A Comparative Study of Text Classification Algorithms on User Submitted Bug Reports

Saniat Javid Sohrawardi, Iftekhar Azam, Shazzad Hosain
Department of Electrical and Computer Engineering
North South University, Dhaka - 1229, Bangladesh
saniat.s@northsouth.edu, iftekhar.azam@northsouth.edu, shazzad@northsouth.edu

***Abstract*—In software industries, various open source projects utilize the services of Bug Tracking Systems that let users submit software issues or bugs and allow developers to respond to and fix them. The users label the reports as bugs or any other relevant class. This classification helps to decide which team or personnel would be responsible for dealing with an issue. A major problem here is that users tend to wrongly classify the issues, because of which a middleman called a bug triager is required to resolve any misclassifications. This ensures no time is wasted at the developer end. This approach is very time consuming and therefore it has been of great interest to automate the classification process, not only to speed things up, but to lower the amount of errors as well.**

In the literature, several approaches including machine learning techniques have been proposed to automate text classification. However, there has not been an extensive comparison on the performance of different natural language classifiers in this field. In this paper we compare general natural language data classifying techniques using five different machine learning algorithms: Naive Bayes, kNN, Pegasos, Rocchio and Perceptron. The performance comparison of these algorithms was done on the basis of their apparent error rates. The data-set involved four different projects, Httpclient, Jackrabbit, Lucene and Tomcat5, that used two different Bug Tracking Systems - Bugzilla and Jira. An experimental comparison of pre-processing techniques was also performed.

I. Introduction

Bug Tracking Systems (BTS) generally allow the users to submit issue reports, let them communicate on the bug report page by posting comments, as well as let the developers collaborate while resolving the issues. Clearly, BTS helps large projects keep track of their numerous software bugs and keep their public face by having developers and engineers respond to the reporters. However, large and successful projects frequently tend to face enormous amounts of issue reports. Hence unavoidable inefficiencies occur due to the time required to sort the issues as well as human errors. For example, a GNOME project using Bugzilla BTS has received 46,352 issue reports in one week[1]. That means approximately 6,000 issue reports were required to be addressed every day, but only 430 were opened in the whole week. Not only that, recent research suggests that about 33.8% of issue reports in the BTS are misclassified [1] that causes time loss due to developers requiring to start working on the issue and then realizing that it has to be passed on to a different department. This issue is typically addressed by utilizing a middleman known as a bug triager, who has to manually go through each report and assign it to the appropriate department. This task can be extremely time consuming and can still lead to quite a lot of human errors. Thus it is of great interest to use efficient machine learning techniques in order to either aid or completely eliminate the requirement of this middleman. The goal is to decrease the amount of human errors as well as to decrease the amount of time spent on manual classification.

Our practical research is based on issue reports from two prevalent Bug Tracking Systems, namely Bugzilla[2] and Jira[3]. These systems behave like any traditional BTS, requiring users to register and sign in to submit bug reports. The input fields slightly vary from one BTS to another, but all of those have the description and the comments that are of interest to us for the classification process.

The test data has been collected from previous research published by Herzig, Just and Zeller [1], where they manually classified a large set of bug reports from five different projects. In their paper, they provided a link to their classification CSV files[4]. Their research suggested that the impact of misclassified issue reports could have adverse effects on bug prediction systems, resulting in about 39% of the files that never even had a bug being marked defective. This serves as an additional incentive for drastic improvement in the quality of issue classification.

We defined our problem to be the classification of text data files into either of the two major classes, ”bug” or ”not bug”. For this purpose we used five different algorithms with several preprocessing techniques. Our goal was not only to achieve automatic classification, but to get some insight into how the performance of these algorithms compare with each other, given our dataset. Unlike other papers, we focused on comparing a large number of algorithms from different fields of data mining for text classification of bug reports. Additionally, we did some extra comparison of how each type of stemming affects the accuracy of the algorithms.

For our comparison, we considered the following five prominent machine learning algorithms, all of which use bag-of-words data representation.

1) Naive Bayes.
2) K-nearest neighbor
3) Pegasos from Support Vector Machines

[1]https://bugzilla.gnome.org/page.cgi?id=weekly-bug-summary.html the statistic mentioned in this paper was extracted on 12th May 2014

[2]http://www.bugzilla.org/

[3]https://www.atlassian.com/software/jira

[4]http://www.softevo.org/bugclassify

978-1-4799-5421-6/14/$31.00 © 2014 IEEE

TABLE I. SOURCES OF DATASET (NORMALIZED)

PROJECT	BUG TRACKER	SAMPLE COUNT
Httpclient	Jira	554
Jackrabbit	Jira	2380
Lucene	Jira	2047
Tomcat5	Bugzilla	298

4) Rocchio classifier / Nearest Centroid classifier
5) Perceptron classifier from Neural Networks

The first two algorithms, Naive Bayes and K-nearest neighbor algorithms were chosen as they are the most commonly used algorithms for text classification and we felt that they should be present as a base for comparison. Both are well suited for our discrete dataset and are quite simple to understand. The latter three were chosen from different fields based on recommendations from several papers. In a paper by Lee et.al. [2] it was mentioned that Support Vector Machines (SVM) are considered to be the one of the best text classifiers and showed that Neural Networks (NN) are quite competitive in performance. As a matter of fact, they mentioned that NN algorithms outperform SVM when the dataset is relatively small. We chose Pegasos [3] from SVM because of its simplicity and efficiency. From NN, we picked the Perceptron algorithm as a popular representative. Rocchio is another popular algorithm for text classification and is mentioned by Guo et al. [2] as well as Sebastiani [4]. We chose it as a variation to kNN as they are quite similar and yet each has its own drawbacks and advantages.

Rochhio was chosen as it is conceptually simpler than K-nearest neighbor and quite efficient at training and testing. It is also a renowned algorithm for text classification as mentioned by Guo et.al [2] who also mentions the popularity of the above mentioned fields.

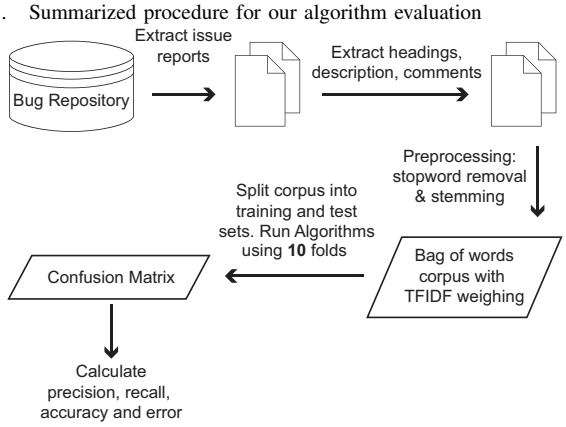
Our research has showed that Perceptron, a Neural Networks classifier, outperformed the rest of the classifiers on our dataset.

The paper starts off with the description of the dataset and the extraction process in Section II, followed by the evaluation procedures in Section III where we talked about the pre-processing techniques we used for the formation of the final dataset. Section IV contains the discussion of the results of our analysis. The threats to validity of our research and the related works are in the Sections V and VI respectively. We finally conclude our paper with Section VII where we also talk about possible extensions to our work.

II. DATA SET

We extracted a large number of issue reports from five different projects that used either one of the two ubiquitous Bug Tracking Systems, Bugzilla or Jira. Then we normalized the initial dataset to get equal numbers of BUG and NON-BUG reports to prevent biased learning. The name of the projects, the BTS, and the issue report counts are shown in Table I. These specific projects were chosen because the issue reports were already manually classified in a previous research [5] and the authors were kind enough to publish the dataset on their web page. For each project, they provided separate CSV files that contained issue report identification numbers together with their correct classifications.

Fig. 1. Summarized procedure for our algorithm evaluation



The issue reports themselves, however, had to be extracted manually from each of the corresponding BTS web sites. Since we only required the basic textual data, we decided to extract them from the individual issue report web pages. This task was made less cumbersome as the issue report page links were consistent with the IDs provided in the CSV files. First we automatically generated simple HTML pages that contained all of the required issue report links. Then we used a Mozilla Firefox plugin called, "Down Them All"[5] to download all the issue report pages. Thus we ended up with all the web pages, proper and reliable classifications that were ready for extraction of the required text.

III. EVALUATION PROCEDURE

The first step was to extract the raw text to be used for classification. This was followed by the pre-processing of the text and the generation of a proper dataset for the classification algorithms. The classification algorithms were cross-validated using the widely used 10 folds on a normalized, discrete dataset. At the end, the performance of each algorithm was recorded and tabulated. The entire process is summarized in Figure 1.

A. Extraction

This step involved coding a simple Java program to grab the headings, description and comments from the issue report pages and saving them to TXT files, one for each individual issue report. This was accomplished with the help of a Java library called JSOUP[6] meant for HTML file parsing.

B. Removal of Stop Words

Stop words had to be removed to eliminate the number of irrelevant features, thus decreasing the noise in the dataset. We used the publicly available list of SMART[7] stop-words for our purpose. Additionally we included a few extra characters to the SMART list for our specific purpose. The actual removal was quite trivial. It involved an intermediate step in our Java code, right after extraction, where the words from our Stop Word list were ignored and not passed on to the next step of the dataset generation.

[5] http://www.downthemall.net/
[6] http://jsoup.org/
[7] http://jmlr.org/papers/volume5/lewis04a/a11-smart-stop-list/english.stop

Fig. 2. Effects of Stemming

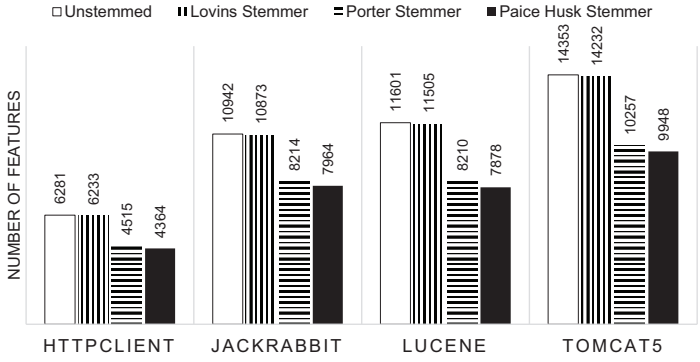
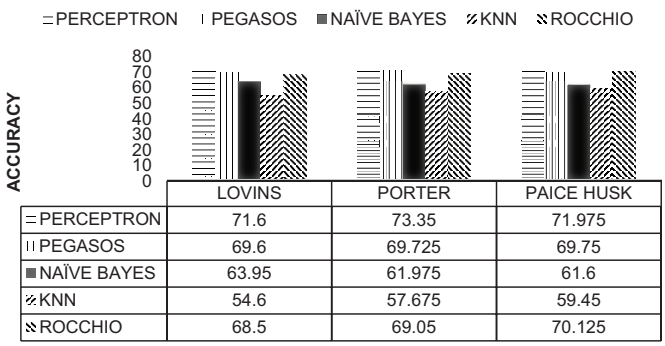


Fig. 3. Stemmer Performances



	LOVINS	PORTER	PAICE HUSK
PERCEPTRON	71.6	73.35	71.975
PEGASOS	69.6	69.725	69.75
NAÏVE BAYES	63.95	61.975	61.6
KNN	54.6	57.675	59.45
ROCCHIO	68.5	69.05	70.125

C. Stemming

We evaluated the following three stemming [6] algorithms by running them through each one of our projects to see which one results in the smallest number of features[8].

1) Lovins Stemmer
2) Porter Stemmer
3) Paice Husk Stemmer

Figure 2 shows the results of the tests. "Paice/Husk" Stemmer, which is known to be quite aggressive [7], managed to bring down the number of features to the lowest amount, making it an attractive choice for us. Figure 3 shows the comparison of performances of the algorithms after each type of stemming. The performance was evaluated by measuring the algorithm accuracy.

Before classification, the text data was loaded to form a "bag-of-words" dataset. The earliest mentions of the bag-of-words model can be found in Harris's article [8]. This model is represented by a table containing frequencies of words from a dictionary subset that is created by taking all the words that exist in the project. The list of words is generally orderless and disregards grammar rules. Thus the columns represent frequencies of words and rows represent the files that contain them, hence bag-of-words.

The frequencies of words were later weighed using TF-IDF (Term Frequency - Inverse Document Frequency) [9]. This measure helps in finding out the importance of each word and eventually setting its priority.

Fig. 4. Confusion matrix legend

	BUG	NOT BUG	
BUG	True Positive (TP)	False Negative (FN)	Sensitivity
NOT BUG	False Positive (FP)	True Negative(TN)	Specificity
	Precision	Negative Predictive Value	**Accuracy**

All the algorithms mentioned were run on the weighed bag-of-words dataset using a Java library called JSAT [9].

IV. Results & Discussion

All of the algorithms were run on the four separate projects as well as a "union" of all four projects. The "union" test set was created to see how a complete, project-independent generalization would affect the classification accuracy.

Figure 5 reports the confusion matrices for Httpclient and Jackrabbit projects, while Figure 6 contains the ones for Lucene and Tomcat5 projects. Figure 4 provides the format of the confusion matrix which is used in Figures 5 and 6. The tables also contain Recall and Precision values.

Precision specifically refers to the True Positive values while Negative Predictive Value is for the False Positive ones. However both can be generalized as Precision. Recall is the generalized term for Sensitivity and Specificity. They are calculated by:

$$Sensitivity = \frac{TP}{TP + FN}$$

$$Specificity = \frac{TN}{TN + FP}$$

$$Precision = \frac{TP}{TP + FP}$$

$$NegativePredictiveValue = \frac{TN}{TN + FN}$$

We observe the following from the confusion matrices:

- The Perceptron algorithm showed very stable and high readings for both precision and negative predictive values. This gives it top accuracy levels.
- Recall values of most projects were quite unstable and did not hold much pattern. However, the kNN algorithm had a near perfect 98.24% Sensitivity (Recall on BUG conditions) in the tomcat5 project. This suggests that the performance of a classification algorithm is highly specific to the dataset.

The performance of the algorithms in Figure 7 was based on the apparent error rate, which was calculated by,

[8]Here we are referring to words as features

[9]https://code.google.com/p/java-statistical-analysis-tool/

Fig. 5. Confusion matrices for Httpclient and Jackrabbit corpuses

	Httpclient				Jackrabbit			
Naïve Bayes		BUG	NOT BUG	Recall		BUG	NOT BUG	Recall
	BUG	206	71	74.4	BUG	933	257	78.4
	NOT BUG	125	152	54.9	NOT BUG	550	639	53.7
	Precision	62.2	68.2	**64.6**	Precision	62.9	71.3	**66.1**
kNN		BUG	NOT BUG	Recall		BUG	NOT BUG	Recall
	BUG	179	98	64.6	BUG	275	915	23.1
	NOT BUG	139	138	49.8	NOT BUG	80	1109	93.3
	Precision	56.3	58.4	**57.2**	Precision	77.5	51.5	**58.2**
Pegasos		BUG	NOT BUG	Recall		BUG	NOT BUG	Recall
	BUG	183	94	66.1	BUG	841	349	70.7
	NOT BUG	96	181	65.3	NOT BUG	313	876	73.7
	Precision	65.6	65.8	**65.7**	Precision	72.8	71.5	**72.2**
Rocchio		BUG	NOT BUG	Recall		BUG	NOT BUG	Recall
	BUG	151	126	54.5	BUG	728	462	61.2
	NOT BUG	44	233	84.1	NOT BUG	191	998	83.4
	Precision	77.4	64.9	**69.3**	Precision	79.2	68.4	**72.6**
Perceptron		BUG	NOT BUG	Recall		BUG	NOT BUG	Recall
	BUG	187	90	67.5	BUG	923	267	77.6
	NOT BUG	95	182	65.7	NOT BUG	271	918	77.2
	Precision	67.5	66.9	**66.6**	Precision	77.3	77.4	**77.3**

$$Apparent\ Error\ rate = \frac{Number\ of\ docs\ erred\ on}{Number\ of\ docs\ in\ the\ test\ set}$$

It is effectively the exact opposite of the accuracy calculated in Figures 5 and 6. The resultant error rates of all five algorithms on all four projects, the union set as well as the average error rate of all the algorithms are displayed in Fig. 7.

- On average, the Perceptron algorithm performed the best throughout almost all the projects. This suggests that our dataset must be linearly separable, since Perceprton works best on such datasets. K-nearest neighbor algorithm preformed the worst overall. Although, kNN did out perform all the rest in the tomcat5 project.
- The performance of all the algorithms in the union set was lower compared to other individual projects. This shows that it is practically less feasible to create a generalized bug classifier by combining the text data from all the issue reports and training a classifier.

V. Threats to validity

- In reality, the errors in classification have costs and risks associated with them. Some errors possess greater significance than others. However, we have not assigned either of these two in our calculations.
- We have only considered the apparent error rate for our comparison, however, we could have used the true error rate of a classifier to improve the accuracy. True error rate is the error rate of the classifier if it was tested on the true distribution of cases in the population [10].
- Our dataset could be regarded as quite limited, with 2,402 being the largest number of files in a project. Typically, a larger dataset ensures a more convincing training. However, we showed that even datasets as low as this provide quite efficient results, achieving an error rate as low as 14.6%.

VI. Related Works

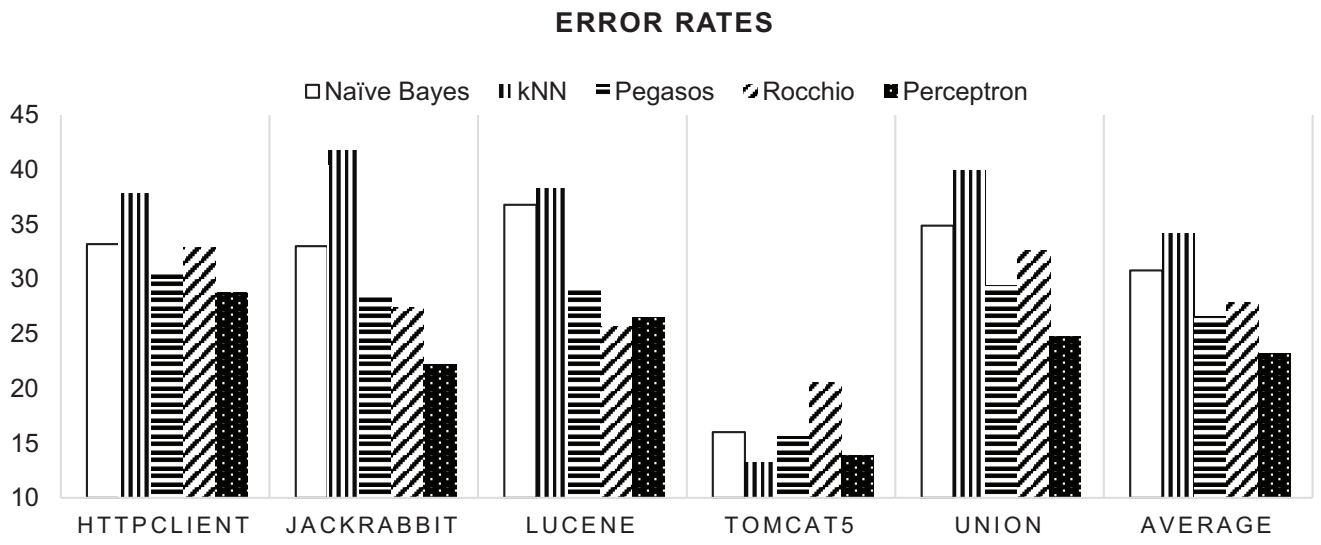
Herzig, Just and Zeller [1] wrote a wonderful paper justifying why automated issue report classification is important. While in the paper they started with manually classifying more than 7,000 issue reports stemming from five different projects, their paper was based on generating statistics on how much misclassification exists in the reports and how it could affect future prediction of defective or error prone files. Their findings suggested that every third issue report is misclassified as a Bug. This statistic motivated us to take up the initiative to explore bug classification using the text from issue reports.

Domatti et al. [11] presented an investigation of using text categorization to automatically assign bugs to developers based on the description of the bug submitted by the reporter. This

Fig. 6. Confusion matrices for Lucene and Tomcat5 corpuses

	Lucene				Tomcat5			
Naïve Bayes		BUG	NOT BUG	Recall		BUG	NOT BUG	Recall
	BUG	595	442	57.4	BUG	56	93	37.6
	NOT BUG	361	676	65.2	NOT BUG	43	106	71.1
	Precision	62.2	61.5	**61.3**	Precision	56.6	53.3	**54.4**
kNN		BUG	NOT BUG	Recall		BUG	NOT BUG	Recall
	BUG	338	699	32.6	BUG	91	58	61.1
	NOT BUG	160	877	84.6	NOT BUG	50	99	66.4
	Precision	67.9	55.6	**58.6**	Precision	64.5	63.1	**63.8**
Pegasos		BUG	NOT BUG	Recall		BUG	NOT BUG	Recall
	BUG	742	295	71.5	BUG	107	42	71.8
	NOT BUG	347	690	66.5	NOT BUG	41	108	72.5
	Precision	68.1	70	**69**	Precision	72.3	72	**72.1**
Rocchio		BUG	NOT BUG	Recall		BUG	NOT BUG	Recall
	BUG	695	342	67	BUG	114	35	76.5
	NOT BUG	193	844	81.4	NOT BUG	71	78	52.3
	Precision	78.3	71.2	**74.2**	Precision	61.6	69	**64.4**
Perceptron		BUG	NOT BUG	Recall		BUG	NOT BUG	Recall
	BUG	732	305	70.6	BUG	103	46	69.1
	NOT BUG	265	772	74.4	NOT BUG	39	110	73.8
	Precision	73.4	71.6	**72.5**	Precision	72.5	70.5	**71.5**

Fig. 7. Comparison of Naive Bayes, kNN, Pegasos, Rocchio and Perceptron based on error rates on our dataset



was one of the first papers of its kind and they focused on one open source project, Eclipse, that used the Bugzilla BTS. They used only the Naive Bayes classifier for their classification step to assign more than two classes to the data which brought out only about 30% accuracy.

Antoniol et al. [11] used Naive Bayes, Alternating Decision Trees and Logistic Regression to find out whether the text of issues posted on the BTS is enough to classify them into bugs or non-bugs. They randomly chose resolved issues from Mozilla, Eclipse and JBoss projects, which they manually classified and then used on the learning algorithms. Therefore similar to ours, this paper compared different algorithms, although they considered only three.

Another quite interesting paper that involved comparison of algorithms was published by Santana et al. [12] where they performed clustering based on the textual data from the bug-only issue documents. Their experimental dataset was quite small, because it came from a single private project, containing only 713 records in total. They performed their analysis using kNN, Naive Bayes and Decision Tree algorithms. Tests on their data showed that Naive Bayes and kNN performed similarly and better than Decision Tree algorithms.

All of the above papers have one thing in common - they all dealt with pre-labeled datasets and therefore utilized supervised training to train their algorithms. However, Naghwani and Verma [13] used a different information retrieval approach in their journal entry. They made the use of a Suffix Tree Clustering algorithms using Carrot2 framework in order to find patterns in the textual data of the issue reports and cluster them up. The labeling was later performed on the clusters that were formed. This technique, even though sounds quite unorthodox, it is quite similar to ours throughout the pre-processing stage.

Most of the papers discussed above involve the use of classification algorithms to analyze natural language that is extracted from the BTS of their individual projects. Almost all of them worked with the Bugzilla BTS system, thus restricting the variety in the test data. Most of these papers did not involve enough comparison of the performance between different kinds of algorithms, which we would have liked to see more of.

VII. Conclusion and Future Works

In this paper, we presented an extensive comparison of different algorithms to assist in automatic issue report classification. The issue reports for the training and test data were extracted from two different types of ubiquitous Bug Tracking Systems. These issue reports were earlier classified manually to their proper types in a paper by Herzig, Just and Zeller [1]. Unlike existing papers, our comparison included a much broader variety of algorithms. We also compared several stemming algorithms used to preprocess our textual data. The experimental results have shown that Paice/Husk Stemmer was more aggressive, achieving the lowest number of distinct features. We applied five different algorithms to classify the issue reports into bug and non-bug. The results showed Perceptron, a Neural Networks classifier, to be the best performing classifier for our type of data, achieving an average error rate of only 23.68%. Judging by the performance of natural language classifiers in other contexts, the average error rate of Perceptron seemed pretty low. Therefore for classification of our type of dataset, that is, user submited Bug Reports, Perceptron seems to be a good choice.

We chose not to classify the issues into finer grained categories such as RFE (Request For Enhancement), IMPR (Improvements), DOC (Documentation), REFAC (Refactor Source Code) and Other (as indicated in [1]), because our dataset was small and consisted of mainly bugs and non-bugs . The other fine grained categories were relatively small in numbers and were not giving optimal results. However, this sort of finer grained categorization can be achieved with a much larger dataset consisting of a greater variety of classifications.

References

[1] H. Kim, J. Sascha, and Z. Andreas, "It's not a bug, it's a feature: How misclassification impacts bug prediction," in *Proceedings of the 2013 International Conference on Software Engineering*, Piscataway, NJ, USA, 2013, pp. 392–401.

[2] W. Zaghloul, S. M. Lee, and S. Trimi, "Text classification: neural networks vs support vector machines," *Industrial Management & Data Systems*, vol. 109, no. 5, pp. 708–717, 2009.

[3] S.-S. Shai, S. Yoram, and S. Nathan, "Pegasos: Primal estimated sub-gradient solver for svm," in *Proceedings of the 24th International Conference on Machine Learning*, ser. ICML '07, 2007, pp. 807–814.

[4] F. Sebastiani, "Machine learning in automated text categorization," *ACM computing surveys (CSUR)*, vol. 34, no. 1, pp. 1–47, 2002.

[5] A. Giuliano, A. Kamel, D. P. Massimiliano, K. Foutse, and G. Yann-Gaël, "Is it a bug or an enhancement?: A text-based approach to classify change requests," in *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, Ontario, Canada, 2008, pp. 304–318.

[6] What is stemming. [Online]. Available: http://www.comp.lancs.ac.uk/computing/research/stemming/general/index.htm

[7] Paicehusk. [Online]. Available: http://www.comp.lancs.ac.uk/computing/research/stemming/Links/paice.htm

[8] Z. Harris, "Distributional structure," *Word*, vol. 10, no. 23, pp. 146–162, 1954.

[9] A. Rajaraman and J. D. Ullman, *Mining of massive datasets*. Cambridge University Press, 2011.

[10] J. R. Nolan, "Estimating the true performance of classification-based nlp technology," in *Workshop On From Research To Commercial Applications: Making NLP Work In Practice*, 1997, pp. 23–28.

[11] D. S. Joy, A. Ruchi, G. R. M. Reddy, and K. S. Sowmya, "Bug classification: Feature extraction and comparison of event model using naive bayes approach," *Computing Research Repository*, vol. abs/1304.1677, 2013.

[12] A. Santana, J. Silva, P. Muniz, F. Arajo, and R. M. C. R. de Souza, "Comparative analysis of clustering algorithms applied to the classification of bugs." in *ICONIP (5)*, vol. 7667. Springer, 2012, pp. 592–598.

[13] N. K. Nagwani and S. Verma, "Software bug classification using suffix tree clustering (stc) algorithm," *International Journal of Computer Science and Technology*, vol. 2, no. 1, pp. 36–41, March 2011.