# Automatic Classification of Bug Reports for Mobile Devices: An Industrial Case Study

**Renata F. Lins[1], Flávia A. Barros[1], Ricardo B. C. Prudêncio[1], Wallace N. Melo[2]**

[1]Centro de Informática (CIn) – Universidade Federal de Pernambuco (UFPE)
Recife – PE – Brazil

[2]CIn UFPE – Motorola Cooperation Project
Recife – PE – Brazil

{rfl4, fab, rbcp}@cin.ufpe.br, wallacen@motorola.com

***Abstract.*** *When a failure is found during software testing activities, a bug report (BR) is written and stored in product management tools. In order to prioritize the errors to fix, a BR triage process is performed to identify the most critical errors. This is specifically relevant in the context of mobile applications due to the fast development cycle, which results on a high number of BRs to evaluate daily. In this paper, Machine Learning (ML) and Natural Language Processing (NLP) techniques are investigated to automatically classify the criticality of BRs in the context of a real mobile environment, and a prototype was developed. Results on a corpus of 9,785 BRs were very satisfactory, reaching up to 0.79 of AUC and meeting the performance level required by the considered application.*

## 1. Introduction

The demand for innovative and high-quality mobile applications makes the development and maintenance of these devices a constant challenge for companies. In order to guarantee that products meet the requirements established for their operation, in addition to providing quality, reliability, security and robustness, companies invest a large part of their resources in error detection practices, which are associated with prevention, discovery or correction of inappropriate behavior [Wagner 2006]. Thus, during the development cycle of a product, different types of tests are carried out, in order to assess that the functionalities have been correctly developed and integrated.

In general, companies use project management tools to have a clear and centralized overview of the development stages. When a test case fails, a bug report (BR) describing the encountered error is created and reported by the testers in these tools. The BRs are then inspected in a triage process, which aims to identify the criticality of the error reported, assisting with the efficient and strategic allocation of resources necessary for error fixing. The most critical issues are those that, if not corrected, can block the software development or even delay the launch of a particular product on the market. Therefore, critical errors must be solved more quickly.

Clearly, the triage process of critical BRs is highly relevant for monitoring, maintaining and improving product quality, especially in large companies, where the daily amount of new BRs can be high. However, it is a costly, labor intensive and time-consuming activity for developers, which is usually carried out manually. Mobile application environments deal with dynamic scenarios, where resources are limited and development cycles tend to be fast [Podgurski et al. 2003]. As a solution, the triage process can

be supported by the use of Machine Learning (ML) techniques to automatically classify the criticality of BRs. Automating the triage process of error reports during the development stage aims to bring improvements to the process of detecting and correcting critical errors, reducing associated costs, ensuring efficient resolution of problems and resulting in software quality.

From the above perspective, this paper focuses on the automatic classification of criticality of BRs in the context of mobile devices. The goal is to help developers quickly and accurately identify reports that address critical errors in order to readily mitigate them. This work was carried out within the context of a research cooperation project between Motorola Mobility (a Lenovo Company) and CIn-UFPE, with the objective of developing tools to support the BR triage for the different types of tests performed by the company's test teams. The prototype developed follows a process that has three main steps: (1) Preprocessing of BRs, based on Natural Language Processing (NLP) techniques; (2) Classification of BR criticality, based on ML algorithms, and (3) Visualization of results, through the user interface.

The proposed process was implemented in a prototype that was tested with historical data from 9.784 BRs of the partner company, labeled as critical or non-critical during manual error triage. In the experiments, two distinct preprocessing techniques and five ML algorithms from different families were evaluated. Finally, statistical tests were performed to identify the strategies that produced the best results for the criticality classification of the BRs in the considered mobile development environment. The results obtained were very satisfactory (with AUC rates that reached 0.79).

The contributions of this paper can be summarized as follows:

- An investigation of ML techniques for criticality classification of BRs on mobile devices, an application context that is not well explored in the literature;
- A prototype implemented in a real environment, focusing on the development of a system to support the triage process in the partner company;
- Experiments carried out to evaluate the best configurations of techniques for the developed system, aiming to identify the best practices for future work.

The remaining of this paper is organized as follows. Section 2 presents the related work, followed by Section 3 that presents the paper proposal. Section 4 presents the implementation details of the developed prototype. The experimental results are presented and discussed in Section 5. Finally, Section 6 concludes the paper with final considerations and future work.

## 2. Related Work

During the software life cycle, errors can occur in different phases, whether in the development, in the testing phase, or even after the release of a product. The taxonomy employed by [Avizienis et al. 2013] distinguishes failures from defects. Failures are the observable impact of errors. A failure is an event that occurs when the behavior observed by the product deviates from the correct behavior that it should perform, according to the previously established requirements. The defect is the cause of the error. In turn, errors can be classified considering their consequences, ranging from trivial to critical.

Different types of tests are performed during the design of a product, mainly in the development phase. When a failure is observed during the execution of a test, the responsible tester can submit a bug report (BR), so that actions can be taken to correct the problem. A BR is a document that must contain relevant information regarding the observed failures that occurred in the product, such as: summary, textual description of the erroneous behavior, steps to reproduce the error, version of the product, examples of code, severity and priority level, and comments [Bettenburg et al. 2008]. Thus, the main objective of a BR is to help stakeholders to identify potential problems and mitigate them.

This section briefly presents some works related to the topic of our research, which focuses on the criticality classification of bug reports for mobile devices. It is worth mentioning that in the available literature we did not find any work with the same objective as ours. Thus, some of the works discussed here were developed with the objective of classifying criticality of BRs, but in a desktop context, while other works deal with processing mobile device BRs automatically, however with other purposes.

### 2.1. Criticality Classification - General Context

It is possible to find in the literature different studies that deal with the automatic classification of BRs criticality in the context of desktop applications. We initially highlight the publications [Uddin et al. 2017, Zhang et al. 2015], which present an overview of the studies carried out and the different techniques used for this purpose.

It is worth mentioning more specific works, such as [Otoom et al. 2019], which proposes a classifier of BRs in two classes: corrective reports (focusing on correcting defects) and improvement reports (with a focus on maintenance). The proposal is to help developers to identify the type of error reported and allocate the necessary resources for the proper treatment of each case. The results showed an accuracy of 93.1% using Support Vector Machines and a text representation based on the occurrence of selected keywords, rather than using the whole document term matrix.

We also cite the work of [Goseva-Popstojanova and Tyo 2018]. In order to identify vulnerabilities in software, the authors propose a supervised and an unsupervised ML approach for the classification of bug reports. Textual information was extracted from the reports (title, summary and description), followed by the application of preprocessing techniques and text vectorization, so that the data could be used by the algorithms. Three different approaches were used to generate textual vectors: Bag-of-Words with Frequency (BF), Term Frequency (TF) and Term Frequency - Inverse Document Frequency (TF-IDF). Supervised algorithms had better overall performance compared to unsupervised algorithms for this specific problem.

### 2.2. Classification of BRs in the Context of Mobile Applications

The work developed by [Runeson et al. 2007] focuses on the identification of duplicate BRs submitted in project management tools. The authors investigated the use of different NLP techniques, analyzing a database from Sony Mobile. Techniques such as tokenization, stemming, removal of stopwords, word representation based on weighted term frequency, and calculation of similarities using the cosine similarity between words were used in the development of a prototype. This study was able to identify up to 40% of duplicate reports in the BR repository.

Finally, we highlight the work by [Tong and Zhang 2021], which proposes a new strategy for ranking critical reports based on crowdsourcing. This strategy has recently gained popularity in the software testing arena. In this modality, a large amount of BRs from testing activities are collected. The authors adopted a technique to extract textual features from the reports by retaining verbs and nouns, combining this information with image features from the screenshots that are attached by using the Spatial Pyrimid Matching (SPM). This information is then summarized in a hash table using locality-sensitive hashing (LSH), creating an effective strategy for ranking critical reports by measuring the entropy of the information.

## 3. Developed Work

This work proposes a process to assist in the triage of BRs, developing an intelligent solution based on NLP techniques and ML algorithms. The final objective is the automatic classification of the criticality of BRs in the context of mobile application development. Therefore, it is worth noting that the proposed process can be easily adapted to other usage scenarios.

The process is composed of three distinct modules: (1) Preprocessing, where each BR is transformed into a vector representation, based on NLP techniques (Section 3.1); (2) Criticality classification of BRs, where an ML model performs the classification of BR criticality, based on its vector representation (Section 3.2); and (3) User interface, which enables the visualization of classification results (Section 3.3). The process was also implemented in a prototype, evaluated experimentally with a real dataset from the partner company (Section 4), and different NLP and ML techniques were compared in the experiments (Section 5), in which statistical tests were performed to evaluate specific components of the proposed system.

### 3.1. Document Preprocessing

The error reports used in this work are stored in a repository maintained by the partner company, previously labeled as critical and non-critical, which contains several fields that describe the problems encountered during software development and testing. The fields used to conduct our study on criticality classification are: summary, description and priority. The first two fields contain textual information, while the last one is a numeric field (with numbers assigned ranging from 1 to 3).

The textual fields (summary and description) consist of raw, unstructured text. These fields were processed in two steps: (a) cleaning and normalization of the text, with the objective of creating a list of words (vocabulary) to represent the documents; and (b) transformation of the text into a vector representation of each document based on the vocabulary extracted in the first step. The vector is then concatenated with the numeric data (priority), where each dimension represents a feature used by the ML algorithm in the classification step. The techniques used in the preprocessing module of the implemented prototype will be presented in Section 4.2.

### 3.2. Classification of Criticality

A classifier can be defined as a function $f : X \rightarrow Y$, which will map an instance $x_i \in X$ (space of features) to a class $y_i \in Y$ (space of classes). In our context, each instance is the

vector representation of the extracted information from the BRs. The class attribute, in turn, indicates the criticality of the error, which is the variable to be predicted (i.e., whether the error is critical or non-critical). In our case, the criticality classification problem is a binary classification task. The classification model is induced using a collection of BRs with known criticality class, being a supervised learning problem. The ML algorithms used in the prototype will be presented in Section 4.3.

Another aspect considered in our work was the problem of imbalanced classes in the training set. Real data usually suffer with unbalanced classes, which can significantly compromise the performance of algorithms [He and Garcia 2009]. Therefore, balancing the classes in the training set helps to improve the generalization ability of the algorithms, so that the classification does not become biased toward the majority class. In the application context, the tendency is to observe a greater number of non-critical BRs in relation to critical BRs. In order to address this issue, we used an oversampling approach to equal the number of reports in both classes.

### 3.3. User Interface

In order to facilitate the use of the developed classifier, a user interface was implemented, through which the user can feed the input data (BRs that must be classified), as well as visualize the result of the automatic classification of criticality. Thus, the application allows classifying BRs quickly and simply, providing support to the different phases of software development. Figure 1 shows how the developed application works.

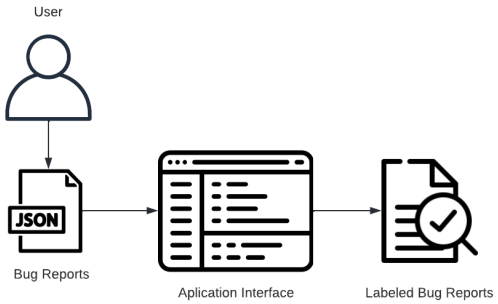


**Figure 1. Application Operation.**

## 4. Implemented Prototype

Based on the process described in the previous sections, a prototype was developed. Figure 2 illustrates how the experimental evaluation was performed in general. The entire prototype was developed in Python, and the steps are detailed as follows.

### 4.1. Data Collection and Description

The dataset used in the case study is maintained by the partner company using the Jira management tool[1]. BRs are related to various tests performed on mobile applications during the different stages of the development cycle. Altogether, 9,784 labeled reports were retrieved, of which 2,838 are critical, which demonstrates an imbalance between the classes of 29.01% of the examples belonging to the positive class.

[1]Jira is a trademark of Atlassian Pty Ltd

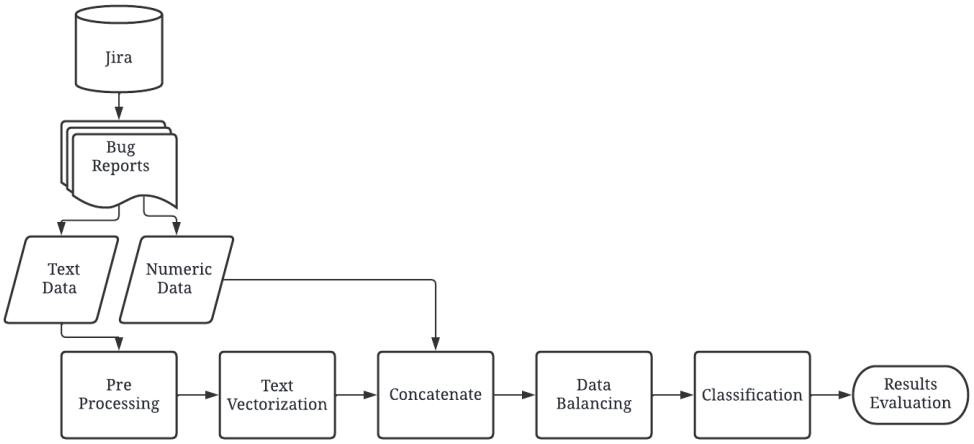


**Figure 2. Evaluation Process.**

### 4.2. Document Preprocessing

The steps used to clean and normalize documents and create a vocabulary of words for vector representation are described below. These steps were applied to the summary and description fields of the BRs.

- Tokenization: It can be described as the process of dividing the text into smaller pieces (called tokens), which can be composed of words, syllables, or letters [Gasparetto et al. 2022]. In the prototype, the NLTK[2] Tokenize module was used, adopting words as tokens and using white spaces as a separator;
- Normalization: The Regular Expression (RegEx) technique was used to identify string patterns. All words have been converted to lowercase letters. Symbols, punctuation marks, line breaks, non-ASCII characters and numbers were also removed, as they do not add relevant information to our context;
- Stopwords Removal: This step aims to eliminate non-discriminating words, which occur very frequently in a corpus. In our prototype, the words contained in the NLTK Python package were removed from the input text (with some exceptions, e.g., “on”, “off”, “not”). It was also necessary to remove words according to the application domain, for that, a list of words with their frequency of occurrences was created [Saif et al. 2014]. The objective was to identify very frequent words that did not add relevant information (e.g., “actual”, “expected”);
- Uncommon Words Removal: Words with less than 5 occurrences were also removed, as they are generally considered rare words or misspellings;
- Lemmatization: In this step, words are grouped according to their lemma, that is, their canonical form (e.g., boys $\rightarrow$ boy) [Jivani 2011]. The NLTK WordNetLemmatizer class was used;
- Stemming: This preprocessing step aims to map the words to their stem, removing affixes (prefixes and suffixes), which are morphic elements added to a word in order to provide it with other meanings (e.g., atypical $\rightarrow$ typical). The algorithm adopted here was Krovetz Stemming[3]. The result of the application is a

[2]NLTK - www.nltk.org

[3]Krovetz Stemmer - pypi.org/project/KrovetzStemmer

real word, unlike other Stemming algorithms considered more aggressive (such as Porter or Lancaster), where the conversion result often produces a non-existent word [Jivani 2011].

The total number of words extracted from the corpus was 7,193. After the definition of the vocabulary, the vector representation of the text is created. In the developed prototype, two representation techniques were considered, as described below.

- Bag-of-Words representation: each dimension of the vector representation contains the TF-IDF value associated with a specific word in the vocabulary (totaling 7,193 dimensions). This representation was implemented using the TfIdfVectorizer[4] method, and was first proposed by [Luhn 1957, Jones 1972];
- Word2Vec representation: this model is pre-trained on the Google News[5] database (which has 300 dimensions, with fixed vectors for each word), proposed by [Mikolov et al. 2013]. The model was trained using the Skip-Gram architecture with negative sampling. Only the words present in the corpus are mapped to their Word2Vec representation.

Preprocessing is applied separately to the textual data (summary and description). The resulting representations are then concatenated with the numeric data (priority) to create a unique representation of the BRs, used as input by the learning algorithms. All methods parameters were set to default settings.

### 4.3. Classification

In the classification phase, the considered ML algorithms are described below. The SKLearn[6] library was used, and the parameters are shown in Table 1.

- Logistic Regression: The Logistic Regression model for classification tasks is used to calculate the probability of classes through linear combinations of the predictor attributes;
- Gaussian Naive Bayes: The Gaussian Naive Bayes algorithm is a variation of Naive Bayes, based on Bayes' Theorem, which assumes independence between terms, where classes are modeled through a normal distribution;
- KNN: The algorithm performs the classification of new instances by calculating the distance between the k nearest examples (calculating the similarity between instances), using the majority rule to assign a class;
- SVM: Support Vector Machines are non-linear classifiers that map instances to high-dimensional spaces, so that it is possible to build a separating hyperplane between classes;
- Random Forest: The Random Forest algorithm builds a collection of decision trees, where the attributes used to split the leaves are chosen randomly through the bootstrap method. Classifications are made by averaging the results of the collection of decision trees.

During the training phase, a balancing technique was considered in order to minimize any possible bias of the algorithms towards the majority class. To deal with this

[4]TF-IDF - scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer
[5]Word2Vec - github.com/mmihaltz/word2vec-GoogleNews-vectors
[6]SKLearn - https://scikit-learn.org

**Table 1. Models Hyperparameters.**

| Algorithms | Parameters |
|---|---|
| LR | Penalty = L2, C = 1, Solver = LBFGS |
| GNB | - |
| KNN | K = 5, Cosine Distance |
| SVM | Kernel = RBF, C =1, Gamma = Scale |
| RF | Estimators = 100, Criteria = Gini |

issue, the Synthetic Minority Over Sampling (SMOTE)[7] technique was applied, where synthetic examples of the minority class are created, which are produced based on the closest examples of samples of the minority class, chosen randomly [Chawla et al. 2002].

### 4.4. Interface

Finally, according to the evaluation of the results, it was possible to choose the best approach, through the scientific method, for the construction and implementation of a prototype to classify the criticality of the error reports. Figure 3 shows the created interface. The user can interact with the application by submitting bug reports in a JSON format, and visualize or download the classification results. For the implementation of the prototype, the Flask[8] web framework was used, implemented in a WSGI[9] server.

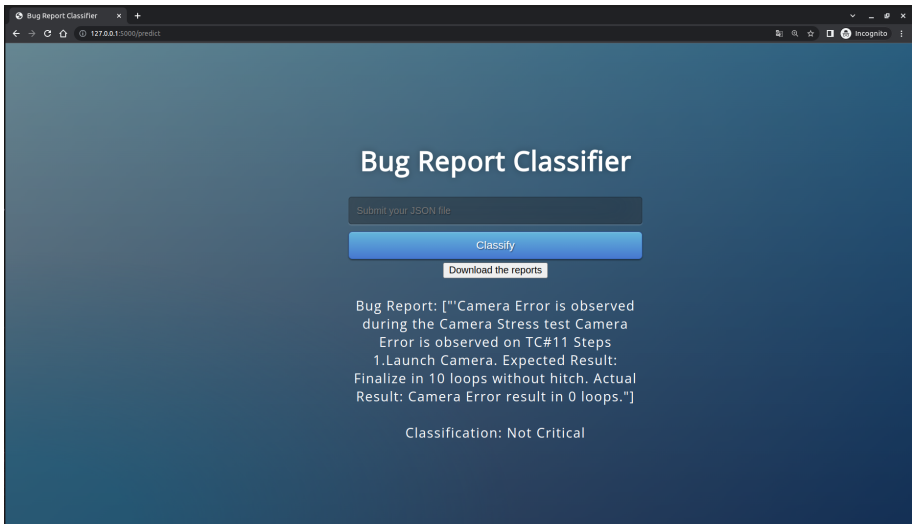


**Figure 3. User Interface.**

## 5. Experimental Evaluation

In this section, we present the results of the conducted experiments in order to evaluate the developed process for classifying the criticality of bug reports. Different aspects were considered in the experiments, such as: (a) the learning algorithm; (b) the textual representation technique; and (c) the usefulness of balancing the training data.

[7]SMOTE - imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.SMOTE
[8]Flask - flask.palletsprojects.com/en/2.2.x
[9]WSGI - wsgi.readthedocs.io/en/latest/what.html

### 5.1. Methodology

In order to evaluate the performance of the models, we used Stratified K-Fold Cross-Validation. The data is partitioned into 10 segments, and subsequent 10 iterations of training and validation are performed. In each round, different k-1 segments are used for training and balanced using SMOTE, and one segment is kept separate for validation. The stratification maintains the original distribution of the data in the validation sets [Refaeilzadeh et al. 2009].

The metric used to evaluate the results is the Area Under the ROC Curve (AUC). The ROC Curve (Receiver Operating Characteristics) is a technique for visualizing, organizing and selecting classifiers based on their performance. The ROC Curve plot shows True Positive Rate against the False Positive Rate. The AUC metric has a relevant statistical property, since it represents the probability of a classifier ranking a positive example correctly, given that it was chosen randomly. Therefore, it can be seen as the ability of a classifier to distinguish between classes [Fawcett 2006]. Considering that the classes are unbalanced, and the positive class (critical BRs) is the class of interest, the AUC metric was chosen for the evaluation of the results.

Finally, to compare the results of the experiments and verify if there is a significant difference regarding the generalization ability of the classifiers, statistical tests were applied to compare the data samples.

### 5.2. Experiment Results

In this section, the results of the criticality classification of the error reports are analyzed. The predictive performance of the five different learning algorithms are compared using the two different textual representations and the data balancing technique.

Table 2 presents the classification results using the AUC metric in two scenarios: applying the algorithms to the original dataset (without using the SMOTE balancing technique), and applying the algorithms to the balanced dataset. The values correspond to the mean of the cross-validation results.

**Table 2. Classification Results.**

| | Original Dataset | | SMOTE | |
|---|---|---|---|---|
| Algorithms | TF-IDF | Word2Vec | TF-IDF | Word2Vec |
| LR | 0,7889 | 0,7511 | 0,7844 | 0,7493 |
| GNB | 0,5444 | 0,6609 | 0,5452 | 0,6366 |
| KNN | 0,7402 | 0,6453 | 0,7211 | 0,6369 |
| SVM | 0,7740 | 0,7660 | 0,7769 | 0,7545 |
| RF | 0,7943 | 0,7500 | 0,7956 | 0,7489 |

As it can be seen, the AUC rates vary around 0.7, reaching 0.79, which indicates the feasibility of using ML for the application in question. Only the result of the GNB algorithm, with the TF-IDF representation, obtained results close to 0.5 (reference value for AUC). The best performance was obtained by the RF algorithm, with rates ranging from 0.74 to 0.79. In general, for the LR, SVM and RF algorithms, the AUC value achieved confirms that the developed models are able to correctly predict the criticality of error reports.

Considering the textual representation technique, the TF-IDF representation produces better results for most algorithms (4 out of 5) compared to the Word2Vec representation. The TF-IDF is calculated based on the set of documents and the corpus used, that is, it takes into account the domain of our application. The W2V model in turn was trained on the Google News database, which may not provide a good representation for our context.

### 5.3. Statistical Tests

To analyze the impact of using a balancing technique, the statistical Mann-Whitney U Test was used, which compares two independent samples. The test compares the results obtained from the original dataset with those from the balanced dataset. The test p-value of 0.7913 indicates the failure to reject the null hypothesis, meaning that the samples come from the same distribution. Therefore, there is no improvement in adopting a balancing method for this specific problem.

To verify the impact of different textual representation techniques, the T-Test was used. First, we compared the samples from the original dataset regarding the TF-IDF against the W2V representations. Then, we compared the samples from the balanced dataset. The p-value obtained was 0.8139 and 0.7306, respectively, which indicates the failure to reject the null hypothesis. That is, the two samples have equal means. However, it is evident (from Table 2) that in some cases there was a significant improvement in the generalization capacity of the algorithms when adopting the TF-IDF method.

Regarding the learning algorithms, the Friedman test was performed along with the Nemenyi post-hoc test, which is used when all classifiers are compared to each other. According to [Demsar 2006], the performance of two classifiers are significantly different if the corresponding average ranks differ by at least the critical difference, and the algorithms that are not significantly different from each other are connected. The results of the post-hoc test can be visually analyzed with the Critical Difference diagram in Figure 4. In the graph, the RF, SVM and LR are considered the best group of algorithms, followed by the KNN and GNB. Therefore, the groups of algorithms that are connected have no significant difference from each other.

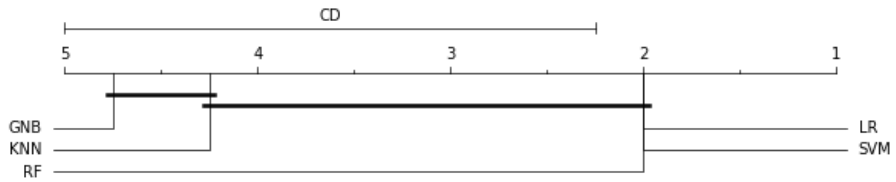


**Figure 4. CD - Critical Difference.**

## 6. Conclusion

The study presented an experimental evaluation of different Natural Language Processing techniques and Machine Learning algorithms to propose an application capable of classifying the criticality level of error reports within the context of mobile applications. A statistical comparison was made between textual representation techniques and different algorithms in order to find a suitable solution for the issue.

It is possible to conclude that, although neural language models represent the state of the art with regard to NLP techniques for text representation, frequency-based techniques, such as TF-IDF, still prove to be competitive in terms of their performance.

In general, the Random Forest algorithm showed the best result for the classification of criticality, considering that it adopts the bootstrap method for resampling the attributes for the construction of the trees. This feature makes the algorithm particularly effective when dealing with problems that have unbalanced classes.

Finally, a prototype was built based on the proposed process to help classify the criticality of error reports in a mobile application environment. The prototype was built using the TF-IDF representation and the Random Forest algorithm, considering that they presented a better overall performance.

Directions for future work include investigating the explainability of models, identifying relevant attributes for the predictions using different techniques. Another line of study is to identify concept shift regarding data streams.

## Acknowledgments

This work has been supported by the research cooperation project between Motorola Mobility (a Lenovo Company) and CIn-UFPE. The authors have also been supported by CAPES and CNPq (Brazilian funding agencies).

## References

Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2013). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1:11 – 33.

Bettenburg, N., Just, S., Schröter, A., Weiss, C., Premraj, R., and Zimmermann, T. (2008). What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, page 308–318, New York, NY, USA. Association for Computing Machinery.

Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). Smote: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357.

Demsar, J. (2006). Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7:1 – 30.

Fawcett, T. (2006). Introduction to roc analysis. *Pattern Recognition Letters*, 27:861–874.

Gasparetto, A., Marcuzzo, M., Zangari, A., and Albarelli, A. (2022). A survey on text classification algorithms: From text to predictions. *Information*, 13.

Goseva-Popstojanova, K. and Tyo, J. (2018). Identification of security related bug reports via text mining using supervised and unsupervised classification. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 344–355.

He, H. and Garcia, E. A. (2009). Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering*, 21:1263–1284.

Jivani, A. (2011). A comparative study of stemming algorithms. *International Journal of Computer Applications in Technology*, 2:1930–1938.

Jones, K. S. (1972). A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28:11–21.

Luhn, H. P. (1957). A statistical approach to mechanized encoding and searching of literary information. *IBM Journal of Research and Development*, 1:309–317.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *Proceedings of Workshop at ICLR*.

Otoom, A. F., Al-jdaeh, S., and Hammad, M. (2019). Automated classification of software bug reports. In *Proceedings of the 9th International Conference on Information Communication and Management*, ICICM 2019, page 17–21.

Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., and Wang, B. (2003). Automated support for classifying software failure reports. In *25th International Conference on Software Engineering*, pages 465–475.

Refaeilzadeh, P., Tang, L., and Liu, H. (2009). Cross-validation. *Encyclopedia of Database Systems*, pages 532–538.

Runeson, P., Alexandersson, M., and Nyholm, O. (2007). Detection of duplicate defect reports using natural language processing. In *29th International Conference on Software Engineering (ICSE'07)*, pages 499–510.

Saif, H., Fernández, M., He, Y., and Alani, H. (2014). On stopwords, filtering and data sparsity for sentiment analysis of twitter. In *LREC*.

Tong, Y. and Zhang, X. (2021). Crowdsourced test report prioritization considering bug severity. *Information and Software Technology*, 139.

Uddin, J., Ghazali, R., Deris, M. M., Naseem, R., and Shah, H. (2017). A survey on bug prioritization. *Artificial Intelligence Review*, 47:145 – 180.

Wagner, S. (2006). A literature survey of the quality economics of defect-detection techniques. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, page 194–203.

Zhang, J., Wang, X., Hao, D., Xie, B., Zhang, L., and Mei, H. (2015). A survey on bug-report analysis. *Science China Information Sciences*, 58:1 – 24.