ISE Coursework Task 1

Tool Building Project

March 28, 2025

1 Introduction

For large projects (such as in deep-learning like TensorFlow), there are often too many bug reports submitted per day to be processed. When a report is related to performance, it becomes even more complex as there often isn't a precise benchmark to compare to. Even experienced engineers can spend weeks trying to understand a dozen reports for a popular project. Below I will discuss some of the reasons why bug reports can be difficult to decipher, and how machine learning techniques can assist the process of bug classification and triage.

Bug reports are expressed in natural language text which are often vast and ambiguous, and can often include spelling and grammatical errors, incorrect punctuation, etc. Bettenberg et al. [1] investigated the quality of bug reports and found that they often include incomplete and incorrect information, which can make it challenging for someone to read and understand the root problems of each report, and decide whether it is something they need to fix.

Bug triage is the process of reviewing, prioritizing and allocating software defects by analysing and classifying the detected bugs based on their severity. After this stage, a bug will have either been allocated to a developer to fix or deferred to next release (meaning it has been acknowledged but is of lower priority).

Bug Report Classification is used to speed up the maintenance of projects through automation of the previously discussed process by categorizing bug reports based on their characteristics. It is useful in prioritizing performance-related bugs and filtering out non-issues such as user errors, or removing duplicates.

For my Tool Building Project, I have built a bug report classification based on the baseline code provided in Lab 1 that classifies whether a bug report is performance related or not.

2 Related Work

Through my research, I have found a paper by Murphy and Cubranic [2] that explores the process of automatic bug triage using text categorization. Their approach is very similar to how I plan to tackle performance-related bug classification - they utilise supervised Bayesian learning to predict whether a developer should work on a bug, building the model using a collection of 15,859 bug reports from a large open-source project.

A pro of this approach is their use of Bayesian learning. The Naive-Bayes classifier is computationally efficient which is important for real-time bug triage, and handles high-dimensional feature spaces well which text classification requires.

One con is that, although they have helped reduce the manual effort required, the method has only achieved a 30% correct prediction rate. This means that 70% of reports would still require human assignment. Another con is that they have developed this model specifically for the Eclipse project, meaning it would likely need modification for adapted use.

A very early work in 2008 by Antoniol et al. [3] also makes use of Naive-Bayes classifiers to determine whether the text of the issues posted in bug tracking systems is enough to classify them into corrective maintenance and other kinds

of activities.

3 Solution

To address this problem, I have proposed a supervised learning approach utilising the Naive-Bayes (NB) classification algorithm with TF-IDF vectorization.

The reason why I have selected Multinomial Naive-Bayes (MNB) is because it is well-suited for text classification tasks. MNB is efficient and scalable, making it perfect for the analysis of bug reports in the large datasets used. My main rationale behind changing from the baseline's use of Gaussian Naive-Bayes to Multinomial is that while Gaussian is better suited for continuous numerical data, Multinomial is superior for discrete data.

TF-IDF [4] has been chosen because it effectively captures the significant terms within each bug report relative to the entire dataset, helping the model focus on distinguishing terms rather than unimportant ones.

When splitting the data for training and testing the model, randomization improves the model ability to generalize, and ensures that it can perform on unseen data and not just specified examples. We iterate this process to improve the model's stability, allowing us to generate an average for performance metrics and remove stochastic bias.

4 Setup

The key steps for our solution are:

1. Text preprocessing
2. Feature extraction (using TF-IDF)
3. Model training
4. Evaluation metrics

We first remove all necessary columns of data from the databases (csv's), merging the Title and Body sections into one labelled as text. This merged column is then utilised to classify the whether a bug is performance-related or not, passing through the text preprocessing and feature extraction.

4.1 Text preprocessing

In our text preprocessing, we first remove any HTML tags from the text, then remove any emojis from the text using regex-based filtering. We then complete stopword removal using a combination of NLTK's predefined list and a custom stopwords list, and followed this by normalizing all text to lowercase and removing non-alphanumeric characters.

4.2 Feature extraction

Using TF-IDF, text is then converted into numerical feature vectors, weighting them based on how important a particular word is to a bug report. We will also use n-gram to generate context among words.

4.3 Model training

For each iteration, the dataset is split into training and testing subsets using randomization. Following normal procedure, 70% is allocated to training the model, and the remaining 30% is used for testing against the metrics.

4.4 Evaluation metrics

For my solution, I am using four commonly used metrics for binary classification problems:

- Accuracy - measures overall correctness
- Precision - ability of a model to identify only the relevant data points
- Recall - ability of a model to find all relevant data points
- F1 Score - a combination of precision and recall and used as a measure of predictive performance

As a baseline, I will be using the results of the Lab 1 code against the metrics to compare to my adapted approach with the intention of beating all four metrics. I will be comparing both models on five datasets covering deep-learning projects with a large number of bug reports.

5 Experiments

For my statistical test, the five datasets total to 3,712 GitHub reports, with TensorFlow having the most at 1490, down to Caffe with 286.

5.1 Experiment metrics

	My Model	Baseline	Improvement (%)
Accuracy	0.8540	0.5554	+53.76%
Precision	0.8576	0.6351	+35.07%
Recall	0.6262	0.7110	-11.92%
F1 Score	0.6549	0.5364	+22.11%

Table 1: TensorFlow

	My Model	Baseline	Improvement (%)
Accuracy	0.8647	0.6408	+34.94%
Precision	0.6727	0.6138	+9.60%
Recall	0.5896	0.7581	-22.23%
F1 Score	0.6060	0.5666	+6.96%

Table 2: PyTorch

	My Model	Baseline	Improvement (%)
Accuracy	0.8572	0.5659	+51.47%
Precision	0.7924	0.6316	+25.45%
Recall	0.7260	0.6955	+4.38%
F1 Score	0.7456	0.5469	+36.35%

Table 3: Keras

	My Model	Baseline	Improvement (%)
Accuracy	0.8946	0.6000	+49.10%
Precision	0.7947	0.6077	+30.78%
Recall	0.6559	0.7471	-12.22%
F1 Score	0.6880	0.5370	+28.17%

Table 4: MXNet

	My Model	Baseline	Improvement (%)
Accuracy	0.8960	0.5414	+65.52%
Precision	0.6340	0.5633	+12.56%
Recall	0.5448	0.6513	-16.36%
F1 Score	0.5454	0.5613	-2.94%

Table 5: Caffe

The overall average results for all databases:

	My Model	Baseline	Improvement (%)
Accuracy	0.8733	0.5807	+50.96%
Precision	0.7503	0.6103	+22.69%
Recall	0.6285	0.7126	-11.67%
F1 Score	0.6480	0.5496	+18.13%

Table 6: Overall Average Results

5.2 Discussing observations

The first main observation from the results obtained is the overall accuracy improvement compared to the baseline. My model has, on average, correctly identified performance-related bugs 87.33% of the time. This is an average improvement of 50.96% from the baseline approach. This clearly indicates that the use of Multinomial Naive-Bayes with TF-IDF vectorization is significantly better than the baseline's use of Gaussian.

Except for tests on our Keras dataset, my model's recall metric worsened by 11.67% on average. However, the precision metric showed consistent (22.69% on average). This indicates that the model is more cautious in classifying a bug report as performance related. This results in having fewer false positives, but giving more false negatives. Due to our F1 Score metric being up 18.13% on average, this shows that the higher precision outweighs the recall losses.

Generally, we can also see varied improvements across the datasets, especially in the accuracy metric (Caffe having 65.52% and PyTorch having 34.94%). This suggests that the way certain projects have their bug reports written could make them easier to classify with TF-IDF vectorization.

6 Reflection

Although my model demonstrated significant improvements across the baseline's metrics, there are a few main limitations that could be addressed in the future.

The first is the decrease in my model's recall metric across the test datasets. This indicates that it is missing performance-related bugs, which could prove problematic in real-world applications where this identification is crucial. Any future improvements could focus on balancing both precision and recall metrics using by: collecting more data, fine-tuning model hyperparameters, use a different machine learning algorithm.

As mentioned above, using a different ML algorithm could potentially improve the recall metric. While the model currently proves Multinomial Naive-Bayes as effective, other algorithms like CNN-based deep learning or SVMs may provide better results.

Currently, my model is only evaluated on individual projects at a time, so we have no idea how well a model trained on one project may work when applied to data from other projects. An improvement that could be implemented is cross-project generalization, testing the model's ability to recognise general patterns of performance-related issues rather than project-specific terminology. This is important as it demonstrates that the features used are actually about performance issues.

7 Conclusion

In conclusion, this project developed an improved Bug Report Classification system that effectively identifies performance-related bugs. By replacing Gaussian Naive-Bayes with Multinomial, the model showed significant improvements in accuracy (50.96%), precision (22.69%), and F1 Score (18.13%) when compared to the baseline.

The results attained validate the original hypothesis that machine learning techniques (specifically text classification) can improve bug triage efficiency by automatically identifying performance-related reports. This approach could substantially reduce the amount of manual effort required by developers in this identification process, allowing them to focus on actually addressing the critical issues.

Future work includes studying the possible improvements brought by using other deep-learning algorithms, cross-generalizing the data, and evaluating other methods to improve the recall metric of the model.

8 Artifact

https://github.com/ofirmstone/ISEbrclassification

9 References

[1] Bettenburg, Nicolas, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. "What makes a good bug report?." In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering, pp. 308-318. 2008.

[2] Murphy, G., and Davor Cubranic. "Automatic bug triage using text categorization." In Proceedings of the sixteenth international conference on software engineering & knowledge engineering, pp. 1-6. Citeseer, 2004.

[3] Antoniol, Giuliano, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. "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, pp. 304-318. 2008.

[4] Terdchanakul, Pannavat, Hideaki Hata, Passakorn Phannachitta, and Kenichi Matsumoto. "Bug or not? bug report classification using n-gram idf." In 2017 IEEE international conference on software maintenance and evolution (ICSME), pp. 534-538. IEEE, 2017.