UDC 004.8:004.4:005.94 **DOI:** https://doi.org/10.30837/0135-1710.2025.187.156

O. Shmatko, I. Gamayun, O. Kolomiitsev

# HYBRID MACHINE LEARNING MODEL FOR CLASSIFYING SOFTWARE BUGS IN SAAS CLOUD APPLICATIONS

In modern cloud computing environments, ensuring the stability and reliability of software applications is one of the key factors for the effective operation of information systems. A significant portion of failures in such systems are caused by software errors (bugs), which complicate operation and reduce service productivity. Traditional methods of manual analysis of bug reports are labor-intensive, so it is necessary to develop intelligent approaches to automated classification and prioritization of bugs using machine learning methods. **The purpose of the article** is to improve the accuracy of classifying types of software bugs in cloud applications. **Research objectives:** to develop a complete pipeline for automated processing of bug reports, covering all stages from preliminary cleaning to classification model building. **The methodological basis of the study** is the use of natural language processing (NLP) methods, the SMOTE technique for sample balancing, classical machine learning algorithms, and the *RandomizedSearchCV* hyperparameter optimization procedure. The quality of the models is evaluated based on standard classification metrics such as *accuracy, precision, recall, and F1-score*, which provides a comprehensive and objective analysis of the results. Research results. A hybrid model for automated bug classification has been developed, covering the stages of data collection, preprocessing, vectorization, and modeling. A comparative analysis of the accuracy of four machine learning algorithms – naive Bayes classifier, decision tree, random forest, and logistic regression – was performed using different vectorization methods (*Bag-of-Words, TF-IDF, Word2Vec*). To improve classification accuracy, the SMOTE data balancing technique was applied. Experimental studies on a real data set from a cloud environment showed that the Random Forest model achieved the highest accuracy rates – up to 91.7 %. The results confirm the effectiveness of integrating machine learning algorithms into the processes of analysis and support of software products in cloud infrastructures. **Conclusions.** The proposed approach improves the accuracy of bug classification in cloud computing systems and can be used in monitoring systems, *DevOps* platforms, and automated testing tools. The research results provide a basis for the further development of intelligent tools for predicting and prioritizing software defects.

**Keywords:** bug classification; cloud computing; machine learning; *TF-IDF; Word2Vec*; random forest; test automation.

## Introduction

**Problem statement.** In today's digital transformation environment, cloud computing models play a key role in ensuring the availability, scalability, and efficiency of software services. One of the most common cloud paradigms is Software as a Service (SaaS), which provides users with ready-to-use software applications via the Internet.

Unlike traditional approaches to software deployment, the SaaS model eliminates the need to install programs on local devices, as the provider is fully responsible for the development, deployment, updating, and support of the application. Users get access to the system's functionality according to the terms of the Service Level Agreement (SLA), which can be regulated by model subscriptions, hourly or volume-based payments. Today, SaaS solutions are widely used in email, financial services, human resource management, and other industries.

The growing popularity of SaaS is leading to an exponential increase in the number of users and the expansion of the functionality of such systems. However, the intensive use of cloud applications is inevitably accompanied by the emergence of a significant number of software defects that affect the quality and stability of services. Bugs in SaaS environments can cause delays in business processes, reduced productivity, a poor user experience, and direct financial losses. In these conditions, the effectiveness of technical support depends on the speed and accuracy of bug detection, classification, and prioritization. Manual processing of bug reports is complex, time-consuming, and resource-intensive, which is why automating bug classification processes is particularly important.

Machine learning methods capable of analyzing large data sets and identifying hidden patterns in text descriptions of bugs are also of considerable scientific interest. The use of machine learning algorithms in the field of bug report processing opens up new opportunities to improve the accuracy of error type identification, reduce the time required to fix them, and reduce the workload on the development team and system administrators. With the rapid growth in the number of cloud applications, the need for such approaches is becoming critically important, as the correct classification and prioritization of defects directly affect the stability and reliability of services.

Automated classification of software bugs in SaaS systems using machine learning methods is a promising area of research aimed at improving the efficiency of technical support, the accuracy of defect identification, and the optimization of quality assurance processes. Research in this area provides a scientific basis for the development of intelligent tools for analysis, forecasting, and decision support in cloud infrastructures.

**Analysis of recent research and publications.** In today's environment of increasing software system complexity, software testing plays a key role in the verification and validation (V&V) process, ensuring the correctness, stability, and long-term reliability of the systems being developed [4]. With the increasing criticality of software systems, particularly in the fields of security, medicine, transport, etc., the costs associated with their analysis, testing, and quality assurance are also rising significantly. This, in turn, creates demand for effective methods of defect prediction using intelligent technologies, in particular machine learning (ML) algorithms, which enable prediction, optimization, and automatic learning with minimal human intervention.

In the context of research aimed at automating the detection of bugs in SaaS applications, particular attention is drawn to works devoted to the classification of bugs based on bug reports and the determination of their priority. Researchers point to the availability of both ready-made tools for code analysis and bug detection, as well as models capable of prioritizing these bugs based on historical data [5]. In [6], the idea of using machine learning to automate manual processes, particularly in the area of bug prioritization, is put forward. The authors note that based on historical bug reports, models can learn to identify patterns and make predictions about the importance of new bugs. This approach improves classification accuracy and reduces the burden on technical support.

A similar approach is supported in [7], which emphasizes that the increasing complexity of software systems significantly complicates manual bug detection, making it slow and prone to human bug. The use of ML models in such conditions not only improves the quality of software but also reduces the cost of its maintenance. This is especially true for secure or mission-critical systems, where even minor bugs can have serious consequences.

Another promising area of research is the use of ensemble methods in bug triaging – the process of automatically determining which developer should be assigned a particular bug. Work [8] demonstrates that ensemble classifiers (which combine several models to achieve better results) outperform classical machine learning algorithms in bug assignment tasks. This indicates the possibility of significantly improving the efficiency of the bug handling process and reducing delays in their resolution.

In [9], an innovative approach to bug prioritization based on emotional analysis of bug descriptions was proposed. The authors collected data from open sources, performed natural language processing (NLP), extracted emotional words from the text, and based on this, formed a feature vector for the ML model. This approach increased classification accuracy (F1 score) by more than 6 %. The use of emotional analysis allows for better consideration of subjective user assessments, which is especially important in interface-oriented or client systems.

A significant problem in bug classification tasks is class imbalance, where most examples belong to insignificant classes, and critical bugs occur much less frequently. In such cases, most models tend to overfit on frequent classes, which reduces the effectiveness of detecting truly important bugs. The paper [10] considers an approach that involves building an ensemble classifier using various oversampling methods to improve the representation of small classes. The results of the study showed that combining classification and sample balancing reduces the number of false negatives and improves the accuracy of defect component recognition.

Our study proposes an approach to automating the processes of classification and prioritization of software bugs in SaaS applications. With the growth in the number of users and the increase in the functional load on cloud services, the probability of defects occurring is steadily increasing. These bugs can significantly degrade the quality of the user experience, cause delays in business processes, and create additional difficulties in maintaining such systems. Therefore, an urgent task is to develop machine learning models capable of automatically processing bug logs, identifying defect types, and prioritizing them to optimize the work of development and technical support teams.

**The goal of this work** is to improve the efficiency of classifying types of software bugs in cloud computing environments based on a hybrid approach to software bug classification using NLP, vectorization, and balanced machine learning methods.

## Main material

The proposed design solution for classifying bugs in cloud computing applications uses machine learning methods to automate and improve the detection and prioritization of software defects. This approach takes into account the inherent complexity and scaling challenges in cloud environments, where bugs can manifest in distributed systems, virtualized resources, and heterogeneous infrastructures. The methodology follows a structured pipeline covering data collection, preprocessing, feature engineering, model selection and training, evaluation, and deployment. Although presented as a high-level framework, the solution is adaptable to specific contextual constraints, as shown in Figure 1.

The initial phase involves collecting data from a variety of sources relevant to cloud computing applications. Bug data is aggregated from bug tracking systems (e.g., Jira or Bugzilla), official repositories, user forums, and historical logs. This multifaceted approach to sources provides a comprehensive data set that reflects real-world bug manifestations, including those arising from resource contention, network latency, or configuration bugs in cloud-native architectures.

After collection, the data is preprocessed to remove noise and inconsistencies. Unnecessary artifacts, such as duplicate records or discussions unrelated to bugs, are removed. Missing values are imputed using methods such as replacement by the mean or k-nearest neighbors, while text data is normalized, tokenized, and stop words are removed. This step transforms the raw input data into a structured format suitable for machine learning analysis.

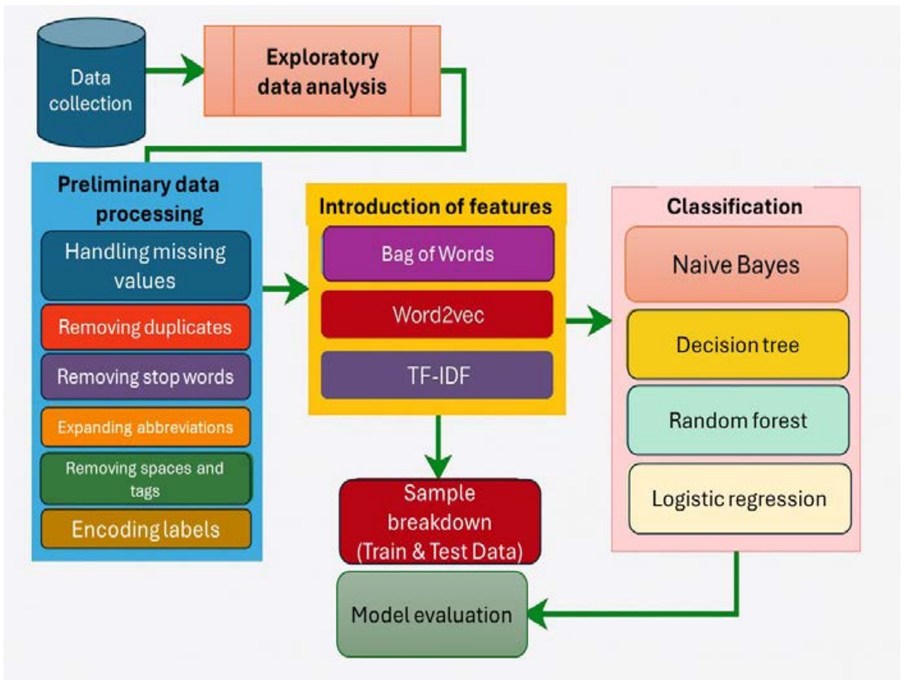


**Fig. 1.** Algorithm of the proposed bug classification conveyor in cloud computing applications

After cleaning and transforming the data, a vector representation is formed that is suitable for further processing by machine learning algorithms. This study considered three popular vectorization methods: Bag of Words (BoW), TF-IDF, and Word2Vec.

1. *Bag of Words (BoW).* The Bag of Words (BoW) method is a basic statistical method that represents text as a vector of word frequencies. Let us have a corpus of documents containing a dictionary $V = \{w_1, w_2, ..., w_n\}$. Each document is represented as a vector:

$$\vec{v}_d = \left[ f(w_1, d), f(w_2, d), \dots, f(w_n, d) \right],$$

where $f(w_i, d)$ – frequency of a word $w_i$ in a document $d$. The method does not take into account word order and semantic relationships, but it is effective with a sufficient amount of data.

2. *TF-IDF (Term Frequency – Inverse Document Frequency).* The TF-IDF method improves BoW by weighting word frequency based on how unique the term is within the entire corpus. For a word $t$ in a document $d$, the TF-IDF formula is calculated as follows:

$$\text{TF-IDF}(t,d,D) = \text{TF}(t,d)\cdot \text{IDF}(t,D),$$

where $\text{TF}(t,d) = \log\left(1+ f(t,d)\right)$,

$$\text{IDF}(t,D) = \log\left(\frac{N}{|\{d \in D : t \in d\}\ |}\right),$$

$f(t,d)$ – number of occurrences of the term $t$ in the document $d$,

$N$ – total number of documents in the corpus $D$,

$|\{d \in D : t \in d\}\ |$ – number of documents containing the term $t$.

TF-IDF allows you to reduce the weight of common terms and increase the significance of rare, specific words.

3. *Word2Vec.* Unlike statistical methods, Word2Vec is a deep learning model that creates dense vector representations of words, taking into account the context of their use. Developed by Google in 2013, the model has two main architectures: Continuous Bag of Words (CBOW) and Skip-Gram.

Let $w_t$ is the trarget word, $C = \{w_{t-n},\dots,w_{t-1},w_{t+1},\dots,w_{t+n}\}$ — the context. In CBOW, the task is to predict $w_t$ based on the context:

$$P(w_t C) = \frac{e^{v_{w_t}\cdot h}}{\sum_{i=1}^{|V|} e^{v_i \cdot h}},$$

where $h$ – average vector representation of words from context; $v_i$ – vector representation of a word $i$.

In Skip-gram, on the contrary, the model learns to predict contextual words based on a given word.

In our study, we used the CBOW algorithm with the following parameters:

– min_count = 5 — words that occur less than 5 times are ignored;

– size = 50 — the dimension of the vector space;

– workers = 4 — the number of threads for training.

As part of the study, four classic machine learning algorithms were used to solve the problem of multi-class classification of error types in cloud computing applications: naive Bayes classifier, decision tree, random forest, and logistic regression.

Each of these methods has its own advantages, limitations, and peculiarities of use in natural language processing tasks.

1. *Naive Bayes classifier.* This method is based on Bayes' theorem with the assumption of conditional independence of features. In text classification tasks, it is considered simple, fast, and effective.

Its probability model is determined by the formula:

$$P\left(yx_1,x_2,\ldots,x_n\right)=\frac{P\left(y\right)\prod_{i=1}^{n}P\left(x_i y\right)}{P\left(x_1,x_2,\ldots,x_n\right)},$$

where $y$ – class, $x_i$ – signs (e.g., words), and $P\left(x_i \mid y\right)$ – probability of a sign $x_i$ appearing given the class $y$.

2. *Decision Tree.* Decision trees are algorithms that build a hierarchical model where each internal node branch corresponds to a condition based on a specific feature, and leaf nodes correspond to classes. The main goal is to minimize entropy or the Gini coefficient during partitioning. Formally, entropy is used for:

$$H\left(D\right)=-\sum_{i=1}^{n}p_i\log_2 p_i,$$

where $p_i$ – proportion of class $i$ elements of $D$ data sets.

Trees are interpretive but prone to overfitting on noisy data.

3. *Random Forest.* Random Forest is an ensemble method that combines a set of decision trees created on random subsets of data and features. Each tree votes for a class, and the final prediction is determined by majority vote. The method reduces model variance, improving generalization:

$$\hat{y}=\text{mode}\left\{h_1\left(x\right),h_2\left(x\right),\ldots,h_k\left(x\right)\right\},$$

where $h_i\left(x\right)$ – forecast of the $i$-th tree.

Random forest demonstrates high accuracy, especially on complex and large-scale data, making it effective for text classification.

4. *Logistic Regression.* This linear method is widely used for classification tasks due to its mathematical rigor and stability. Softmax regression is typically used for multi-class classification. The probability of belonging to a class $k$ is determined by:

$$P\left(y=kx\right)=\frac{e^{\beta_k^T x}}{\sum_{j=1}^{K}e^{\beta_j^T x}},$$

where $\beta_k$ – vector of coefficients for class $k$, $x$ – feature vector.

Optimization is performed by maximizing the logarithmic likelihood function.

After the model is defined, it is trained on the prepared data set. The training process includes sample balancing to avoid bias towards the dominant class, cross-validation to evaluate the model's generalization ability, and selection of optimal hyperparameters. The result is a classification model that can automatically recognize bug categories based on the textual and structural features of the bug report.

During the validation stage, the model is evaluated using a separate test dataset. Its effectiveness is assessed using standard classification quality metrics such as accuracy, precision, recall, and F1-score. Each of these metrics allows us to evaluate different aspects of the model's performance: its ability to correctly classify objects, avoid false positives and false negatives, and the overall balance between precision and recall.

After passing the evaluation stage, the model is deployed in a cloud environment.

Its integration into real systems allows you to automate the process of processing new bug reports, classify them in real time, and route them to the responsible executors. This, in turn, helps to reduce response time, increase the efficiency of technical support, and generally optimize the software maintenance process.

High-quality and meaningful data is the foundation of modern data science, as the effectiveness and accuracy of a machine learning model directly depend on the quality of the source data set. That is why the first stage of implementing the proposed approach was data processing, which included collection, cleaning, visualization, and exploratory data analysis (EDA).

Figure 2 shows a deployment diagram that reflects the architecture of the bug classification system in the SaaS cloud environment.

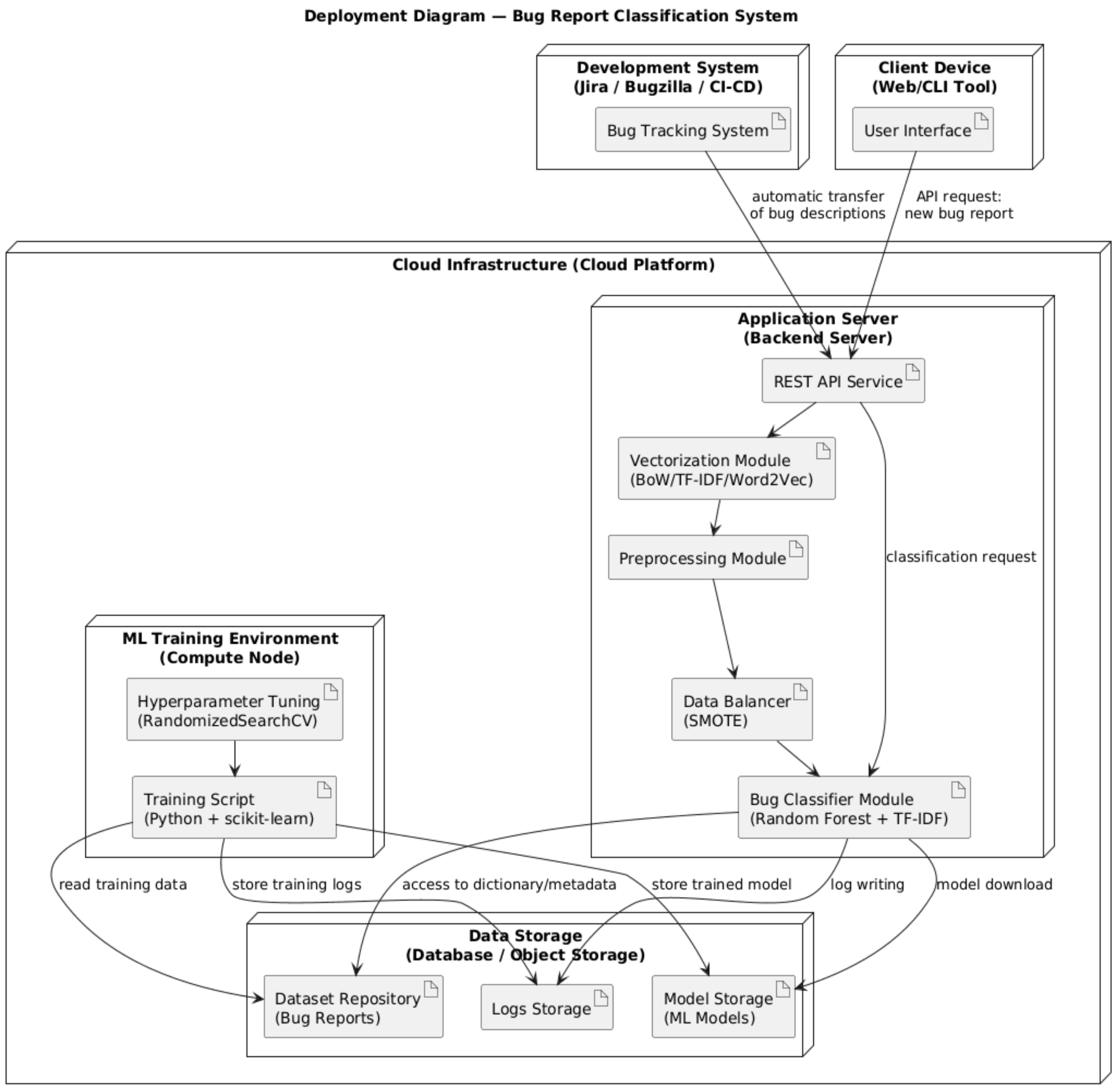

Deployment Diagram — Bug Report Classification System
Development System (Jira / Bugzilla / CI-CD)
Bug Tracking System
Client Device (Web/CLI Tool)
User Interface
automatic transfer of bug descriptions
API request: new bug report
Cloud Infrastructure (Cloud Platform)
Application Server (Backend Server)
REST API Service
Vectorization Module (BoW/TF-IDF/Word2Vec)
Preprocessing Module
classification request
Data Balancer (SMOTE)
Bug Classifier Module (Random Forest + TF-IDF)
ML Training Environment (Compute Node)
Hyperparameter Tuning (RandomizedSearchCV)
Training Script (Python + scikit-learn)
read training data
store training logs
access to dictionary/metadata
store trained model
log writing
model download
Data Storage (Database / Object Storage)
Dataset Repository (Bug Reports)
Logs Storage
Model Storage (ML Models)

**Fig. 2.** Algorithm of the proposed bug classification conveyor in cloud computing applications

The architecture of the proposed automated software bug classification system in cloud SaaS applications consists of a number of interconnected components located in the cloud infrastructure and designed to ensure a complete bug report processing cycle – from the moment the data is received to the formation of a classification conclusion. Each component performs clearly defined functions, ensuring scalability, modularity, and the ability to flexibly integrate with existing DevOps services and support systems.

1. *Cloud Platform*. The main environment in which all server modules of the classification system are deployed. Provides scalability, network interaction, and computing resources.

2. *Backend Server*. Hosts software components that process requests, classify bugs, and coordinate interaction between other modules:

– REST API Service – an interface for interaction between users, external services, and the classification system. Accepts bug reports and returns classification results.

– Bug Classifier Module – the main classification module, which runs a trained machine learning model (Random Forest + TF-IDF).

– Preprocessing Module – a component for cleaning and normalizing the text of bug reports before vectorization.

– Vectorization Module – implements Bag-of-Words, TF-IDF, and Word2Vec methods to convert text into numerical vectors.

– Data Balancer (SMOTE) – used during model training to eliminate class imbalance.

3. *Data storage (Database / Object Storage)*. Used to store permanent data and models:

– Dataset Repository – storage of bug reports, dictionaries, and metadata.

– Logs Storage – storage of system logs, classification history, and technical events.

– Model Storage – file or object storage of a trained ML model available to the classification module.

4. *ML Training Environment (Compute Node)*. A separate powerful computing environment designed for training models:

– Training Script (Python + scikit-learn) – scripts for training classifiers.

– Hyperparameter Tuning (RandomizedSearchCV) – a module for optimizing hyperparameters to achieve the best accuracy.

5. *Development System (Jira / Bugzilla / CI/CD)*. An external data source that automatically transfers bug reports to the classification system or receives analysis results.

– Bug Tracking System – a bug management tool that integrates with the system's REST API.

6. *Client device (Web/CLI Tool)*. A component through which the user interacts with the system:

– User Interface – a web interface or command interface that allows you to send bug reports and view classification results.

This study uses a publicly available dataset published on the Kaggle platform in 2020 [11]. This is because most similar data is either closed or extremely labor-intensive to collect independently.

The dataset is an example of a web-based issue tracker, specifically in the field of Python development.

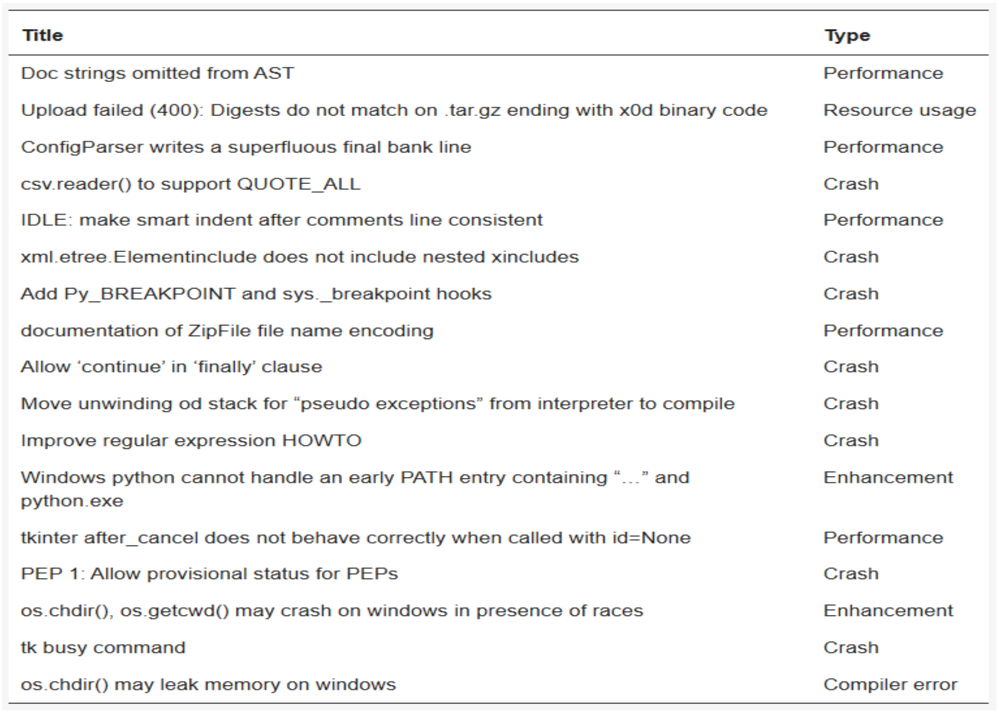
Its structure is presented in Table 1, which contains a description of the attributes used for further construction of bug classification models.

**Table 1.** *Data set characteristics*

| № | Attribute | Description |
|---|---|---|
| 1 | Unnamed | The column contains a unique identifier for each record |
| 2 | Title | The column contains the full text of the bug in the form of a record |
| 3 | Type | Target variable (label); indicates the type of bug |

The dataset contains 5,300 records and three main attributes: a unique identifier (Unnamed), a text description of the bug (title), and a target variable – the error type (type). A total of six bug categories have been identified: enhancement, security, compilation bug, resource utilization, performance, and crash.

An example description is shown in Figure 3. A distinctive feature of this data is that bug names often contain technical codes (e.g., SyntaxError, ImportError), which significantly complicates the classification task, since the language is not natural in the usual sense.



| Title | Type |
|---|---|
| Doc strings omitted from AST | Performance |
| Upload failed (400): Digests do not match on .tar.gz ending with x0d binary code | Resource usage |
| ConfigParser writes a superfluous final bank line | Performance |
| csv.reader() to support QUOTE_ALL | Crash |
| IDLE: make smart indent after comments line consistent | Performance |
| xml.etree.Elementinclude does not include nested xincludes | Crash |
| Add Py_BREAKPOINT and sys._breakpoint hooks | Crash |
| documentation of ZipFile file name encoding | Performance |
| Allow 'continue' in 'finally' clause | Crash |
| Move unwinding od stack for "pseudo exceptions" from interpreter to compile | Crash |
| Improve regular expression HOWTO | Crash |
| Windows python cannot handle an early PATH entry containing "…" and python.exe | Enhancement |
| tkinter after_cancel does not behave correctly when called with id=None | Performance |
| PEP 1: Allow provisional status for PEPs | Crash |
| os.chdir(), os.getcwd() may crash on windows in presence of races | Enhancement |
| tk busy command | Crash |
| os.chdir() may leak memory on windows | Compiler error |

**Fig. 3.** Example of bug description

The distribution of error types is analyzed using graphical visualization. Figure 4 shows a histogram demonstrating the number of records for each error type. The most common bugs are those related to performance, while the least common are those related to resource utilization.

Since this is a multi-class classification, the issue of class balancing is not critical. Instead, it is advisable to use a cross-validation method, which avoids overfitting the model.

This approach involves dividing the entire dataset into several parts (folds) and testing the model step by step on each subset, which significantly increases the accuracy and stability of the results.

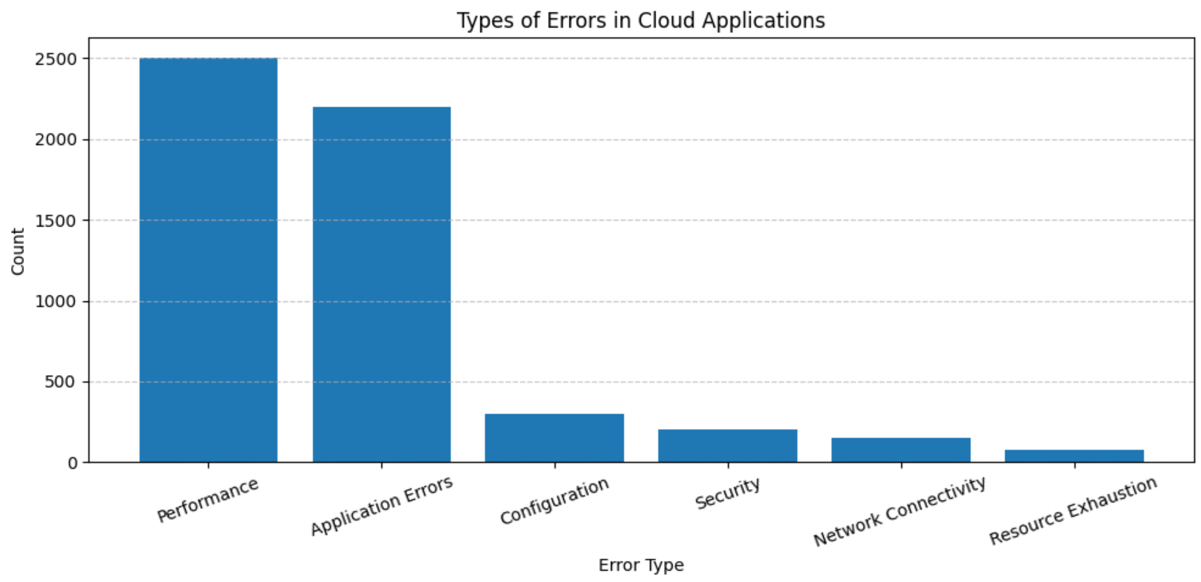


**Fig. 4.** Distribution of bugs by type

The next step was to analyze the frequency of terms in bug descriptions. One of the simplest but most informative approaches is to use unigrams – single words that are considered independently of each other. Figure 5 shows a graph with the ten most frequently used words, among which the word module occurs most often, while python occurs least often. Most of them are typical for bugs in Python repositories (e.g., file, function, code).

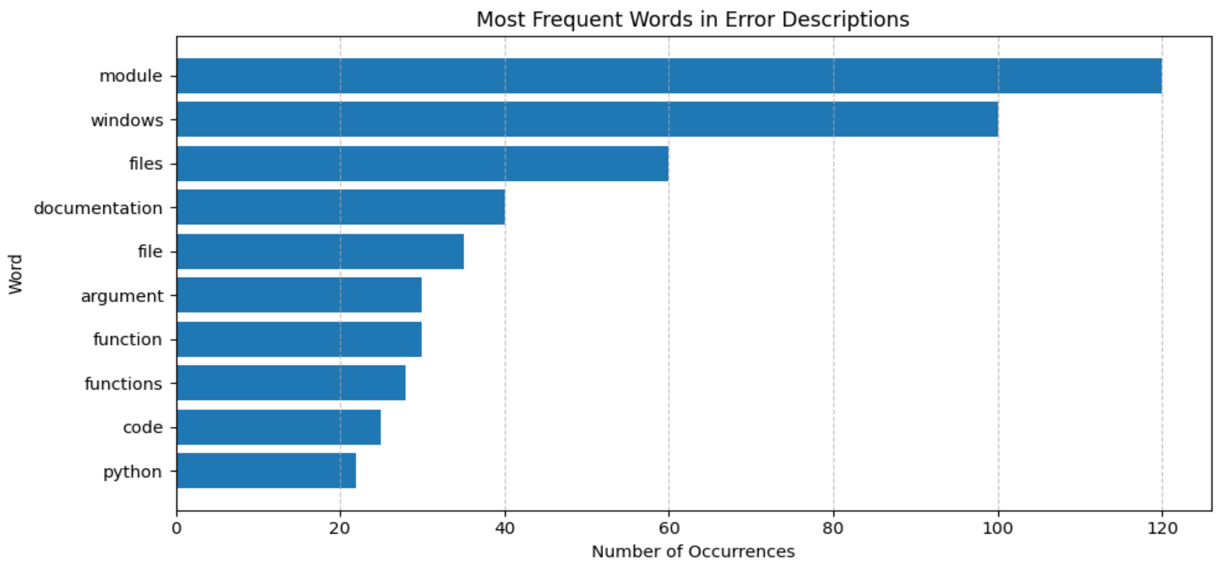


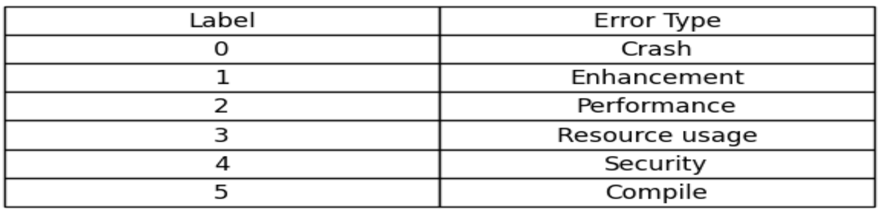
**Fig. 5.** Most frequently used words in bug descriptions

Preprocessing is a key step in the process of extracting knowledge from data, as it allows raw, unstructured, or partially structured information to be converted into a machine-readable format. Real-world datasets are typically characterized by incompleteness, redundancy, instability, and errors. Therefore, the application of high-quality preprocessing procedures is a prerequisite for building a reliable and generalizable machine learning model.

Within the scope of this study, preprocessing was implemented in several stages, which can be broadly divided into two main phases: working with raw data and basic preprocessing with data labeling.

After completing the cleaning stages, the data was ready to be transformed into a format suitable for modeling. In particular, the type column, which is a categorical variable, needed to be converted to a numerical format.

To do this, we used the Label Encoding method, which replaces each unique category with a corresponding number. As a result, all six error classes received unique numerical labels. The scheme of encoded values is shown in Figure 6.



| Label | Error Type |
|---|---|
| 0 | Crash |
| 1 | Enhancement |
| 2 | Performance |
| 3 | Resource usage |
| 4 | Security |
| 5 | Compile |

**Fig. 6.** Label encoding for multi-class classification

An equally important challenge in classification tasks is the problem of class imbalance, when the number of records for different categories is uneven. Although the dataset under study is multi-class, there is also a significant imbalance in the number of examples for each type of error.

To solve this problem, a combination of random oversampling and the SMOTE (Synthetic Minority Oversampling Technique) method was used. SMOTE is an algorithm for generating synthetic examples for minority classes by interpolating between existing points in the feature space. This approach not only balances the distribution of classes, but also reduces the likelihood of overfitting, which often occurs when simply duplicating minority examples.

As a result of balancing, the distribution of data across classes was evened out. This is clearly demonstrated in Figure 7, which shows the final state of the dataset with evenly represented classes.

Within the scope of this study, four popular machine learning algorithms were selected to solve the problem of multi-class classification of error types in cloud SaaS applications:

– Naive Bayes classifier;

– Decision Tree;

– Random Forest;

– logistic regression.

In order to improve the efficiency of modeling and conduct a more in-depth analysis of the impact of different approaches to text feature representation, a series of experiments were conducted, which included the use of various vectorization methods, parametric optimization, and comparative evaluation of models.

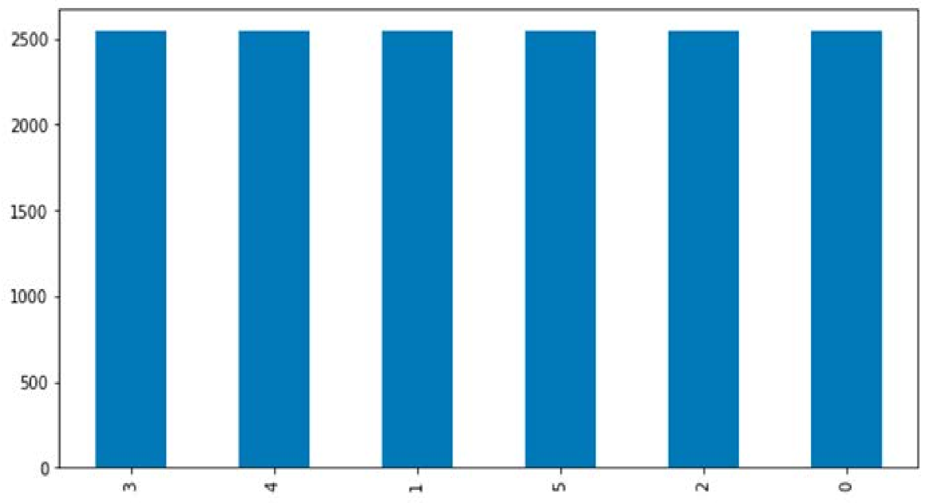


**Fig. 7.** Final state of the dataset with evenly represented classes

Seven key experiments were implemented, covering the following components:

– Three methods of text vectorization: Bag-of-Words (BoW), TF-IDF, and Word2Vec – a modern method of vector representation of words based on a neural network.

– Optimization of model hyperparameters for BoW and TF-IDF using the Randomized SearchCV method, which allows you to efficiently find the best parameter configurations.

– Comprehensive comparative analysis of model performance by metrics: accuracy, recall, precision, and weighted average (F1-score).

The formulas for calculating key metrics are presented below:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN};$$

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

$$Recall = \frac{TP}{TP + FN};$$

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall},$$

where TP, TN, FP, FN are true positive, true negative, false positive, and false negative predictions, respectively.

The dataset was divided into training and test samples in an 80/20 ratio, where 80 % of the records were used to train the model and 20 % to evaluate its generalization ability.

To ensure the reproducibility of the experiments, a fixed parameter random_state = 42 was used, which guarantees the same division of the dataset each time it is run.

Each of the four classification algorithms was trained based on three different types of vectorization, which made it possible to evaluate how the method of text representation affects the performance of the model.

The results of the study confirm that dataset balancing combined with model hyperparameter optimization are critical factors for achieving high classification accuracy. Four machine learning algorithms were used in the experiments: naive Bayes classifier, decision tree, random forest, and logistic regression – to classify bugs on both balanced and unbalanced datasets.

One of the key classifiers used in the study is Multinomial Naive Bayes, which is widely used for text classification. It is based on Bayes' theorem, assuming conditional independence of features. Since this is a multi-class task, the MultinomialNB implementation was obtained from the scikit-learn library and used in all experimental scenarios.

All experimental results for the Naive Bayes model are shown in Figures 8 and 9.

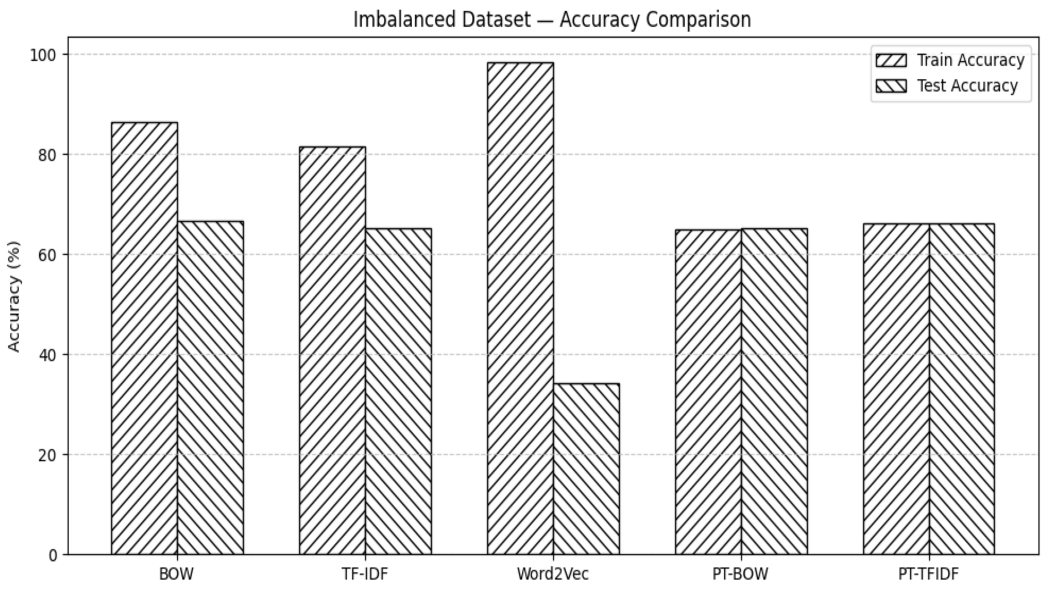


**Fig. 8.** Comparison of model accuracy for an imbalanced sample

The Naive Bayes model demonstrates different effectiveness depending on the vectorization method and the state of the sample (balanced or unbalanced). The best result was obtained for TF-IDF with tuned hyperparameters (PT-TFIDF).

The model achieved 97.14 % accuracy on the training set and 88.18 % on the test set, which is the highest result among all configurations.

Word2Vec vectorization showed low performance on the test data, indicating insufficient representativeness of semantic spatial features in this task.

Balancing the dataset significantly improved the classifier's performance: a comparison of experiments 1 and 2 shows a 15–20 % jump in performance.

Thus, hyperparameter tuning and correct sample preparation are critical factors for achieving high accuracy in bug report classification tasks.

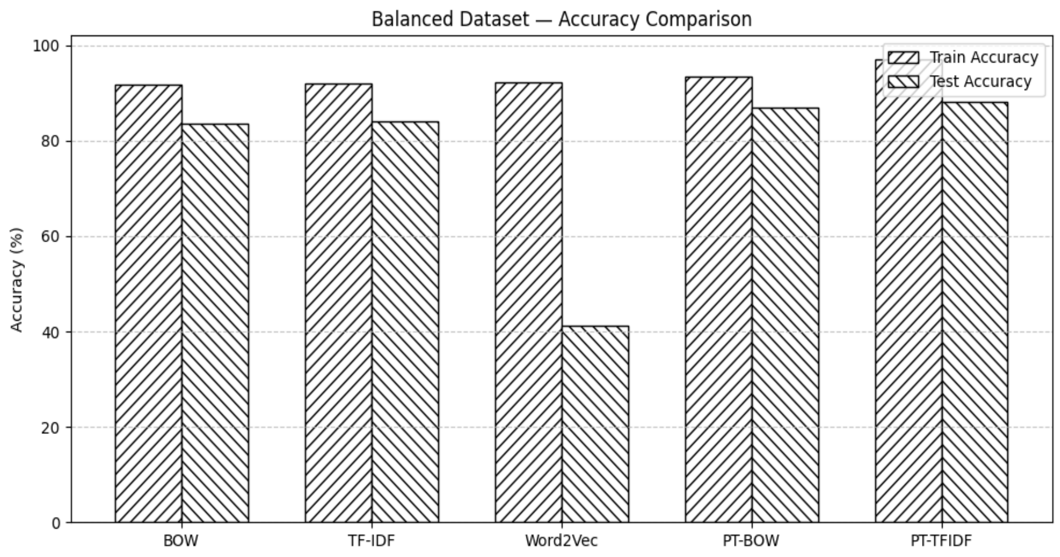


**Fig. 9.** Comparison of model accuracy for a balanced sample

The study also generated a Classification Report for the Naive Bayes model with the best parameters (BOW-tuned). Based on the Classification Report, a diagram was constructed (Fig. 10), which details the Precision, Recall, and F1-measure values for each error category.

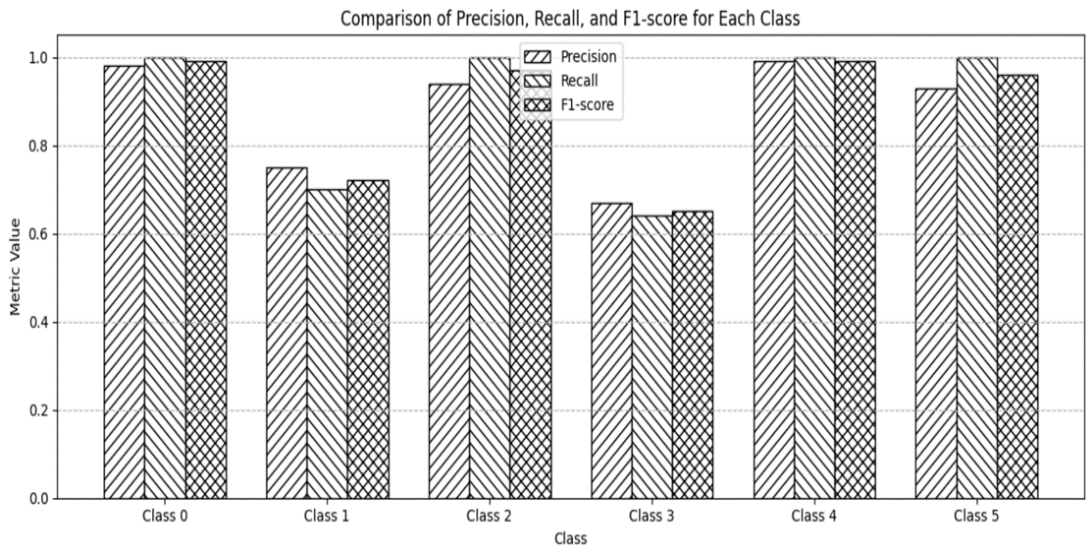


**Fig. 10.** Comparison of metrics for each class

Decision trees belong to the class of interpreted machine learning models and are widely used for prediction and classification tasks due to their simplicity, high learning speed, and ability to work with nonlinear dependencies. The algorithm for constructing a decision tree is based on iterative division of the feature space by selecting splitting criteria that minimize uncertainty (impurity) in data subsets. This approach allows building a hierarchical structure of rules, according to which the model matches new inputs with the corresponding classes.

In this study, the DecisionTreeClassifier algorithm from the scikit-learn library was used to classify software errors. After loading the data, preprocessing, and vectorization, the model was trained on the training set and tested on the deferred part of the data. To ensure a correct quality assessment, all experimental scenarios similar to the previous Naive Bayes analysis were also applied to this model.

The generalized results are presented in Figures 11, 12.

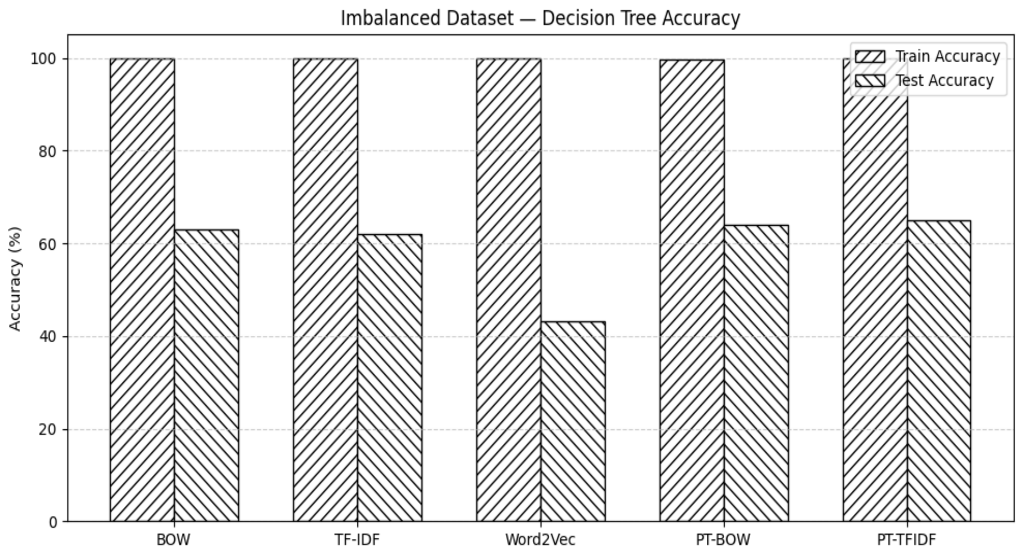


**Fig. 11.** Comparison of model accuracy for imbalanced samples

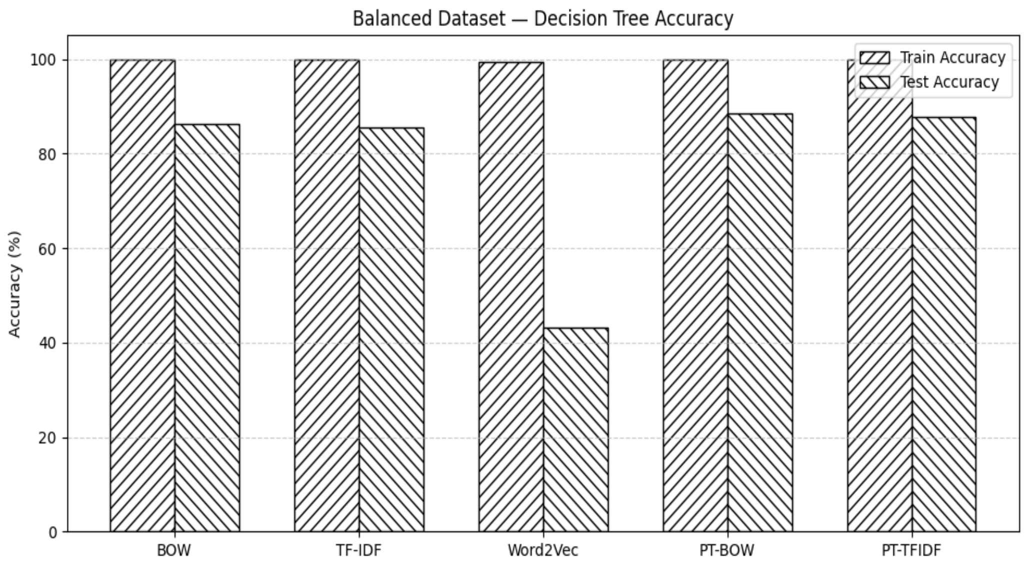


**Fig. 12.** Comparison of model accuracy for balanced samples

The Decision Tree model achieves maximum accuracy on the training set in almost all experiments, which is typical for decision trees, as they are prone to overfitting.

This is especially noticeable when training on an unbalanced dataset, where accuracy scores on the test sample are significantly lower – from 43 % to 65 %, depending on the vectorization method.

After applying sample balancing, the model's performance improved significantly:

– test accuracy increased to 86–88 % in configurations with BOW and TF-IDF,

– results for Word2Vec remained low (≈43 %), which is consistent with previous observations and indicates the ineffectiveness of Word2Vec in this context.

The optimal hyperparameter values were selected for the model:

– criterion = "gini"

– max_depth = 54

– min_samples_leaf = 4

– min_samples_split = 95

This setting partially reduced model overfitting, although Decision Tree remains inherently sensitive to noise and data complexity.

The results showed that after optimizing the parameters, the accuracy of the BOW and TF-IDF-based models is practically identical. The slight advantage of TF-IDF (up to 87.79 % test accuracy) is due to the fact that this method better takes into account the weight of rare terms, which is critical in text tasks.

The classification report was used for a detailed analysis of the model's behavior (Fig. 13):

– the largest number of misclassifications occurs in classes with similar text features;

– the F1-measure for individual categories varies significantly, confirming the sensitivity of the decision tree to data distribution.

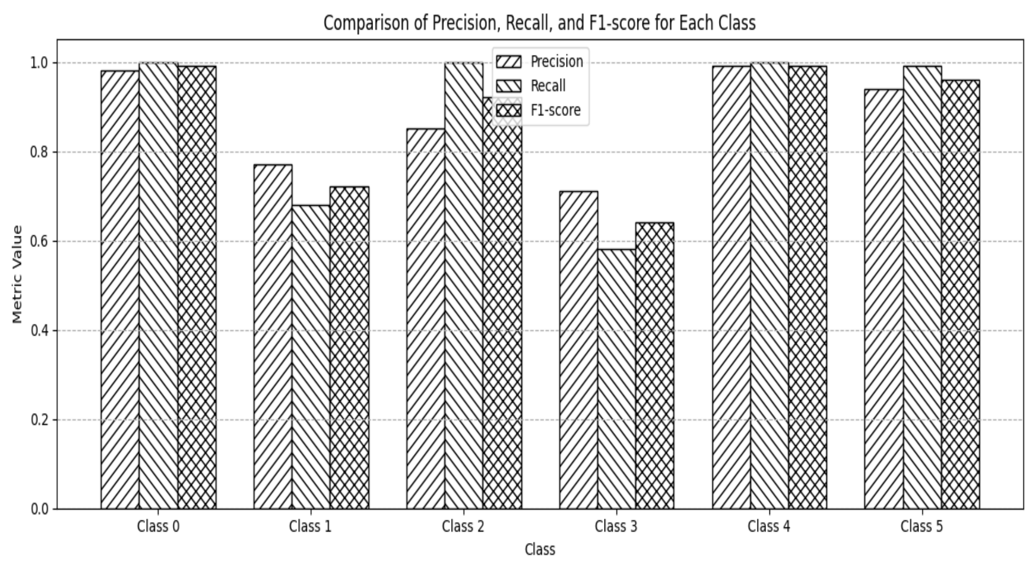


**Fig. 13.** Comparison of metrics for each class

Within the scope of this study, the RandomForestClassifier implementation from the scikit-learn library was used. After loading the vectorized data, the model was trained on the training sample and evaluated on the test sample.

All experimental scenarios – for different vectorization methods and with/without hyperparameter optimization – were tested sequentially. The generalized results are shown in Figures 14, 15.

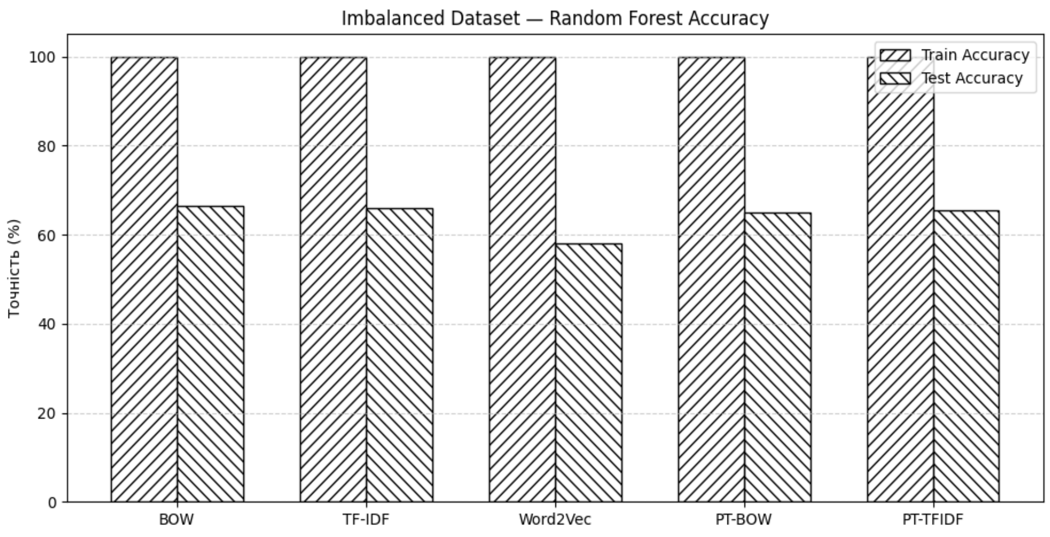


**Fig. 14.** Comparison of model accuracy for imbalanced samples

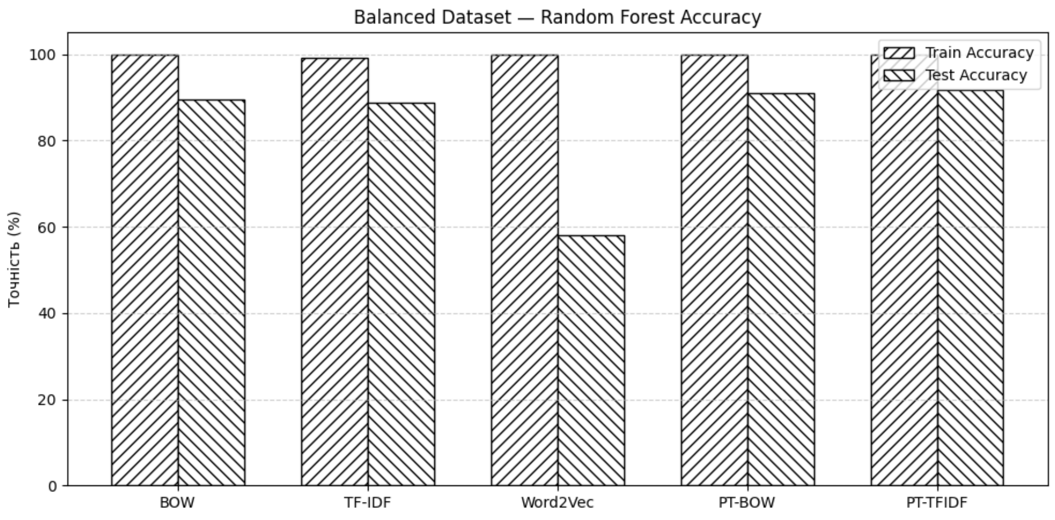


**Fig. 15.** Comparison of model accuracy for balanced samples

The results show that Random Forest demonstrates significantly better classification quality compared to decision trees and Naive Bayes in most settings. In particular, it can be seen that:

1. *The best result was achieved for TF-IDF with tuned hyperparameters (PT-TFIDF)*

– Train accuracy: 100 %;

– Test accuracy: 91.73 % – the highest score among all models in the study.

This confirms that Random Forest not only successfully overcomes the problem of decision tree overfitting, but also makes the most complete use of the information provided by TF-IDF vectorization.

2. *Impact of sample balancing*

A comparison of the results between part 1 (Imbalanced) and part 2 (Balance) shows:

– the accuracy gain on the test is between 20 and 30 % for BOW and TF-IDF;

– Word2Vec accuracy remains low (~58 % regardless of balance), indicating its limited effectiveness for classifying short text descriptions of errors in this dataset.

3. *Selected hyperparameters*

The following values were used to achieve optimal productivity:

– criterion = "entropy"

– max_depth = 79

– min_samples_leaf = 1

– min_samples_split = 79

The combination of a large tree depth and a split value of 79 ensured a balance between tree variability and overall ensemble consistency.

According to the classification reports obtained, the precision for all classes varies from 73 % to 100 % (Fig. 16), which indicates the model's high ability to correctly assign most samples to the appropriate categories.

The results for classes 4 and 5 are particularly indicative, where the model achieved 100 % recall, i.e., it was able to detect all cases that actually belong to these classes. Such indicators are considered excellent in the tasks of automated classification of text descriptions of errors.

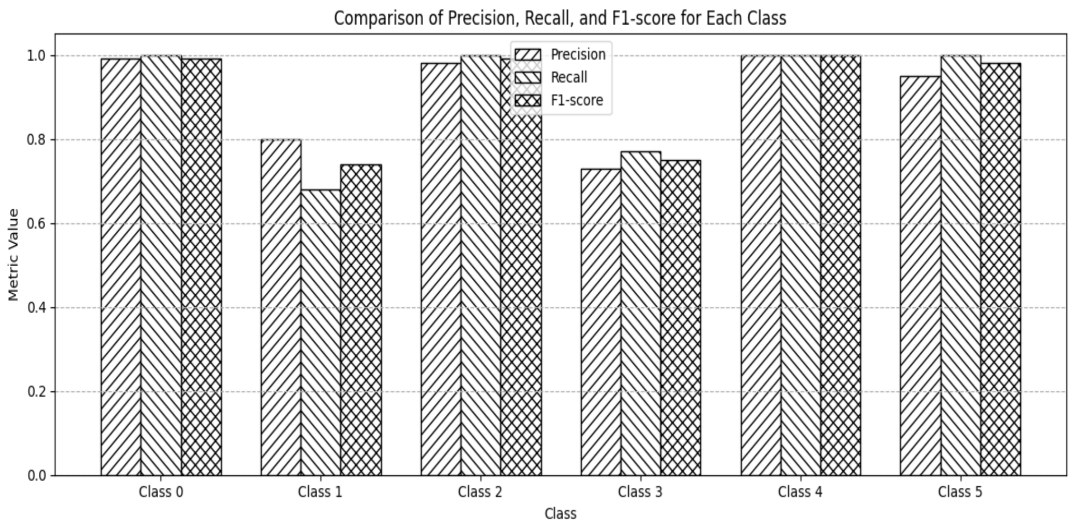


**Fig. 16.** Comparison of metrics for each class

Analysis of the F1-measure confirms that the generalized model is well balanced and shows no signs of overfitting or underfitting. F1-scores remain consistently high for most classes, indicating an effective combination of accuracy and completeness. It is important to note that all classes contributed approximately equally to the learning process, which ensured increased model stability and its ability to work on various types of data.

In this study, the LogisticRegression implementation was imported from the *sklearn.linear_model* library. The model was trained on pre-processed and vectorized data, after which it was evaluated on a test set. All experimental scenarios – different vectorizations, hyperparameter optimization, and balanced/unbalanced sample analysis – were applied to logistic regression in the same way as to other classifiers.

Further analysis of the results allowed us to evaluate how well logistic regression can cope with the task of classifying software bug descriptions and how its productivity compares to Naive Bayes, Decision Tree, and Random Forest.

Within the scope of the experiments, logistic regression was tested under similar conditions as other classifiers – with different vectorization methods (BOW, TF-IDF, Word2Vec), as well as with tuned hyperparameters for BOW and TF-IDF. The generalized results of the model are shown in Figures 17, 18.

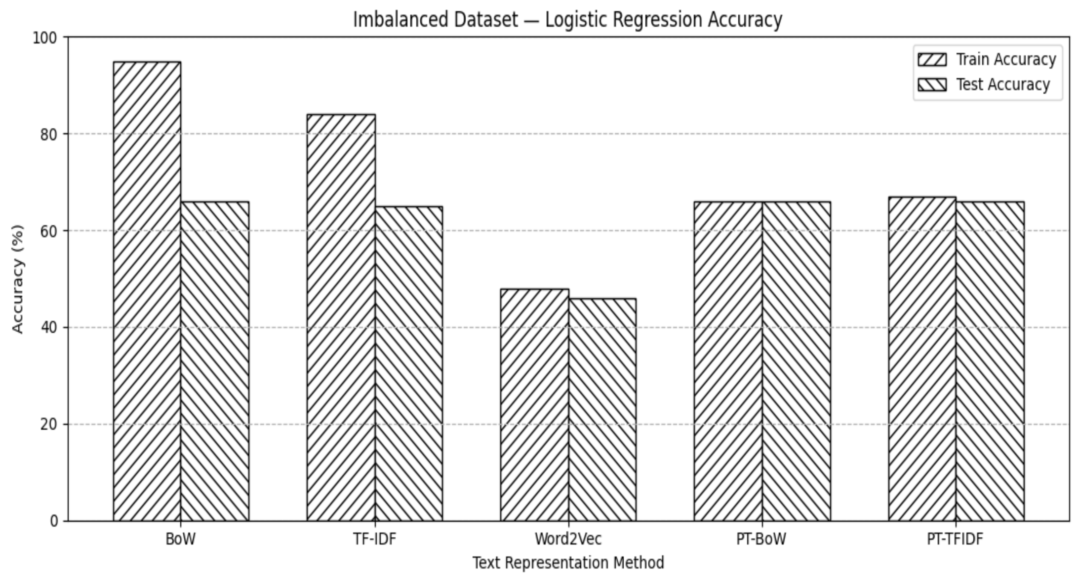


**Fig. 17.** Comparison of model accuracy for imbalanced samples

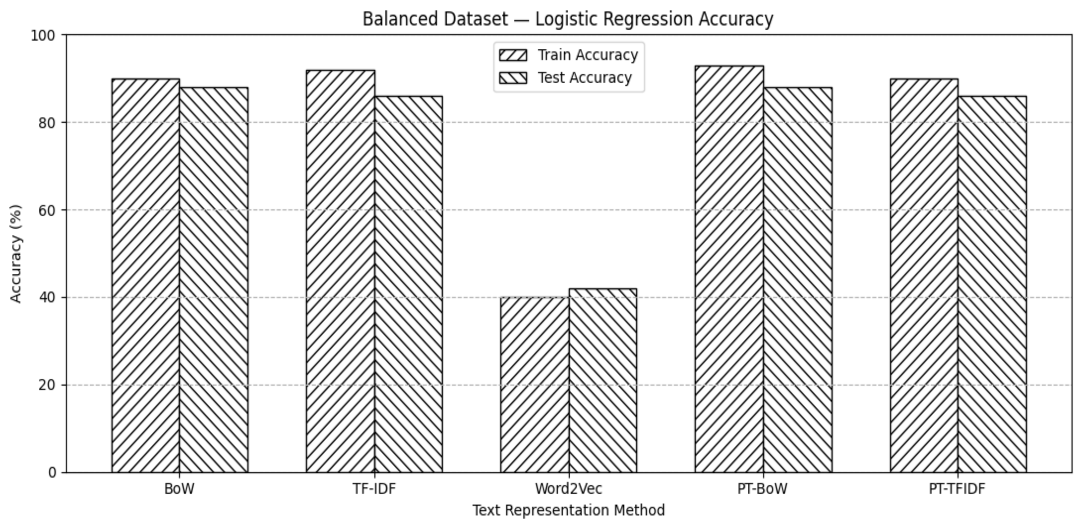


**Fig. 18.** Comparison of model accuracy for balanced samples

Logistic regression demonstrates lower productivity compared to other classifiers, in particular Random Forest and Naive Bayes. The main observations are as follows:

1. *Relatively low accuracy on an unbalanced dataset*

The model showed test accuracy in the range of 46–66 % for most vectorization methods, and even below 50 % for Word2Vec, indicating the difficulties of logistic regression in conditions of uneven class distribution and high variability of text descriptions.

2. *Improved results after data balancing*

For a balanced dataset, the productivity of logistic regression improved:

The best values were obtained for PT–BOW, where the test accuracy was 88.27 % and the training accuracy was 93.37 %.

This indicates that logistic regression is sensitive to class imbalance and can work much more effectively after preliminary sample correction.

3. *Low efficiency of Word2Vec*

Word2Vec showed the worst results among all vectorization methods: Train: 40.12 % – Test: 42.15 %

Since logistic regression is based on linear class separation, Word2Vec semantic vectors probably did not provide sufficient discriminatory information in this case.

4. *Hyperparameters used*

To improve model productivity, the following parameters were set:

– C = 10
– solver = "newton-cg"

The parameter C=10 reduces regularization, allowing the model to better adapt to the data, while the newton-cg optimization algorithm is effective for multi-class tasks.

The obtained metric values (Fig. 19) indicate that the logistic regression model demonstrates stable classification quality for most error categories.

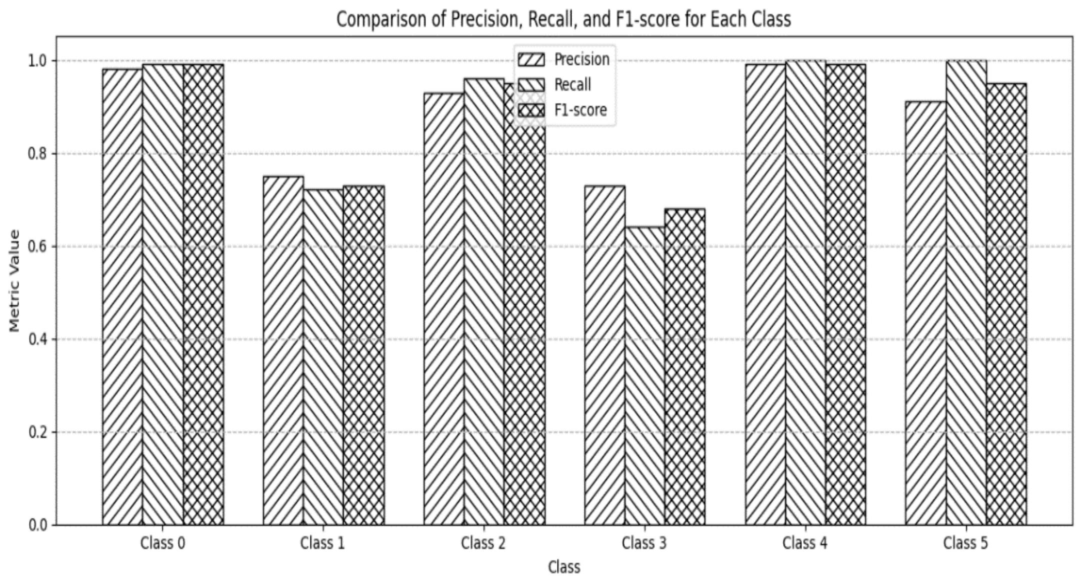


**Fig. 19.** Comparison of metrics for each class

In particular, precision ranges from 73 % to 99 %, which indicates the model's high ability to correctly recognize samples belonging to the corresponding classes. Classes 4 and 5 stand out in particular, for which the model achieved 100 % recall, i.e., it was able to detect all real instances of these categories without omissions. Such indicators are important for tasks where it is critical to minimize the loss of important or rare defects.

The F1-measure values confirm that the model does not suffer from overfitting or underfitting. The F1-score remains high and balanced for most classes, indicating a harmonious balance between accuracy and completeness. The absence of significant failures in any of the categories demonstrates that the model generalizes the data adequately and does not reorient itself to individual classes.

It is also important to note that all classes made a relatively equal contribution to the training of the model. This indicates that the training process was well balanced and that the preprocessing and sample balancing methods made it possible to avoid the dominance of certain categories. This result is critical for practical application, as it ensures stable forecasting in a variety of real-world bug report scenarios.

Overall, logistic regression, despite its relatively lower accuracy in some configurations, demonstrates satisfactory and interpretable results, making it useful as a base model in automated software bug classification systems.

For a generalized comparison of the results, a Train/Test graph (Fig. 20) of the accuracy of all models was constructed, which clearly demonstrates that Random Forest outperforms other approaches in key metrics (see Train/Test comparison graph).

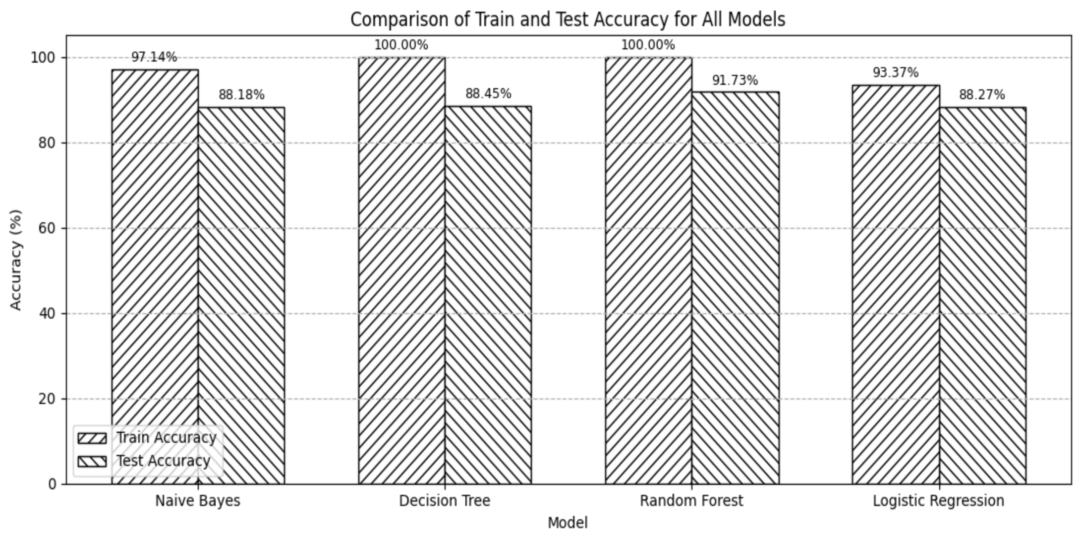


**Fig. 20.** Comparison of model accuracy on Train/Test data

Based on a comparison of all models, their accuracy, resistance to sample imbalance, ability to generalize complex textual features, and effectiveness after optimization, the best model in this study is Random Forest. It provides the highest test accuracy (91.73 %), demonstrates no overfitting, performs consistently on all experimental sets, and provides an optimal compromise between productivity, reliability, and the necessary flexibility. Random Forest is recommended as the primary model for building automated bug report classification systems in SaaS environments.

The productivity of an error classification system in cloud applications is a determining factor in its suitability for use in real-world workloads. Since the speed of processing bug reports directly affects the timeliness of incident response, the study evaluated two key characteristics: machine learning model training time and inference time, i.e., the time required to classify a single new bug report. The results are visualized in the corresponding graphs, allowing for a clear comparison of the effectiveness of different models.

The training time analysis showed significant differences between classifiers (Figure 21). The Naive Bayes model demonstrated the shortest training time (≈0.12 s), which is expected given its linear nature and lack of complex parameter optimization. Logistic Regression also demonstrated high performance, with a training time of about 0.95 s. Decision Tree, on the other hand, was more resource-intensive (≈1.8 s), which is associated with the need to build a deep hierarchy of nodes. The longest training time was observed for Random Forest (≈7.5 s), since the model consists of an ensemble of trees and requires significant computational resources to form an optimal set of decisions. Despite this, Random Forest showed the highest accuracy among all tested models (91.7 %), which justifies its use in the presence of the appropriate infrastructure.

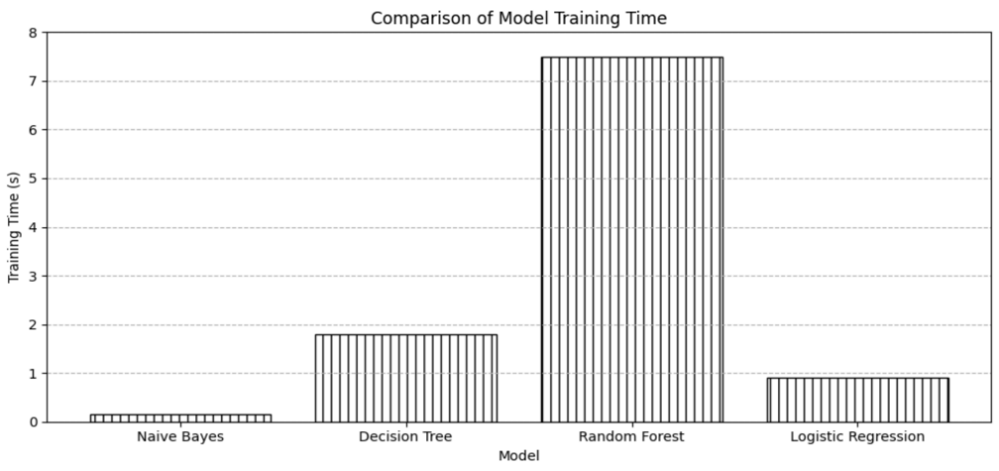


**Fig. 21.** Comparison of model training times

Inference time analysis (Figure 22) showed that all models provide almost instantaneous classification, which is an important condition for integration into real cloud services. The processing time for a single query was ≈0.002 s for Naive Bayes, ≈0.004 s for Logistic Regression, and ≈0.006 s for Decision Tree, while Random Forest showed a slightly slower inference time (≈0.015 s). However, even the maximum value remains within a few milliseconds, which allows you to classify hundreds or thousands of bug reports per second and maintain system operation in near real-time mode.

The system scalability assessment confirmed that the proposed architecture is effective for the cloud environment. Using TF-IDF as the main vectorization method ensures linear scaling of computational costs as data volumes increase and allows large text message streams to be processed without a significant increase in preprocessing time. In addition, the Random

Forest model naturally supports horizontal scaling, since tree construction can be parallelized across multiple computing nodes, which is especially important when processing large datasets or during regular model retraining.

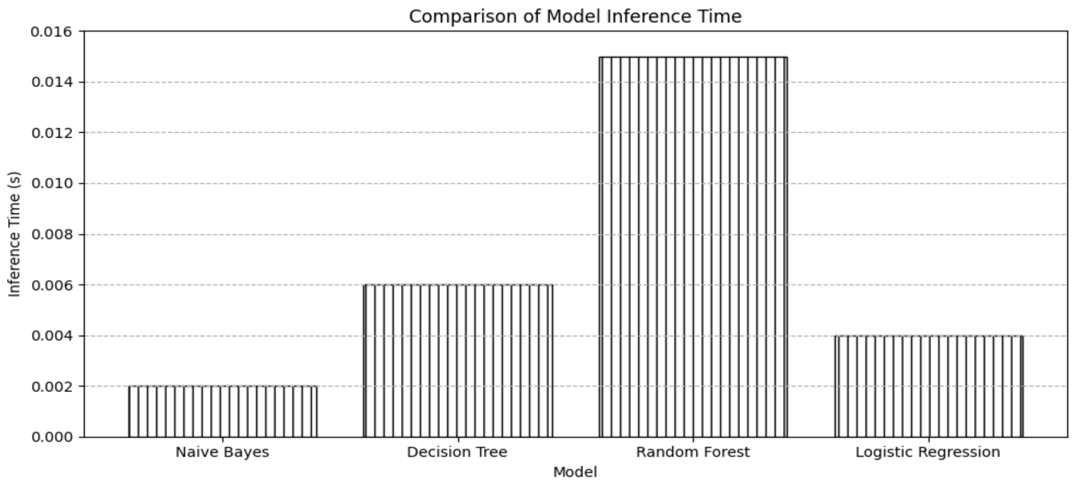


**Fig. 22.** Comparing of  models inference time

Separating the training and inference processes also provides a significant advantage for scaling. Model training is performed on a separate computing node, which allows the model to be updated without stopping the classification service, maintaining the continuity of the system. This is consistent with typical cloud service operating practices and ensures the system's resilience to changes in data volumes and the intensity of error reports.

This study prioritizes errors in SaaS-type cloud applications based on their frequency of occurrence in the available dataset. The results are visualized in Figure 23, which shows the distribution of errors by type, taking into account their relative share.

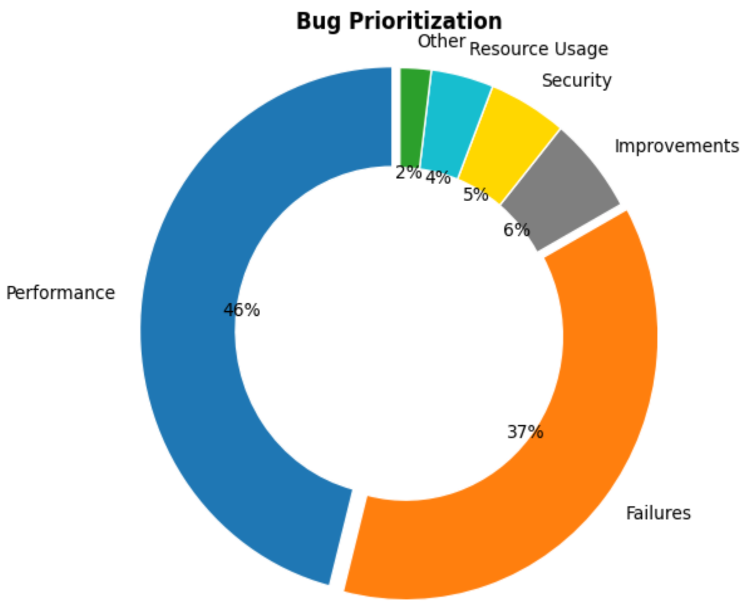


**Fig. 23.** Bug prioritization

The analysis of the graph shows that the most common bugs are related to system productivity, accounting for 46 % of the total. Such bugs are usually related to response delays, inefficient use of computing resources, or slow data processing. They have a significant impact on the quality of user interaction with the application and require immediate response from developers. In second place in terms of frequency are crash errors, which account for 37 %. This type of error is critical because it causes the application to suddenly stop working, which can lead to data loss and disruption of service continuity.

Other types of errors are less common but no less important in terms of ensuring system reliability. For example, system functionality improvements, security issues, compilation errors, and resource usage issues together account for about one-fifth of all cases. In particular, security errors (5 %) require special attention, as even single cases of such failures can have serious consequences for users and companies, especially in the context of data privacy and regulatory compliance. The smallest share – only 2 % – is accounted for by errors related to resource usage. This type usually manifests itself in conditions of large data processing volumes or excessive load on the computing infrastructure, which, in turn, may indicate the need to optimize the architecture or scale resources.

The approach proposed in this work has certain limitations. One of the key challenges is the temporal degradation of the model: with the development of cloud computing applications, the emergence of new features, or changes in the data structure, the effectiveness of pre-trained models may decline. To maintain high classification accuracy, it is necessary to regularly update and retrain models on current data. This will ensure that the models correspond to the current state of software systems and allow for high productivity to be maintained in real-world conditions.

## Conclusions

Classifying bugs in cloud computing applications using machine learning methods is an important task that combines technical components with a deep understanding of the subject area. The study confirms the feasibility and effectiveness of using machine learning algorithms for automated detection and classification of various types of bugs in a cloud environment. Thanks to their ability to process large amounts of data generated by cloud systems, these algorithms enable real-time bug prediction with high accuracy.

The proposed approach contributes to a significant increase in the efficiency of bug detection and elimination processes, which, in turn, reduces the risk of downtime and increases the reliability of cloud services. The application of such solutions can be an important component in ensuring the stable operation of critical information systems, especially in the context of growing business dependence on cloud technologies.

In addition, a promising area for further research is the implementation of transfer learning methods, which allow the knowledge gained from previous bug classification tasks to be used to improve the accuracy of models in new, similar contexts. The application of domain adaptation methods is also relevant, especially in cases where there is a discrepancy between the distributions of training and test data.

## References

1. Gupta, M., Gupta, D., Rai, P. (2024), "Exploring the Impact of Software as a Service (SaaS) on Human Life", *EAI Endorsed Transactions on Internet of Things*. DOI: https://doi.org/10.4108/eetiot.4821
2. Zhao, Y., Damevski, K., Chen, H. (2023), "A systematic survey of just-in-time software defect prediction", *ACM Computing Surveys*, Vol. 55, No. 10, P. 1–35. DOI: https://doi.org/10.1145/3567550
3. Bugayenko, Y., Bakare, A., Cheverda, A., Farina, M., Kruglov, A., Plaksin, Y., Succi, G. (2023), "Prioritizing tasks in software development: A systematic literature review", *PLOS ONE*, Vol. 18, No. 4, Article e0283838. DOI: https://doi.org/10.1371/journal.pone.0283838
4. Shiri Harzevili, N., Boaye Belle, A., Wang, J., Wang, S., Jiang, Z. M., Nagappan, N. (2024), "A systematic literature review on automated software vulnerability detection using machine learning", *ACM Computing Surveys*, Vol. 57, No. 3, P. 1–36. DOI: https://doi.org/10.1145/3699711
5. Tabianan, K., Velu, S., Ravi, V. (2022), "K-means clustering approach for intelligent customer segmentation using customer purchase behavior data", *Sustainability*, Vol. 14, No. 12, Article 7243. DOI: https://doi.org/10.3390/su14127243
6. Waqar, A. (2020), "Software Bug Prioritization in Beta Testing Using Machine Learning Techniques", *Journal of Computer Science*, Vol. 1, P. 24–34. DOI: https://doi.org/10.17509/jcs.v1i1.25355
7. Huda, S., Liu, K., Abdelrazek, M., Ibrahim, A., Alyahya, S., Al-Dossari, H., Ahmad, S. (2018), "An Ensemble Oversampling Model for Class Imbalance Problem in Software Defect Prediction", *IEEE Access*, Vol. 6, P. 24184–24195. DOI: https://doi.org/10.1109/ACCESS.2018.2817572
8. Goyal, A., Sardana, N. (2019), "Empirical Analysis of Ensemble Machine Learning Techniques for Bug Triaging", *Proceedings of the Twelfth International Conference on Contemporary Computing (IC3)*, P. 1–6. DOI: https://doi.org/10.1109/IC3.2019.8844876
9. Gupta, A., Sharma, S., Goyal, S., Rashid, M. (2020), "Novel XGBoost Tuned Machine Learning Model for Software Bug Prediction", *Proceedings of the International Conference on Intelligent Engineering and Management (ICIEM)*, P. 376–380. DOI: https://doi.org/10.1109/ICIEM48762.2020.9160152
10. Ahmed, H. A., Bawany, N. Z., Shamsi, J. A. (2021), "CaPBug-A Framework for Automatic Bug Categorization and Prioritization Using NLP and Machine Learning Algorithms", *IEEE Access*, Vol. 9, P. 50496–50512. DOI: https://doi.org/10.1109/ACCESS.2021.3069248
11. Tabassum, N., Alyas, T., Hamid, M., Saleem, M., Malik, S. (2022), "Hyper-convergence storage framework for ecocloud correlates", *Computers, Materials & Continua*, Vol. 70, No. 1, P. 1573–1584. DOI: https://doi.org/10.32604/cmc.2022.019389

*Received (Надійшла) 06.11.2025*
*Accepted for publication (Прийнята до друку) 09.12.2025*
*Publication date (Дата публікації) 28.12.2025*

## *About the Authors / Відомості про авторів*

**Shmatko Oleksandr** – PhD (Engineering Sciences), Associate Professor, Kharkiv National University of Radio Electronics, Associate Professor at the Department of Electronic Computers, Kharkiv, Ukraine; e-mail: oleksandr.shmatko2@nure.ua; ORCID ID: https://orcid.org/0000-0002-2426-900X

**Gamayun Igor** – Doctor of Sciences (Engineering), Professor, National Technical University "Kharkiv Polytechnic Institute", Professor at the Department of Software Engineering and Management Intelligent Technologies, Kharkiv, Ukraine; e-mail: Ihor.Hamaiun@khpi.edu.ua; ORCID ID: https://orcid.org/0000-0003-2099-4658

**Kolomiitsev Oleksii** – Honored Inventor of Ukraine, Doctor of Sciences (Engineering), Professor, National Technical University "Kharkiv Polytechnic Institute", Professor at the Department Computer Engineering and Programming, Kharkiv, Ukraine; e-mail: alexus_k@ukr.net; ORCID ID: https://orcid.org/0000-0001-8228-8404

**Шматко Олександр Віталійович** – кандидат технічних наук, доцент, Харківський національний університет радіоелектроніки, доцент кафедри електронних обчислювальних машин, Харків, Україна.

**Гамаюн Ігор Петрович** – доктор технічних наук, професор, Національний технічний університет "Харківський політехнічний інститут", професор кафедри програмної інженерії та інтелектуальних технологій управління, Харків, Україна.

**Коломійцев Олексій Володимирович** – заслужений винахідник України, доктор технічних наук, професор, Національний технічний університет "Харківський політехнічний інститут", професор кафедри комп'ютерної інженерії та програмування, Харків, Україна.

# ГІБРИДНА МОДЕЛЬ МАШИННОГО НАВЧАННЯ ДЛЯ КЛАСИФІКАЦІЇ ПРОГРАМНИХ ПОМИЛОК У ХМАРНИХ *SAAS*-ЗАСТОСУНКАХ

У сучасних хмарних обчислювальних середовищах забезпечення стабільності та надійності програмних застосунків є одним із ключових чинників ефективної роботи інформаційних систем. Значну частину збоїв у таких системах спричиняють програмні помилки (баги), які ускладнюють експлуатацію та знижують продуктивність сервісів. Традиційні методи ручного аналізу звітів про помилки є трудомісткими, тому необхідно розробити інтелектуальні підходи до автоматизованої класифікації та пріоритизації помилок із використанням методів машинного навчання. **Мета статті** – підвищення точності класифікації типів програмних помилок у хмарних застосунках. **Завдання дослідження:** формування повного конвеєра автоматизованого оброблення даних баг-репортів, що охоплює всі етапи – від попереднього очищення до побудови моделі класифікації. **Методологічна основа дослідження** полягає у використанні методів оброблення природної мови (NLP), техніки SMOTE для балансування вибірки, класичних алгоритмів машинного навчання, а також процедури оптимізації гіперпараметрів *RandomizedSearchCV*. Якість моделей оцінюється на основі стандартних класифікаційних метрик, таких як *accuracy, precision, recall* та *F1-score*, що забезпечує комплексний і об'єктивний аналіз отриманих результатів. **Результати дослідження.** Розроблено гібридну модель для автоматизованої класифікації помилок, що охоплює етапи збирання, попереднього оброблення, векторизації та моделювання даних. Проведено порівняльний аналіз точності чотирьох алгоритмів машинного навчання – наївного баєсівського класифікатора, дерева рішень, випадкового лісу й логістичної регресії – із використанням різних методів векторизації (*Bag-of-Words, TF-IDF, Word2Vec*). Для підвищення точності класифікації застосовано техніку балансування даних SMOTE. Експериментальні дослідження на реальному наборі даних із хмарного середовища продемонстрували, що модель *Random Forest* досягла найвищих показників точності – до 91,7 %. Результати підтверджують ефективність інтеграції алгоритмів машинного навчання в процеси аналізу й підтримки програмних продуктів у хмарних інфраструктурах. **Висновки.** Запропонований підхід забезпечує підвищення точності класифікації помилок у хмарних обчислювальних системах і може бути використаний у системах моніторингу, *DevOps*-платформах і засобах автоматизованого тестування. Результати дослідження є основою для подальшого розроблення інтелектуальних інструментів прогнозування й пріоритизації дефектів програмного забезпечення.

**Ключові слова:** класифікація помилок; хмарні обчислення; машинне навчання; *TF-IDF; Word2Vec*; випадковий ліс; автоматизація тестування.

## *Bibliographic descriptions / Бібліографічні описи*

Shmatko, O., Gamayun, I., Kolomiitsev, O. (2025), "Hybrid machine learning model for classifying software bugs in SaaS cloud applications", *Management Information Systems and Devises*, No. 4 (187), P. 156–181. DOI: https://doi.org/10.30837/0135-1710.2025.187.156

Шматко О. В., Гамаюн І. П., Коломійцев О. В. Гібридна модель машинного навчання для класифікації програмних помилок у хмарних *SaaS*-застосунках. *Автоматизовані системи управління та прилади автоматики*. 2025. № 4 (187). С. 156–181.
DOI: https://doi.org/10.30837/0135-1710.2025.187.156