UNIVERSIDAD SAN FRANCISCO DE QUITO USFQ

Colegio de Posgrados

Classification of Software Bugs Using Supervised Learning Models

Proyecto de Titulación

Jack Ricardo Narváez Salazar

Israel Pineda, Ph.D.

Director de Trabajo de Titulación

Trabajo de titulación de posgrado presentado como requisito para la obtención del título de Magíster en Ciencia de Datos

Quito, 01 de diciembre de 2024

UNIVERSIDAD SAN FRANCISCO DE QUITO USFQ
COLEGIO DE POSGRADOS

HOJA DE APROBACIÓN DE TRABAJO DE TITULACIÓN

Classification of Software Bugs Using Supervised Learning Models

Jack Narváez

Nombre del Director del Programa:	Felipe Grijalva
Título académico:	Ph.D. en Ingeniería Eléctrica
Director del programa de:	Ciencia de Datos
Nombre del Decano del colegio Académico:	Eduardo Alba
Título académico:	Doctor en Ciencias Matemáticas
Decano del Colegio:	Ciencias e Ingenierías
Nombre del Decano del Colegio de Posgrados:	Dario Niebieskikwiat
Título académico:	Doctor en Física

Quito, diciembre 2024

© DERECHOS DE AUTOR

Nombre del estudiante:	Jack Ricardo Narváez Salazar
Código de estudiante:	00338780
C.I.:	1751370725
Lugar y fecha:	Quito, 01 de diciembre de 2024.

ACLARACIÓN PARA PUBLICACIÓN

Nota: El presente trabajo, en su totalidad o cualquiera de sus partes, no debe ser considerado como una publicación, incluso a pesar de estar disponible sin restricciones a través de un repositorio institucional. Esta declaración se alinea con las prácticas y recomendaciones presentadas por el Committee on Publication Ethics COPE descritas por Barbour et al. (2017) Discussion document on best practice for issues around theses publishing, disponible en http://bit.ly/COPETheses.

UNPUBLISHED DOCUMENT

Note: The following graduation project is available through Universidad San Francisco de Quito USFQ institutional repository. Nonetheless, this project – in whole or in part – should not be considered a publication. This statement follows the recommendations presented by the Committee on Publication Ethics COPE described by Barbour et al. (2017) Discussion document on best practice for issues around theses publishing available on http://bit.ly/COPETheses.

DEDICATORIA

"A Dios, por ser mi luz y guía en cada paso del camino, y a mi familia, por su amor incondicional, paciencia y apoyo inquebrantable. Sin su fe en mí, este logro no habría sido posible."

RESUMEN

El presente trabajo aborda la problemática de la clasificación automática de errores en software mediante el uso de técnicas avanzadas de aprendizaje supervisado. La motivación principal radica en la necesidad de mejorar la detección y clasificación de errores para optimizar los procesos de desarrollo y mantenimiento de software. Se utilizaron repositorios públicos de GitHub como fuente de datos, analizando commits mediante herramientas como Radon y Flake8 para extraer características clave del código, como la complejidad y el número de líneas. Se emplearon los modelos XGBoost y RandomForestClassifier, con técnicas de balanceo de datos como SMOTE y undersampling para abordar el desbalance de clases presente en los datos. El análisis comparativo entre ambos modelos mostró que XGBoost logró un desempeño superior, destacando en precisión y recall, especialmente en clases con mayor representación. Los resultados demuestran la efectividad de combinar preprocesamiento exhaustivo, como imputación de valores faltantes y transformación logarítmica, con modelos robustos y ajuste de hiperparámetros para mejorar la clasificación. Como conclusión, este enfoque proporciona un marco reproducible y escalable para la detección de errores en software, con aplicaciones potenciales en sistemas de desarrollo automatizado. Futuros trabajos pueden explorar la integración de técnicas de aprendizaje profundo y ampliar el análisis a otros lenguajes de programación.

Palabras clave: clasificación de errores, aprendizaje supervisado, XGBoost, RandomForestClassifier, balanceo de datos, GitHub, Radon, Flake8, detección de errores.

ABSTRACT

This work addresses the challenge of automatic bug classification in software using advanced supervised learning techniques. The primary motivation lies in the need to improve error detection and classification to optimize software development and maintenance processes. Public GitHub repositories were used as data sources, analyzing commits with tools like Radon and Flake8 to extract key code features, such as complexity and line count. The XGBoost and RandomForestClassifier models were employed, utilizing data balancing techniques like SMOTE and undersampling to address the class imbalance in the dataset.The comparative analysis between the two models showed that XGBoost achieved superior performance, particularly excelling in precision and recall for classes with higher representation. The results demonstrate the effectiveness of combining thorough preprocessing, such as missing value imputation and logarithmic transformation, with robust models and hyperparameter tuning to enhance classification. In conclusion, this approach provides a reproducible and scalable framework for software bug detection, with potential applications in automated development systems. Future work could explore the integration of deep learning techniques and expand the analysis to other programming languages.

Key words: bug classification, supervised learning, XGBoost, RandomForestClassifier, data balancing, GitHub, Radon, Flake8, bug detection.

TABLA DE CONTENIDO

ÍNDICE DE FIGURAS

Classification of Software Bugs Using Supervised Learning Models

Israel Pineda, *Senior Member, IEEE,* Jack Narváez, *Member, IEEE,*

***Abstract*—The automated classification of software bugs plays a crucial role in improving software quality and accelerating the debugging process. This study compares the performance of two widely used supervised learning algorithms, RandomForestClassifier and XGBoost, in classifying software bugs extracted from open-source repositories. By utilizing a structured dataset generated from GitHub commits, including metrics such as code complexity, maintainability index, and number of lines, we systematically preprocess the data through outlier imputation, logarithmic transformations, normalization, and balancing techniques like SMOTE and undersampling. The results demonstrate the advantages of XGBoost in handling imbalanced datasets and improving precision in complex classifications, especially in minority bug types. These findings highlight the potential of supervised learning to streamline bug classification, providing a foundation for future research in software reliability engineering.**

I. Introduction

SOFTWARE bugs are a significant challenge in software development, impacting reliability, maintainability, and user satisfaction. Traditional debugging relies on manual classification and prioritization, which is time-consuming and prone to errors. Leveraging machine learning, particularly supervised learning models, offers a data-driven approach to automate bug classification based on historical code data and commit messages.

"The most crucial phase of any software, which necessitates intensive testing, is software defect detection. It also occupies the most significant position in the software development life cycle (SDLC). (Khalid, 2023)"

This research investigates the effectiveness of RandomForestClassifier and XGBoost for classifying software bugs, utilizing features such as code complexity, maintainability index, and code structure extracted from GitHub repositories. By addressing challenges such as imbalanced datasets and feature scaling, we aim to establish a robust methodology for improving the accuracy and reliability of bug classification models.

"With the recent widespread availability of open source repositories, it has become possible to use data-driven techniques to discover patterns of bug manifestation. (Harer et al., 2018)."

I. Pineda and J. Narváez are with Universidad San Francisco de Quito USFQ

II. State of the Art

The evolution of machine learning has significantly influenced software engineering. Random forests and gradient boosting algorithms like XGBoost have become prominent due to their robustness and adaptability. Random forests use ensemble techniques to combine decision trees, reducing overfitting and variance. In contrast, XGBoost employs sequential boosting, optimizing residual errors iteratively for high accuracy.

"Transferring learning techniques can leverage knowledge gained from previous bug classification problems to improve the performance of bug classification in cloud computing applications (Tabassum, 2023)."

Studies have demonstrated the superior performance of XGBoost in handling complex data structures and imbalanced datasets. However, its computational intensity can be a limitation. RandomForestClassifier remains a popular choice for rapid prototyping due to its simplicity and efficiency. This study bridges the gap in literature by comparing these algorithms specifically for software bug classification, leveraging real-world datasets from GitHub.

"Classification is a major task of data analysis using machine learning algorithms that allow the machine to learn associations between instances and decision labels, from which an algorithm builds a model to predict the labels of new instances for a specific sample data (Khleel & Nehéz, 2021)."

III. Materials and Methodology

A. Data Collection

We have a task called fetch_commits which uses the GitHub API to extract commit metadata from repositories such as python/cpython, pandas-dev/pandas, django/django and scikit-learn/scikit-learn. The extracted data included commit messages and associated file changes by filtering out those libraries containing the keywords “fix” and “bug”, and analyzed using tools such as radon for code metrics and flake8 for static code analysis.

"One of the main challenges in the identification of bug candidates is to check whether the reported warnings by static detectors are corresponding with the issued bug class or not (Shiri Harzevili et al., 2023)."

B. Feature Extraction

The process_commit task attempts to perform a detailed analysis of the commit file content using the ast (Abstract

Syntax Tree) library and code analysis tools like radon and flake8, with the goal of extracting metrics that describe the quality and structure of the code. This function attempts to parse the file content as an abstract syntax tree (ast.parse). This is useful for exploring the structure of functions, classes, and other elements of the code without executing it.

"The activity of software defect prediction is necessary in order to enhance the effectiveness of quality assurance process. It can help to develop a qualitative product with limited amount of resources in a limited time period. (Iqbal1, 2019)"

The following key features were derived from each file:

- maintainability_index: Calculates a maintainability index of the code, using mi_visit, that estimates the maintainability of the code.
- complexity: Sum of the cyclomatic complexity of each code block (function or class) in the file.
- Normalization: StandardScaler was used to scale features for uniformity.
- num_lines: Total number of lines in the file.
- num_functions and num_classes: Number of functions and classes in the file.
- avg_function_length: Average length of functions (number of AST elements in each function).
- avg_num_parameters: Average number of parameters per function.
- nesting_levels: Maximum nesting level in control structures (if, for, while).
- num_comments: Number of comments in the file.
- duplicated_code_warnings: Detects duplicate code fragments in the file.
- num_imports: Number of libraries imported into the file.
- cyclic_dependencies: Variable to record cyclic dependencies, although it is currently a "placeholder".

We then use the pylint library to perform a static analysis of the code and look for certain common error patterns in the code (pylint_output). If pylint detects specific errors, the error type is classified into one of the following errors.

- "syntax-error" -> SyntaxError
- "undefined-variable" -> NameError
- "unused-import" -> ImportError
- "attribute-defined-outside-init" -> AttributeError

Finally, we obtain an error dataset with approximately 20 thousand records (4 thousand records for each type of error) with which we can continue working.

C. Dataset Analysis

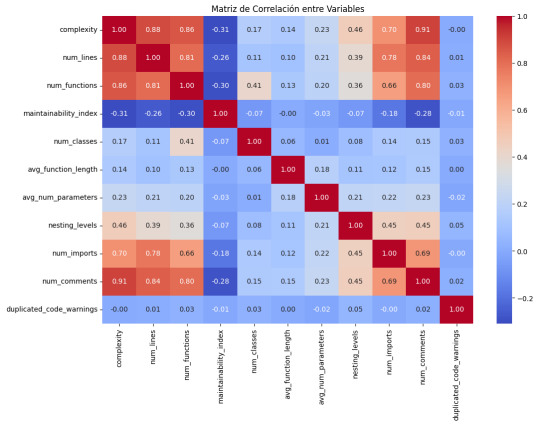

Figure 1. Correlation Matrix

Figure 1 shows the correlation of the numerical variables.

High Correlations:

- num_lines and num_functions have an extremely high correlation (0.81), suggesting that as the number of lines increases, the number of functions tends to increase as well. This may be because larger and more complex projects generally contain more functions.
- complexity and num_lines show a high correlation (0.88), which is intuitive since longer code in terms of lines is likely to be more complex.
- num_comments is positively correlated with complexity (0.91), num_lines (0.84), and num_functions (0.80). This suggests that as code complexity or size increases, comments also tend to increase, possibly to make it easier to understand.

Moderate Correlations:

- num_imports has a moderate correlation with num_lines (0.78) and num_comments (0.69). This could indicate that in larger projects, more dependencies or libraries need to be imported, which is reflected in a higher number of imports.

Low Correlations:

- avg_function_length has low correlations with almost all variables, indicating that the average function length does not directly depend on the number of functions, classes, or overall code complexity.
- nesting_levels shows low correlations with most variables, suggesting that nesting depth does not vary significantly as a function of other code metrics.

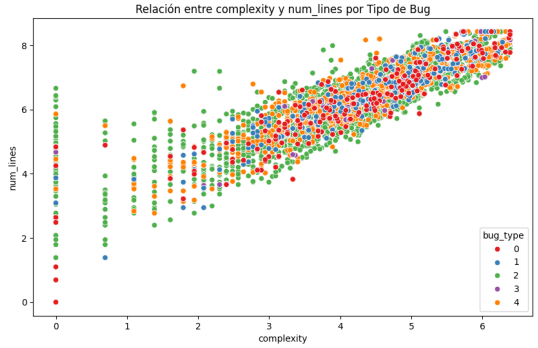

Figure 2. Relationship between complexity and num_lines

Figure 2 shows a scatter plot illustrating the relationship between code complexity (on the x-axis) and the number of lines of code (num_lines) (on the y-axis) for different bug types.

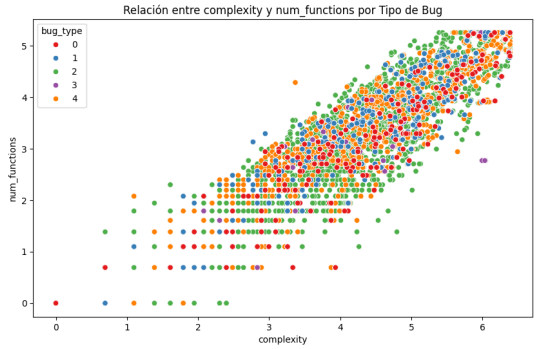

Figure 3. Relationship between complexity and num_functions

Figure 3 shows a scatter plot illustrating the relationship between code complexity (on the x-axis) and the number of functions of code (num_functions) (on the y-axis) for different bug types.

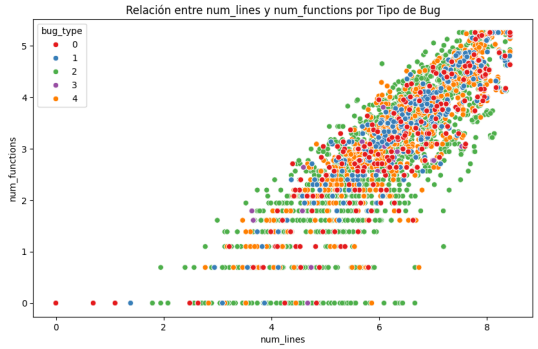

Figure 4. Relationship between num_lines and num_functions

Figure 4 shows a scatter plot illustrating the relationship between code num_lines (on the x-axis) and the number of functions of code (num_functions) (on the y-axis) for different bug types.

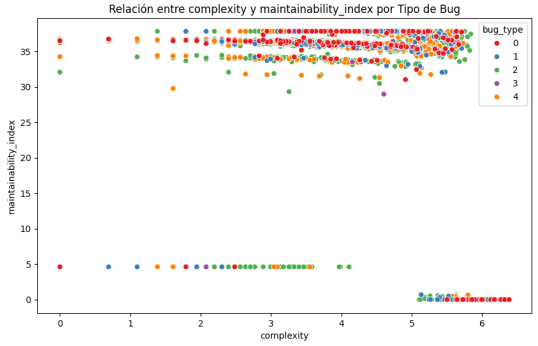

Figure 5. Relationship between complexity and mantainability_index

Figure 5 shows a scatter plot illustrating the relationship between code complexity (on the x-axis) and the mantainability of code (mantainability_index) (on the y-axis) for different bug types.

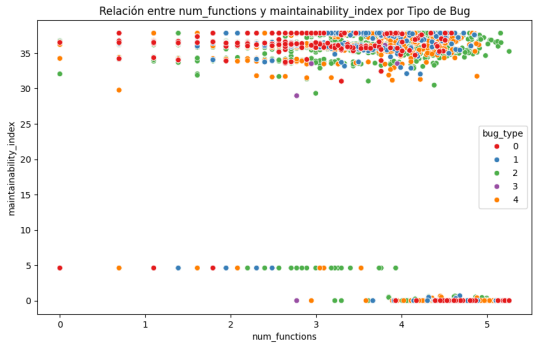

Figure 6. Relationship between num_functions and mantainability_index

Figure 6 shows a scatter plot illustrating the relationship between number of functions of code (on the x-axis) and the mantainability of code (mantainability_index) (on the y-axis) for different bug types.

Analysis:

1) Skewed distributions: Some features, such as num_classes, duplicated_code_warnings, cyclic_dependencies, and is_bug, have extremely skewed distributions or take a dominant value. This can affect the performance of certain machine learning models. Log transformation might be useful for those features with continuous values and positive skew, but it would not be useful for binary or categorical variables such as is_bug.
2) Log transformation: Consider applying log transformations on features that have a large value range and positive skew, such as maintainability_index, complexity, num_lines, num_functions, num_comments, and num_imports. The transformation might make these distributions more like a normal distribution and facilitate convergence in linear models or models sensitive to the data distribution.

3) Categorical variables: Features like bug_type and is_bug seem to represent categories (discrete values). In these cases, it is best not to apply logarithmic transformations, but make sure they are properly encoded for classification models.
4) High-frequency variables at zero: Some features, such as cyclic_dependencies and duplicated_code_warnings, show very dominant values of zero. If these values are frequent and represent a significant proportion, you might consider treating these values specially, perhaps as a binary indicator.

D. Outliers Processing

Outliers are data points that deviate significantly from the rest of the data distribution. They can arise due to measurement errors, data entry errors, or represent rare but valid phenomena. Addressing outliers is crucial in machine learning and statistical analysis. For this reason we are going to impute the outliers.

Variables with Highly Skewed Distributions: Variables with long tails at one end often have significant outliers. These variables are likely to benefit from a logarithmic transformation and outlier processing.

Variables with Wide Value Ranges and Frequency Spikes: Some variables may have very high extreme values compared to the majority of the data, suggesting that they have outliers. Common examples are the number of lines of code (num_lines) or the number of functions (num_functions).

We implemented a function that identifies and replaces outliers in selected variables, using the interquartile range (IQR).

The IQR is defined as the difference between the third quartile (Q3) and the first quartile (Q1), and the thresholds for detecting outliers are calculated as:

$$\text{Lower limit} = Q1 - 1.5 \times IQR$$

$$\text{Upper limit} = Q3 + 1.5 \times IQR$$

Any value below the lower bound or above the upper bound is considered an outlier. The implemented function replaces these outliers with the nearest bound (lower or upper).

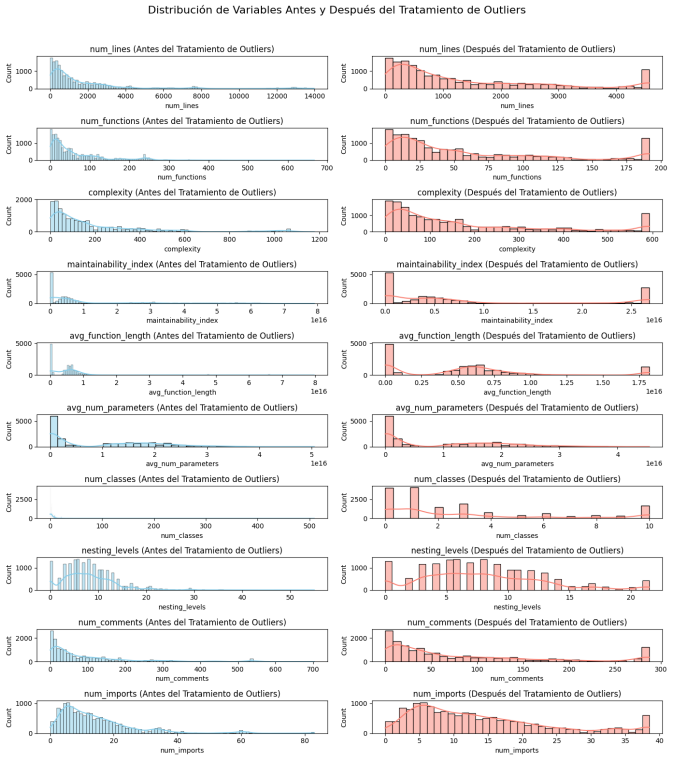

Figure 7. Distribution of variables before and after outlier treatment

Figure 7 shows a comparison between the distributions of key variables in the dataset before and after outlier treatment.

The plots show a clear comparison of the effect of outlier imputation on each of the variables. This approach, rather than removing outliers, helps to maintain the overall information of the distribution without extreme values skewing the data too much. The subsequent normalization has also helped to bring the variables to a more uniform scale, which can improve the performance of the models to be trained.

This technique ensures that outliers are handled effectively, preserving the consistency and integrity of the data for subsequent analysis.

E. Logarithmic Transformation

If a variable has a long tail on the right side (skewed distribution), log transformation can make the distribution more symmetrical, making analysis and interpretation easier.

Logarithmic transformation can reduce the influence of outliers in the analysis because it compresses large values more significantly than small values.

By applying the logarithmic transformation on the selected variables, you have managed to reduce the skewness in their distribution, and the histograms show better symmetry compared to the original data. This is especially useful when the data has a high skewness or extreme values, as in this case.

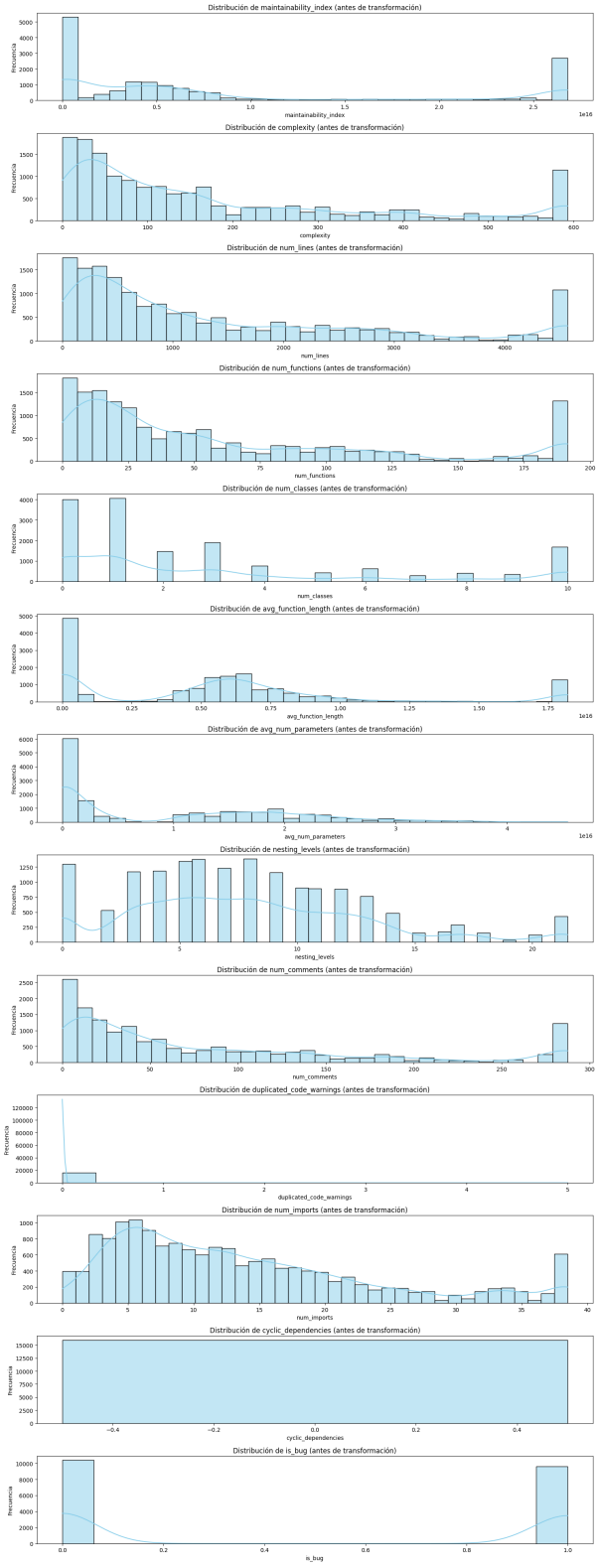

Figure 8. Variable distributions before logarithmic transformation

Figure 8 shows the distribution of variables in the dataset before applying any transformations.

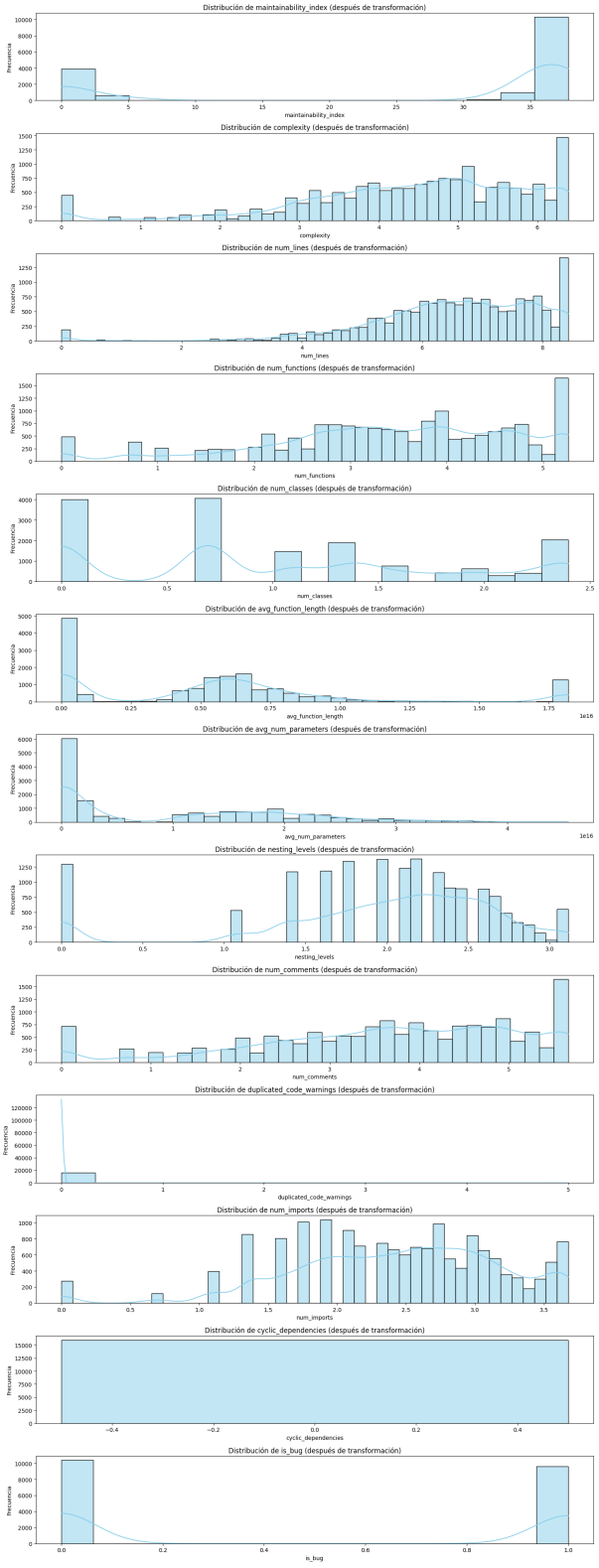

Figure 9. Variable distributions after logarithmic transformation

Figure 9 shows the distribution of variables in the dataset after applying any transformations.

The logarithmic transformation of the variables complexity, num_lines and num_functions smoothed out the long tails and made the distributions closer to normal, which is ideal.

After normalization, the distributions are centered around zero, ready to be used in scale-sensitive models.

The variable maintainability_index: maintains its bimodal shape even after transformation and normalization, which is expected since it did not have such a pronounced asymmetry as the other variables.

Normalization is still useful to keep this variable in the same range as the others.

F. Normalization

Data normalization is an essential step to prepare a dataset before training a machine learning model. In this work, we implement a complete data processing pipeline including null value imputation, categorical variable encoding, and feature normalization.

To handle missing values in the predictor variables, the mean-based imputation strategy was used. This ensures that no information is lost due to null values, replacing them with the average of the corresponding column.

The categorical variable bug_type was encoded into a numeric format using LabelEncoder so that it can be interpreted by machine learning models.

The predictor variables (X) and the target variable (y) were separated to split the data into training and test sets. The split was done in a ratio of 80%-20%.

To ensure that the numerical features are on the same scale, standardization was applied using StandardScaler.

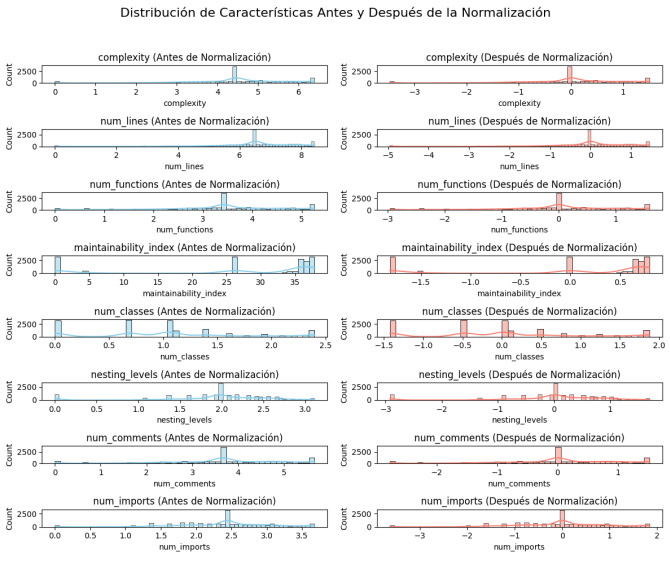

Figure 10. Distribution of Features Before and After Normalization

Figure 10 illustrates the distributions of selected features before and after normalization. It compares the original scales of the features (on the left) to their normalized versions (on the right).

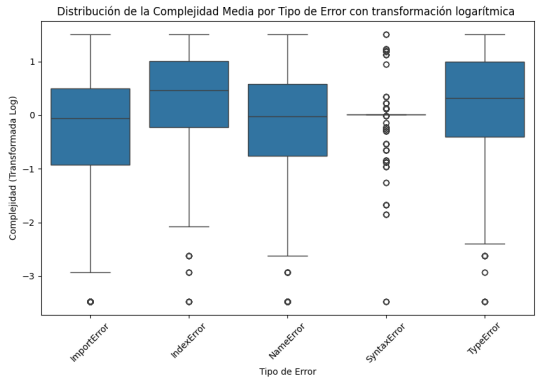

Figure 11. Distribution of Average Complexity by Error Type with Logarithmic Transformation

Figure 11 is a boxplot showing the distribution of the log-transformed complexity for different types of errors: ImportError, IndexError, NameError, SyntaxError, and TypeError.

G. Model Training: XGBoost VS RandomForestClassifier

Supervised learning means that these models are trained with a dataset where each input includes features (inputs) and a known target label or class (output). The goal of the model is to learn the relationship between the features and labels so that, once trained, it can correctly predict the labels of new inputs.

"The supervised machine learning algorithms try to develop an inferring function by concluding relationships and dependencies between the known inputs and outputs of the labeled training data, such that we can predict the output values for new input data based on the derived inferring function. (Hammouri, 2018)"

RandomForestClassifier:

It is a model based on an ensemble of decision trees (a "forest"). It builds multiple independent decision trees in parallel and combines their predictions (usually using majority voting for classification) to arrive at a final decision. It uses bagging (bootstrap aggregating) techniques to improve accuracy and reduce overfitting.

"The random forest feature selection helps to reduce the correlation between the trees. If we use every feature then most of the trees will have the same decision nodes and they will act very similar, which can increase the variance (Thomas & Kaliraj, 2024)."

This model uses the bagging technique (Bootstrap Aggregating) to improve accuracy and reduce overfitting.

- Training process:
  1) The training set is divided into several samples by sampling with replacement.
  2) For each sample, an independent decision tree is built, using only a random selection of the available features.

3) Each tree is trained to make predictions, and the final prediction is made by majority vote of all the trees (in the case of classification).

- Key Features:
  1) Each tree in the forest is trained on a random sample of the dataset, which helps reduce the variance of the model.
  2) Trees in the forest do not depend on each other and can be built in parallel, making the model fast and efficient.

  The important hyperparameters were:
  - **n_estimators:** As in XGBoost, this controls the number of trees in the forest. Values between 100 and 500 are explored here as well.
  - **max_depth**: Maximum depth of trees. Values like 10, 20, and 30 limit the depth and thus the complexity of trees. It can also be left at None to allow trees to grow until they become overfit.
  - **min_samples_split:** Minimum number of samples needed to split a node. Higher values (e.g. 10) make the model more restrictive, while lower values (e.g. 2) allow for more splitting and detail.
  - **min_samples_leaf:** Minimum number of samples needed in a leaf. Increasing this value makes the leaves have more data and reduces overfitting.
  - **bootstrap:** Controls whether each tree is trained with samples with replacement (True) or without replacement (False). True is the default setting and is effective in reducing overfitting.

XGBoost:

This is a boosting model that also uses decision trees, but instead of building them in parallel, it builds them sequentially.

Each new tree attempts to correct the mistakes made by the previous trees, giving it a more adaptive learning approach.

It uses advanced boosting techniques (such as gradient descent and optimization) to improve performance, being very effective at capturing complex relationships in data. Both models are widely used in classification and regression tasks and can handle high-dimensional and complex data.

"In tree boosting algorithms, eXtreme or XGBoost are used to aid in the exploitation of all hardware and memory resources available, allowing for its deployment in computing environments, tuning the model and enhancing the algorithm (Aquil & Ishak, 2020)"

- Training process:
  1) It starts with a tree that attempts to predict the outcome. At the end of the tree, errors are identified using a loss function (such as squared error for regression or log-loss for classification).
  2) Each new tree is trained on the residue of the previous trees, trying to improve where the previous ones failed.
  3) The model continues adding trees until it reaches a specified number of trees or until the error is minimized.
- Key Features:
  1) XGBoost employs advanced optimizations, such as regularization and null value handling techniques, that allow it to be extremely accurate in prediction tasks.
  2) Its implementation uses parallel processing, efficient memory management and tree pruning techniques, making it one of the fastest tools for boosting.

In XGBoost (XGBClassifier), the most relevant hyperparameters in this project are:

- **n_estimators:** Number of trees in the model. Values between 100 and 500 are explored here.
- **max_depth:** Maximum depth of each tree, which controls the level of detail in the model. Values like 3, 5, 10, and 20 help regulate complexity.
- **learning_rate:** Model learning rate. This hyperparameter controls how much each tree contributes to updating predictions. Low values (e.g., 0.01) make the model more accurate but require more trees; higher values (e.g., 0.3) can make the model faster but less accurate.
- **subsample:** Fraction of samples used in each tree to avoid overfitting. A value of 0.5 indicates that each tree uses 50 % of the data, while a value of 1.0 indicates that all are used.
- **colsample_bytree:** Fraction of features that each tree uses for training, similar to subsample but for columns. A value of 0.5 indicates that each tree uses only 50% of the features.
- **scale_pos_weight:** This parameter adjusts the weight of the classes in imbalanced problems. Higher values indicate greater weight for the minority class.

Using RandomizedSearchCV for Hyperparameter Optimization:

Using RandomizedSearchCV for Hyperparameter Optimization RandomizedSearchCV is a hyperparameter search technique that allows you to randomly select combinations of hyperparameters from a predefined range and test a limited number of combinations. In both cases:

- **param_distributions:** Specifies the range of values for each hyperparameter. Distributions and values are defined for the hyperparameters of each model.
- **n_iter=50:** Controls the number of random combinations to test. This choice limits the number of tests to make the search time-efficient.
- **cv=3:** Divides the data into 3 partitions to perform cross-validation and estimate the performance of the model.
- **random_state=42:** Ensures reproducibility of the results.

- **n_jobs=-1:** Uses all available CPU cores to speed up the process.
- **scoring='accuracy':** The primary metric used to select the best combination of hyperparameters based on accuracy.

Training Process:

In both cases, after setting up RandomizedSearchCV, the model is trained on the balanced dataset (using fit_resample for oversampling and undersampling). The difference is that:

- XGBoost performs sequential learning, where each new tree improves the errors of the previous tree. This makes XGBoost's accuracy tend to be higher on complex problems, but it can take longer.
- RandomForestClassifier trains all trees independently and then takes a majority vote for the final prediction. This approach is generally faster and more efficient on less complex problems.

IV. Results and Discussion

A. Confusion Matrix

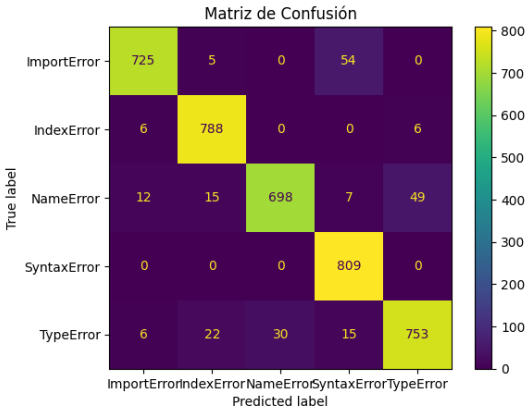

Figure 12. RandomForestClassifier confusion matrix

Figure 12 represents the confusion matrix of a classification model (in this case, a RandomForestClassifier) for predicting five categories of software errors: ImportError, IndexError, NameError, SyntaxError, and TypeError.

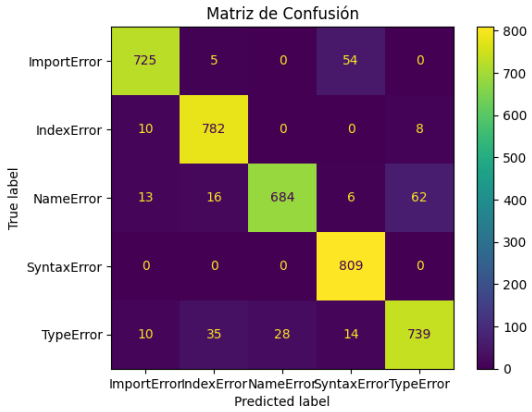

Figure 13. XGBoost confusion matrix

Figure 13 represents the confusion matrix of a classification model (in this case, XGBoost) for predicting five categories of software errors: ImportError, IndexError, NameError, SyntaxError, and TypeError.

- The confusion matrix of XGBoost and RandomForestClassifier show some differences in the amount of misclassifications between classes.
- XGBoost appears to have slightly better accuracy for some classes, which could be due to its ability to more finely tune the complexities of the data.

B. Classification Metrics (Precision, Recall, F1-Score)

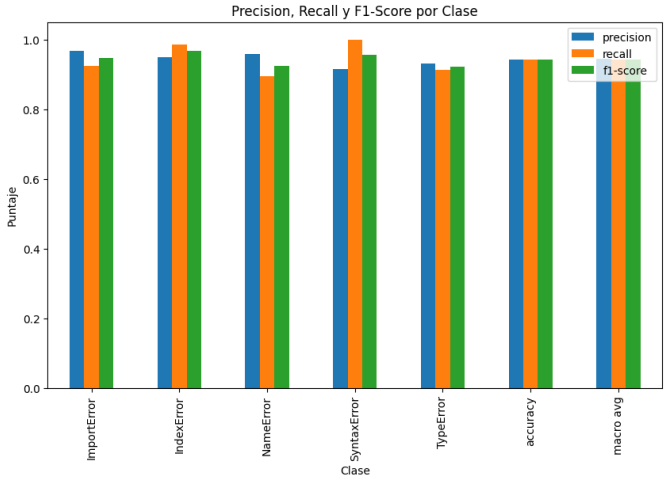

Figure 14. Precision, Recall, F1-Score RandomForestClassifier

Figure 14 shows the precision, recall, and F1-score for each class in a classification model, specifically a RandomForestClassifier. It provides a detailed performance evaluation across five error types: ImportError, IndexError, NameError, SyntaxError, and TypeError, along with aggregate metrics like overall accuracy and macro averages.

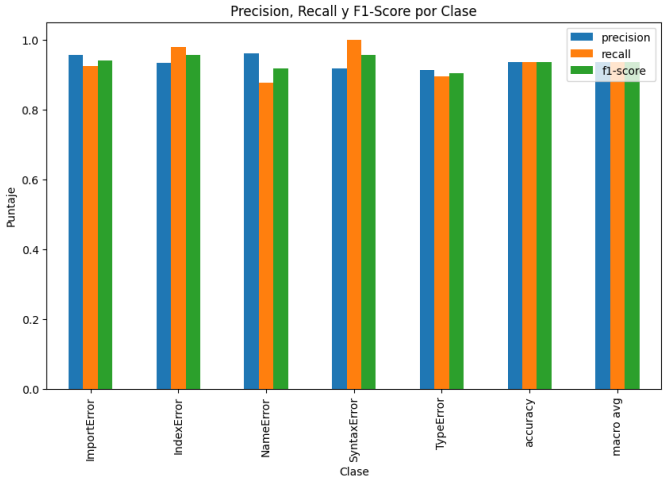

Figure 15. Precision, Recall, F1-Score XGBoost

Figure 15 shows the precision, recall, and F1-score for each class in a classification model, specifically a XGBoost. It provides a detailed performance evaluation across five error types: ImportError, IndexError, NameError, SyntaxError, and TypeError, along with aggregate metrics like overall accuracy and macro averages.

- Both models show very competitive results in terms of precision and recall for several classes.
- In particular, XGBoost could show better results in cases where classes are unbalanced or when the model needs more granularity in classification due to its boosting technique, which allows targeting prediction errors in successive iterations.
  "Machine learning techniques help developers to retrieve useful information after the classification and enable them to analyse data from different perspectives. Machine learning techniques are proven to be useful in terms of software bug prediction (Aleem, Capretz, & Ahmed, 2015)."

The link to the github repository is as follows:https://github.com/jack-narv/Classification_Software_Bugs.git

V. Future Works

- **Increase the dataset:** Getting more examples of minority errors (such as TypeError) could help improve the model, especially in less represented classes.
- **Use additional dimensionality reduction techniques:** Methods such as PCA (Principal Component Analysis) could reduce the complexity of the model and improve its interpretability.
- **Explore other model architectures:** Since this is a multi-class classification problem, exploring more complex neural network architectures or ensemble models could lead to better performance compared to decision tree-based models.

VI. Conclusiones

- With the XGBoost model tuned and the hyperparameters optimized, the overall performance shows an accuracy of approximately 93%. This suggests that the model is generally effective at classifying different types of errors in the code.
- The logarithmic transformation and normalization of variables contributed to improving the distribution of the data, reducing the bias in some of the numerical characteristics such as complexity and num_lines.
- Although the model is accurate overall, there are still difficulties in classifying some classes accurately, such as TypeError.
- This classification model can be a useful tool for software development teams, since it allows anticipating and preventing errors in the code before its deployment, improving software quality and reducing debugging times.

References

[1] Hammouri A., Hammad M., Alnabhan M., Alsarayrah F., (2018). Software bug prediction using machine learning approach. ResearchGate. Retrieved from https://www.researchgate.net/publication/323536716_Software_Bug_Prediction_using_Machine_Learning_Approach

[2] Harer, J. A., Kim, L. Y., Russell, R. L., Ozdemir, O., Kosta, L. R., Rangamani, A., ... & Lazovich, T. (2018). Automated software vulnerability detection with machine learning. Draper and Boston University. Retrieved from https://arxiv.org/abs/1803.04497.

[3] Khleel, N. A. A., & Nehéz, K. (2021). Comprehensive study on machine learning techniques for software bug prediction. International Journal of Advanced Computer Science and Applications, 12(8), 1-10. Retrieved from https://www.researchgate.net/publication/354330980_Comprehensive_Study_on_Machine_Learning_Techniques_for_Software_Bug_Prediction

[4] Harzevili, N. S., Shin, J., Wang, J., Wang, S., & Nagappan, N. (2023). Automatic static bug detection for machine learning libraries: Are we there yet? arXiv preprint. Retrieved from https://arxiv.org/abs/2307.04080.

[5] Aleem, S., Capretz, L. F., & Ahmed, F. (2015). Benchmarking machine learning techniques for software defect detection. International Journal of Software Engineering & Applications (IJSEA), 6(3), 1–13. Retrieved from https://doi.org/10.5121/ijsea.2015.6302

[6] Thomas, N. S., & Kaliraj, S. (2024). An improved and optimized random forest-based approach to predict software faults. SN Computer Science, 5(530). https://doi.org/10.1007/s42979-024-02764-x

[7] Tabassum N., Namoun A., Alyas T., Tufail A., Taqi M. & Kim K. (2023). Bug Classification and Prioritization in Cloud Computing Systems. Applied Sciences, 13(2880), 1-24. https://doi.org/10.3390/app13052880

[8] Khalid, A., Badshah, G., Ayub, N., Shiraz, M., & Ghouse, M. (2023). Software Defect Prediction Analysis Using Machine Learning Techniques. Sustainability, 15(6), 5517. https://doi.org/10.3390/su15065517

[9] Aquil, M. A. I., & Ishak, W. H. W. (2020). Predicting software defects using machine learning techniques. International Journal of Advanced Trends in Computer Science and Engineering, 9(4), 6609–6616. https://doi.org/10.30534/ijatcse/2020/352942020

[10] Iqbal, A., Aftab, S., Ali, U., Nawaz, Z., Sana, L., Ahmad, M., & Husen, A. (2019). Performance analysis of machine learning techniques on software defect prediction using NASA datasets. International Journal of Advanced Computer Science and Applications, 10(5), 300–308. https://doi.org/10.14569/IJACSA.2019.0100538