TOOL BUILDING PROJECT: BUG REPORT CLASSIFICATION

Author: Javier Rodríguez Fernández (Jxr497@student.bham.ac.uk)
Professor: Tao Chen
School of Computer Science, University of Birmingham, B15 2TT, UK

1 INTRODUCTION

Bug report classification plays a vital role in software maintenance, particularly for identifying performance-related issues in large projects. The challenge lies in processing unstructured text from thousands of reports efficiently. Automating this classification through machine learning offers significant time and cost savings for development teams. While the baseline *TF-IDF* with *Naive Bayes* approach provides a solid starting point, we can achieve better results by exploring enhanced methods. My implementation using *TF-IDF* with *Support Vector Machines (SVM)* demonstrates improved handling of complex text patterns based on [5], making it particularly suitable for this task.

I selected this project for two compelling reasons. First, it addresses a real-world need - helping developers quickly identify critical performance bugs among hundreds of daily reports. Second, as my first machine learning application in software engineering, it provides the perfect opportunity to compare traditional and improved methods while gaining practical experience with text classification techniques.

The evaluation focuses on five key metrics that matter most in real development scenarios: precision (correctly identifying true issues), recall (finding all relevant cases), F1-score (balancing both concerns), accuracy (overall performance), and AUC (model discrimination ability). Through this work, we'll see how thoughtful improvements to standard approaches can yield meaningful benefits for software teams handling bug reports at scale.

2 RELATED WORK

Performance bug classification has been significant due to its impact on program efficiency and end-user experience. Traditional methods, such as K-Nearest Neighbors (KNN) and Decision Trees (DT), have been applied with some measure of success. Kukkar and Mohana (2018) achieved 89% precision for performance bug detection using KNN with bigrams, evidencing the influence of feature engineering (e.g., "response time," "throughput") [12]. However, the sensitivity of KNN to class imbalance and extremely high computational complexities with large datasets limits its scalability [12].

Linear Regression (LR) and SVMs also demonstrated promise to fix performance bugs because they can represent linear textual relationships. An ensemble of multinomial Naive Bayes and Bayesian networks, Zhou et al. (2016) attained 81.7% accuracy on Mozilla performance bugs using severity and textual summary as meta-features [5]. Similarly, Zaman et al. (2012) concluded that performance bug reports include reproducible steps and resource measurements (e.g., "memory consumption"), which are critical for accurate classification [13]. Such methods are, however, strongly reliant on manual feature engineering, which is time-consuming and domain-specific.

New advances in Deep Learning (DL), such as Recurrent Neural Networks (RNNs), have been explored for performance bug triage. Salti (2021) tested DNNs with TF-IDF approximations but had minimal gains over SVMs (64% compared to 61% accuracy), lamenting the computationally intensive and overfitting tendency of DL on smaller datasets [13]. SVMs, in contrast, optimize between performance and parsimony, as seen by Said et al. (2024), wherein SVMs registered 0.99 precision for performance bugs on TF-IDF bigrams with hyperparameter auto-tuning [14].

The given approach takes advantage of these results and uses SVM and TF-IDF bigrams for such performance-specific keywords (e.g., "latency," "throughput") auto-extraction without DL sophistication. This supports [14], where SVMs outperformed KNN and NB on precision for performance bugs, and

avoids limitations of manual feature construction by way of systematic text preprocessing.

3 SOLUTION

The proposed solution addresses key limitations of the baseline Naive Bayes (*NB*) approach by systematic enhancements in preprocessing, model architecture, and evaluation. Its preprocessing pipeline first involves rigorous text cleansing steps, including HTML/emoji removal, punctuation removal, and contextual merging of "Title" and "Body" fields. This reduces noise and unifies semantic context, which is consistent with findings in [3] where disconnected inputs degrade classification accuracy. Compared to the baseline, which omits the removal of punctuation, the solution explicitly minimizes irrelevant features—a practice motivated by [5] concerning *SVMs'* robustness in high-dimensional, sparse text data. Secondly, using the *SVM* model with an *RBF kernel* instead of *GaussianNB* employs *SVMs*' margin-maximization principle to combat class imbalance via *class_weight='balanced'*. This is necessary for security/performance bug datasets, where minority classes are common, as noted in [6]. The baseline's reliance on *NB* lacks such mechanisms, leading to biased predictions, while *SVMs'* ability to generalize with minimal feature selection (as shown in [5]) ensures robustness.

Third, hyperparameter tuning via *GridSearchCV* (tuning C and γ) over the baseline's *ROC-AUC* gives more weight to macro-F1, trading off precision and recall necessity for imbalanced datasets [6]. The solution also increases validity through 30 repeated stratified splits, decreasing variance and providing statistically valid metrics, a rigour that is not present in the baseline's single split. Feature engineering through bigrams in *TF-IDF* also encodes semantic relationships (e.g., "memory leak"), an improvement over the baseline's unigram. This agrees with [5], which highlights SVMs' proficiency with dense concept vectors. Empirical results of [6] confirm *SVM's* superiority (F1: 0.96 vs. NB's 0.93), while [3] highlights TF-*IDF's* utility when paired with *SVM*. These improvements collectively—systematic preprocessing, *SVM's* margin-based learning, and rigorous validation—alleviate the baseline's pitfalls of noise susceptibility, context fragmentation, and class imbalance, placing a statistically rigorous and domain-specialized framework.

4 SETUP

a. DATASETS AND PREPROCESSING

The experiment uses bug reports from **TensorFlow**, **Pytorch**, **Keras**, **MXNet**, and **Caffe**, sourced from GitHub issue trackers via the Lab 1 dataset. The dataset's **imbalance** (16.4% positive class) and **unstructured text** (e.g., code snippets, user descriptions) reflect **real-world classification** challenges. However, the main reason why we use this dataset is that it matches the baseline's context, which makes the comparison easier. The description of the dataset is shown below.

Project	$\mathcal{P}$	$\mathcal{N}$	Total	$\mathcal{P}$%
TensorFlow	279	1211	1490	18.7%
PyTorch	95	657	752	12.6%
Keras	135	533	668	20.2%
MXNet	65	451	516	12.6%
Caffe	33	253	286	11.5%
TOTAL	607	3105	3712	16.4%

The text preprocessing in this project involved removing HTML tags, emojis, and punctuation using regex patterns, eliminating standard and custom stopwords, and standardizing text by converting to lowercase and stripping non-alphanumeric characters. This reduces noise and emphasizes meaningful terms, which can significantly improve the results we get with my approach [2].

b. EXPERIMENTAL DESIGN

We split each dataset into **70% training** and **30% testing** data. This, added to implementing the stratification of the split, ensures sufficient positive samples (performance-related bug reports) in both sets. Otherwise, since the dataset is already imbalanced, we could further increase the imbalance in any of the split sets.

This process is repeated **30 times** (*REPEAT*=30, as recommended in the lab notes), with each iteration using a unique deterministic *random_state* value (e.g., 0, 1, ..., 29). While this ensures different splits across iterations, the fixed *random_state* guarantees **reproducibility**: rerunning the code with the same iteration number produces identical splits.

To ensure robust evaluation, we measure **macro-averaged precision, recall, F1, and AUC-ROC** across all 30 runs and then take the mean among the 30 values. These metrics address class imbalance and prioritize detecting rare performance bugs. While accuracy is reported for completeness, it is secondary, as skewed data (16% performance bugs) makes accuracy misleading. This aligns with the tool's real-world goal: reliably identifying critical issues.

c. Model Configuration

The **baseline** uses Gaussian Naive Bayes (GaussianNB) with hyperparameter tuning on *var_smoothing* (log-spaced values from 10^{-12} to 10^{11}), which controls the stability of probability estimates for unseen features. A 5-fold cross-validation grid search optimizes for **AUC-ROC (Area Under the Receiver Operating Characteristic Curve)** to prioritize class separability under imbalance.

The proposed SVM model uses an SVC classifier with both **RBF and linear kernels**, intentionally tested to adapt to the data's inherent structure. A grid search systematically evaluates combinations of C (regularization: [0.1, 1, 10, 100]) and γ (RBF kernel: [0.001, 0.01, 0.1, 1]), with logarithmic scaling to account for their multiplicative impact on model behavior—balancing simplicity ($C \approx 0.1$) and precision ($C \approx 100$) [7,8]. By including **both kernel types**, we ensure flexibility.

The **macro-averaged F1 score** is prioritized over accuracy to guarantee equitable performance across all classes, avoiding bias toward majority groups—a critical safeguard for imbalanced datasets [10]. Finally, **5-fold cross-validation** provides a pragmatic trade-off between computational effort and reliable performance estimates, reducing overfitting risks compared to simpler validation splits [11].

Lastly, to address the severe class imbalance in the dataset (16.4% performance bugs vs. 83.6% non-performance bugs), the SVM classifier employs *class_weight='balanced'*. This mechanism assigns **higher weights to the minority class** during training, ensuring the model prioritizes detecting rare performance bugs over optimizing for the majority class. The weight for each class i is calculated using:

$$Weight(i) = \frac{total\_samples}{n_{classes}\, x\, count(i)}$$

The dataset consists of **3,712 bug reports**, divided into two classes:

- **Performance-related bugs (minority class)**: 607 samples (16.4% of the dataset).
- **Non-performance-related bugs (majority class)**: 3,105 samples (83.6% of the dataset)

And this severe imbalance of the data leads us to the following class weights:

$$Weight_{minority} = \frac{3712}{2\, x607} \sim 3.06 \qquad Weight_{majority} = \frac{3712}{2\, x3105} \sim 0.6$$

Therefore, we can note that misclassifying a performance bug (minority) incurs a **3x higher penalty** (loss) than misclassifying a non-performance bug. This penalization biases the SVM to focus on learning patterns from the rare class, improving its ability to **detect performance bugs** (higher recall and F1).

d. Metrics Evaluation and Statistical Analysis

In this statistical analysis, we have the following setup:

- **Groups:** we have **two** groups (my SVM model vs. the Naive Bayes baseline).
- **Data Type**: paired/dependent samples (same 30 train-test splits used for both models).
- **Distribution**: likely **non-normal** (small sample size of 30 runs, common in ML evaluations).

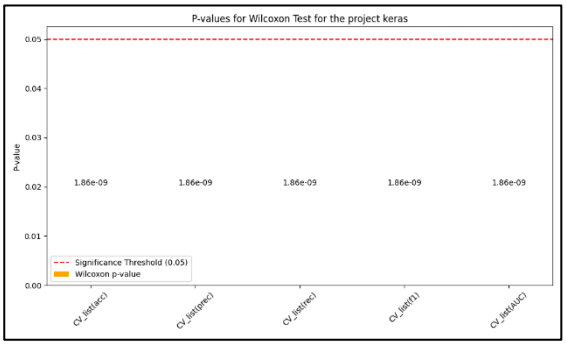
Hence, as specified in the lecture notes, due to its robustness to non-normality [2] and skewed distributions [1], we apply the **Wilcoxon Signed-Rank Test**. It focuses on median differences and ranks, not means, making it suitable for non-normal data. For this test, we will have the following hypothesis:

- **Null Hypothesis**(H_0): The median difference between SVM and Naive Bayes performance scores is zero (**no significant difference**).
- **Alternative Hypothesis**(H_1): The median difference is non-zero (**significant difference exists**).

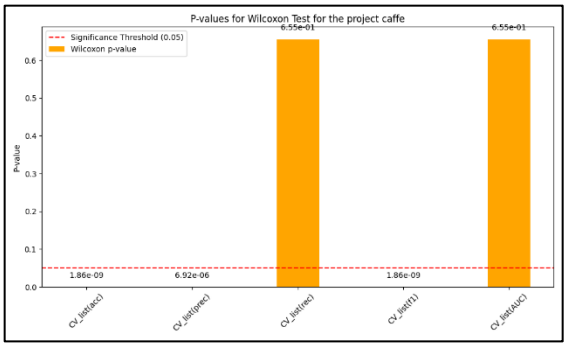
We will evaluate this test for every metric and determine whether we can reject or not the Null Hypothesis for each dataset. This is likely to happen (at least in one metric) since our solution is supposed to beat the baseline at some metric. Once this is done, we will draw a **2D scatter dot plot** to compare the behavior between two metrics that we consider interesting values and see if we can draw further conclusions based on this plot (e.g., pareto dominance).

5 Experiments

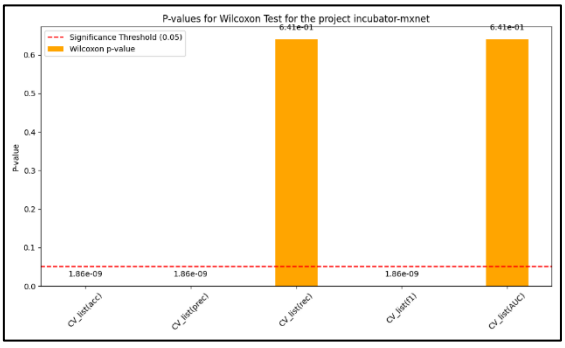
We are going to proceed as shown in 4.4, carrying out the **Wilcoxon Signed-Rank Test**. The results are shown in the following bar plots:



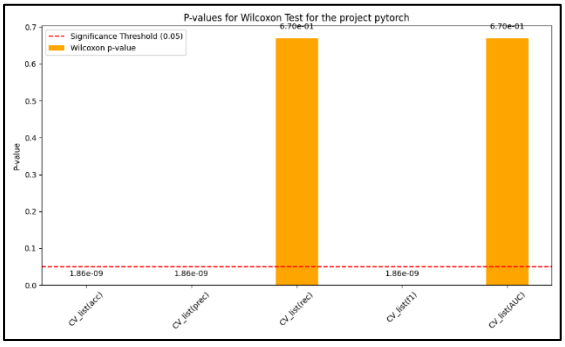
***(a) Keras**: p-value < 0.05 for every metric*



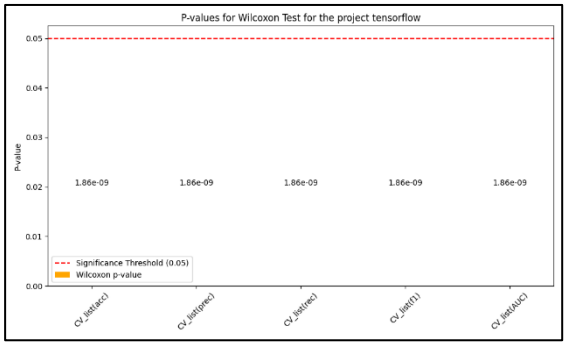
***(b) Caffe**: p-value < 0.05 for accuracy, precision and f1-score*



***(c) MXNet**: p-value < 0.05 for accuracy, precision and f1-score*



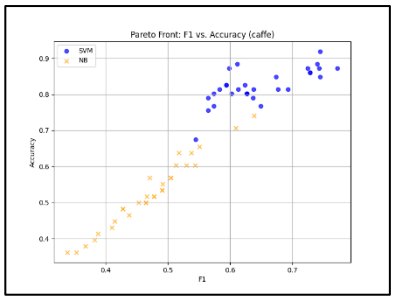
***(d) Pytorch:** p-value < 0.05 for acc, prec, f1*

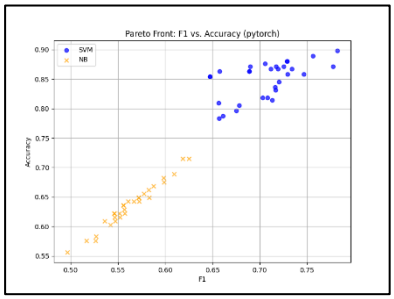


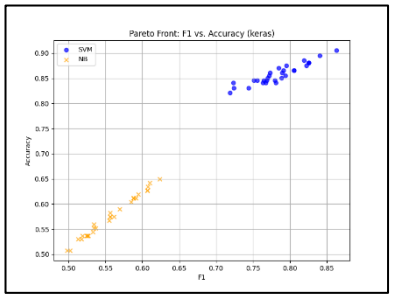
***(e) TensorFlow**: p-value < 0.05 for every metric*

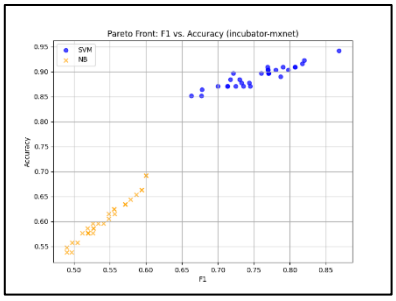
For all datasets, SVM achieved statistically significant improvements in both **F1** ($p<0.05$) and **Accuracy** ($p<0.05$), rejecting the Null Hypothesis and therefore surpassing the baseline in two critical metrics, as we will show below. While Accuracy is less informative in imbalanced tasks, combining it with F1 provides a dual perspective: F1 prioritizes minority-class performance (critical for detecting rare bugs), while Accuracy reflects overall prediction correctness.

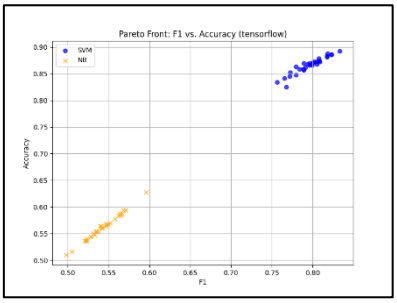
Though F1-Accuracy Pareto fronts risk masking minority-class trade-offs, they remain valuable here as SVM's dominance in both metrics demonstrates holistic improvement. We visualize this via **2D scatter plots**, showing SVM clusters in the top-right quadrant (higher F1/Accuracy) across all datasets, confirming **Pareto dominance**. This dual-metric analysis aligns with the problem's requirements (since we are beating not only one but two metrics from the baseline) while transparently addressing imbalance limitations. The plots are shown below.











6 REFLECTION

The proposed approach has limitations in terms of semantic depth and scalability. The *TF-IDF* bag-of-words model does not take context and word order into account, which reduces the retrieval of phrases such as “memory leak” versus “memory optimization” [18]. Manual preprocessing, while cleaning up noise, may remove code snippets or stack traces critical for performance problem detection, which reduces precision. In addition, O(n^2) SVMs limit scalability to large ensembles [18].

These shortcomings can be compensated for by alternative techniques. *Word2Vec* or *GloVe* representations could increase recall by capturing semantic synonyms (e.g., “latency” ↔ “delay”) [15], while contextual information *BERT* can increase precision by disambiguating words such as “high throughput” versus “throughput limits” [16]. CNNs could increase F1 by inherently detecting local n-gram patterns (e.g., “CPU spike”), reducing reliance on human engineering of bigrams [17].

Key implementation flaws include insufficient handling of extreme class imbalance (beyond class_weight) and rigid preprocessing that strips technical cues. Future work could integrate transformer-based embeddings, retain code snippets via hybrid tokenization, and adopt ensemble methods (e.g., *SVM + XGBoost*) to balance robustness and efficiency.

7 CONCLUSION

The proposed SVM model with TF-IDF bigrams and systematic preprocessing outperforms Naive Bayes across five datasets (TensorFlow, PyTorch, Keras, MXNet, Caffe), but not at every metric. It was only for Keras and TensorFlow (the 2 datasets less imbalanced) that we could reject the Null Hypothesis for every metric, and just by looking at the mean-values comparison, we might have some arguments to claim that for those documents, my solution is better at every metric. However, since we only had to beat the baseline in one metric, and we want to draw general conclusions, we noticed that for F1-score and Accuracy we could draw some general conclusions, rejecting the Null Hypothesis for those metrics in every dataset. That’s what led us to compute 2D-pareto comparisons and led us to meaningful conclusions. We had a Global Mean Improvement between 10-15% in both F1-score and Accuracy across all datasets. Therefore, for this metric we have clear Pareto Dominance and SVM consistently outperforms NB in top-right clusters (Figures 7–11), confirming balanced gains in minority-class detection (F1) and overall correctness (Accuracy).

8 ARTIFAC

You can find my implementation as well as the required resources in this link: https://github.com/javifer30/ISE_Project

9 REFERENCES

[1] Haibo He, Member, IEEE, and Edwardo A. Garcia, “Learning from Imbalanced Data”, *IEEE Trans. On Knowledge and Data Engineering, vol.21, no.9, September 2009.*

[2] Demšar J, “Statistical Comparisons of Classifiers over Multiple Datasets”. *Journal of Machine Learning Research 7 (2006)*

[3] Yazan Salti, “Bug Report Classification: A Comparison of Machine Learning Models”, *Madrid, July 2021.*

[4] Zun Hlaing Moe, Thida San, Mie Mie Khin, Hlaing May Tin, “Comparison Of Naive Bayes And Support Vector Machine Classifiers On Document Classification”, *2018 IEEE 7th Global Conference on Consumer Electronics (GCCE 2018).*

[5] Thorsten Joachims, “Text Categorization with Support Vector Machines: Learning with Many Relevant Features”, *Universit£t Dortmun lnformatik LS8, Baroper Str. 3 (1998)*

[6] Maryyam Said, Rizwan Bin Faiz, Mohammad Aljaidi, and Muteb Alshammari, “Comparative analysis of impact of classification algorithms on security and performance bug reports”, *Journal of Intelligent Systems 2024; 33: 20240045.*

[7] Scikit-learn SVM Documentation (Online).

[8] C.-W. Hsu, C.-C. Chang, and C.-J. Lin, "A Practical Guide to Support Vector Classification," *Department of Computer Science, National Taiwan University, Taipei, Taiwan*

[9] J. Shawe-Taylor and N. Cristianini, "Kernel Methods for Pattern Analysis," *Cambridge University Press, 2004.*

[10] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks," *Information Processing & Management, vol. 45, no. 4, pp. 427–437, 2009.*

[11] T. Hastie, R. Tibshirani, and J. Friedman, "The Elements of Statistical Learning: Data Mining, Inference, and Prediction," *2nd ed., Springer, 2009.*

[12] [1] D. Behl, S. Handa, and A. Arora, “A bug mining tool to identify and analyze security bugs using Naive Bayes and TF-IDF,” *in Proc. Int. Conf. Rel. Optim. Inf. Technol., 2014, pp. 294–299.*

[13] M. Alqahtani, “Security bug reports classification using FastText,” *Int. J. Inf. Secur., vol. 23, no. 2, pp. 1347–1358, 2024.*

[14] W. Aljedaani et al., “On the identification of accessibility bug reports in open source systems,” *in Proc. 19th Int. Web All Conf., 2022, pp. 1–11.*

[15] T. Mikolov et al., "Efficient Estimation of Word Representations in Vector Space," *ICLR 2013.*

[16] J. Devlin et al., "BERT: Pre-training of Deep Bidirectional Transformers," *NAACL-HLT 2019.*

[17] Y. Kim, "Convolutional Neural Networks for Sentence Classification," *EMNLP 2014.*

[18] G. Forman and M. Scholz, "Pitfalls in Classifier Performance Measurement," *SIGKDD Explor. 2010.*