Proceedings of the International Conference on
Formal Methods and Foundations of Artificial Intelligence
Eszterházy Károly Catholic University
Eger, Hungary, June 5–7, 2025
pp. 115–128
DOI: 10.17048/fmfai.2025.115

Detection of God Class and Data Class code smells based on an automatic machine learning tool

Nasraldeen Alnor Adam Khleel, Károly Nehéz

Department of Information Engineering, University of Miskolc,
Miskolc, H-3515, Hungary
{nasr.alnor,aitnehez}@uni-miskolc.hu

Abstract. Code smells are symptoms of poor design or incomplete implementation that can degrade software quality and maintainability. Detecting them is crucial for improving software reliability and guiding refactoring efforts.

Traditional detection methods rely on predefined rules or thresholds, which are inflexible and prone to errors, while modern machine learning approaches require significant expertise and large, balanced datasets.

To address these challenges, we propose an automated code smell detection method using AutoGluon, an AutoML framework that streamlines model selection, hyperparameter tuning, and handling of imbalanced datasets.

To evaluate the effectiveness of the proposed method, experiments were conducted using two code smell datasets: God Class and Data Class. The performance of the method was evaluated using six different metrics: accuracy, precision, recall, F-measure, Matthew's correlation coefficient (MCC), and the area under the receiver operating characteristic curve (AUC).

Additionally, we have also compared our proposed method with state-of-the-art code smell detection methods. Experimental results show that AutoGluon achieves high predictive performance—up to 0.98 accuracy for God Class and 1.00 for Data Class, which often matches or outperforms state-of-the-art methods, demonstrating the potential of AutoGluon for efficient and scalable code smell detection.

Keywords: code smells, software metrics, machine learning, AutoGluon Tool

1. Introduction

Code smells are signs of poor design that go against basic design rules [10, 12]. Finding these issues is important because it helps fix and improve the code, making

the software better and less likely to fail. These problems usually happen when developers are in a rush, use weak designs, or write quick but flawed code [10].

Software metrics play a crucial role in detecting code smells by providing measurable data about the quality and structure of code. These metrics act as objective indicators that help developers assess the health of a software system, allowing them to pinpoint areas that may require refactoring [3, 12, 13]. By analyzing different aspects such as complexity, coupling, cohesion, and size, software metrics help identify potential design flaws that may lead to maintainability issues, poor performance, or increased technical debt. When left unaddressed, these issues can make the codebase harder to understand, modify, and scale, ultimately increasing development costs and the risk of software failures [7, 16].

To enhance the accuracy and efficiency of code smell detection, machine learning techniques can be applied to analyze software metrics and automatically classify code as clean or smelly [4, 5]. Supervised learning algorithms, such as decision trees, random forests, and deep learning models, can be trained on historical datasets containing labeled code samples with identified code smells. These models learn patterns from the software metrics and can predict the presence of code smells in new, unseen code [1, 11].

Traditional methods for detecting code smells rely on rigid rules and static thresholds, which lack adaptability to different projects, require high maintenance, and often ignore the broader context of the code, leading to inaccuracies. These methods also struggle to scale for large or complex software systems. Modern machine learning approaches, while more adaptable, face challenges such as dependency on large labeled datasets, difficulty handling imbalanced data, the need for expert knowledge to tune models, high computational costs, and the risk of overfitting [12, 13, 16].

One powerful tool for automating the detection of code smells using machine learning is AutoGluon. AutoGluon is an open-source AutoML framework that simplifies the process of training and tuning machine learning models. AutoML frameworks provide a helpful solution for both beginners and experts in machine learning. For beginners, they make it easier to build high-performing ML models by handling complex tasks automatically. For experts, AutoML allows them to set up best practices—like choosing models, combining multiple models, tuning settings, preparing data, and splitting datasets—just once. After that, they can apply these steps repeatedly without needing to do everything manually. This helps experts use their knowledge more efficiently across different projects without constant hands-on work.

So, AutoGluon can address the challenges of traditional and modern techniques in detecting code smells by automating model selection, hyperparameter tuning, and handling imbalanced data with built-in techniques like class weighting and resampling, making code smell detection more efficient, scalable, and accessible without requiring extensive expertise or manual adjustments. By using AutoGluon, developers can easily apply machine learning to detect code smells without requiring deep expertise in model selection and hyperparameter tuning.

By leveraging software metrics with machine learning techniques, particularly with AutoGluon, developers can build intelligent, automated systems for detecting code smells[6, 17].

The contributions of this research can be summarized as follows: (i) Development of an automated code smell detection methodology using AutoGluon (AutoML): This study introduces a novel approach that automates model selection, tuning, and evaluation for code smell detection, reducing manual intervention and improving efficiency in software quality assessment. (ii) Comprehensive empirical evaluation on real-world datasets: Various AutoGluon models were evaluated on real software datasets using multiple performance metrics, addressing challenges such as class imbalance and identifying the most impactful software metrics through feature importance analysis. (iii) Facilitation of scalable and reproducible code quality assessments: The research contributes a practical and data-driven methodology that can be integrated into software development workflows, providing consistent, automated, and interpretable code smell detection, thus supporting empirical software engineering research and industrial applications.

2. Related work

Many traditional and modern methods for detecting code smells have been proposed in previous research works [1, 3–5, 11–13, 16].

Arcelli et al. [3] presented an approach for identifying code smells that involves the use of various ML techniques. The results indicate that all techniques performed satisfactorily; however, the imbalanced data adversely affected the performance of certain models.

Mhawish and Gupta [16] presented an approach for predicting code smells using ML techniques and software metrics. The authors utilized datasets obtained from Fontana et al., and their experimental results showed that the accurate prediction of code smells can be significantly facilitated by employing ML techniques.

Cruz et al. [4] conducted an assessment of seven ML algorithms to identify four distinct types of code smells, while also analyzing the influence of software metrics on the detection of code smells. The experimental results found that ML algorithms can perform well in detecting bad code smells, and metrics play a fundamental role in detecting bad code smells.

Dewangan et al. [5] proposed an approach based on six ML algorithms to predict code smells based on four datasets obtained from 74 open-source systems. The proposed approach's effectiveness was assessed using various performance metrics, and two feature selection methods were implemented to improve the accuracy of the predictions. The experimental results showed that their approach achieved high prediction accuracy.

Khleel and Nehéz [1, 11–13] presented various classical machine and advanced learning algorithms with different data balancing methods to detect code smells based on a set of Java projects. The authors examined four datasets related to code smells (God class, data class, feature envy, and long method) and compared

the results using various performance metrics. The experiments demonstrated that the models proposed, along with data balancing methods, exhibited improved performance in detecting code smells. In addition, the results were compared with those of state-of-the-art code smell detection methods. A comparison of the experimental results indicates that their method outperforms state-of-the-art code smell detection methods.

After reviewing previous studies in code smells detection, based on our knowledge, there are no studies that applied AutoML tools for this issue. Therefore, our study focuses on applying a new method for code smell detection, which is based on the AutoGluon Tool.

3. Proposed methodology

In this study, we present a systematic approach for automated code smell detection using AutoGluon. The choice of methodology in this study was guided by the need for scalability, automation, and robustness in code smell detection. So, AutoGluon was selected as the AutoML framework due to its ability to automate feature selection, model tuning, and ensemble construction while effectively handling imbalanced datasets [6], compared with traditional workflows such as Decision Trees, Random Forests, Support Vector Machines, k-Nearest Neighbors, and XGBoost, as well as deep learning models including CNN, LSTM, and GRU that rely on manual thresholding or tuning [3, 12, 13, 16].

Experimental results demonstrate that AutoGluon-based models achieve high accuracy—up to 0.98 for God Class and 1.00 for Data Class, which often outperform or match the state-of-the-art, highlighting the potential of AutoML to deliver accurate, efficient, and reproducible code smell detection suitable for integration into continuous quality assurance processes.

The methodology follows a structured pipeline that includes key stages such as software metrics, data modeling and collection, data preprocessing, feature selection, models building and performance evaluation. Each of these steps plays a crucial role in ensuring the accuracy and effectiveness of the detection model. Figure 1 provides an overview of the proposed methodology, with detailed explanations of each stage outlined in the following sections.

3.1. Software metrics, data modeling and collection

Software metrics are crucial for creating prediction models that help improve software quality by identifying and predicting software defects, such as bugs and code smells [12]. These metrics reveal patterns and signs that are linked to issues in the software [13]. Many studies have shown that these metrics are effective in predicting vulnerabilities in the code [12]. These metrics reveal patterns and signs that are linked to issues in the software [3, 11–13]. Additionally, researchers have demonstrated that software metrics can also be used to evaluate how reusable a piece of software is [14, 19].

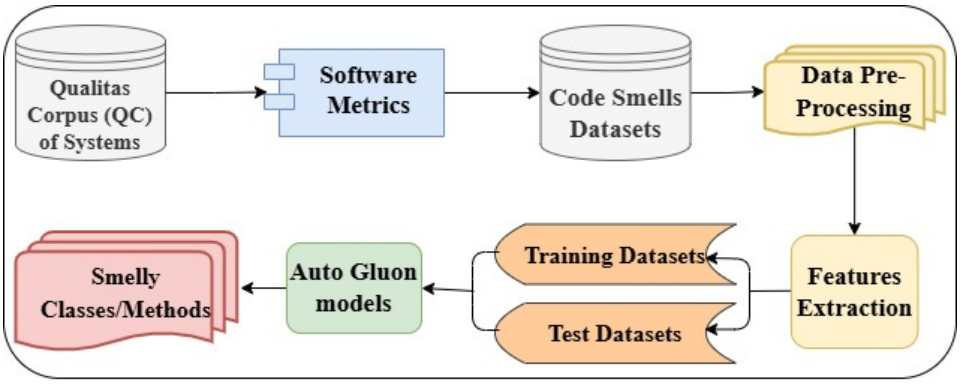

Figure 1. Overview of the Proposed Method for Code Smells Detection.

There are two main types of software metrics: static code metrics and process metrics. Static code metrics are directly extracted from the source code, while process metrics come from the source code management system, based on historical changes in the code over time. Process metrics reflect how the code evolves, including changes in the code itself, the number of changes made, and information about the developers [1, 11].

In various studies, McCabe's Cyclomatic Complexity and Halstead metrics were commonly used as independent variables to analyze code smells. McCabe's Cyclomatic Complexity measures the number of independent control paths in a program, indicating its structural complexity [13]. So, McCabe's Cyclomatic Complexity metrics focus on software quality, including cyclomatic complexity, essential complexity, design complexity, and lines of code [13].

Halstead metrics calculate program length, volume, difficulty, and effort based on operators and operands, reflecting the cognitive complexity of the code. Halstead divides software metrics into three categories: base measures, derived measures, and lines of code [3, 13, 15].

Choosing the right dataset is a key step in machine learning (ML) because classification models work better when the dataset closely matches the problem being studied. In this study, the detecting code smells models use supervised learning, which depends on a large set of software metrics as input data. Having well-structured datasets is important for training ML models effectively and ensuring that the results can be applied to different cases [10, 13].

For our analysis, we used the Qualitas Corpus, a collection of software systems compiled by researchers E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble [20]. This dataset includes many Java-based systems of different sizes and application types, as listed in Table 1, and Table 2 Lists the two specific code smells that we have investigated.

Table 1. Summary of project characteristics [3].

Number of systems	Lines of code	Number of packages	Number of classes
74	6,785,568	3420	51,826

Table 2. Lists the two specific code smells that we have investigated [3].

Code smells	Description	Affected entity
God Class	A God Class is a type of code smell that occurs when a single class takes on too many responsibilities, violating the Single Responsibility Principle.	Class
Data Class	A Data Class is a type of code smell where a class primarily exists to store data without meaningful behavior or logic.	Class

3.2. Data pre-processing and feature selection

Pre-processing the collected data is one of the critical stages before constructing the model. To generate a high-performing model, data quality must be taken into account [12]. Not all data collected is immediately suitable for training and model development. The quality of input features has a significant impact on model performance and, ultimately, prediction outcomes.

Data pre-processing encompasses a series of techniques aimed at improving the dataset by handling noise, removing irrelevant outliers, addressing missing values, and converting feature types to compatible formats. In this study, a clean and validated dataset was used to minimize the need for extensive preprocessing [12, 16].

Normalization was applied to scale numerical feature values to a uniform range (0 to 1), which helps enhance model learning efficiency [11, 16]. Specifically, Min–Max normalization was used. The normalization formula used is described in Equation (3.1).

Feature Selection is another crucial component of the modeling pipeline, as it aims to reduce dimensionality by retaining only the most informative and discriminative features relevant to the target variable [13]. This process eliminates irrelevant, redundant, or noisy variables, which may otherwise decrease model accuracy and increase training time [1, 4].

In this study, we adopted an embedded feature selection method, which is inherently integrated within the model training process. Unlike filter methods that evaluate features independently of any learning model, or wrapper methods that evaluate subsets of features through repetitive training cycles (which can be computationally expensive), embedded methods assess feature importance during model construction. This allows the algorithm to automatically prioritize features that improve performance and ignore those that do not contribute meaningfully.

AutoGluon, the AutoML framework used in this study, performs this embedded

selection as part of its training pipeline, making it both computationally efficient and well-suited for handling high-dimensional software metric datasets. Additionally, feature scaling was applied to ensure that all selected features were on a comparable scale, further supporting consistent learning behavior across models.

$$x_i = (x_i - Xmin)/(Xmax - Xmin) \tag{3.1}$$

Where max(x) and min(x) represent the maximum and minimum value of the attribute x, respectively.

3.3. Models building and evaluation

In this study, model development and evaluation were conducted using AutoGluon, an open-source AutoML framework that automates the machine learning pipeline and supports both novice and advanced users. AutoGluon simplifies tasks such as data preprocessing, feature engineering, model selection, and hyperparameter tuning, enabling rapid development of high-performing models.

It supports a diverse range of algorithms, including LightGBM, RandomForestEntr, LightGBMXT, XGBoost, and deep neural networks, all of which are automatically trained and combined through ensemble techniques such as WeightedEnsemble-L2 [2, 8]. For this work, the dataset was split into 80% for training and validation (handled internally through cross-validation) and 20% for independent testing. AutoGluon applied classification-appropriate defaults such as log loss for optimization and automated selection of learning rates and batch sizes.

The evaluation of the trained models was based on standard metrics derived from the confusion matrix, including accuracy, precision, recall, F1-score, and MCC, which is a statistical metric used to assess the performance of binary classification models. It takes into account true and false positives and negatives and is regarded as a balanced measure, even if the classes are of very different sizes. Additionally, AUC was employed to assess the discriminative ability of the classifiers, illustrating the trade-off between True Positive (TP) and False Positive (FP) rates across different thresholds [9, 18].

As shown in Figure 2, the confusion matrices for each dataset confirm the models' effectiveness in predicting smelly and non-smelly code. The mathematical formulations of these metrics are defined in Equations (3.2) to (3.7).

Overall, AutoGluon's automation and ensemble strategy proved effective for detecting code smells such as God Class and Data Class, delivering robust performance with minimal manual intervention.

$$Accuracy = ((TP + TN))/((TP + FP + FN + TN)) \tag{3.2}$$

$$Precision = (TP)/((TP + FP)) \tag{3.3}$$

$$Recall = (TP)/((TP + FN)) \tag{3.4}$$

$$F - Measure = ((2 * Recall * Precision))/((Recall + Precision)) \tag{3.5}$$

$$\text{MCC} = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{3.6}$$

$$\text{AUC} = \frac{\sum_{\text{ins}_i \in \text{Positive Class}} \text{rank}(\text{ins}_i) - \frac{M(M+1)}{2}}{M \cdot N} \tag{3.7}$$

4. Experimental results and discussion

The experimental evaluation aimed to assess the performance of the proposed AutoGluon-based methodology for detecting code smells using real-world datasets, which are God Class and Data Class, sourced from the Qualitas Corpus. Multiple models were automatically trained and optimized by AutoGluon, including LightGBM, RandomForestEntr, LightGBMXT, XGBoost, and the ensemble model WeightedEnsemble-L2. Each model's performance was evaluated using standard classification metrics such as accuracy, precision, recall, F1-score, MCC, and AUC. As shown in Tables 3 and 4, for the God Class dataset, all five top-performing models (including LightGBM, XGBoost, and LightGBMXT) achieved high levels of accuracy, ranging from 0.97 to 0.98, with corresponding precision and recall values consistently reaching 0.96 or higher. The MCC and AUC values for these models also remained close to or at 1.00, indicating excellent discriminatory power. Similarly, for the Data Class dataset, the models demonstrated outstanding performance, with LightGBM, RandomForestEntr, and WeightedEnsemble-L2 achieving perfect scores (1.00) across all metrics, while XGBoost and LightGBMXT followed closely with slightly lower, yet still impressive, values.

Table 3. Evaluation results for the top five models – God Class dataset.

Models	Performance Measures					
	Accuracy	Precision	Recall	F-measure	MCC	AUC
LightGBM	0.97	0.94	1.00	0.96	0.95	1.00
RandomForestEntr	0.98	0.96	1.00	0.98	0.97	1.00
LightGBMXT	0.98	0.96	1.00	0.98	0.97	0.99
XGBoost	0.98	0.96	1.00	0.98	0.97	0.98
WeightedEnsemble-L2	0.98	0.96	1.00	0.98	0.97	0.99

In terms of training efficiency (Table 5), the AutoGluon-based models achieved consistently strong performance on the God Class dataset. The best results were achieved by LightGBMXT and WeightedEnsemble-L2, both reaching a validation accuracy of 0.985 and a test accuracy of 0.988. These models also had relatively low

Table 4. Evaluation results for the top five models – Data Class dataset.

Models	Performance Measures					
	Accuracy	Precision	Recall	F-measure	MCC	AUC
LightGBM	1.00	1.00	1.00	1.00	1.00	1.00
RandomForestEntr	1.00	1.00	1.00	1.00	1.00	1.00
LightGBMXT	0.98	1.00	0.97	0.98	0.97	0.99
XGBoost	0.97	1.00	0.94	0.97	0.95	1.00
WeightedEnsemble-L2	1.00	1.00	1.00	1.00	1.00	1.00

training times (0.36s and 0.45s, respectively), showing a strong balance between accuracy and efficiency. Notably, XGBoost performed similarly in test accuracy (0.988) but trained faster (0.175s), making it the most efficient model in terms of runtime. For the Data Class dataset, performance was even more impressive. LightGBM and WeightedEnsemble-L2 achieved perfect scores on both validation and test sets (1.000). While LightGBMXT required significantly longer training time (2.48s), it still maintained high accuracy (0.985 validation, 0.988 test).

Table 5. Training Time (seconds) and Models Performance.

Models	God Class Dataset		
	fit-time	score-val	score-test
LightGBM	0.405785	0.970588	0.976190
RandomForestEntr	0.657616	0.970588	0.988095
LightGBMXT	0.361412	0.985294	0.988095
XGBoost	0.175200	0.970588	0.988095
WeightedEnsemble-L2	0.450221	0.985294	0.988095
Models	Data Class Dataset		
	fit-time	score-val	score-test
LightGBM	0.273476	1.000000	1.000000
RandomForestEntr	0.646896	0.985294	1.000000
LightGBMXT	2.481219	0.985294	0.988095
XGBoost	0.180894	0.985294	0.976190
WeightedEnsemble-L2	0.360617	1.000000	1.000000

The confusion matrices (Figure 2) and AUC (Figure 3) visually confirmed the models' ability to accurately distinguish between smelly and non-smelly classes. The confusion matrices underline the efficacy of AutoGluon models in accurately detecting code smells. The minimal or absent false classifications demonstrate the robustness of these models. In addition to accuracy, precision, recall, F-measure, and MCC, we also report AUC scores to assess the discriminative ability of the classifiers. As shown in Tables 3 and 4, AUC values for both God Class and Data Class datasets are consistently high, ranging from 0.98 to 1.00 across all models. These

results confirm that the classifiers are not only highly accurate but also robust in distinguishing smelly from non-smelly classes. This finding is particularly important when dealing with imbalanced datasets, where accuracy alone can sometimes mask poor performance in minority classes. The consistently high AUC values demonstrate that AutoGluon-based models achieve excellent sensitivity-specificity trade-offs, reinforcing their suitability for automated software quality assurance tasks.

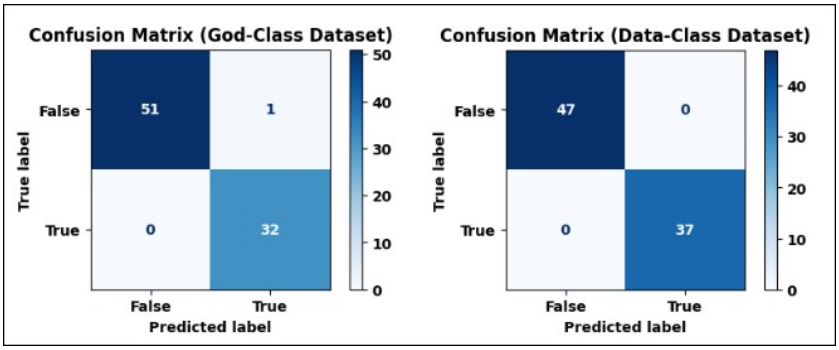

Figure 2. Confusion Matrix for the models over all datasets.

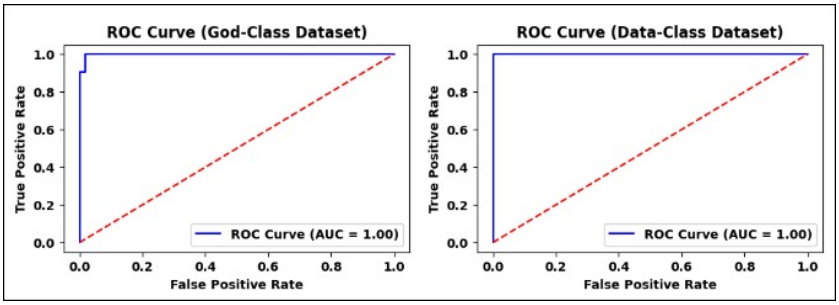

Figure 3. AUC for the models over all datasets.

AutoGluon provides feature importance scores based on permutation shuffling, which quantifies how much the model's performance decreases when each feature is randomly shuffled. Higher importance scores indicate that the model relies more on that feature for making predictions. Therefore, feature importance analysis (Figures 4 and 5) revealed that a subset of software metrics significantly influenced the prediction outcomes, validating the effectiveness of embedded feature selection techniques used in the pipeline.

In comparison with previous state-of-the-art approaches (Table 6). We compared our results with the results obtained in previous studies based on the accuracy. The values marked with "-" indicate that the approaches that did not use data balancing techniques or did not provide results for the performance measure

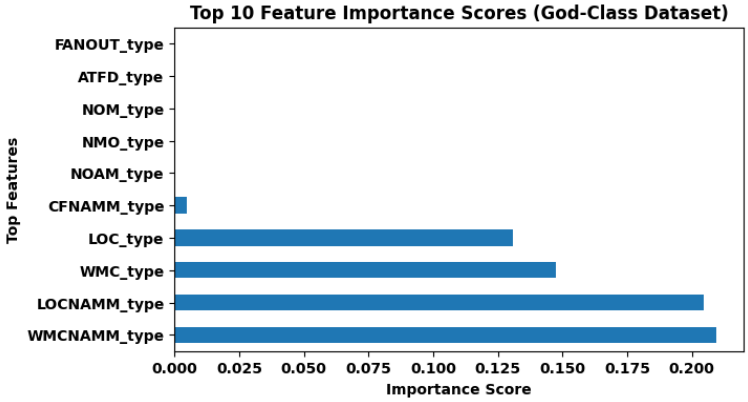

Figure 4. Feature importance scores for the models – God Class Dataset.

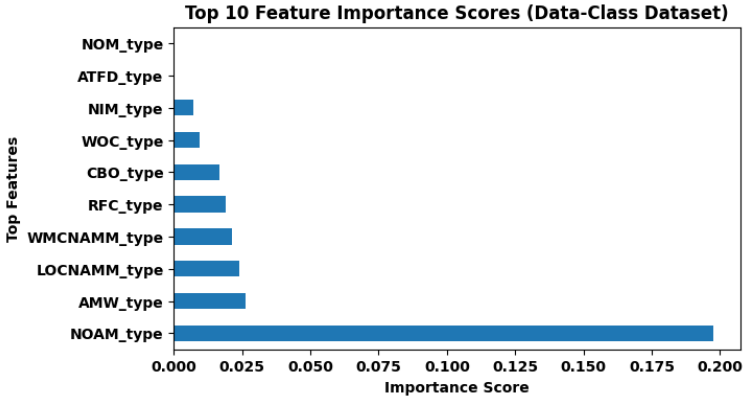

Figure 5. Feature importance scores for the models – Data Class Dataset.

in a particular data set. Additionally, our proposed models are highlighted in bold text. According to the Table, some of the results in the previous studies are better than ours, but in most cases, our method outperforms the other state-of-the-art approaches and provides better predictive performance. These findings collectively demonstrate the practicality, robustness, and accuracy of the AutoGluon framework in automating code smell detection, reducing reliance on human expertise, and offering scalable solutions for software quality assurance. Future studies could focus on testing these models on larger datasets or in real-world scenarios to further validate their effectiveness and generalizability.

Temporal and Long-Term Development Context of God and Data Classes: Code smells such as God Classes and Data Classes rarely emerge instantaneously; in-

Table 6. Comparison of the proposed models with other existing approaches based on accuracy.

Approaches	**Data Balancing Techniques**	**God Class**	**Data Class**
Decision Tree [12]	Random oversampling	0.98	1.00
K-Nearest Neighbors [12]	Random oversampling	0.97	0.96
Support Vector Machine [12]	Random oversampling	0.96	0.97
XGBoost [12]	Random oversampling	0.96	1.00
Multi-Layer Perceptron [12]	Random oversampling	0.97	0.98
Bi-LSTM [13]	Random oversampling	0.96	0.99
GRU [13]	Random oversampling	0.96	0.98
Bi-LSTM [13]	Tomek links	0.96	0.95
GRU [13]	Tomek links	0.96	0.99
Random Forest [3]	–	0.96	0.98
Naive Bayes [3]	–	0.97	0.97
Random Forest [16]	–	–	0.99
CNN [6]	SMOTE	0.96	0.98
XGBoost [2]	SMOTE	0.99	–
SVM [2]	SMOTE	0.97	–
KNN [2]	SMOTE	0.97	–
Random Forest [8]	–	0.69	0.70
Naive Bayes [8]	–	0.82	0.75
SVM [8]	–	0.74	0.83
KNN [8]	–	0.80	0.82
Our proposed LightGBM	–	0.97	1.00
Our proposed RandomForestEntr	–	0.98	1.00
Our proposed LightGBMXT	–	0.98	0.98
Our proposed XGBoost	–	0.98	0.97
Our proposed WeightedEnsemble-L2	–	0.98	1.00

stead, they evolve gradually as projects grow in size and complexity, often persisting across multiple releases. Their longevity reflects not only design flaws but also the developmental pressures and shortcuts taken during software evolution. In this study, the analysis was based on static snapshots of systems from the Qualitas Corpus, so project timelines and release histories were not explicitly captured. Nevertheless, prior research indicates that both God Classes and Data Classes often become long-lived entities, contributing to technical debt across the lifecycle of software projects. Integrating AutoML-based detection into CI/CD pipelines offers a way to address this challenge by enabling continuous monitoring of these smells, allowing teams to identify their emergence early, track their growth, and guide timely refactoring. In this way, automated detection not only classifies existing design problems but also supports sustainable software evolution by mitigating the accumulation of long-lived technical debt.

5. Conclusion

Code smell detection involves identifying patterns in the source code that indicate potential problems with design or implementation, even if the code is working correctly.

This study presented an automated approach to detecting code smells using AutoGluon, an AutoML framework that simplifies the machine learning workflow by automating tasks such as model selection, hyperparameter tuning, feature selection, and data balancing.

By leveraging software metrics and structured data from the Qualitas Corpus, the proposed methodology was evaluated on two prominent code smell types: God Class and Data Class. The experimental results confirmed the effectiveness of the approach, with all models achieving high accuracy, precision, recall, F1-score, MCC, and AUC.

The use of AutoGluon allowed for efficient training and performance optimization without manual tuning, making the methodology accessible to both novice and expert users. The models also showed resilience to imbalanced data and highlighted the impact of key software metrics through feature importance analysis.

Moreover, comparisons with existing state-of-the-art approaches demonstrated that the proposed method either outperformed or matched traditional and deep learning-based techniques, further validating its competitiveness and reliability.

Overall, the integration of AutoML techniques into code smell detection offers a promising pathway toward automated, interpretable, and scalable software quality assessment. Future work may include expanding the methodology to detect additional types of code smells, applying the approach to larger and more diverse software projects, and exploring hybrid AutoML strategies for further performance gains.

Acknowledgements. The authors gratefully acknowledge the financial assistance from the Institute of Information Science, Faculty of Mechanical Engineering and Informatics, University of Miskolc.

References

[1] N. A. Adam Khleel, K. Nehéz: *Optimizing LSTM for Code Smell Detection: The Role of Data Balancing.* Infocommunications Journal 16.3 (2024), DOI: 10.36244/ICJ.2024.3.5.

[2] K. Alkharabsheh, S. Alawadi, V. R. Kebande, Y. Crespo, M. Fernández-Delgado, J. A. Taboada: *A comparison of machine learning algorithms on design smell detection using balanced and imbalanced dataset: A study of God class*, Information and Software Technology 143 (2022), p. 106736, DOI: 10.1016/j.infsof.2021.106736.

[3] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, A. Marino: *Comparing and experimenting machine learning techniques for code smell detection*, Empirical Software Engineering 21.3 (2016), pp. 1143–1191, DOI: 10.1007/s10664-015-9378-4.

[4] D. Cruz, A. Santana, E. Figueiredo: *Detecting bad smells with machine learning algorithms: an empirical study*, in: Proceedings of the 3rd international conference on technical debt, 2020, pp. 31–40, doi: 10.1145/3387906.3388618.

[5] S. Dewangan, R. S. Rao, A. Mishra, M. Gupta: *A novel approach for code smell detection: an empirical study*, IEEE Access 9 (2021), pp. 162869–162883, doi: 10.1109/ACCESS.2021.3133810.

[6] N. Erickson, J. Mueller, A. Shirkov, H. Zhang, P. Larroy, M. Li, A. Smola: *Autogluon-tabular: Robust and accurate automl for structured data*, arXiv preprint arXiv:2003.06505 (2020), doi: arXivpreprintarXiv:2003.06505.

[7] T. Guggulothu, S. A. Moiz: *Code smell detection using multi-label classification approach*, Software Quality Journal 28.3 (2020), pp. 1063–1086, doi: 10.1007/s11219-020-09498-y.

[8] S. Jain, A. Saha: *Rank-based univariate feature selection methods on machine learning classifiers for code smell detection. Evol. Intell. 15 (1), 609–638*, 2022, doi: 10.1007/s12065-020-00536-z.

[9] S. Jain, A. Saha: *Improving performance with hybrid feature selection and ensemble machine learning techniques for code smell detection*, Science of Computer Programming 212 (2021), p. 102713, doi: 10.1016/j.scico.2021.102713.

[10] A. Kaur, S. Jain, S. Goel, G. Dhiman: *A review on machine-learning based code smell detection techniques in object-oriented software system (s)*, Recent Advances in Electrical & Electronic Engineering (Formerly Recent Patents on Electrical & Electronic Engineering) 14.3 (2021), pp. 290–303, doi: 10.2174/2352096513999200922125839.

[11] N. A. A. Khleel, K. Nehéz: *Deep convolutional neural network model for bad code smells detection based on oversampling method*, Indonesian Journal of Electrical Engineering and Computer Science 26.3 (2022), pp. 1725–1735, doi: 10.11591/ijeecs.v26.i3.pp1725-1735.

[12] N. A. A. Khleel, K. Nehéz: *Detection of code smells using machine learning techniques combined with data-balancing methods*, International Journal of Advances in Intelligent Informatics 9.3 (2023), pp. 402–417, doi: 10.26555/ijain.v9i3.981.

[13] N. A. A. Khleel, K. Nehéz: *Improving accuracy of code smells detection using machine learning with data balancing techniques*, The Journal of Supercomputing 80.14 (2024), pp. 21048–21093, doi: 10.1007/s11227-024-06265-9.

[14] N. Medeiros, N. Ivaki, P. Costa, M. Vieira: *Vulnerable code detection using software metrics and machine learning*, IEEE Access 8 (2020), pp. 219174–219198, doi: 10.1109/ACCESS.2020.3041181.

[15] B. Mehboob, C. Y. Chong, S. P. Lee, J. M. Y. Lim: *Reusability affecting factors and software metrics for reusability: A systematic literature review*, Software: Practice and Experience 51.6 (2021), pp. 1416–1458, doi: 10.1002/spe.2961.

[16] M. Y. Mhawish, M. Gupta: *Predicting code smells and analysis of predictions: using machine learning techniques and software metrics*, Journal of Computer Science and Technology 35.6 (2020), pp. 1428–1445, doi: 10.1007/s11390-020-0323-7.

[17] L. M. Paladino, A. Hughes, A. Perera, O. Topsakal, T. C. Akinci: *Evaluating the performance of automated machine learning (AutoML) tools for heart disease diagnosis and prediction*, Ai 4.4 (2023), pp. 1036–1058, doi: 10.3390/ai4040053.

[18] F. Pecorelli, D. Di Nucci, C. De Roover, A. De Lucia: *A large empirical assessment of the role of data balancing in machine-learning-based code smell detection*, Journal of Systems and Software 169 (2020), p. 110693, doi: 10.1016/j.jss.2020.110693.

[19] K. Z. Sultana, V. Anu, T.-Y. Chong: *Using software metrics for predicting vulnerable classes and methods in Java projects: A machine learning approach*, Journal of Software: Evolution and Process 33.3 (2021), e2303, doi: 10.1002/smr.2303.

[20] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, J. Noble: *The qualitas corpus: A curated collection of java code for empirical studies*, in: 2010 Asia pacific software engineering conference, 2010, pp. 336–345, doi: 10.1109/APSEC.2010.46.