CodePure : Visual Studio Code Extension For Code Smells Detection

1st Mark Mounir Adly
Computer Science
Misr International University
Cairo, Egypt
mark2107794@miuegypt.edu.eg

2nd Mohammad Hesham Elsayed
Computer Science
Misr International University
Cairo, Egypt
mohammad2109652@miuegypt.edu.eg

3rd Omar Hosny Badrawy
Computer Science
Misr International University
Cairo, Egypt
omar2108721@miuegypt.edu.eg

4th Zeyad Abdelnasser Elzayaty
Computer Science
Misr International University
Cairo, Egypt
zeyad2103469@miuegypt.edu.eg

5th Sarah Nabil
Computer Science
Misr International University
Cairo, Egypt
sara.abdullah@miuegypt.edu.eg

6th Saja Saadoun
Computer Science
Misr International University
Cairo, Egypt
sajatarek2001@gmail.com

***Abstract*—Code smells are superficial signs of deeper issues that may exist in software codebases and have a detrimental impact on readability and maintainability. Early detection enhances the quality of software. In this paper, we introduce CodePure, an addon for Visual Studio Code that detects code smells in real time across a variety of programming languages. By computing software metrics (such Lines of Code, Cyclomatic Complexity, and Weighted Methods per Class) and submitting them to a trained model housed on a web server for smell classification, CodePure combines static code analysis and machine learning. To find the most accurate detection method, experiments were carried out on labeled datasets using a variety of machine learning models, yielding encouraging results across important smell classes. The development process, experiments, and their results are covered in this publication, and the integration of the detection engine into the VS Code environment.**

***Index Terms*—Code smell detection, software metrics, machine learning, Visual Studio Code extension, static code analysis, software quality.**

I. Introduction

Detecting code smells has been a significant focus in both academic research and industry due to its importance in ensuring maintainable and high-quality software. Code smells are not bugs per se, but they indicate deeper design flaws that may hinder software maintainability, scalability, and overall quality over time. The concept of "code smells" was popularized by Martin Fowler in 1999 [6], describing them as symptoms of suboptimal code structure that suggest the need for refactoring to improve long-term code health.

Over the years, considerable effort has been dedicated to identifying and categorizing common code smells, such as God Classes, Long Methods, and Duplicated Code [11], [13]. These patterns can significantly degrade code readability, complicate maintenance efforts, and accumulate technical debt [15], [18]. This issue becomes increasingly critical in the context of modern, large-scale software systems, where rapid iteration, evolving requirements, and tight deadlines often lead to compromises in code quality.

Research has shown that code smells tend to emerge gradually throughout the software lifecycle, often as a result of shortcuts taken during maintenance or development under time pressure [20]. Left unaddressed, these smells contribute to technical debt, increasing future maintenance costs and lowering system reliability [14], [18]. Early detection of these design issues is vital to reduce future effort and ensure the long-term sustainability of software products.

While many tools have been developed to aid in code smell detection — including static analysis utilities and machine learning-based approaches — most existing solutions still face notable limitations. Many rely on rigid rule sets or manually tuned thresholds, which can result in high false positive rates or an inability to detect more nuanced or context-dependent smells. Despite progress in applying machine learning, such as bi-level optimization and data-driven techniques, current methods often struggle with generalization and precision.

This creates a growing need for intelligent, automated tools capable of adapting to various codebases and detecting complex code smells accurately. Our proposed tool, CodePure, addresses this gap by integrating machine learning techniques within a VS Code extension, providing real-time feedback to developers. CodePure automatically analyzes code, computes software metrics, and leverages a trained ML model deployed on AWS SageMaker to detect potential code smells. The results are visualized through an intuitive dashboard and code annotations within the editor, helping developers take immediate action to improve code quality.

The remainder of this paper is structured as follows: Related Works, Methodology, Implementation, Dataset, Experiments and Results, and Conclusion.

II. Related Work

The paper "Detecting Code Smells using Machine Learning Techniques: Are We There Yet?"**[2]** addressed the main problem of subjectivity and inconsistency in the results produced by traditional code smell detection tools. The researchers aim to explore whether machine learning (ML) techniques can offer a more objective and reliable approach to detecting code smells. To tackle this problem, the authors conducted a replication study by using a different dataset configuration to evaluate the capabilities of ML techniques under more realistic conditions. They contributed by expanding the experimental setup to include datasets containing instances of multiple types of smells, rather than focusing on single-smell datasets as done in prior studies. The dataset used comprised 74 Java software systems, initially developed for the Qualitas Corpus project. The primary result of the study was that, under a more realistic dataset setting, the performance of ML techniques dropped significantly—up to 90% in F-measure compared to the previous study—suggesting that the problem of using ML for accurate code smell detection remains unsolved. One critical observation is that the study successfully highlights the shortcomings of ML in real-world scenarios, but it would have been more insightful if the authors proposed potential improvements or hybrid approaches to overcome these limitations.

The paper titled "Automatic detection of bad smells in code: An experimental assessment"**[4]** addresses the inconsistency in code smell detection across different automatic tools, a problem that makes it challenging for developers to rely on these tools for software maintenance. The researchers contributed by experimentally evaluating four widely-used detection tools on various versions of the GanttProject software, assessing how each tool performs in identifying code smells. Using this open-source dataset, they found significant variations between the tools, with minimal agreement on detected smells, thus showing that these tools provide inconsistent results. While the paper effectively highlights the limitations in current detection technologies, it falls short in proposing specific solutions to improve the precision and agreement between these tools. Additionally, the dataset used is limited in scope, making the results less generalizable.

The paper "On the Evaluation of Code Smells and Detection Tools"**[17]** tackles the problem of inconsistencies and lack of agreement between different code smell detection tools, which complicates the trustworthiness of these tools for software developers. The authors conducted an empirical assessment of four detection tools (inFusion, JDeodorant, PMD, and JSpIRIT) using two open-source projects, MobileMedia and Health Watcher, to evaluate their precision, recall, and agreement in detecting God Class, God Method, and Feature Envy smells. The study found wide variations in performance, with recall ranging from 0% to 100%, suggesting that the tools are unreliable in their current form. Although the paper does well in pointing out discrepancies between tools, it offers little in terms of actionable recommendations for improving their consistency.

The paper titled "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis"**[1]** addresses the limitations of traditional heuristic-based code smell detection methods, which are often subjective and inconsistent, by investigating the use of machine learning (ML) techniques to enhance detection accuracy. The researchers conducted a systematic literature review of studies published between 2000 and 2017, focusing on the effectiveness of various ML models such as Decision Trees and Support Vector Machines, using datasets derived from these studies. Their meta-analysis revealed that JRip and Random Forest were the most effective classifiers, but also identified key gaps such as that a wrong selection of the machine learning algorithm to use impacts the performance of code smell prediction models by up to 40% in terms of F-Measure and a lack of standardized datasets. While the paper provides a comprehensive review, it lacks concrete suggestions for overcoming these challenges and does not incorporate more recent advancements, limiting the relevance of its conclusions.

The paper titled "Detecting code smells using industry-relevant data"**[12]** addresses the problem of detecting code smells, which are patterns in software code associated with increased defects and higher maintenance effort, typically identified using predefined, arbitrary rules. These rules often lack industrial relevance, leading to unreliable conclusions. To solve this, the researchers evaluated the performance of eight machine learning algorithms on a dataset (MLCQ) built from industry-relevant projects reviewed by experienced developers. The dataset contained over 14,000 reviews of code snippets labeled with four types of code smells: Blob, Data Class, Feature Envy, and Long Method. The study found that the Random Forest and Flexible Discriminant Analysis algorithms performed the best, with performance varying depending on the type of smell. The highest median MCC value was 0.81 for detecting Long Method, while the lowest was 0.31 for Feature Envy. Although the study provides valuable insights into using machine learning for code smell detection, it is limited by the lack of diversity in predictors and the reliance on proprietary metrics, which could hinder reproducibility and generalization in future research. Moreover, the performance differences between algorithms were minimal, indicating the need for new predictive features to improve detection.

The paper titled "An Experience Report on Using Code Smells Detection Tools" **[5]** explores the effectiveness of various code smell detection tools by comparing their results on multiple versions of the GanttProject software. The main problem addressed is the inconsistency and ambiguity in detecting code smells using different tools, which complicates code quality evaluation and maintenance. The researchers conducted an experimental evaluation using five popular tools: JDeodorant, Stench Blossom, InFusion, iPlasma, and PMD, and compared their effectiveness on detecting code smells like God Class, Long Method, and Feature Envy. The dataset consisted of multiple versions of GanttProject, with detailed comparisons based on metrics like number of packages, classes, and methods. The main result showed significant differences in the detection capabilities of these tools, with some tools like JDeodorant and Stench Blossom performing inconsistently across different releases of the same software. The paper emphasizes that there is no single "best" tool for all code smells, as results vary based on the detection rules and metric thresholds.

The paper titled "A Hybrid Approach to Detect Code Smells using Deep Learning" **[8]** addresses the problem of detecting code smells, which are structural weaknesses in software code that indicate deeper design issues, leading to high maintenance costs and reduced software quality. To tackle this, the authors propose a hybrid method combining an unsupervised deep auto-encoder for dimensionality reduction with a supervised artificial neural network (ANN) for classification. The approach is tested on four types of code smells—God Class, Data Class, Feature Envy, and Long Method—using datasets derived from 74 open-source projects. The results demonstrate high accuracy, with precision and recall values exceeding 96%, highlighting the importance of feature reduction for enhancing detection performance.

The paper titled "Code Smell Detection Using Multi-Label Classification Approach"**[7]** addresses the challenge of detecting multiple overlapping code smells in software, which are critical indicators of design flaws impacting maintainability. Traditional approaches focus on single-label detection and fail to consider the correlation among smells. To tackle this, the researchers propose a multi-label classification (MLC) approach, enabling simultaneous detection of multiple code smells while accounting for their correlation. They created a multi-label dataset by combining method-level code smells, "Long Method" and "Feature Envy," using data from 74 open-source Java projects, balancing it to address disparity instances that previously reduced classifier performance. Experimental results showed that incorporating correlation through MLC techniques, particularly Classifier Chains and Label Combination, improved detection accuracy to over 96%, significantly outperforming single-label approaches. The findings highlight the potential of MLC for realistic, accurate, and practical code smell detection, offering a valuable tool for prioritizing refactoring tasks in software development.

The paper titled "Automatic Detection of Bad Smells from Code Changes" **[9]** addresses the problem of maintaining code quality by detecting "bad smells," which are indicators of poor design choices that affect code maintainability and readability. The authors propose a lightweight, real-time approach to automatically detect these smells as code changes are implemented using an Eclipse plug-in tool called JFly. The tool monitors code changes, calculates relevant software metrics, and applies predefined detection rules to identify nine different bad smells, such as Long Parameter List, Brain Method, and Lazy Class. The dataset used includes multiple Java projects, such as Payments Auditing and Smart Payment Router for performance testing, and three industrial projects (Payment Gateway, Settlement, and Fees) for correctness evaluation. The results show that JFly performs efficiently, with an average detection time of less than three seconds, and achieves high precision and recall values in detecting bad smells, especially for Long Parameter List and Brain Method. However, a key limitation is that the approach does not account for larger codebases and might struggle with complex dependencies between classes, which led to some false positives and false negatives during real-world testing.

The paper titled "Java Quality Assurance by Detecting Code Smells"**[19]** addresses the problem of inefficient and labor-intensive traditional software inspection methods that fail to scale for large systems. The authors propose automating the detection and visualization of code smells to make software quality assurance more practical and efficient. They contributed by developing a tool named jCOSMO, a prototype code smell detector and visualizer for Java code, which automates the inspection process by identifying design and coding issues such as "large classes" or "typecasts" and visualizing these smells to assist developers in understanding and improving code quality. The dataset used for evaluation was the source code of a Java-based system named CHARTOON, consisting of 147 classes and 46,000 lines of code. The main results show that jCOSMO successfully identified code smells and provided useful visualizations that enabled maintainers to pinpoint problematic areas and suggest refactoring. However, a limitation of the paper is that it only presents a qualitative evaluation through a single case study without quantitative performance metrics, making it difficult to generalize the tool's effectiveness across various software projects. Additionally, while the tool provides valuable visualization capabilities, the authors could have extended the work to automate more advanced refactoring suggestions based on detected smells.

III. Methodology

A. Proposed System Overview

The proposed system, **CodePure**, is developed as an intelligent Visual Studio Code (VS Code) extension designed to detect code smells through a combination of static code analysis and machine learning techniques. The system architecture, depicted in Fig. 1, integrates multiple modular components including metric extraction, machine learning prediction, and developer feedback.

The process begins with a code listener embedded in VS Code, which ensures that the analyzed source file meets three criteria: (1) it belongs to a supported programming language, (2) it contains no syntax errors, and (3) the backend server is operational. Upon successful validation, the code parsing module extracts relevant components (such as classes, methods, and control structures) and stores them in JSON format.

Following extraction, the system computes key software metrics, including Lines of Code (LOC), Depth of Inheritance (DIT) and Weighted Methods per Class (WMC), and other important metrics using well-defined calculation rules. These metrics are encapsulated in a JSON file and transmitted via HTTP request to the machine learning backend.

The backend leverages a cloud-based architecture, utilizing an AWS API Gateway to forward requests to an AWS Lambda function. This Lambda function invokes the machine learning model hosted on AWS SageMaker. Prior to deployment, the model was trained using a robust pipeline that includes data preprocessing (handling missing values, removing duplicates, addressing class imbalance with SMOTE), feature selection, and the evaluation of multiple classifiers to select the best-performing model.

Once predictions are generated, the results are delivered through AWS Simple Queue Service (SQS) to ensure reliable, asynchronous communication with the extension. The VS Code extension retrieves the results, updates its dashboard with both the calculated metrics and the detected code smells, and enables developers to provide feedback, thereby closing the loop between automated analysis and human insight.

To ensure robust machine learning performance, the proposed system employs a structured pipeline encompassing data preprocessing and model training. The initial phase involves comprehensive data preparation, including strategic sampling and feature extraction to refine the dataset and enhance representational balance. These steps are designed to mitigate common data issues and to ensure the model is trained on a well-distributed and informative feature space. The selection of features is guided by their predictive significance, contributing to the overall accuracy and reliability of the system.

Following data cleaning and feature selection, these features serve as input to several candidate machine learning models. The system experiments with multiple algorithms — including Random Forest, Gradient Boosting, XGBoost, Logistic Regression, and K-Nearest Neighbors (KNN) — each wrapped in a **MultiOutputClassifier** to handle the multi-label nature of code smell detection.

Model evaluation is conducted using stratified cross-validation to ensure fair performance estimation across all labels. Based on accuracy and generalization performance, the K-Nearest Neighbors (KNN) model is selected as the best-performing classifier. This final model is deployed on AWS SageMaker, allowing the extension to send extracted metrics via HTTP requests and receive real-time code smell predictions efficiently.

B. *Software Development*

- Visual Studio Code IDE: Visual Studio Code (VS Code) was selected as the development platform for the CodePure extension due to its popularity, extensive documentation, and wide range of supportive tools and libraries. According to recent surveys, VS Code is currently the most widely used IDE, making it an ideal choice for developing extensions that integrate seamlessly with modern development workflows.
- Tree-sitter Library: The Tree-sitter library is employed for code parsing and analysis. It allows precise extraction of user code components, enabling the system to scan source files and customize detection targets. This flexibility is essential for calculating the software metrics used both in the dataset preparation and during live analysis in the extension.
- k-Nearest Neighbors Model: The machine learning backend relies on the k-Nearest Neighbors (KNN) classifier. KNN is a simple yet effective pattern recognition and classification algorithm where a sample is classified by a majority vote among its k nearest neighbors. The system was trained and tested on a large dataset using an 80%–20% split to achieve high predictive accuracy. The final trained model is deployed on AWS SageMaker, enabling real-time prediction responses for the extension.

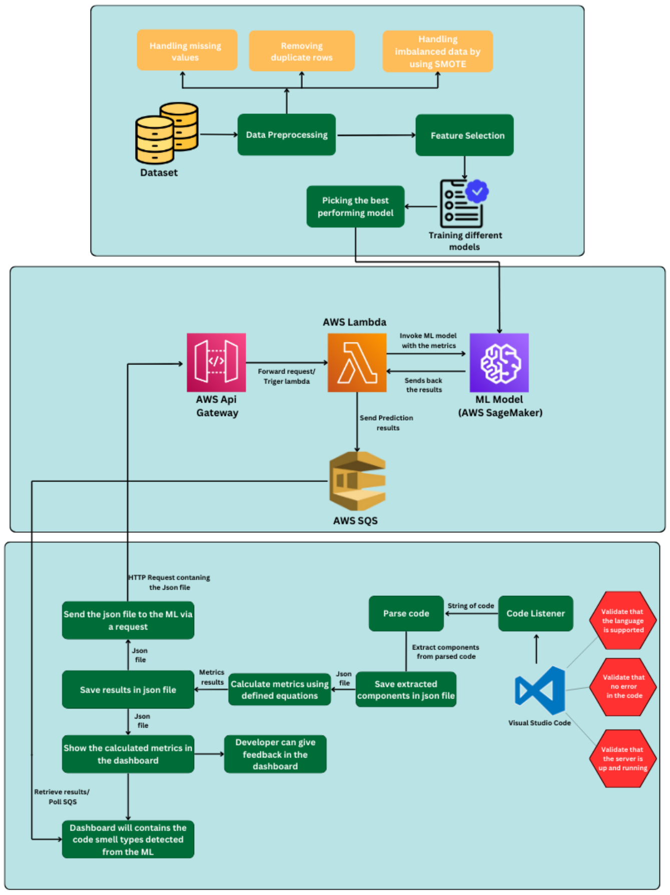
Handling missing values
Removing duplicate rows
Handling imbalanced data by using SMOTE
Dataset
Data Preprocessing
Feature Selection
Picking the best performing model
Training different models
AWS Lambda
Invoke ML model with the metrics
Forward request/ Triger lambda
Sends back the results
AWS Api Gateway
ML Model (AWS SageMaker)
Send Prediction results
AWS SQS
HTTP Request contaning the Json file
Send the json file to the ML via a request
Parse code
String of code
Code Listener
Validate that the language is supported
Extract components from parsed code
Json file
Metrics results
Save results in json file
Calculate metrics using defined equations
Json file
Save extracted components in json file
Validate that no error in the code
Visual Studio Code
Json file
Show the calculated metrics in the dashboard
Developer can give feedback in the dashboard
Validate that the server is up and running
Retrieve results/ Poll SQS
Dashboard will contains the code smell types detected from the ML

Fig. 1. System overview of the proposed CodePure approach.

IV. IMPLEMENTATION

A. *A. VS Code IDE Extension*

The CodePure system is implemented as a Visual Studio Code extension, designed to integrate seamlessly into the developer's workflow. The extension offers an intuitive graphical interface that allows users to detect code smells easily and efficiently. As shown in Fig. 2, users can simply open their code project in the IDE, and the extension will automatically analyze the files upon saving or on-demand. The system works in the background without interrupting the user's development flow, providing notifications or visual highlights when a code smell is detected.

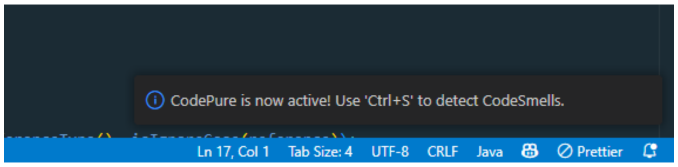
CodePure is now active! Use 'Ctrl+S' to detect CodeSmells.

Fig. 2. extension ready for detection

B. User's Code Components Extraction

Once the extension is activated, it begins by parsing the user's source code to extract all relevant components. This includes classes, methods, functions, and other structural elements. Using the Tree-sitter library, the system generates a structured representation of the code, capturing the key components needed for analysis. The extracted components are stored in a JSON file format as shown in fig 3, which serves as the input for subsequent metric calculations.

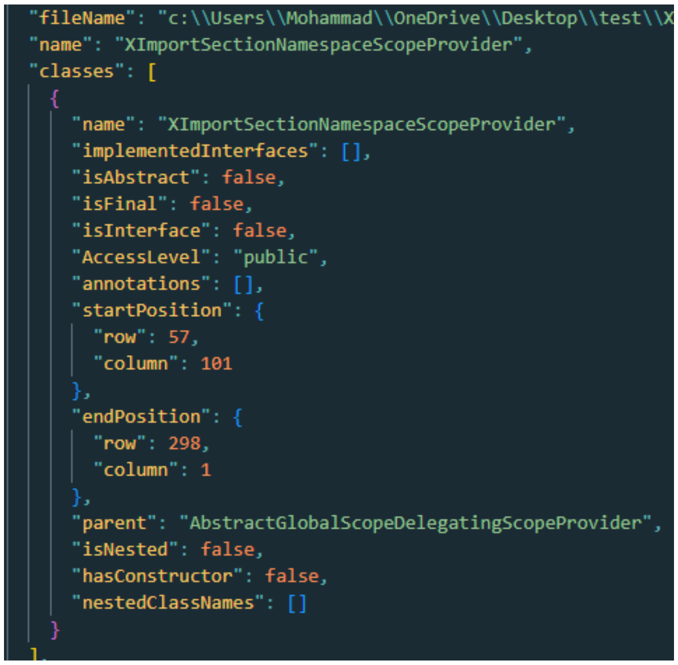
```
"fileName": "c:\\Users\\Mohammad\\OneDrive\\Desktop\\test\\X
"name": "XImportSectionNamespaceScopeProvider",
"classes": [
  {
    "name": "XImportSectionNamespaceScopeProvider",
    "implementedInterfaces": [],
    "isAbstract": false,
    "isFinal": false,
    "isInterface": false,
    "AccessLevel": "public",
    "annotations": [],
    "startPosition": {
      "row": 57,
      "column": 101
    },
    "endPosition": {
      "row": 298,
      "column": 1
    },
    "parent": "AbstractGlobalScopeDelegatingScopeProvider",
    "isNested": false,
    "hasConstructor": false,
    "nestedClassNames": []
  }
],
```

Fig. 3. classes extracted

C. Metrics Calculation

With the code components extracted, the system proceeds to calculate a set of predefined software metrics. These include lines of code (LOC), cyclomatic complexity (CC), weighted methods per class (WMC), coupling between objects (CBO), and others. The calculated metrics are saved in a separate JSON file as shown in fig 4, which represents the feature set used by the machine learning model to predict potential code smells and to be viewed in our dashboard. This modular separation ensures that the system can be easily updated or extended with additional metrics in the future.

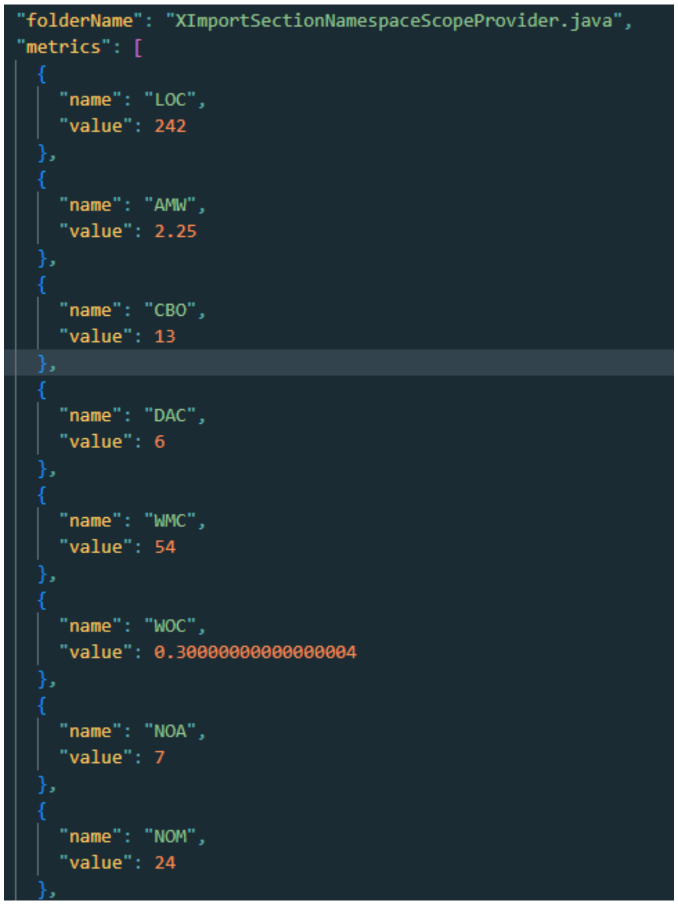
```
"folderName": "XImportSectionNamespaceScopeProvider.java",
"metrics": [
  {
    "name": "LOC",
    "value": 242
  },
  {
    "name": "AMW",
    "value": 2.25
  },
  {
    "name": "CBO",
    "value": 13
  },
  {
    "name": "DAC",
    "value": 6
  },
  {
    "name": "WMC",
    "value": 54
  },
  {
    "name": "WOC",
    "value": 0.30000000000000004
  },
  {
    "name": "NOA",
    "value": 7
  },
  {
    "name": "NOM",
    "value": 24
  },
```

Fig. 4. metrics calculated

D. Amazon Web Services

The collected metrics are transmitted to an AWS SageMaker endpoint, where the trained machine learning model is hosted. The model processes the incoming feature set and returns a prediction indicating whether a code smell is present and, if so, its type. This cloud-based deployment enables scalability and offloads the computational burden from the user's local machine. The extension then displays the prediction results directly within the IDE, providing actionable feedback to the developer in real time.

V. DATASET

Our project utilizes a multi-label dataset designed for detecting code smells. This datasets is crucial for training and evaluating machine learning models aimed at identifying problematic code structures that may affect maintainability and readability. The dataset is concerned with classes, each annotated with multiple code smell labels. These labels indicate the presence of different code smells such as ”God Class,” ”Brain Class,” ”Data Class,” and others. Additionally, the dataset contains several software metrics, including Weighted Methods per Class (WMC), Tight Class Cohesion (TCC), and other attributes.

The dataset underwent several preprocessing operations to optimize model performance. Sampling was conducted using

SMOTE to address class imbalance by generating synthetic examples for underrepresented classes. A hybrid feature selection technique was then applied, combining filter-based and embedded methods to identify the most relevant features. Initially, features were filtered based on their ROC-AUC scores to assess individual predictive power. This was followed by embedded feature selection using Lasso regression, which further refined the feature set by penalizing less informative attributes.

Key software metrics were selected for feature extraction Percentage Number of Added Services (PNAS), Depth of Inheritance Tree (DIT), Number of Added Services (NAS), Number of Accessor Methods (NOAM), Tight Class Cohesion (TCC), Data Abstraction Coupling (DAC), Weight of a Class (WOC), Number of Attributes (NOA), Foreign Data Provider (FDP), Number of Methods (NOM), Average Method Weight (AMW), and Coupling Between Objects (CBO). These metrics were chosen as they have the most significant influence on the prediction results.

VI. Experiments and Results

To evaluate the accuracy and robustness of our model's predictions, we conducted extensive testing using the CodePure extension on a diverse set of medium-sized GitHub repositories sourced from our dataset**[16]**.

The results of these experiments were encouraging for certain smell types and revealed areas for further improvement in others. Data Class detection showed consistently high accuracy, exceeding 95% across nearly all tested repositories. This suggests that the features and heuristics used by CodePure are particularly effective in identifying classes that primarily store data without significant behavior. In contrast, the accuracy for detecting God Classes—classes that tend to centralize too much functionality—was slightly lower, falling between 80% and 85%. This decline reflects the more nuanced nature of God Classes, which can vary significantly depending on the context and design patterns used. The model struggled more with Schizophrenic Class detection, achieving only 55% to 60% accuracy. This is likely due to the complex and often subjective nature of identifying classes that exhibit inconsistent responsibilities or behaviors.

To further validate our model, we extended our testing to include two new external datasets **[3]**, **[10]** that were not part of the original training or tuning process. These datasets introduced new coding styles and domain contexts, allowing us to assess the generalizability of the model. On these datasets, the model achieved 78% accuracy in determining whether a code smell was present or not, and 63% accuracy in classifying the specific type of code smell. While these figures indicate solid performance in unfamiliar environments, they also highlight the need for continued refinement, particularly in improving the classification of more ambiguous or borderline cases. Overall, the experiments demonstrate that CodePure is a promising tool for automated code smell detection, with strong performance in certain areas and clear opportunities for enhancement in others.

VII. Conclusion

In this research, we presented **CodePure**, a system designed for automated code smell detection within Visual Studio Code. Through our **Related Works** section, we explored the landscape of existing tools and approaches, highlighting gaps in automation, multi-language support, and integration with modern development environments. Our **Methodology** outlined a clear pipeline starting from data preprocessing, where we cleaned the dataset using techniques like SMOTE for balancing, and carefully selected twelve critical software metrics to ensure meaningful feature extraction.

In the **Implementation** phase, we built a Visual Studio Code extension leveraging the Tree-sitter library for accurate extraction of code components like classes and methods, followed by metric calculation and JSON serialization. The backend machine learning model K-Nearest Neighbor classifier—was trained on a robust, multi-label dataset and deployed on AWS SageMaker, enabling seamless communication between the extension and the prediction engine.

Our **Dataset** section provided insights into the sources, preparation, and characteristics of the data used, while the *Experiments and Results* demonstrated the performance of various models, where KNN outperformed others like Random Forest and XGBoost in accuracy and stability under multi-label classification settings.

Overall, CodePure represents a significant step forward in providing developers with real-time, automated, and accurate feedback on code quality within their everyday development environment.

Recommendations: For future work, we recommend expanding the system's support to additional programming languages beyond Java to increase its adoption. Incorporating more advanced models such as deep learning or ensemble methods may enhance the detection of complex code smells. Additionally, integrating continuous learning mechanisms that adapt to evolving project-specific patterns and offering personalized recommendations to developers could further improve the practical impact of CodePure.

References

[1] Muhammad Ilyas Azeem, Fabio Palomba, Lin Shi, and Qing Wang. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. *Information and Software Technology*, 108:115–138, 2019.

[2] Dario Di Nucci, Fabio Palomba, Damian A Tamburri, Alexander Serebrenik, and Andrea De Lucia. Detecting code smells using machine learning techniques: Are we there yet? In *2018 ieee 25th international conference on software analysis, evolution and reengineering (saner)*, pages 612–621. IEEE, 2018.

[3] Ehsan Esmaili, Morteza Zakeri, and Saeed Parsa. Code smells and quality attributes dataset. Figshare, August 2023. Accessed: 2025-05-06.

[4] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. Automatic detection of bad smells in code: An

experimental assessment. *J. Object Technol.*, 11(2):5–1, 2012.

[5] Francesca Arcelli Fontana, Elia Mariani, Andrea Mornioli, Raul Sormani, and Alberto Tonello. An experience report on using code smells detection tools. In *2011 IEEE fourth international conference on software testing, verification and validation workshops*, pages 450–457. IEEE, 2011.

[6] Martin Fowler. *Refactoring: improving the design of existing code*. Number 1. Addison-Wesley Professional, 2018.

[7] Thirupathi Guggulothu and Salman Abdul Moiz. Code smell detection using multi-label classification approach. *Software Quality Journal*, 28(3):1063–1086, 2020.

[8] Mouna Hadj-Kacem and Nadia Bouassida. A hybrid approach to detect code smells using deep learning. In *ENASE*, pages 137–146, 2018.

[9] Maen Hammad and Asma Labadi. Automatic detection of bad smells from code changes. *International Review on Computers and Software*, 11(11):1016–1027, 2016.

[10] Vladimir Kachanov and Sergey Markov. Trusted code smells dataset. Zenodo, 2023. Accessed: 2025-05-06.

[11] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *2009 16th Working Conference on Reverse Engineering*, pages 75–84. IEEE, 2009.

[12] Lech Madeyski and Tomasz Lewowski. Detecting code smells using industry-relevant data. *Information and Software Technology*, 155:107112, 2023.

[13] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 350–359. IEEE, 2004.

[14] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139, 2004.

[15] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2009.

[16] Binh Nguyen Thanh, Hanh Le Thi My, Binh Nguyen Thanh, and Minh Nguyen Nhat. ml-codesmell: A code smell prediction dataset for machine learning approaches. Figshare, October 2022. Accessed: 2025-05-06.

[17] Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio Sant'Anna. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development*, 5:1–28, 2017.

[18] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 403–414. IEEE, 2015.

[19] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 97–106. IEEE, 2002.

[20] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? In *2012 28th IEEE international conference on software maintenance (ICSM)*, pages 306–315. IEEE, 2012.