of significance. Next, we use the standard independent t-test to test the population means. The p-value of this t-test on these two samples is equal to 0.7138, which means we fail to

reject the null hypothesis of equal population means with 5% significance.

Last samples we test are from Netbeans and Proprietary datasets. The p-value of the Levene test for these two samples is 0.0216 thus we reject the null hypothesis of equal variances. The p-value of the Welch's t-test is equal to 0.0039, which again means we can reject the null hypothesis that the two samples have the same population mean with 5% significance.

4.2 Window Size

The size of the time window is an important question that needs to be addressed when dealing with machine learning. A naive approach uses all samples from a dataset to train the classifier, but a more advanced approach considers what effect does the size of the window have on the data and how it can improve the performance. In this section, we try to address this concern by evaluating the performance of the classifier for different sizes of the window.

To determine weather a different size of the window helps the performance is a difficult endeavour, we attempt it by employing three different approaches, each approach can be described by two properties like this:

1. Fixed number of bugs in each period, variable size of the train set
2. Variable number of bugs in each period, fixed size of the train set
3. Variable number of bugs in each period, variable size of the train set

The Firefox dataset is used for testing of all approaches. The model that is used is Support Vector Machine with Stop-Words removed and TF-IDF weighing.

4.2.1 First Approach

In the first approach, we first remove all bugs from the whole dataset that were fixed by developers with less than 20 fixed bugs. Then we split

the remaining dataset into 8 bins. First bin contains 1000 randomly selected bug reports from period 1. All subsequent bins (2-8) contain 500 randomly selected reports from periods 2-8. There are 8 periods, first period is about one year long, all the other periods are about half a year long each. All periods combined add up to the time span of the whole dataset and each $n+1$ period contains bugs older than n period. We train the classifier on period 1, then 1-2, 1-3, 1-4, 1-5, 1-6, 1-7 and 1-8. We test each trained classifier on the same cross-validation set of 300 bug reports that are newer than bugs in period 1.

The disadvantage of this approach is that you select fixed number of bugs from each period. In real world, each period can contain different number of bug reports based e.g. on season. Another disadvantage is that you remove developers that do not meet the criteria of 20 bugs fixed based on the whole dataset. If some developers were very active in the past but are no longer very relevant, they will not be removed from a dataset that is constructed from periods in which the developer no longer fulfills the criteria.

The result is visualized on figure 4.4. You can see that about the best result is achieved with the size of the window equal to 30 months. Unfortunately, the performance of the classifier for other sizes is very similar and the results is therefore quite inconclusive.

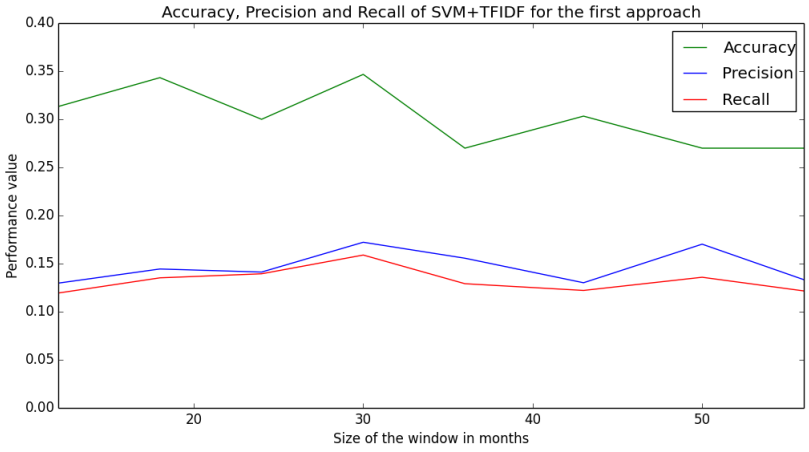

Figure 4.4: Performance of the classifier for different size of the window, 1st approach

4.2.2 Second Approach

In the second approach, we employ the same periods, except period 1 that is about 15 months long. Another difference is that each train set contains 3000 randomly selected bug reports from period 1, 1-2, 1-3 etc. and the size of the sample is in this case therefore fixed, what is different is the period from which they were selected. From each bin, all bug reports that were fixed by a developer with less than 30 fixed bugs within the same train set are removed. We also remove all bug reports from cross-validation set that are not in the train set. We do this to get more relevant results from macro-averaged metrics, because otherwise the result of such metrics would be very close to zero.

The disadvantage of the second approach is that it does not really tell us what size of the window can be used to get the best performance. What it shows is whether the size of the window from which an equal size of sample is selected matters.

Figure 4.5 shows the plot of this approach. The performance decreases as the size of the window increases, which is expected. Precision and recall however increases when the size of the window is around 45 months. (4)

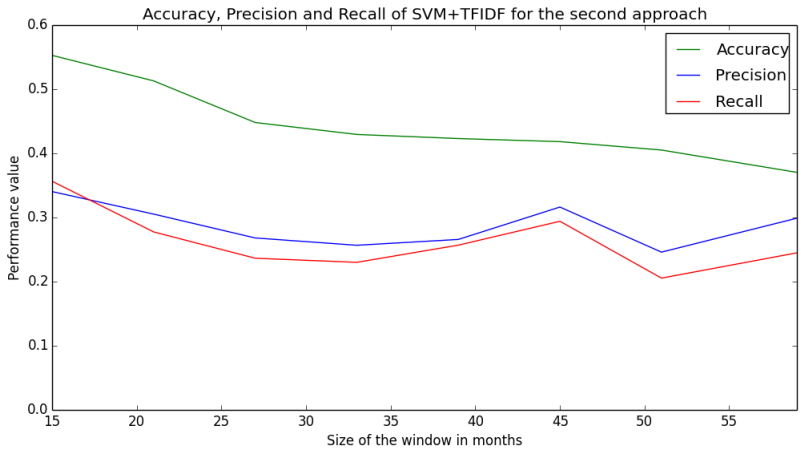

Figure 4.5: Performance of the classifier for different size of the window, 2nd approach

4.2.3 Third Approach

Finally, in the third approach, we again use the same periods as in the second approach. The difference is that we select all bug reports as train set.

The advantage is that this approach is closest to reality (no random selection). The disadvantage is, however, that the results are skewed by a variable number of samples in each period. The last major disadvantage is that the number of classes (developers) increases significantly each time samples from the next period are added. Figure 4.6 shows the plot of this approach.

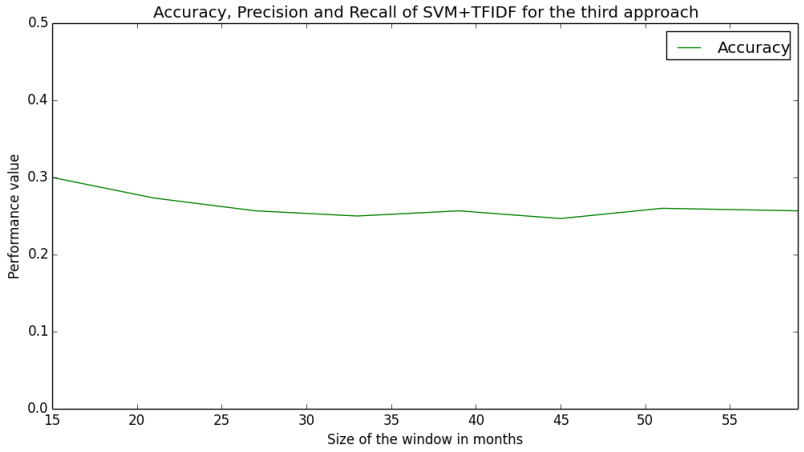

Figure 4.6: Performance of the classifier for different size of the window, 3rd approach

4.3 Time-Window Analysis

In this section, we analyze the distribution of topics with respect to time for Proprietary and Firefox datasets. We model 30 topics using Latent Dirichlet Allocation in our approach.

4.3.1 Firefox Data

All our Firefox bug reports were created within 62 months. You can see the distribution on figure 4.7.

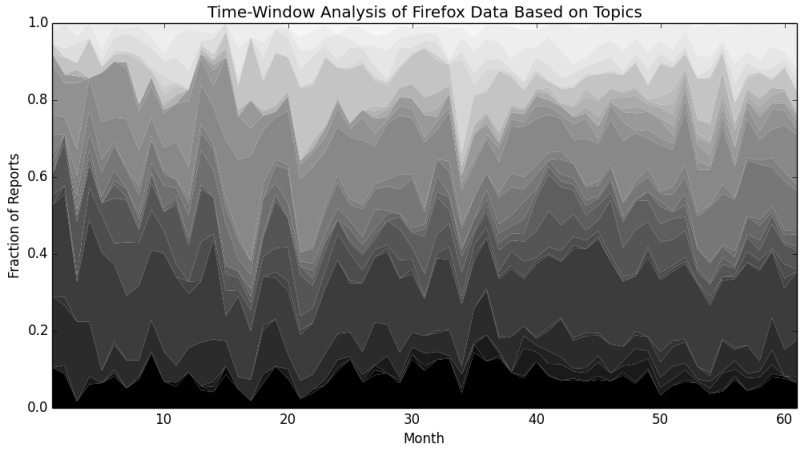

Figure 4.7: Time-Window Analysis of Firefox Data by Topics

4.3.2 Proprietary Data

The proprietary bug reports were created within 48 months. You can see the distribution on figure 4.8.

4.4 Baseline

In this section, we establish a baseline for our models. Our baseline is very simple, the number of bug report that is assigned to each developer is counted and the developer with highest number of reports is selected as a prediction for each subsequent call of the `predict` function.

Figure 4.9 shows the performance of the baseline model on Firefox data. While the accuracy of this model is relatively high (18%), the precision and recall values are much lower (1% and 5%), which can be explained by the way macro-averaged metrics work, see TODO [(5)].

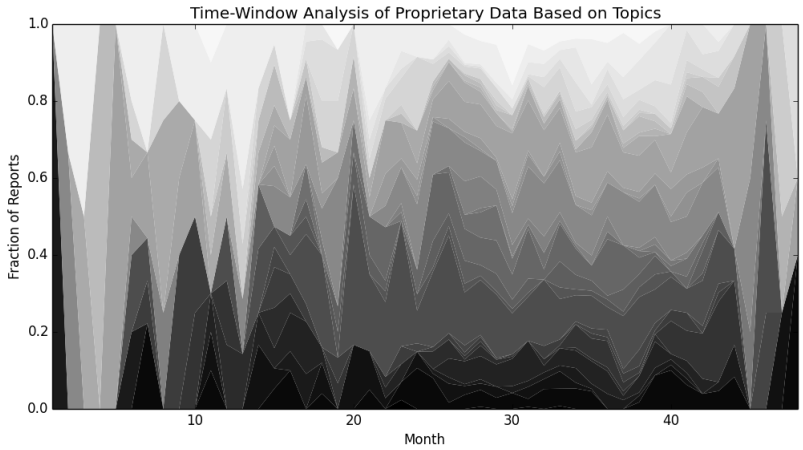

Figure 4.8: Time-Window Analysis of Proprietary Data by Topics

4.5 Stop-Words Removal

Stop-words removal is a technique that can increases performance of the model. In this section, we determine whether the increase is significant enough to use it for subsequent comparisons of models.

Figure 4.9 shows the increase in performance after all stop-words were removed from the feature vector of the Firefox data. You can see that the performance of the classifier slightly increased on all models. Accuracy increased by 3%, 0% and 1% on SVM, Naive Bayes and CART model respectively. Precision value of the SVM model decreased by 1% but increased by 6% with Naive Bayes and by 1% with CART. Finally, Recall values of SVM and Naive Bayes increased by 4% and 2% respectively while it slightly decreased on the CART model by about 1%.

The results of the other datasets were similar in nature. We therefore conclude that the performance boost of stop-words removal is significant enough to warrant better results, which matches the conclusion of Čubranić and Murphy [1] (see chapter 2). Thus, in our evaluation, we always use stop-words removal in combination with other feature extraction/selection methods.

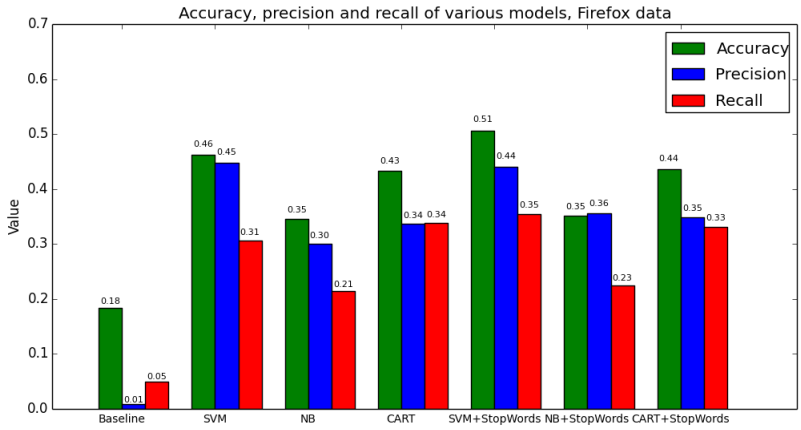

Figure 4.9: Stop-Words removal on Firefox data.

4.6 Comparison of Models

In the following text, comparison of the models is presented. We show the accuracy, precision and recall of these models on three datasets. (6) (7) This comparison allows us to conclude what model is best for this application.

4.6.1 Firefox Data

On Figure 4.10, you can see the performance of the chosen models on Firefox data. The SVM model with TF-IDF weighing achieves the best performance with accuracy of 57%. It's precision and recall also outperforms all the other approaches with values of 51% and 45%. The same model with LSI takes second place. The only model other than SVM that approaches SVM with it's performance is CART, especially with TF-IDF weighing.

4.6.2 Netbeans Data

The performance of the models on Netbeans data is shown on Figure 4.11. It is clear that the SVM model with TF-IDF weighing performs best even on the Netbeans data, although in this case the LSI feature extraction technique does not seem to perform as well as in the

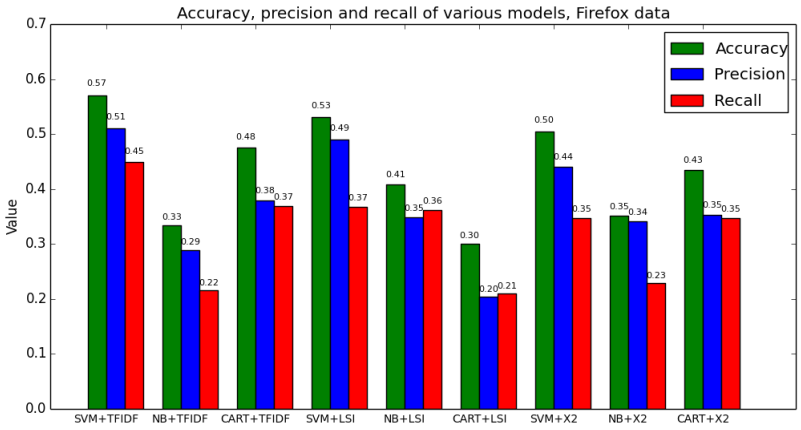

Figure 4.10: Comparison of models on Firefox data.

previous case. The accuracy, precision and recall of the approach are 53%, 53% and 49% respectively. Sole SVM model and SVM+χ^2 model perform similarly as far as precision is concerned, while the accuracy and recall values are lagging behind by a considerable margin. None of the other models offer better performance than even sole SVM model on this data, which suggests that SVM is a very good choice in this domain.

4.6.3 Proprietary Data

In this case, the performance of all models decreased a lot in comparison with the other datasets. However, even in this case, the SVM model with TF-IDF offers the best performance of 37% accuracy, 32% precision and 31% recall. The same model with LSI also shows quite a good performance and quite surprisingly, the Naive Bayes model with LSI performs quite well as far as precision and recall is concerned. Figure 4.12 presents the results in graphical manner.

4.6.4 Conclusion

The comparison shows that SVM with TF-IDF weighing performs best on all datasets. Not only that, it also generalizes very well, because there was no need to readjust the parameters of the model to get the best or

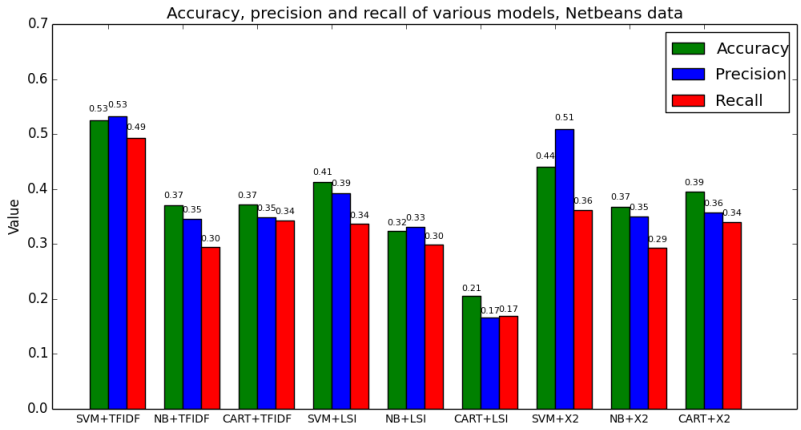

Figure 4.11: Comparison of models on Netbeans data.

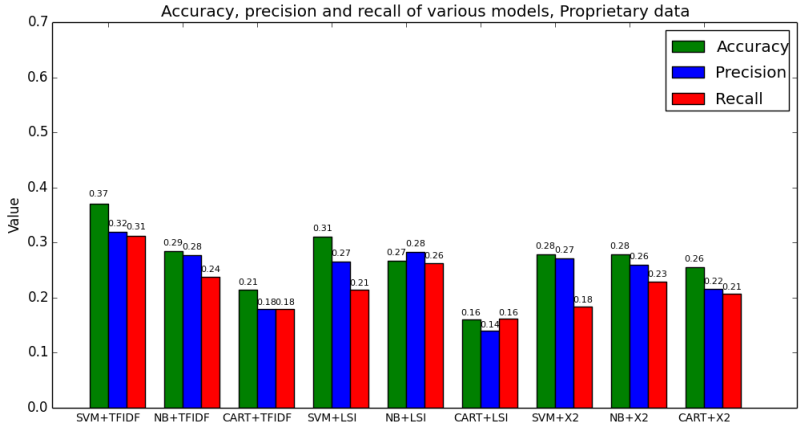

Figure 4.12: Comparison of models on the Proprietary data.

nearly the best performance for all datasets. The disadvantage of the model is that it is the most computationally complex one, because SVM is the slowest of the three models, there are a lot of classes and there are a lot of features. This can be at least partially dealt with by using χ^2 feature extraction in conjunction with TF-IDF while sacrificing some of the performance.

4.7 Comparison of Datasets

The main purpose of this section is to compare proprietary dataset with open-source dataset. The open-source dataset used for this is from the Firefox project. We selected equal number of reports from both datasets to construct an accurate comparison. (8) (9)

4.7.1 Naive Bayes Model

The first model for comparison is Naive Bayes. The only used feature extraction method for this model is stop-words removal.

First plot (Figure 4.13) represents accuracy of the Naive Bayes model. You can already see that the open source data perform better than the proprietary data. Accuracy of the classifier is 35% vs 24% when x (see above) is set to 30. In the following text, we will always compare the results of this setting of parameter x as we believe that is the best production value for the given size of the dataset.

Second plot (Figure 4.14) is a representation of precision and recall of the same model. Precision of the classifier is 46% and 22% for the open source and proprietary data, respectively. Recall value is 29% for the open source data and 19% for the proprietary data.

4.7.2 Support Vector Machine Model

The second model used for comparison of the proprietary dataset with the open-source dataset is Support Vector Machine model with TF-IDF weighing and stop-words removal as this model shows the most promising results.

Figure 4.15 visualizes accuracy of the model. Even with this model, the open source data perform much better with accuracy value of 54% vs 33% of the proprietary data.

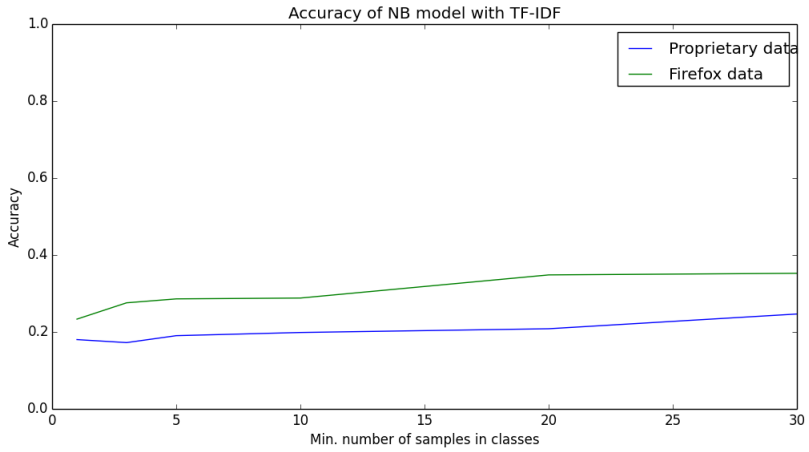

Figure 4.13: Comparison of accuracy of Naive Bayes model.

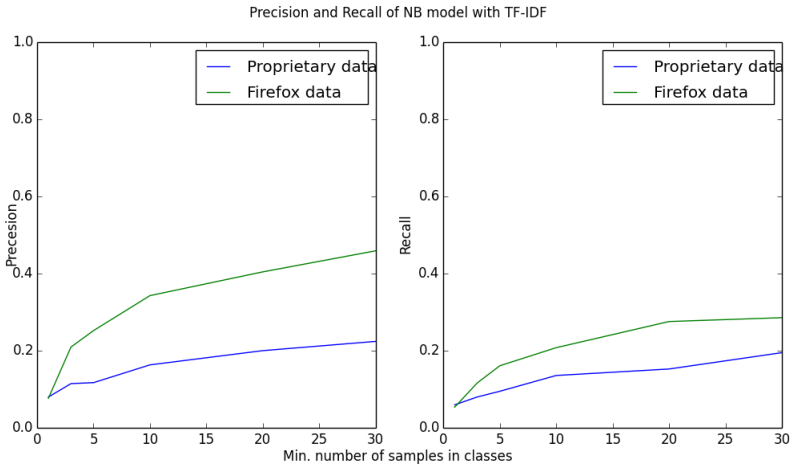

Figure 4.14: Comparison of precision and recall of Naive Bayes model.

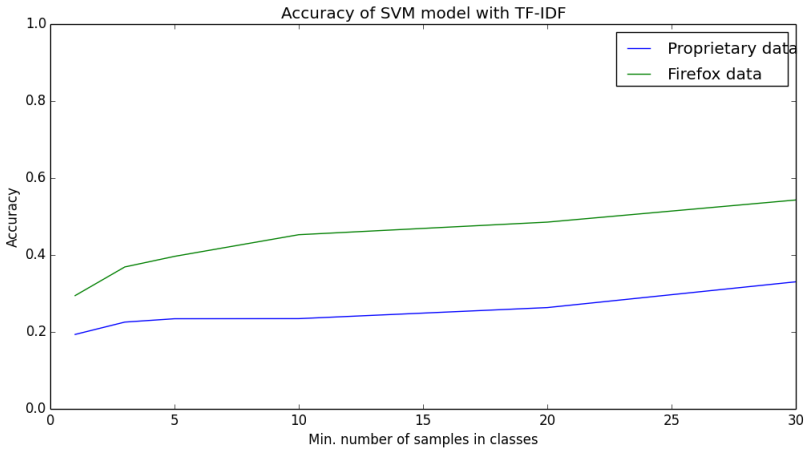

Figure 4.15: Comparison of accuracy of SVM model.

Precision and recall of the SVM model is pictured on figure 4.16. It again clearly shows that the precision value of the open source data is higher (again about 54 %) than the precision value of the proprietary data (32%). The recall value is 49% for the open-source data and 26% for the proprietary data.

4.7.3 Conclusion

The results clearly show that the open-source data provide much better performance. The number of classes is very similar and the number of reports used for training too, so it is clear that the quality of the open-source reports is higher. (10)

4.8 Comparison of Performance for Variable Number of Recommendations

We look at the performance of the models for different number of recommendations in this section. The plots will show the performance for number of recommendations from 1 to 10. (11)

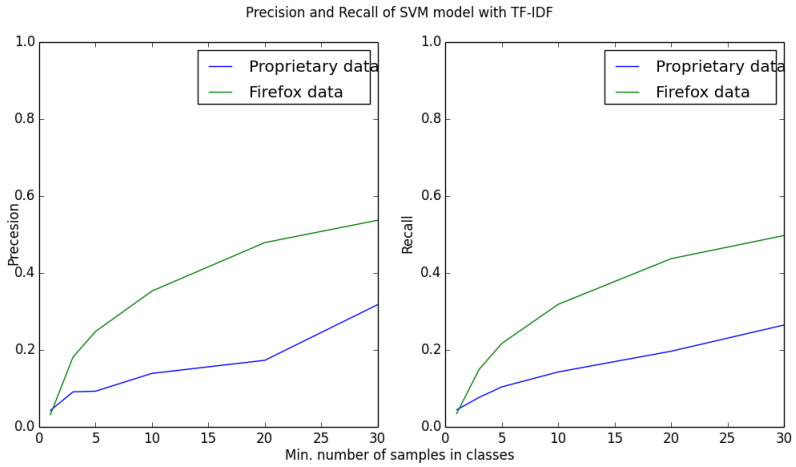

Figure 4.16: Comparison of precision of SVM model.

4.8.1 Support Vector Machine Model

For this comparison, we, again, take the SVM model with TF-IDF weighing and stop-words removed. Figure 4.17 shows how the accuracy increases when the number of recommendations increases, which is expected as the more there are recommendation, the higher the chance of a hit is. (12)

It is apparent from the plot that the highest performance boost happens when the number of recommendations changes from one to two both for the proprietary data and the Firefox data.

4.8.2 Conclusion

From the results in this section, we concluded that the best number of recommendations to suggest to users is 5. The performance does not increase very much with more recommendations and it would probably only lead to confusion if the number of recommendations was too high.

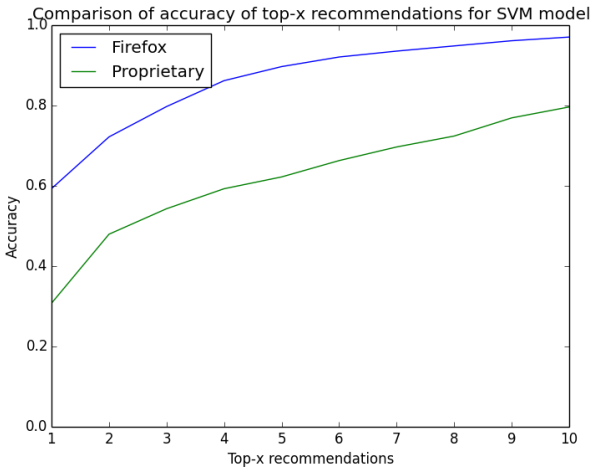

Figure 4.17: Comparison of accuracy of top-x recommendations for SVM model.

5 Conclusion

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Bibliography

[1] G. Murphy, "Automatic bug triage using text categorization," *Citeseer*, 2004. [Online]. Available: http://scholar.google.cz/scholar?hl=en&q=automatic+bug+triage+using+text+categorization&btnG=&as_sdt=1,5&as_sdtp=#1

[2] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" *Proceeding of the 28th international conference on Software engineering - ICSE '06*, p. 361, 2006. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1134285.1134336

[3] S. N. Ahsan, J. Ferzund, and F. Wotawa, "Automatic Software Bug Triage System (BTS) Based on Latent Semantic Indexing and Support Vector Machine," *2009 Fourth International Conference on Software Engineering Advances*, pp. 216–221, Sep. 2009. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5298419

[4] J. Xuan, H. Jiang, Z. Ren, J. Yan, and Z. Luo, "Automatic Bug Triage using Semi-Supervised Text Classification." *SEKE*, 2010. [Online]. Available: http://researchers.lille.inria.fr/~xuan/page/paper/seke_10.pdf

[5] R. Shokripour, "Automatic Bug Assignment Using Information Extraction Methods," 2011.

[6] M. Alenezi, K. Magel, and S. Banitaan, "Efficient Bug Triaging Using Text Mining," *Journal of Software*, vol. 8, no. 9, pp. 2185–2190, Sep. 2013. [Online]. Available: http://ojs.academypublisher.com/index.php/jsw/article/view/10840

[7] K. Somasundaram and G. C. Murphy, "Automatic categorization of bug reports using latent Dirichlet allocation," *Proceedings of the 5th India Software Engineering Conference on - ISEC '12*, pp. 125–130, 2012. [Online]. Available: http://www.scopus.com/inward/record.url?eid=2-s2.0-84858300807&partnerID=40&md5=07612d59d9b681f002dd3b436ad964bb$\delimiter"026E30F$nhttp://dl.acm.org/citation.cfm?doid=2134254.2134276

[8] J.-w. Park, M.-w. Lee, J. Kim, S.-w. Hwang, and S. Kim, "C OS T RIAGE : A Cost-Aware Triage Algorithm for Bug Reporting Systems," *Artificial Intelligence*, pp. 139–144, 2011.

[9] F. Thung, X.-b. D. Le, and D. Lo, "Active Semi-Supervised Defect Categorization," 2015.

[10] X. Xia, D. Lo, X. Wang, and B. Zhou, "Dual analysis for recommending developers to resolve bugs," *Journal of Software-Evolution and Process*, no. July 2010, pp. 481–491, 2015. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/smr.504/full

[11] V. V. Asch, "Macro- and micro-averaged evaluation measures [[BASIC DRAFT]]," pp. 1–22, 2013.

A Appendix One

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

To do...

- ☐ 1 (p. 2): Describe other chapters
- ☐ 2 (p. 7): citation needed
- ☐ 3 (p. 11): What else? There should probably be more.
- ☐ 4 (p. 17): Try to figure out why
- ☐ 5 (p. 19): Replace with reference to the chapter/section which describes the metrics
- ☐ 6 (p. 21): What models? Explain or provide reference.
- ☐ 7 (p. 21): Explanation of these measurement metrics should be somewhere and perhaps a reference would be in order.
- ☐ 8 (p. 24): Maybe it would be useful to compare with the Netbeans data too.
- ☐ 9 (p. 24): And also it could be quite interesting to use the CART model too.
- ☐ 10 (p. 26): Or maybe there is something else? Try to figure out if that is really the case, if it is, it might be better to provide a better explanation.
- ☐ 11 (p. 26): More than one model might be in order.
- ☐ 12 (p. 27): Precision and recall would be useful too maybe.