An Intelligent Bug Severity Classification System Using TF-IDF, SMOTE, and Logistic Regression

Lie Qian

1 Introduction

Background and Motivation:

The reliability and quality of a software programme have always been an important and challenging aspect of software design. With the increasing size and complexity of software, software developers and system operators need to invest a lot of time in evaluating and resolving anticipated and unanticipated bugs that may negatively affect the user experience. Bug reports, as the core document reflecting software defects, usually contain information such as defect descriptions, operating environments, expected behaviours, and actual problems, and their contents cover both the severity of the bugs and the priority of the bugs. Bug reports, as the core document reflecting software defects, usually contain information such as defect descriptions, operating environment, expected behaviour, and actual problems. For a long time, this task mainly relied on system operation and maintenance personnel to complete manually, which is time-consuming and labour-intensive, and prone to inconsistency due to subjective judgement. Therefore, how to automatically and accurately classify bug reports has become a critical necessity to improve software quality and reduce maintenance costs.

- **Problem Description:**
  A software bug is a failure in the program which causes unexpected or unwanted outputs, Our aim is to make an automated evaluation tool that processes and classifies bug reports onBugzilla Eclipse Bug Reports dataset. Specifically, the tool will extract textual features from the ‘Short Description’ of a bug report and automatically predict the severity of the bug, which is usually classified into one of the following categories
  - **Blocker:** blocks the work of testing and development.
  - **Critical:** is crashes, loss the data, memory leak.
  - **Major:** is a major loss of function.
  - **Minor:** is a trivial function loss or other problem is easy to solve.
  - **Normal:** is a regular issue, resulting in a loss of some functions under specific conditions.
  - **Trivial:** is a cosmetic problem that does not significantly impact the system.

In real scenarios, due to the large number of ‘normal’ bugs and the small number of samples of blocker, critical, major and other key categories, the direct use of traditional classification models often results in poor recognition of a few categories. For this reason, this paper not only combines preprocessing techniques such as text cleaning, word shape reduction, TF-IDF feature extraction, and data enhancement, but also employs strategies such as SMOTE, random oversampling, and hyper-parameter tuning, in order to improve the classification performance and robustness of the model under unbalanced datasets.

The goal of this research is to build a system that can automatically classify and predict the severity of bugs, thus helping the development team to quickly identify the most critical issues among the huge number of bug reports and rationally allocate maintenance resources to improve software quality and user experience.

---

2 Related Work

In the field of software defect prediction and defect report severity classification, researchers have long been committed to building automated and intelligent analysis models to alleviate the problems of high cost and high error of manual determination. This task essentially combines three major challenges of natural language processing, machine learning, and unbalanced data modeling, and centers on accurately extracting key semantic features from complex, heterogeneous reports and completing high-confidence classification decisions accordingly.

Kukkar et al. (2019) [1] proposed a composite architecture that fuses deep learning with traditional integrated learning: the model first utilizes Convolutional Neural Networks (CNNs) to learn hierarchical representations of defective descriptive text, and then combines Random Forests with Boosting algorithms to complete a multi-category classification task. The method has an average 96.34 % accuracy on several open-source project datasets (e.g., Eclipse, Firefox), demonstrating the potential of deep semantic modeling and multi-model integration in bug severity classification. It is worth to

note that its reliance on high-quality structured inputs and a large number of samples limits its generalization ability in low-resource environments, and there is no explicit countermeasure for the problem of extreme category imbalance.

In contrast, Shatnawi and Alazzam (2022) [2] focus on dual-task prediction of priority and severity of Eclipse defect reports, building a system architecture centered on feature selection, text vectorization, and a variety of traditional machine learning classifiers (e.g., SVM, KNN, AdaBoost, Random Forest). They extract key attributes from metadata (OS, components, version numbers, etc.), and characterize text summaries through TF-IDF and use SMOTE techniques to mitigate data bias. The experimental shows the filtering feature and integrated learning strategy has significant effects in improving the model stability and F1 scores, and especially achieves considerable performance improvement in dealing with minority class problems.

Compared with the above studies, the method proposed in this paper seeks to maintain the dual balance of prediction accuracy and model interpretability while simplifying the modeling path. We process highly sparse short text representations with TF-IDF and incorporate SMOTE intelligent synthetic samples to significantly improve the recognition of key classes such as blocker and critical while maintaining overall model accuracy. More importantly, the Logistic Regression classifier adopted in this study is highly transparent and can intuitively reflect the role of each feature in the classification boundaries, thus providing a more practically valuable interpretation framework for model deployment and practical use.

3 Solution

3.1 General Description

In real-world software defect management scenarios, bug reports often contain short text descriptions and other features. Since the percentage of critical severity samples is often very small, traditional machine learning models tend to be biased towards recognizing the mainstream categories, such as ”normal,” and inaccurately predicting a few categories, such as “blocker” or “critical”. The traditional machine learning models tend to be biased towards recognizing mainstream categories such as ”normal,” while inaccurately predicting a few categories, such as “blocker” or “critical”. To this end, we introduce the **SMOTE** (Synthetic Minority Over-sampling Technique) method after preprocessing the text to alleviate the category imbalance, and utilize **Logistic Regression** to classify the severity of the bugs into multiple categories, and ultimately achieve better recognition performance for the minority categories.

The core idea of this research is as follows:

- Customized Text Cleaning: Accurately filter the redundant symbols and noise in the original text;
- TF-IDF: using the clever combination of word frequency and inverse document frequency to convert text into high-dimensional sparse features;
- SMOTE oversampling: Intelligent interpolation to expand a few categories after vectorization, breaking the shackles of data imbalance;
- Logistic Regression multi-class prediction: Constructing classification boundaries with interpretability in a multi-sample environment;
- Evaluation and Visualization: Presenting a panoramic view of the model’s performance through multiple metrics, such as accuracy, macro-averaged F1 scores, and confusion matrices.

3.2 Data Preprocessing and Cleaning

In the data preprocessing stage, we are committed to building a rigorous yet flexible cleaning process to ensure that each bug report is carefully cleaned before entering the feature extraction process, so as to eliminate noise and highlight meaningful information.

3.2.1 Text Cleaning

First of all, for the “Short Description” field in the bug report, we design a set of customized cleaning strategies, whose goal is to eliminate redundant and irrelevant noise while keeping the information whole. The steps are as follows:

- Underline Replacement: In order to prevent boundary blurring during word segmentation, all underscores (_) are replaced with standard half-width spaces;
- Filtering short numeric tokens: Given that tokens with a length of less than 3 and containing numbers often represent version numbers or other short strings with no substantive information, such tokens are eliminated;
- Retaining the rest of the Token: while ensuring that redundant information is removed, words that are critical to subsequent categorization are strictly retained to avoid information loss caused by over-cleaning.

This hierarchical cleaning not only ensures the semantic integrity of the text but also provides a clean and high-quality input to the subsequent quantization process.

3.2.2 TF-IDF vectorization

After text cleaning, we introduced the TfidfVectorizer tool to transform the processed text into a numerical sparse matrix. The main parameters are set as follows:

- `ngram_range=(1,2)`: captures both the fine-grained features of a single word (unigram) and the contextual associations of a double phrase (bigram);
- `stop_words='english'`: reduces noise interference by excluding common stop words;
- `max_features=None`: allows an unlimited dictionary size to ensure that no information is left out.

3.3 SMOTE oversampling

Since the "normal" category occupies the majority of the dataset, and the samples of "blocker" and "critical" are extremely scarce, it is difficult for the traditional model to capture their subtle characteristics. Therefore, we introduce the SMOTE technique after obtaining the TF-IDF feature matrix of the training set. This method generates new synthetic samples by interpolating the feature space of the minority samples, thus breaking the monotonicity and overfitting problems caused by simple replication, and enabling the model to learn the effective patterns of the minority classes in a more balanced sample distribution.

3.4 Classification Model: Logistic Regression

Among the many possible classification algorithms, we choose Logistic Regression as the core classifier, whose main advantage lies in its excellent adaptability to high-dimensional sparse data as well as its excellent interpretation ability. Not only does the model operate efficiently in large-scale feature spaces, but each token or bi-gram corresponds to a weight coefficient, which allows us to intuitively understand which features play a decisive role in distinguishing between severity levels.

3.4.1 Hyper-parameterization

In order to maximize performance in complex data environments, we have carefully tuned the Logistic Regression, and the main hyperparameter settings include:

- `C=30`: A higher inverse regularization parameter allows the model to be more flexible in delimiting the boundaries, especially giving full attention to minority class samples;
- `class_weight='balanced'`: ensures that minority classes are not overlooked due to sample scarcity by automatically adjusting class weights in the loss function;
- `max_iter=270`: increase the upper iteration limit to cope with the challenge of convergence in large-scale high-dimensional feature spaces;
- `solver='lbfgs'`: this optimization algorithm shows great efficiency and stability when dealing with multi-class problems and sparse data.

3.5 Implementation Process

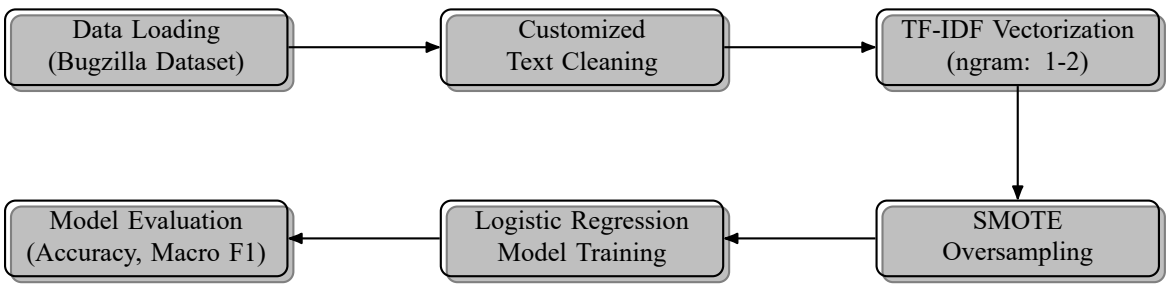

Figure 1: Flowchart of the Proposed Bug Severity Classification System

1. Data loading: The complete dataset from Hugging Face [3] to ensure easy to use.
2. Training/testing partitioning: 80% training set, 20% test set, and the `random_state` is used to ensure the experimental results same.
3. Text preprocessing: `custom_clean_text()` is used to the underscore replacement and short number token delete.

4. The TfidfVectorizer is used to transform the cleaned text into a numeric sparse matrix.
5. SMOTE oversampling: Generate new synthesized samples to balance the category distribution.
6. Model training: The tuned Logistic Regression model is used to accurately fit multiple categories.
7. Model Evaluation: On the test set, the model performance is evaluated visually by calculating the accuracy, Macro F1-score, and plotting the confusion matrix.

---

4 Setup

- In this section, we not only elaborate the hardware and operating system composition of the experimental environment, but also analyze the source and size of the dataset, the selection process of hyper-parameters and the final configuration layer by layer, and finally make a multi-dimensional comparison between the evaluation indexes and the baseline model. The overall scheme not only highlights the technical rigor, but also pursues a fine balance in each step, trying to provide an optimized solution for complex text classification tasks.

4.1 Hardware Configuration and Operating System Environment

In order to cope with the high performance computation required for large-scale text processing and SMOTE oversampling, the experimental platform adopts the following configuration:

Hardware specifications:

- Processor: 13th Gen Intel® Core™ i9-13900H (2.60 GHz),
- Memory: up to 96.0 GB

Operating System:
Windows 11 Professional (version 23H2). Python 3.9.12

4.2 Overview of data sources and sizes

The dataset used in this study is taken from “AliArshad/Bugzilla_Eclipse_Bug_Reports_Dataset” on Hugging Face platform [3], which covers multi-dimensional information in Eclipse Bugzilla. Specifically, the dataset includes the following fields:

- **Project:** The name of the project to which the bug belongs.
- **Bug ID:** a unique identifier for each bug.
- **Severity Label:** identifies the severity level of the defect, such as “blocker”, “critical”, “major”, “minor”, ‘trivial’, and ‘normal’.
- **Resolution Status:** Reflects whether the defect has been fixed, confirmed, etc.
- **Short Description:** provides a concise description of the defect.

It should be noted that the distribution of the data is extremely uneven, with the proportion of “normal” category as high as 81.4%, while the other categories together account for only 18.6%, and this extreme distribution is especially significant in the original 88.7k reports. After rigorous screening and text cleaning, the data fluctuates slightly, but the overall distribution of categories is still clearly characterized by imbalance, which poses a challenge for subsequent model tuning.

4.3 Experimental Process and Core Steps

The entire experimental process is rigorously designed and interlocked, and different strategies are adopted at each stage to achieve overall optimization:

- **Data Segmentation:** The dataset is partitioned according to the ratio of 80% (training set) and 20% (test set), and `random_state=1` is used to ensure the reproducibility of the experimental results.
- **Text Preprocessing:** Customized functions are used to remove underscores and noisy short words in an orderly manner, thus improving the data quality and laying a solid foundation for subsequent vectorization.
- **Feature vectorization (TF-IDF):** With TfidfVectorizer, `ngram_range=(1,2)` and `stop_words='english'` are used for feature extraction without limiting `max_features`, so as to fully capture the potential information of the text.

- **SMOTE oversampling:** Apply SMOTE technique to the training set only, to realize the synthetic expansion of a few classes of samples, to ensure that the training data tends to be balanced in terms of categories, `random_state` is also set to 1.
- **Logistic Regression Model Training:** The model is trained and analyzed using a variety of hyperparameter configurations to explore the optimal parameter settings.
- **Model Evaluation:** By calculating Accuracy, Macro F1-score, Confusion Matrix and other indexes on the test set, the model's ability to recognize and predict a small number of samples is examined in an all-round way.

4.4 Hyper-parameter Tuning and Final Configuration

We explore the multi-dimension of Logistic Regression and TF-IDF, and the core settings are as following:

Logistic Regression hyperparameters

- `C:` select from {0.1, 1, 10, 30} to control the strength of inverse regularization.
- `solver:` selected from {'lbfgs', 'sag'} to ensure that the optimization algorithm balances convergence speed and stability.
- `max_iter:` candidate values are set to {100, 200, 270} to cater for the effect of different iteration counts on model accuracy.
- `class_weight:` take {'balanced', None} to examine whether the class weights are balanced and adjusted.

TF-IDF hyperparameters

- `ngram_range:` compare (1,1) and (1,2) settings to find the effect of a single unigram versus a unigram and a bigram;
- `max_features:` candidate values include {None, 10000, 50000, 80000} to bound the effect of dictionary size on the feature space;
- `stop_words:` {None, 'english'} is selected to check the effect of removing English stop words.

The performance of each hyperparameter combination is accurately evaluated by implementing a 3-fold cross validation (3-fold CV) pre-experiment on the training set. Finally, the optimal configuration was determined by combining Accuracy and Macro F1-score metrics:

- Logistic Regression: `C=30`, `solver='lbfgs'`, `max_iter=270`, `class_weight='balanced'`, and fixed `random_state=1`;
- TF-IDF: `ngram_range=(1,2)` and `stop_words='english'` were chosen, with no limit on `max_features`.

This optimal configuration will serve as the core basis for comparison with the baseline model in all subsequent experiments.

---

5 Experiments

First, I conducted baseline experiments using a Naive Bayes model with two distinct training strategies. I compar with just having Naive Bayes and adding random oversampling (ROS). We performed a comprehensive comparison with the use of SMOTE oversampling with a Logistic Regression classifier. The experiments cover Accuracy and Macro F1-score, Precision, Recall, and F1-score. To reveal the performance strengths, weaknesses, and potential limitations of each model when dealing with unbalanced data.

5.1 Phase I: Plain Bayesian Baseline Comparison

5.1.1 Naive Bayes without oversampling

In the first phase of the experiments, we used the MultinomialNB classifier to train and predict the raw data based on text features that had been transformed by TF-IDF. Despite the 81.4% 'normal' class in the dataset, the model managed to climb to an overall accuracy of about 0.82, while the macro average F1-score hovered at a worrisome 0.18. The classification report revealed that the model is heavily biased in favour of predicting 'normal', but not 'blocker', 'critical', 'minor' and "trivial", the F1-score is extremely low. In this set of experiments, the overall accuracy seemed to be quite impressive, but the model's negligence of the key few categories revealed a serious imbalance problem.

Table 1: Classification Report for Naive Bayes without oversampling (Baseline)

Class	Precision	Recall	F1-score	Support
blocker	1.00	0.01	0.01	159
critical	0.96	0.07	0.14	1067
major	0.00	0.00	0.00	894
minor	1.00	0.01	0.02	579
normal	0.82	1.00	0.90	13951
trivial	0.00	0.00	0.00	419
Accuracy			0.82	
Macro avg	0.63	0.18	0.18	17069
Weighted avg	0.77	0.82	0.75	17069

5.1.2 Naive Bayes + Random Oversampling (ROS)

In order to address the above data imbalance problem, we introduced the RandomOverSampler (ROS) strategy in the training set, which attempted to correct the category skew to a certain extent by simply replicating and expanding the samples of a few categories. The experimental results showed that although the recall and F1-score of a few categories (e.g., 'blocker', 'critical') were significantly improved, the recall and F1-score were also improved. The detailed classification report further revealed the paradox of this strategy: after replicating a small number of samples, the native distribution of the data was severely damaged, resulting in a sharp drop in the prediction accuracy of the 'normal' category, and the overall model performance is greatly affected as a result. The performance of the whole model is thus greatly affected.

Table 2: Classification Report for Naive Bayes with Random Oversampling (ROS)

Class	Precision	Recall	F1-score	Support
blocker	0.12	0.48	0.19	159
critical	0.43	0.69	0.53	1067
major	0.13	0.39	0.19	894
minor	0.09	0.32	0.14	579
normal	0.92	0.53	0.67	13951
trivial	0.10	0.41	0.17	419
Accuracy			0.52	
Macro avg	0.30	0.47	0.31	17069
Weighted avg	0.79	0.52	0.60	17069

Table 3 summarises the Accuracy and Macro F1-score results of the two plain Naive Bayes

Table 3: Baseline Experiment Results for Naive Bayes

Method	Accuracy	Macro F1
Naive Bayes (without ROS)	0.82	0.18
Naive Bayes + ROS	0.52	0.31

5.2 Phase 2: Logistic Regression + SMOTE

We improved the limitations by adopting SMOTE, which interpolates the samples in the high-dimensional feature space composed of TF-IDF, to generate more semantically representative minority samples without destroying the original data distribution. We replaced the classifier with Logistic Regression, which shows excellent performance in handling high-dimensional sparse data, and chose hyperparameters such as `C=30`, `class_weight='balanced'`, `max_iter=270`, and `solver='lbfgs'`. The model maintained the overall accuracy at about 0.81, Macro F1-score jumped to about 0.39. This result not only demonstrates the advantages of SMOTE in improving the recognition of minority classes, but also proves the Logistic Regression in keeping the prediction of majority classes steady.

Table 5 Two plain Naive Bayes in the first stage, visually demonstrating that the Logistic Regression + SMOTE scheme significantly outperforms the traditional baseline method in terms of both overall accuracy and minority class macro average F1-score.

5.3 Analysis of results

It can be seen from the experimental results:

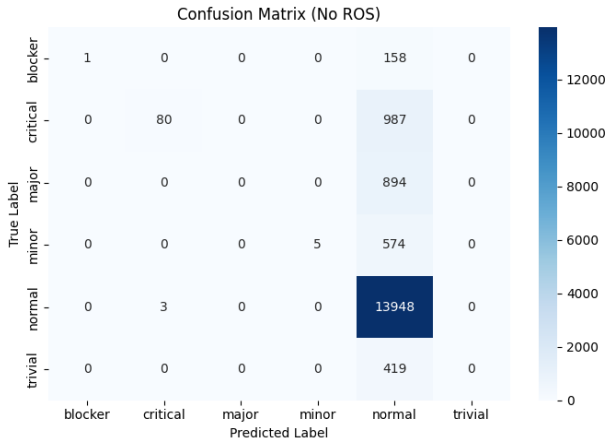

Figure 2: **Naive Bayes with no ROS**

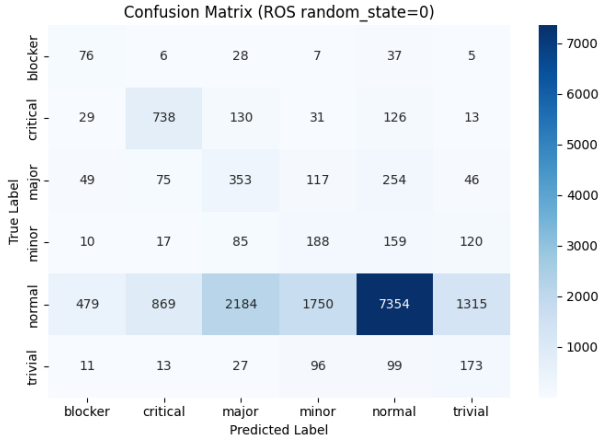

Figure 3: **Naive Bayes with no ROS**

Table 4: Classification Report for Logistic Regression with SMOTE

Class	Precision	Recall	F1-score	Support
blocker	0.34	0.31	0.32	160
critical	0.66	0.71	0.68	1157
major	0.23	0.14	0.18	935
minor	0.16	0.08	0.11	598
normal	0.88	0.92	0.90	14478
trivial	0.25	0.18	0.21	409
Accuracy			0.81	
Macro avg	0.42	0.39	0.40	17737
Weighted avg	0.78	0.81	0.80	17737

Table 5: Comparison of Methods

Method	Accuracy	Macro F1
Naive Bayes (no oversampling)	0.82	0.18
Naive Bayes + ROS	0.52	0.31
Logistic Regression + SMOTE	0.81	0.39

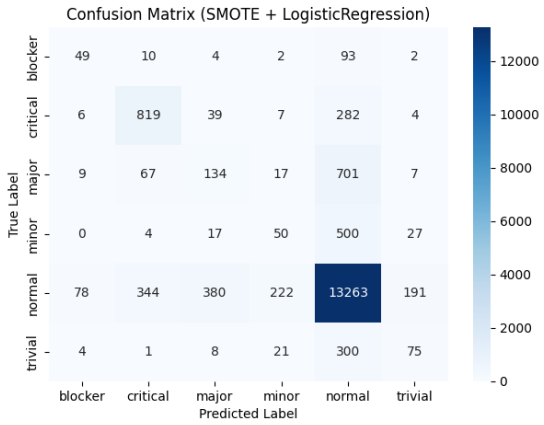

Figure 4: **Logistic Regression + SMOTE**

- When no oversampling technique is used, although the plain Bayesian model achieves high performance in overall accuracy, it relies heavily on majority class prediction, resulting in extremely weak recognition of minority classes and a significantly low Macro F1-score.
- Although random oversampling improves the minority recall and F1-score to some extent, its simple replication destroys the original distribution of the data, resulting in a drastic decrease in the overall accuracy.
- By generating new samples in the feature space, SMOTE effectively preserves the data diversity, and supplemented with the linear advantage of Logistic Regression, the model is able to better balance the recognition ability of the majority and minority categories while maintaining the overall accuracy, which significantly improves the Macro F1-score of key categories.

---

6 Reflection

The automatic bug severity classification scheme based on TF-IDF feature representation, SMOTE oversampling and Logistic Regression classifier proposed in this study has shown big benefits in solving the problem of data distribution imbalance and enhancing the detection ability of key minority classes, but there are still multiple limitations and bottlenecks that need to be broken through at the level of theoretical depth and practical application.

Advantages

- Significantly improve the ability of minority class identification
  SMOTE technology is used to expand the samples of key categories such as ‘blocker’ and ‘critical’ in the high-dimensional feature space, which not only increases the diversity of data, but also allows these originally scarce categories to be more fully displayed in the training stage through linear interpolation. Using the `class_weight='balanced'` parameter of Logistic Regression, the model can better balance the majority and minority classes in the overall prediction process, which significantly improves the F1-score of key classes.

Limitations and Future Improvements

- **Limitations and Improvements of Text Representation Methods**
  Although the current TF-IDF method has the advantages of simplicity and high efficiency, it relies only on word frequency statistics. It cannot detect the implicit context information, semantic relationships and long-distance dependencies in the text. It is obviously insufficient in capturing complex terminology and semantic nuances. In the future, more accurate and efficient semantic understanding can be achieved by directly judging the severity of bugs with the help of the question and answer or classification capabilities of Large Language Models (LLMs).
- **Limitations of SMOTE in High-Dimensional Sparse Space**
  SMOTE de-balances some of the category distributions, and the use of simple linear insertion in the high-dimensional and sparse space constituted by TF-IDF may amplify the original noise. The generated synthetic samples lack sufficient semantic realism. There is some impact on the model generalisation ability.
- **Unidimensionality of Model Selection and Insufficient Nonlinear Expression Capability**
  Logistic Regression shows good training efficiency and interpretation ability on high-dimensional sparse data, but its linear decision boundary is not enough to face the complex and nonlinear relationships in bug reports. Future research can

try the multi-model integration strategy, fusing Random Forest, Gradient Booster, or even Deep Neural Network with existing methods, improving the overall robustness and prediction accuracy of the model with the help of integrated learning; using large language models for preprocessing and feature extraction of the input text, and inputting rich nonlinear features into traditional classifiers, will be a path worth exploring.

- **Limitations of Data Preprocessing and Domain Adaptation**
  Currently adopted text cleaning mainly relies on simple rules (e.g., underline replacement, short number filtering), and although this method reduces noise to a certain extent, it is difficult to comprehensively solve the problems of spelling errors, special symbols, and domain proper names, which may lead to the omission or misinterpretation of key information.

---

7 Conclusion

Aiming at the challenging problem of bug severity classification in Eclipse Bugzilla dataset, this study constructs an automated recognition system based on TF-IDF feature extraction, SMOTE oversampling and Logistic Regression classifier. Through the well-designed text pre-processing and feature construction process, we not only alleviate the data imbalance problem to a certain extent, but also achieve significant improvement in the detection of ‘blocker’, ‘critical’, and other key minority classes. The experimental results show that the Macro F1-score of the traditional plain Bayesian model is much lower than expected, although the overall accuracy of the model is not bad, which mainly reflects that the method is overly biased towards the ‘normal’ category, which is the leading category; in contrast, the method of combining the Logistic Regression with SMOTE maintains the accuracy of about 81%, which is the best result. In contrast, the SMOTE combined with Logistic Regression significantly improves the macro-mean F1-score to 0.39 while maintaining an overall accuracy of about 81%, thus fully verifying the effectiveness of the oversampling strategy in enhancing the performance of minority category identification.

8 Artifact

The source code, experimental data, and related documentation for this project are publicly available and hosted in the following repository. The root directory of the repository contains the following files:

- **requirements.pdf**: Lists all dependencies and their respective versions required to compile and run the code properly.
- **manual.pdf**: Provides detailed instructions on how to install, configure, and use the tool, including an overview of each module’s functionality and step-by-step usage guidelines.
- **replication.pdf**: Offers a complete procedure and relevant details for reproducing the experimental results presented in this report, enabling others to verify the findings.

Repository Link: `https://github.com/superggfun/BugSeverityClassifier`
Please refer to the above documents for detailed instructions on installation, usage, and replication.

References

[1] Kukkar, A., Mohana, R., Nayyar, A., Kim, J., Kang, B.-G., & Chilamkurti, N. (2019). A novel deep-learning-based bug severity classification technique using convolutional neural networks and random forest with boosting. *Sensors*, *19*(13), 2964. https://doi.org/10.3390/s19132964. PMID: 31284398. PMCID: PMC6651582. `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6651582/`

[2] Shatnawi, M. Q., & Alazzam, B. (2022). An assessment of Eclipse bugs’ priority and severity prediction using machine learning. *International Journal of Communication Networks and Information Security*, *14*(1), 62–68. `https://doi.org/10.17762/ijcnis.v14i1.5266`

[3] Lamkanfi, A., Pérez, J., & Demeyer, S. (2013). The Eclipse and Mozilla defect tracking dataset: A genuine dataset for mining bug information. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR)* (pp. 203–206). IEEE. `https://doi.org/10.1109/MSR.2013.6624028` `https://huggingface.co/datasets/AliArshad/Bugzilla_Eclipse_Bug_Reports_Dataset`