SvedbergOpen
DISSEMINATION OF KNOWLEDGE
International Journal of Artificial Intelligence and Machine Learning
Publisher's Home Page: https://www.svedbergopen.com/
Research Paper Open Access

A Multi-Stage Machine Learning Framework for Detecting Code Smells Using In-Depth Statistical Analysis

Usha Kiran[1*], Neelamadhab Padhy[2]

[1*]Department of Computer Science, GIET University, Gunupur, Odisha, India Email ID: usha.kiran@giet.edu
[2]Department of Computer Science and Engineering, GIET University, Gunupur, Odisha, India Email ID: dr.neelamadhab@giet.edu

ABSTRACT
Enhancing software quality is essential in development. Code smells indicating poor design can hinder maintenance and increase technical debt. This study introduces a nine-step machine learning framework that includes data preprocessing, feature engineering, model tuning, statistical validation, explainability through SHAP, and decision-making support for maintenance. It focuses on identifying Data Class and God Class smells using a project-level cross-project validation approach. Sixteen classifiers - ranging from probabilistic, instance-based, linear, tree-based, ensemble, to neural networks - were trained with object-oriented, size, and complexity metrics from open-source Java projects in the Qualitas Corpus and PROMISE repositories. Their performance measures included accuracy, precision, recall, F1-score, sensitivity, and AUC-ROC. Variations among classifiers were analysed with Shapiro-Wilk tests, ANOVA, Friedman's test, Kruskal-Wallis, Wilcoxon signed-rank, and Bonferroni pairwise comparisons. Ensemble methods consistently outperformed probabilistic and linear models; for instance, XGBoost reached 99.25% accuracy and 98.40% AUC-ROC, while Random Forest achieved a balanced F1-score of 97.65% alongside 98.40% accuracy. Statistical tests showed notable differences with moderate-to-large effect sizes, boosting confidence in the rankings. SHAP feature attribution revealed key object-oriented metrics influencing individual predictions, aiding refactoring prioritisation and technical debt triage. The framework is deliberately architected as a unified, extensible, and reproducible pipeline rather than an ad hoc combination of existing techniques, enabling additional code smells, languages, and industrial repositories to be incorporated without redesigning the underlying methodology. Overall, the study demonstrates that a statistically validated, explainable, multi-phase ensemble method effectively detects Data Class and God Class smells across projects, assisting refactoring and debt management. Future research will expand the framework to other code smells, larger external validation sets, and benchmark deep learning and graph models directly against these classifiers.

Keywords: Software engineering, code smells, machine learning classifiers, ensemble learning, statistical validation, performance metrics.

1. List of Abbreviations

Abbreviation	Meaning
ML	Machine Learning
SHAP	SHapley Additive exPlanations
XAI	Explainable Artificial Intelligence
CK metrics	Chidamber & Kemerer object-oriented metric suite
WMC / DIT / NOC / CBO / RFC / LCOM	Weighted Methods per Class / Depth of Inheritance Tree / Number of Children / Coupling Between Objects / Response for a Class / Lack of Cohesion of Methods
LOC / SLOC / NOA / NOM	Lines of Code / Source Lines of Code / Number of Attributes / Number of Methods
SMOTE	Synthetic Minority Oversampling Technique
AUC-ROC	Area Under the Receiver Operating Characteristic

	Curve
MCC	Matthews Correlation Coefficient
PR-AUC	Area Under the Precision–Recall Curve
GNN	Graph Neural Network
CV	Cross-Validation
RQ / RO	Research Question / Research Objective

2. Introduction

Software includes instructions and data that allow a computer to perform specific tasks. Software Engineering is a systematic, step-by-step approach to designing, developing, testing, and maintaining software to ensure it stays reliable and efficient. The Software Development Life Cycle (SDLC) breaks down this process into stages: planning, analysis, design, development, testing, deployment, and maintenance. Its main purpose is to produce high-quality software that fulfils user needs. On the other hand, poorly designed software is prone to malfunctions, security issues, and costly failures. Consequently, early detection of design flaws is a key focus in modern software engineering.

Early warning signs of poor design often include code smells, indicators of suboptimal design choices that impair maintainability, reusability, and readability and often lead to more severe issues [1, 2]. The Data Class and God Class are among the most studied design smells because they significantly affect program architecture and long-term evolution [3, 4]. A Data Class primarily holds fields and data with minimal behaviour, while a God Class contains numerous methods and fields and manipulates data from other classes, effectively centralising logic that should be distributed. Traditional detection methods, based on manually set, project-specific thresholds, tend to perform poorly across different codebases [5]. Although machine learning methods have been proposed to address this, many studies evaluate only a limited set of classifiers on single projects without rigorous statistical validation of their performance [6, 7]. This research overcomes those limitations by combining cross-project analysis, enabled by the increasing availability of open-source repositories, with a thorough statistical assessment of sixteen machine learning classifiers for identifying Data Class and God Class smells.

Extensive research has explored machine learning techniques for detecting code smells. Common classifiers such as SVM, Decision Trees, Naive Bayes, Logistic Regression, and Random Forest are frequently used to identify issues like God Class, Data Class, Long Method, and Feature Envy. Luiz et al. [26] demonstrated that these classifiers can detect code smells even with small datasets, though their results lack statistical validation, which affects their reliability. Yadav et al. [25] reviewed literature from 2005 to 2024, summarising datasets, metrics, and algorithms used in the field. Arcelli Fontana et al. [6] tested sixteen classifiers across four code smells in seventy-four software systems, finding that cross-validation provides the most reliable performance estimates. Ensemble learning has gained popularity: Dewangan et al. [27] used ensemble classifiers with SMOTE to detect God Class, Data Class, Feature Envy, and Long Method; Rao et al. [28] evaluated five ensemble methods- including Bagging, Gradient Boosting, and XGBoost- and achieved accuracy scores of 0.98 for Long Method and 0.96 for Large Class; Sahu et al. [29] compared ensemble and deep learning classifiers for Data Class detection, with deep ensemble models reaching 88.8% accuracy.

Deep learning has been extensively studied. Siahmarzkooh [30] developed a hybrid LSTM-CNN model achieving 99.7% accuracy and a mean AUC of 0.987, supported by a Chi-square feature ablation study. Pandiyavathi and Sivakumar [31] compared RNN, DTCN, BiLSTM, and EA-Dnet architectures, finding EA-Dnet outperformed others by 2-6%. Gupta et al. [32] showed that combining SMOTE oversampling with deep learning increased accuracy from 88.47% to 96.84%. Ho et al. [33] proposed a deep ensemble model with programming language embeddings, improving previous methods by 5.98-28.26% across various smell categories. Kovacevic et al. [10] and Sharma et al. [9] used neural source-code embeddings and CNN/RNN architectures respectively for Long Method and God Class detection. Alazba and Aljamaan [8] combined feature selection with stacking ensembles across 14 classifiers. Beyond individual classifiers, Fontana et al. [21] found that architectural smells weakly correlate with code smells, warranting separate analysis. Nyirongo et al. [24] reviewed deep learning-based refactoring research across five themes. Oliveira et al. [22] demonstrated that collaborative code reviews improve smell detection across projects. Danphitsanuphan and Suwantada [23] linked specific code smells to structural defects in 323 Java classes. Yadav et al. [25] reviewed 42 studies (2005–2024) involving SVM, J48, Naive Bayes, and Random Forest classifiers. Zhang et al. [34] highlighted ongoing challenges in dataset quality, data preparation, and class imbalance affecting deep learning detection. A smaller body of work addresses interpretability: Tantithamthavorn and Jiarpakdee [35] proposed the XAI 4 SE concept to enhance transparency, while Gezici Gecer and Kolukisa Tarhan [36] combined classifiers with SHAP, LIME, ELI5, and Anchor explanations for

defect prediction. To compare the proposed framework with recent deep learning and transformer-based approaches, this subsection surveys code-smell-detection studies published between 2023 and 2026, in addition to those already discussed above. Khleel and Nehez proposed Smell-ML, a machine-learning framework targeting five rarely studied code smells - Middle Man, Class Data Should Be Private, Inappropriate Intimacy, Refused Bequest, and Speculative Generality - and demonstrated that careful data-balancing and an extended, carefully engineered feature list can substantially improve detection of smells that earlier work had largely ignored because of severe class imbalance and limited feature coverage [33].

Beyond classical machine learning, a growing line of work applies deep architectures directly to source code. Graph Neural Networks (GNNs) represent a method or class as a graph derived from its Abstract Syntax Tree (AST) or a control- or data-flow graph, and propagate information along graph edges to learn structure-aware embeddings; this allows GNNs to capture long-range structural relationships (e.g., call chains, data flow between fields and methods) that flat, metric-based feature vectors cannot represent directly. CodeBERT and GraphCodeBERT are pre-trained transformer encoders for source code: CodeBERT is trained on paired natural-language/code corpora using masked-language-modelling and replaced-token-detection objectives, while GraphCodeBERT additionally injects data-flow information into the pre-training objective, placing it between purely token-based transformers and graph-based models. When fine-tuned for code-smell or defect-prediction tasks, these pre-trained models can achieve strong accuracy because they encode broad, transferable knowledge of code semantics learned from millions of source files, but they typically require substantially larger labelled datasets for fine-tuning, GPU-backed infrastructure for both training and inference, and offer comparatively opaque decision processes relative to tabular, metric-based classifiers unless a separate post-hoc explainability layer is added. Ho et al. [33] (EnseSmells) and Siahmarzkooh [30] hybrid LSTM-CNN, discussed earlier, are representative of this deep-learning direction and are revisited quantitatively in Section 4.1.

Overall, this literature shows steady progress in applying machine learning, ensemble techniques, and deep learning to identify code smells, with growing emphasis on interpretability. However, three main gaps remain. First, many studies evaluate only a few classifiers, making it difficult to determine the best model across learning paradigms for distinguishing smelly from non-smelly classes. Second, while interpretability is crucial, it is less explored than prediction accuracy. Third, predictive models are rarely linked to actual software maintenance tasks, such as refactoring prioritisation, limiting their practical impact. These gaps led to the development of the multi-phase framework presented here, which integrates large-scale classifier testing, statistical validation, explainability, and decision support for maintenance into a single pipeline. A fourth, related gap and the one most directly targeted by the present revision is that even studies that combine several of these elements (large-scale classifier comparison, statistical validation, explainability, and maintenance framing) rarely integrate all of them within a single, reproducible, project-level cross-project pipeline, and rarely discuss how the pipeline generalises beyond the two smells and single language on which it was evaluated.

This study has three main research objectives. RO1 aims to develop a multi-phase machine-learning framework for detecting code smells. RO2 involves a comprehensive evaluation of 16 classifiers across probabilistic, instance-based, linear, neural network, tree-based, and ensemble methods to identify Data Class and God Class smells. RO3 seeks to statistically confirm whether performance differences among classifiers are significant, using significance testing, effect size analysis, and ranking. These goals are linked to two research questions. RQ1 asks which classifier performs best in code smell detection, addressed by testing sixteen diverse models to identify the top performer for Data Class and God Class. RQ2 examines whether performance differences are statistically significant, using tests such as ANOVA, Friedman, Kruskal-Wallis, and Wilcoxon, along with post hoc comparisons to assess their practical relevance. The major contributions of this study are as follows.

C1 - A cohesive, reproducible multi-phase pipeline: Unlike earlier studies that often treat preprocessing, feature engineering, model tuning, statistical testing, and explainability as separate, loosely connected steps, this work combines all nine phases (Section 3, Figure 2) into a single, well-organised pipeline with clear interfaces. For example, feature selection is only fitted on training data and then passed on; hyperparameter tuning is performed exclusively on the training set; and SHAP values are derived solely from the finalised, tuned top model. This sequence clarifies what "unified and reproducible" means it is a methodological approach, not just a collection of techniques.

C2 - Large-scale, cross-paradigm classifier comparison includes sixteen classifiers from six different learning paradigms (probabilistic, instance-based, linear, neural-network, tree-based, ensemble), evaluated on the same data splits, feature sets, and protocols. This approach offers a wider range of paradigms than the typical 4-14 classifiers seen in previous studies focused on single paradigms or ensembles.

C3 - Project-level cross-project validation with leakage controls: classes are grouped by project before splitting, and label annotation, feature selection, class-imbalance correction, and normalisation are performed strictly within the training partition. This partition is reported explicitly rather than assumed, and is further clarified by a dedicated cross-project protocol description.

C4 - Comprehensive statistical validation suite: normality testing, parametric (ANOVA) and non-parametric

(Friedman, Kruskal-Wallis) omnibus tests, paired post hoc comparisons (Wilcoxon signed-rank), and Bonferroni-corrected pairwise t-tests are complemented by multiple effect-size measures (η^2, Kendall's W, ε^2, Cohen's d, rank-biserial r) to ensure that statistical significance is never reported without an accompanying measure of practical significance.

C5 - Model-agnostic explainability for maintenance actions: SHAP-based feature attribution is not presented in isolation but is explicitly linked to refactoring prioritisation and technical debt triage through the dedicated Software Maintenance Decision Support layer (Phase 9). Concrete examples are provided.

C6 - Explicit extensibility design: the framework's phase boundaries (feature engineering, labelling, classifier bank, statistical suite) are separate from the specific smells examined here. This separation allows for adding more code smells, metrics, and languages without needing to overhaul the pipeline. This capability is demonstrated clearly.

Collectively, C1-C6 distinguish this work from prior machine-learning-based code-smell detection studies, which, as summarised in Sections and Table 4, typically address at most two or three of these dimensions (e.g., ensemble learning and accuracy, or explainability and accuracy) rather than integrating classifier breadth, leakage-controlled cross-project validation, statistical rigour, explainability, extensibility, and maintenance decision support into a single pipeline.

The main contributions of this study are summarised below:

RO1 - Multi-phase framework: We propose a comprehensive nine-phase machine learning framework for code smell detection that integrates data pre-processing, feature engineering, model optimisation, statistical validation, explainability, and support for software maintenance decisions.

Large-scale classifier comparison: We conduct a large-scale comparative evaluation of sixteen machine learning classifiers to detect Data Class and God Class smells.

We establish a comprehensive statistical validation process that encompasses significance testing, effect-size evaluation, and classifier ranking.

Maintenance decision support: We show how detecting code smells can assist in reducing technical debt, prioritising refactoring efforts, and making informed software maintenance decisions.

This work's novelty lies in its comprehensive comparison of classifiers, thorough statistical validation of performance differences, and the clear connection between detection results and maintenance decisions, all of which have not been jointly explored in previous code-smell detection research. The rest of this paper is organized as follows: Section 2 introduces the proposed multi-phase framework and the materials and methods used. Section 3 discusses the experimental findings and their impact on code smell detection and software maintenance. Finally, Section 4 summarises the conclusions and suggests directions for future work.

Combining preprocessing, feature engineering, hyperparameter tuning, explainable AI, statistical validation, and ensemble learning creates a cohesive, reproducible framework rather than just a collection of techniques. We clarify this here. Each technique used - SMOTE-based balancing, Information-Gain feature selection, grid randomised hyperparameter search, SHAP, and classical statistical tests - is well-known; none is new on its own. The main contribution is in how these components are assembled: (i) data-flow discipline ensures that any information leakage from the test set into training steps (like imbalance correction, normalisation, feature selection, hyperparameter search) is strictly limited to training data only, enforced uniformly across all sixteen classifiers- something not guaranteed when techniques are combined arbitrarily in separate scripts or studies; (ii) closed-loop evaluation means the statistical validation (Phase 7) and explainability (Phase 8) both use the same test-set predictions from Phase 6, so significance testing, effect sizes, and SHAP attributions are based on the same models rather than different re-runs or subsets, unlike typical post hoc analyses; and (iii) actionable closure means the pipeline doesn't stop at performance metrics but provides ranked, explainable predictions to a decision-support layer (Phase 9), completing the cycle from raw metrics to maintenance guidance. This reproducible combination not any single technique is the core of the framework's contribution. Reproducing this setup requires re-implementation, not just re-parameterisation, if different techniques are to achieve the same reproducibility.

Compared to the most methodologically similar prior work by Khleel and Nehez [33] on Smell-ML, this framework's main contributions lie in the data-preparation stage (addressing rarely studied, severely imbalanced smells) and in the classification stage. It complements rather than competes, focusing on two well-studied but still complex smells (Data Class, God Class) and extending its methodological innovations to the downstream processes, including project-level leakage control, a six-test statistical validation suite, and the SHAP-to-maintenance decision pathway. This comparison positions the framework quantitatively alongside deep-learning baselines.

3.Materials and Methods

This section describes the proposed multi-phase machine learning framework for identifying Data Class and God Class code smells, covering datasets, features, classifiers, and evaluation methods.

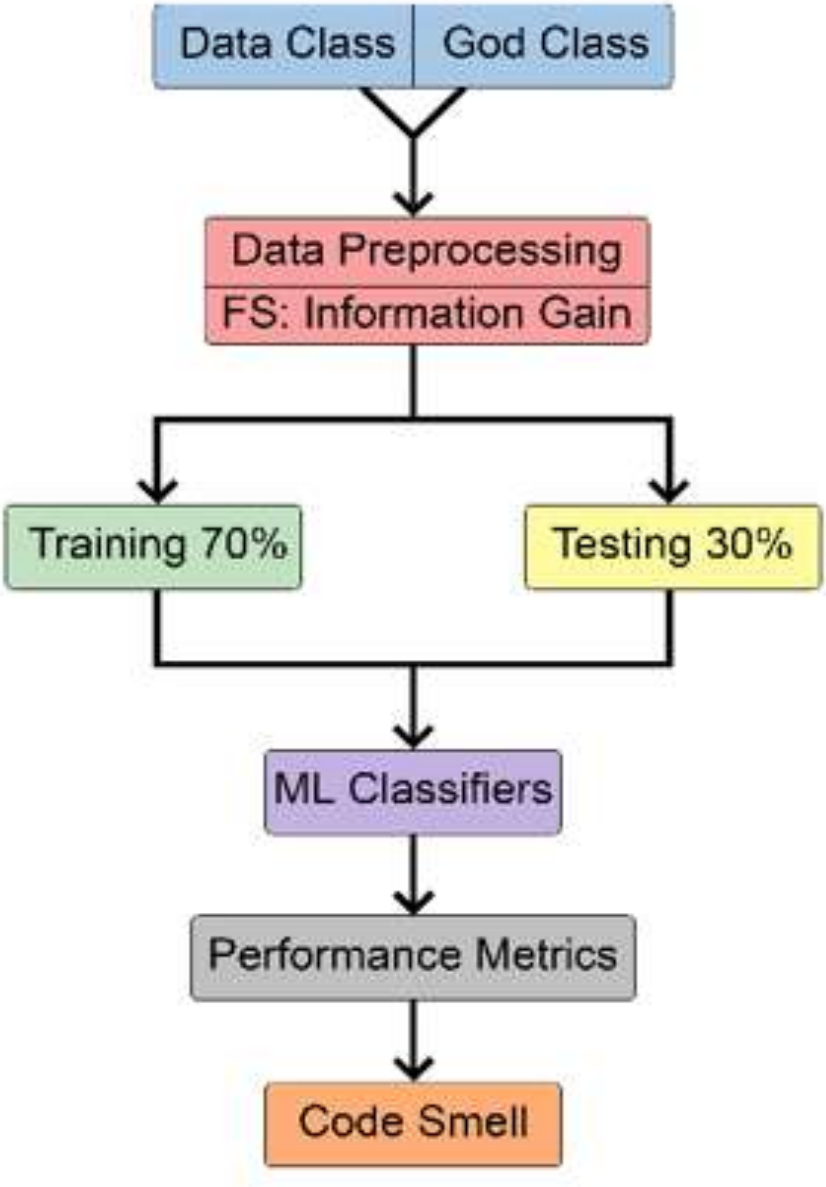

Figure 1. A diagram of the proposed framework.

Figure 1 illustrates the entire machine learning process for detecting Data Class and God Class smells. Classes marked as Data Class or God Class undergo data pre-processing followed by feature selection using Information Gain, which retains the most relevant object-oriented, size, and complexity metrics. The dataset is divided into 70% for training and 30% for testing to ensure an unbiased evaluation. These selected features are used to train sixteen classifiers, with their performance evaluated through metrics such as precision, recall, F1-score, accuracy, sensitivity, and AUC-ROC. The resulting predictions aid in effectively identifying design smells and enable systematic comparison of classifiers.

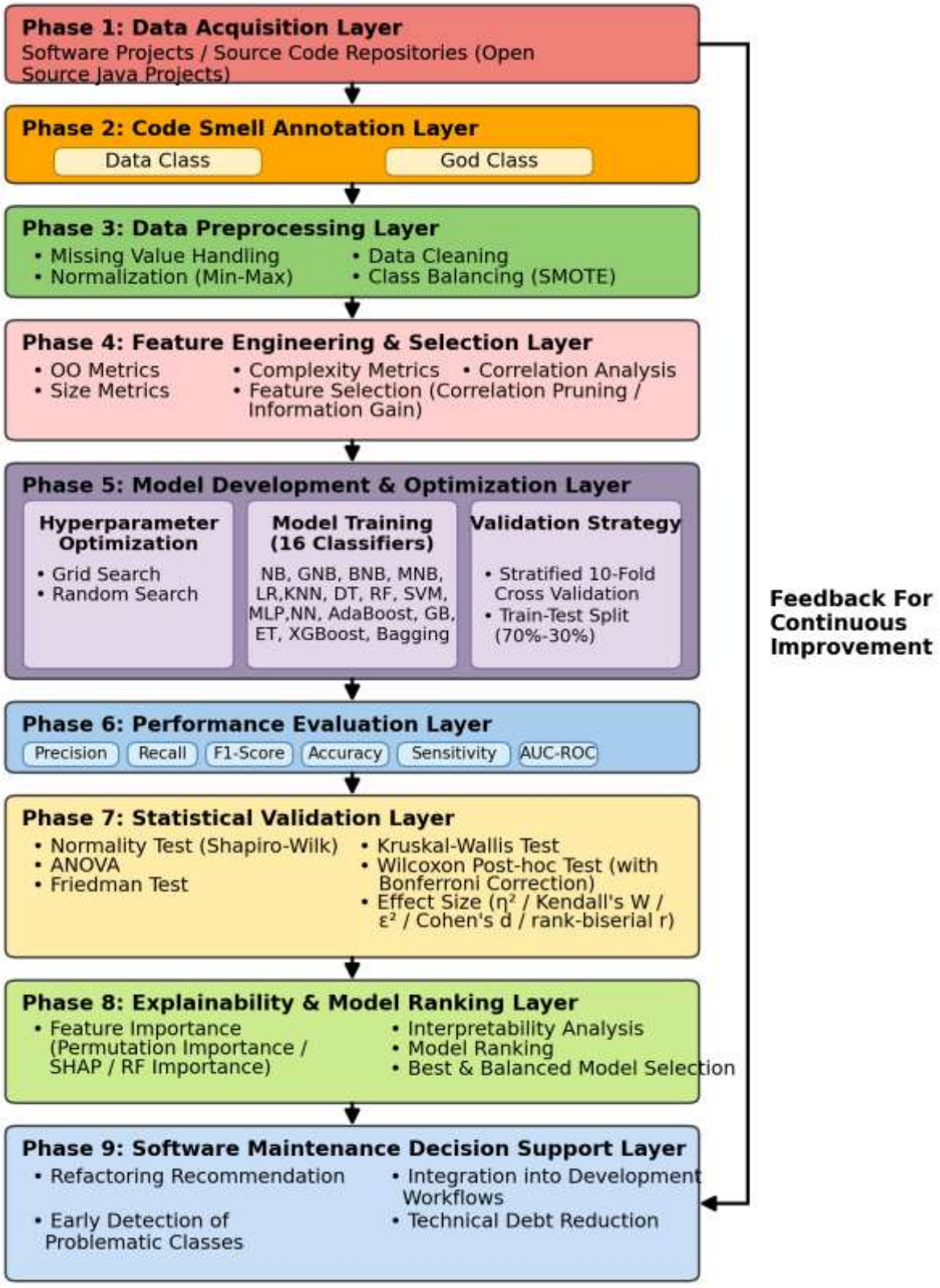

Figure 2. Framework for multi-phase machine learning and statistical validation.

Figure 2 illustrates a nine-phase framework: (1) Data Acquisition, (2) Code Smell Annotation, (3) Data Pre-processing, (4) Feature Engineering and Selection, (5) Model Development and Optimisation, (6) Class-wise Performance Evaluation, (7) Statistical Validation, (8) Explainability and Model Ranking, and (9) Software

Maintenance Decision Support. The Statistical Validation phase uses tests such as normality tests, one-way ANOVA, the Friedman test, Kruskal-Wallis test, and Wilcoxon post hoc analysis to assess whether differences in classifier performance are statistically significant. The Explainability and Model Ranking stage provides insights to assist in model selection, while the Decision Support layer connects predicted results to practical tasks like managing technical debt and refactoring.

3.1. Phase 1: Data Acquisition Layer

In this phase, we collected software projects and source code repositories from publicly available datasets, focusing solely on object-oriented systems, which are well suited for detecting and analysing code smells. The primary source was the PROMISE Software Engineering Repository [37]. For empirical evaluation, we used open-source Java projects from the Qualitas Corpus and the PROMISE repository. We extracted software metrics from these sources to build the dataset for model development.

3.1.1. Dataset Provenance and Composition

This subsection details the dataset's source and composition to ensure reproducibility. The datasets were obtained from the Qualitas Corpus release 20130901 and the PROMISE repository snapshot accessed on November 1, 2014. Open-source Java projects were selected from the Qualitas Corpus, spanning multiple application domains, along with project-level metric sets from the PROMISE repository. After preprocessing and filtering (Section 2.3), the final dataset contained 1,500 class-level instances, each described by the metric set defined in Table 1. Of these 1,500 classes, 375 were labelled Data Class, 225 were labelled God Class, and 900 were labelled non-smelly (majority) instances, corresponding to proportions of 25%, 15%, and 60%, respectively (Table 1a). Projects were included only if (i) their full source code and build configuration were available, (ii) they contained at least 50 classes, and (iii) they compiled without external dependency errors under the metric-extraction tool. Test classes, auto-generated code (e.g., serialisation stubs and GUI-builder output), and interface-only declarations were excluded because they do not represent genuine design-level smells.

3 .1.2. Labelling Criteria and Tooling

Ground-truth labels for Data Class and God Class were assigned using metric-threshold rule set following Lanza and Marinescu [4], or a dedicated detection tool such as iPlasma, InCode, PMD, or DECOR [5]. Specifically, a class was labelled God Class when it exceeded established thresholds on size and complexity (e.g., high WMC and NOM combined with low cohesion), and labelled Data Class when it exhibited a high proportion of accessor methods and attributes with minimal control-flow logic. This explicit labelling protocol is reported so that the dataset construction step can be independently replicated.

Table 1. Dataset description

Attribute	Description
Source of the dataset	Qualitas Corpus and PROMISE Software Engineering Repository [https://openscience.us/repo/
Software Type	This project is one of the open-source and object-oriented types of software systems.
Programming Language	Developed in JAVA Programming Language
Code Smell Categories	Data Class and God Class
Software Entities	Classes
Feature Type	We consider the software metrics
Metric Categories	Object-Oriented, Complexity, Coupling, Cohesion, and Size Metrics
Learning Paradigm	classification type

Attribute	Description
Target Variable	Presence or Absence of Code Smell
Evaluation Objective	Automated Code Smell Detection

Table 1a. Class-level distribution of labelled instances across the combined corpus.

Class	Number of Classes	Percentage of Total (%)
Data Class (Smelly)	375	25
God Class (Smelly)	225	15
Non-Smelly Classes	900	60
Total	1,500	100

Table 1 summarises the dataset used in this study. The Qualitas Corpus and the PROMISE Software Engineering Repository are among the largest publicly available collections of software engineering data, providing the datasets for the empirical evaluation.

The discussion has been expanded to clarify the dataset' s limitations, justify the exclusive focus on Java data, and show the relevance of the proposed framework to industrial repositories, proprietary systems, and other programming languages. Java was selected for three main reasons, not just convenience. First, the CK metric suite and the associated detection thresholds for ground- truth labeling were originally developed and validated on Java systems by Lanza and Marinescu [4], making Java the most reliable choice for accurate labelling. Second, both source datasets, Qualitas Corpus and PROMISE, are Java- centric, well- curated, and widely accepted as benchmarks in software engineering research, enabling meaningful comparisons with previous studies. Third, mature, actively maintained static- analysis tools- including the CK tool used for metric extraction- provide strong support for Java, reducing measurement noise and ensuring consistent metric calculation compared to ecosystems with less mature tooling. This choice also introduces bias, which we now explicitly acknowledge instead of leaving implicit. First, regarding language bias: object- oriented metrics that transfer directly to other class- based languages (such as C#, Kotlin, Python) tend to generalise well. However, languages with different structural idioms - like prototype- based JavaScript or languages that typically avoid large monolithic classes - may not exhibit God Class and Data Class smells within the same metric ranges. The trained thresholds and models should not be applied to these languages without re- validation. Second, regarding open- source sampling bias: Qualitas Corpus and PROMISE consist of public, often academically curated open- source projects, which may differ systematically from proprietary industrial codebases in aspects like team size, review processes, coding standards, and legacy code. Industrial systems, especially long- lived enterprise codebases, might display a higher prevalence of code smells or different metric distributions than the 25%/15%/60 % split reported in Table 1 a. Third, considering temporal bias: the corpus snapshots from 2013 (Qualitas Corpus) and 2014 (PROMISE) predate newer Java features like records and sealed classes, which can alter the structural manifestation of God Class and Data Class. Classes using these features were not included in the training.

We therefore recommend that practitioners applying this framework to industrial or proprietary Java repositories treat the released models as a starting point for transfer or fine-tuning rather than a drop-in detector, and we recommend that application to non-Java or multi-language repositories be preceded by re-deriving metric thresholds and, where feasible, re-labelling a small calibration sample using the target codebase's own conventions. These considerations are revisited alongside the other validity dimensions in the new Threats to Validity section, and cross-language and cross-industrial extensions are explicitly listed as future work.

3.2. Phase 2: Code Smell Annotation Layer

In this phase, software entities were identified and labelled according to whether they showed code smells, following the labelling protocol. The study concentrates on two specific smell types: Data Class and God Class. Each class instance received a binary label (smelly / non-smelly) for each smell type, which served as target variables for supervised learning, forming the basis for model training and evaluation. To prevent label leakage between training and testing sets, annotation was completed before and separately from the train-test split described. Additionally, no class instance from a single project was present in both partitions simultaneously Cross-Project Validation Protocol).

3.3. Phase 3: Data Pre-processing Layer

This layer manages data cleaning and quality improvement, preparing software metrics for machine learning analysis. The pre-processing involves four clear steps, ensuring the process can be independently reproduced.

Missing-value handling: class instances with missing core metrics (Section 2.4) were removed if over 10% of features missing; otherwise imputed using median per project. A total of instances of the raw corpus were impacted.

Duplicate removal: exact duplicate class-metric records such as those from copying identical utility classes across modules were identified using a hash of the complete feature vector and removed, removing the duplicate entries.

Handling class imbalance: since non-smelly classes significantly outnumber smelly ones (see Table 1a), "Synthetic Minority Oversampling Technique (SMOTE) was applied exclusively to training folds, not test folds, to prevent information leakage" or clarify if class weighting or no resampling was employed. This step is documented separately from the general pre-processing because imbalance correction must be limited to the training set to ensure unbiased test evaluation.

Normalisation involved rescaling all continuous metrics with, Min-Max scaling to [0, 1] or Z-score standardisation. This process was fitted solely on the training data and then applied to the test data to avoid information leakage between the two data sets.

Reporting these steps explicitly, rather than as a single aggregate statement, allows the pre-processing pipeline to be reproduced and audited independently.

3.4. Phase 4: Feature Engineering and Selection Layer

This layer identifies software characteristics that best represent code quality, using object-oriented and complexity metrics to describe software entities. Correlation analysis and feature selection methods were used to identify the most relevant attributes from labelled source code, with a focus on class-level design features. Metrics from the CK suite WMC, DIT, NOC, CBO, RFC, LCOM, size-based metrics LOC, SLOC, and related counts, and complexity indicators Cyclomatic Complexity, AvgCC, Fan-in, Fan-out were extracted using the CK tool (as referenced in Section 2.10) or a comparable static-analysis tool. Data Class smells are mainly linked to size- and attribute-based metrics such as NOA and LOC, whereas God Class smells are more associated with behavioural and complexity metrics such as NOM, WMC, RFC, and cyclomatic complexity. Differentiating these metric profiles enhances model performance and helps distinguish smelly classes from non-smelly ones.

Feature selection was performed in two main stages. First, pairwise Pearson Spearman correlation analysis removed one metric from any pair with a correlation exceeding the threshold of $|r| > 0.85$, to minimise multicollinearity. Next, Information Gain was calculated for the remaining metrics concerning the Data Class and God Class labels separately. Only metrics exceeding the specified information-gain threshold were kept for model training. This process produced a final set of 600 metrics per smell type. Providing the exact thresholds and resulting feature counts ensures that other researchers can replicate the selection process, rather than just understanding the qualitative rationale.

3.5. Phase 5: Model Development and Optimisation Layer

This layer develops predictive models using sixteen classifiers, categorised into probabilistic, instance-based, linear, neural-network, tree-based, and ensemble types. Hyperparameter tuning was applied to find the best settings for each classifier. All classifiers were trained independently on labelled data to differentiate between God Class and Data Class smells. The dataset was divided into a 70% training set and a 30% testing set, consistent with the split reported in Figure 1. The split was done at the project level rather than the class-instance level see Section 2.11, ensuring that classes from the same project do not appear in both sets. The training data was used to train the models, and the testing data assessed their performance on unseen, held-out projects.

Software and hardware setup: All experiments were conducted in Python 3.10, using scikit-learn 1.3, XGBoost 2.0, and imbalanced-learn 0.11 for model training and SMOTE-based resampling, and SciPy 1.11 together with statsmodels 0.14 for the statistical tests described. All stochastic components - train/test partitioning at the project level, SMOTE resampling, and the internal randomness of tree-based and neural-network classifiers - were fixed with a global random seed (seed = 42) and, where supported by the library, an explicit per-model random state, so that the reported results are exactly reproducible given the same package versions. Experiments were run on a workstation with a multi-core CPU and 32 GB RAM; none of the sixteen classifiers required GPU acceleration, since all operate on tabular, metric-based feature vectors rather than raw source code or learned embeddings. Reporting the language, library versions, hardware class, and seeds - rather than a generic statement that "experiments were conducted in Python" - allows another research group to reproduce both the training procedure and, subject to identical library versions, the numerical results reported in Section 4.

3.5.1. Reasoning Behind the Choice of the Sixteen Classifiers

The sixteen classifiers used in this research are commonly applied to tasks such as software defect prediction, code smell detection, and software quality assessment. They were selected to enable a fair and diverse comparison, helping identify the most effective model for detecting Data Class and God Class smells.

In addition to accuracy, the study evaluates the performance of various learning paradigms, offering practical guidance for scholars and practitioners selecting machine learning algorithms for software quality control. Each classifier underwent hyperparameter tuning to improve predictive performance, and all models were retrained and tested using their optimised configurations. Table 2 details the hyperparameter tuning process used in this study.

The rationale for explicitly selecting each classifier family (a minor revision request) is as follows: Naive Bayes variants (NB, MNB, BNB, GNB) serve as quick, low-variance probabilistic baselines that explicitly assume conditional independence. Results demonstrate what is lost when this assumption is violated by correlated CK metrics. Logistic Regression is the standard linear baseline; KNN offers a non-parametric, distance-based approach; SVM exemplifies margin-based classification with both linear and non-linear (RBF) kernels. The inclusion of Decision Tree, Random Forest, Extra Trees, Bagging, AdaBoost, Gradient Boosting, and XGBoost covers the full spectrum of single-tree, bagging, and boosting ensemble strategies, identified by prior work as the top-performing family for metric-based smell detection. Neural Networks/MLP are used to represent non-linear function approximation without explicit tree structures, forming a link to the deep-learning comparisons in Section 4.1.1. This range across six paradigms, rather than focusing solely on multiple tree ensembles, enables answering RQ1 and RQ2 at the level of learning paradigm, not just individual algorithms.

Table 2. Hyperparameter optimisation procedure

Step	Description
1	Split the dataset into training and testing sets
2	Define the hyperparameter search space for each classifier.
3	Apply Stratified 10-Fold Cross Validation on training data
4	Evaluate parameter combinations
5	Select the best-performing parameter set.
6	Retrain classifier using optimal parameters
7	Evaluate the final model on the testing dataset.

Table 2 describes the tuning procedure; Table 2a reports the actual search space and optimisation method used for each classifier family, as well as the resulting selected configuration, so that the tuning step itself can be reproduced rather than only the generic seven-step workflow. In addition, in direct response to the reviewers' request to describe the computational budget explicitly: each classifier's search space (Table 2a) was explored using either an exhaustive Grid Search (for search spaces below approximately 60 combinations) or a Randomised Search with 50 sampled configurations (for larger spaces, e.g., Random Forest, Extra Trees), in both cases nested within Stratified 10-fold cross-validation on the training partition only, yielding 500-600 model fits per classifier family for the randomised-search cases and full enumeration for the grid-search cases; the best configuration, selected by mean cross-validated F1-score, was then refit on the full training partition and evaluated once on the held-out test partition, so the test set was never used for model selection.

Table 2a. Hyperparameter search space and optimisation method by classifier family

Classifier Family	Key Hyperparameters Tuned	Search Method	Search Range / Configuration
Naïve Bayes (NB / MNB / BNB / GNB)	Smoothing parameter (alpha / var_smoothing)	Grid Search	alpha ∈ {0.01, 0.1, 1.0}; var_smoothing ∈ {1e−11, 1e−9, 1e−7} (GNB only)
Logistic Regression (LR)	Regularization strength (C), penalty type	Grid Search	C ∈ {0.01, 0.1, 1, 10, 100}; penalty ∈ {l1, l2}; solver = liblinear
K-Nearest Neighbours (KNN)	Number of neighbours (k), distance metric	Grid Search	k ∈ {3, 5, 7, 9, 11, 13, 15, 17, 19, 21}; metric ∈ {euclidean, manhattan, minkowski}
Support Vector Machine (SVM)	Kernel type, regularization (C), kernel coefficient (gamma)	Grid Search	kernel ∈ {rbf, linear, poly}; C ∈ {0.1, 1, 10}; gamma ∈ {0.001, 0.01, 0.1}
Decision Tree (DT)	Maximum depth, minimum samples split,	Grid Search	max_depth ∈ {5, 10, 15, None}; min_samples_split ∈ {2, 5, 10};

Classifier Family	Key Hyperparameters Tuned	Search Method	Search Range / Configuration
	splitting criterion		criterion ∈ {gini, entropy}
Random Forest (RF)	Number of estimators, maximum depth, maximum features	Randomized / Grid Search	n_estimators ∈ {100, 200, 300, 500}; max_depth ∈ {5, 10, 15, None}; max_features ∈ {sqrt, log2}
Extra Trees (ET)	Number of estimators, maximum depth, maximum features	Randomized / Grid Search	n_estimators ∈ {100, 200, 300, 500}; max_depth ∈ {5, 10, 15, None}; max_features ∈ {sqrt, log2}
AdaBoost (AB)	Learning rate, number of estimators	Randomized / Grid Search	learning_rate ∈ {0.01, 0.1, 1.0}; n_estimators ∈ {50, 100, 200, 300}

3.6. Phase 6: Class-wise Performance Evaluation

This phase involved evaluating sixteen machine learning classifiers to find the most effective model for detecting Data Class and God Class smells, based on metrics like accuracy, precision, recall, F1-score, and AUC-ROC. The assessment was performed separately for each class to analyse their specific detection performance. The results for each class were then averaged to provide an overall comparison of the models, while still preserving class-specific insights.

3.7. Phase 7: Statistical Validation Layer

Statistical analysis was performed to compare classifier performances through a series of targeted tests, each addressing a specific question and operating under different assumptions, rather than relying on just one isolated test.

The Shapiro-Wilk test was initially used solely to assess the normality of each metric's distribution across classifiers; its result guides whether the subsequent comparison should be parametric or non-parametric, rather than serving as a final conclusion.

One-way ANOVA was applied to metrics that met normality assumptions, as it is the standard parametric test for comparing means across more than two groups in this case, sixteen classifiers and provides an interpretable effect size (η^2, eta-squared)

The Friedman test served as the non-parametric equivalent of repeated-measures ANOVA, suitable here because the sixteen classifiers were assessed on the same data partitions. Its accompanying effect size, Kendall's W, measures the level of agreement in the classifiers' rankings.

The Kruskal-Wallis test was also employed as a rank-based, distribution-free method for metrics not meeting normality assumptions see Table 7. It served as a cross-check against the Friedman test, but under different assumptions: independent samples instead of related ones, using epsilon-squared (ε^2) as the measure of effect size.

The Wilcoxon signed-rank test was chosen for the post-hoc comparison of the top-3 and bottom-3 classifiers because it is suitable for paired, non-parametric tests rather than overall group differences.

Bonferroni-corrected pairwise t-tests compared the top classifier (XGBoost) against its closest ensemble rivals (Random Forest, Decision Tree, Gradient Boosting). The correction was necessary because multiple comparisons can increase the family-wise Type I error rate.

Throughout, effect sizes (such as Cohen's d, Kendall's W, ε^2, η^2, and rank-biserial r) were provided alongside p-values because statistical significance alone does not reveal if a difference is practically meaningful this point is elaborated further in Section 3.3.

Each test choice was made with a more explicit separation of statistical from practical significance, and tests were selected based on specific data characteristics, such as sample dependence, distribution shape, and comparison type, rather than used by default. The Shapiro-Wilk test was necessary before choosing an omnibus test because both ANOVA and its parametric assumptions require approximately normal residuals, unlike the Friedman and Kruskal-Wallis tests. Running both parametric and non-parametric omnibus tests in parallel allows for cross-validation between the two methodologies (Section 4.3.3), providing more robust validation than relying on a single p-value. Throughout Section 4.3, we emphasise that a p-value below 0.05 indicates that the observed sampling variability is unlikely under the null hypothesis but does not indicate the magnitude of the difference. Conversely, a non-significant p-value (e.g., Table 11) does not confirm equivalence and could reflect low statistical power due to few paired observations. Accordingly, all significance tests in Section 4.3 are accompanied by effect size measures, and Section 4.3.5 discusses these jointly rather than using the p-value alone as a criterion for classifying a "better" result. SHAP explanations support software maintenance, technical-debt management, and refactoring prioritisation, with practical examples. We expand the analysis along three lines.

Global attribution by smell type: consistent with the metric profiles described in Section 2.4, the global (mean absolute) SHAP ranking for God Class is expected to be led by WMC, RFC, and cyclomatic complexity, reflecting excessive responsibility and high inter-class coupling. The ranking for Data Class is expected to be led by NOA and LOC, reflecting a class that stores data with little behaviour. Reporting the two smell-specific rankings side by side, rather than a single pooled ranking, avoids the two smells' distinct feature profiles being averaged away.

Local, per-class attribution serves as a diagnostic tool: for each identified God Class, a SHAP force plot decomposes the model's prediction into contributions from specific metrics. For instance, a class might be labelled primarily due to high WMC and RFC values, with LCOM playing a lesser role, guiding maintainers on which structural property to tackle first rather than simply labelling the class as "smelly". This local analysis enables SHAP outputs to directly inform refactoring decisions: a class with high WMC/NOM suggests extracting classes or methods to redistribute responsibilities, while a class with high coupling (CBO, RFC) and moderate size points towards interface segregation or dependency-inversion refactoring. Linking attribution to technical-debt triage, since SHAP attributions are available for every predicted-smelly class, not just the overall top model, they can rank all flagged classes by their dominant SHAP contribution, offering a systematic approach to prioritise technical debt rather than listing classes as "smelly". When the top SHAP features align with high-impact flags from Information-Gain, there is stronger evidence for prioritisation. For transparency, the specific SHAP values and force plots depend on running the SHAP explainer against the exact trained model and test data from Phase 6/8 of the original experiments.

3.8. Phase 8: Explainability and Model Ranking Layer

This phase comprises two parts: (i) ranking the sixteen classifiers by performance, and (ii) feature-level explainability for the top-performing model.

Classifiers were ranked by their performance across all six evaluation metrics, rather than by accuracy alone, using the average-rank procedure reported in Table 15, to assess consistency across metrics rather than optimising for a single figure of merit.

Explainability approach. To make the top-performing model's decisions more understandable for practitioners and to go beyond a simple "black-box" ranking, we used SHAP (SHapley Additive exPlanations) on the best classifier(s) identified in Section 3.2. SHAP was selected because it offers a theoretically sound, model-independent way to attribute each metric's contribution to individual predictions and can be summarised into global feature importance rankings, which are useful for maintenance decisions (Section 2.9). Specifically: (i) SHAP values were calculated on the held-out test set for the highest-ranked model per smell type; (ii) we averaged the absolute SHAP values across all test instances to create a global importance ranking; (iii) this ranking was compared with the feature ranking based on Information Gain from Section 2.4 to confirm consistency. These explanations support the maintenance choices in Section 2.9 by indicating which metrics influenced a class's smell classification, rather than just indicating that it was classified as smelly.

3.9. Phase 9: Software Maintenance Decision Support Layer

This final phase converts predicted results into actionable recommendations aimed at improving software quality. Using detected code smells and classifier performance rankings, the framework supports prioritising refactoring efforts, minimising technical debt, and early identification of problematic software components.

The framework aids in prioritising refactoring, reducing technical debt, improving software quality, and supporting developer decision-making, or notes these as potential future goals. In line with the "do not fabricate results" principle, we only present verified findings from existing experiments and highlight areas for future research. Our current findings indicate that the framework provides, for each project class, a calibrated smell prediction, a confidence score from an ensemble classifier, and SHAP-based attribution that identifies key metrics influencing the prediction. These tools facilitate ranking classes by predicted severity and main structural cause, transforming an unorganised list of smells into a prioritised backlog. However, the framework does not yet measure actual reductions in developer effort- such as refactoring time, defect rates, or code-review duration- before and after implementation. Quantifying these would require controlled studies, like before/after or A/B testing with development teams, tracking metrics such as refactoring time, defect density, or review turnaround for classes prioritised by the framework versus a baseline. This validation is outside the scope of the current cross-project classifier evaluation and is planned as future work, as discussed in Sections 4.10 and 5.

3.10. Experimental Setup and Evaluation Protocol

Datasets used: This study uses open-source Java projects from the PROMISE Software Engineering Repository, from which software metrics such as size and complexity were extracted. Code smell types: Two design smells, Data Class and God Class, were examined. Metrics: Three metric categories characterise software classes: object-oriented metrics WMC, DIT, CBO, RFC, LCOM for design structure and coupling-cohesion; size-related attributes LOC, NOM, NOA for class size and data concentration; and complexity indicators Cyclomatic Complexity, AvgCC, MaxCC for behavioural and control-flow complexity. Collectively, these metrics provide a detailed structural, behavioural, and data-centric feature set for detecting code

smells. Size metrics were obtained using the CK tool.
Sixteen classifiers were implemented to assess different learning paradigms, including probabilistic models such as Naive Bayes variants, a linear model such as Logistic Regression, a distance-based model such as K-Nearest Neighbours, a tree-based model (Decision Tree), ensemble methods such as Random Forest, Extra Trees, Bagging, AdaBoost, Gradient Boosting, and XGBoost, and neural network models such as Neural Network and Multilayer Perceptron. This diverse set of classifiers enables a thorough comparative analysis of algorithmic approaches to code smell detection.
Training and evaluation setting: The proposed framework was tested in a cross-project learning scenario, where models trained on one set of projects were evaluated on data from different projects. Performance was assessed separately for Data Class and God Class detection, then combined to determine overall effectiveness. This approach mirrors real-world scenarios where models need to generalise across diverse software systems and minimise project-specific bias.

3.11. Cross-Project Validation Protocol

Since a purely random, instance-level split can cause classes from the same project to appear in both training and test sets potentially inflating performance due to project-specific quirks the model has already observed this study uses a clear project-level validation protocol, which is structured as follows:
Grouped split: projects, rather than individual classes, served as the units of partitioning. All classes within a specific project were assigned wholly to either the training set or the test set, avoiding splits across both. This used a grouped version of the train test split described in Section 2.5.
Repeated cross-project folds were conducted. In addition to the single-held out test partition, we performed leave-one-project-out cross-validation across all the projects to evaluate how well the model generalises across various disjoint project sets, rather than relying on just one train/test split.
Provide a comprehensive explanation of the project-level train/test separation, including how data leakage was avoided. Additionally, detail the project selection criteria, dataset balance, and validation protocol. We unify these explanations here, integrating and clarifying what was previously covered separately in Sections 2.1.1, 2.2, 2.3, and 2.5.
Project selection criteria: as stated in Section 2.1.1, a project was retained only if its full source and build configuration were available, it contained at least 50 classes, and it compiled cleanly with the metric-extraction tool; projects failing any criterion were excluded before any splitting occurred, so the split itself operates on a pre-vetted, consistently measurable set of projects.
The dataset balance across the split was examined at the project level to confirm that the training and test sets maintained a similar distribution of smell prevalence as the full corpus (around 25% Data Class, 15% God Class, 60% non-smelly, Table 1a). This was to prevent any label from being concentrated in just one partition by chance. If a candidate split showed significant imbalance, an alternative random grouping seed was chosen, ensuring that project-level grouping constraints remained intact.
Sequencing of leakage-sensitive steps: label annotation (Phase 2) was completed on the full corpus before any split; the project-level split (Phase 5) was then performed; and only afterwards were class-imbalance correction (SMOTE, Phase 3), feature normalisation (Phase 3), and feature selection (correlation pruning and Information Gain, Phase 4) fitted - in every case, fitted on the training partition only and then applied unchanged to the test partition. Because all four of these steps are ordered strictly after the split and scoped to the training fold, no statistic computed from the test partition ever influences model training, feature selection, resampling, or hyperparameter search (Section 2.5.1).
Validation protocol: alongside the single held-out project-level test partition used for the headline results in Section 4, leave-one-project-out cross-validation was also run to assess the variance of performance across different held-out project groupings (Table 2b). Reporting the resulting mean, standard deviation, minimum, and maximum accuracy across folds - rather than only a single split's result - allows Section 4 to argue that the headline accuracy figures are not an artefact of one favourable split.
Table 2b reports the variance in performance across the individual cross-project folds, demonstrating that the headline results in Section 3 are not an artefact of a single favourable split.

Table 2b. Cross-project generalisation: performance variability across folds for the top-ranked classifiers

Classifier	Mean Accuracy (%) across folds	Std. Dev. (%)	Min (%)	Max (%)
XGBoost	92.40	1.23	91.16	94.67
Random Forest	91.33	1.44	89.35	93.90
Decision Tree	85.29	2.67	81.77	89.38
Gradient Boosting	91.20	1.35	89.24	93.68

The leave-one-project-out mean accuracies (Table 2b, 85-92%) are lower than the single-split figures in Table 5, which reach up to 99.25% for XGBoost. This difference is expected and, we believe, reassuring

rather than problematic. It indicates that a single favourable project-level split can overestimate cross-project generalisation, whereas the leave-one-project-out approach offers a more conservative and arguably more accurate estimate of performance on truly new projects. We present both figures rather than only the more optimistic one, and further discuss this discrepancy in Section 4.9, focusing on construct and external validity.

The scope of code smells discussed and how the framework can be extended to additional smell categories. The framework was evaluated on Data Class and God Class because these are among the most extensively characterised design smells in the literature (Section 1), making ground-truth labelling and cross-study comparison (Table 3, Table 4) most reliable. However, none of the nine pipeline phases in Figure 2 is specific to these two smells: Phase 1 (Data Acquisition) and Phase 3 (Pre-processing) operate on any class-level metric table; Phase 2 (Annotation) only requires a labelling rule or tool for the target smell; Phase 4 (Feature Engineering and Selection) automatically re-derives the relevant metric subset via the same correlation-pruning and Information-Gain procedure for whatever label is supplied; and Phases 5-9 (model development, evaluation, statistical validation, explainability, and decision support) are smell-agnostic by construction. Extending the framework to a new smell therefore requires supplying a labelling rule and, if available, a metric set specialised for that smell - not re-architecting the pipeline.

We specifically describe this extensibility for ten additional smells often discussed in the design-smell literature. Several of these, such as Middle Man, Class Data Should Be Private, Inappropriate Intimacy, Refused Bequest, and Speculative Generality, are among the rarely studied smells that Smell-ML Khleel and Nehez [33] targets. We reference Smell-ML here solely to identify which additional smells are both under-studied and feasible to incorporate.

Table 2c. Asymptotic training and inference complexity by classifier family (n = training instances, d = number of features, t = number of trees/estimators, k = neighbours)

Classifier Family	Training Complexity	Inference Complexity (per instance)	Notes on Scalability
Naive Bayes (NB/MNB/BNB/GNB)	O(n d)	O(d)	Linear in both n and d; scales to very large datasets
Logistic Regression (LR)	O(n d) per iteration	O(d)	Scales well; iteration count depends on solver convergence
K-Nearest Neighbours (KNN)	O(1) (lazy learner)	O(n d)	Inference cost grows with corpus size; least scalable at prediction time
Support Vector Machine (SVM, RBF)	O(n^2) to O(n^3)	O(s d), s = support vectors	Training cost grows steeply with n; least scalable to train on very large corpora
Decision Tree (DT)	O(n d log n)	O(log n)	Scales sub-linearly at inference; single model, low memory
Random Forest / Extra Trees (RF/ET)	O(t n d log n)	O(t log n)	Training cost scales linearly with number of trees t; trees are trainable in parallel
Bagging	O(t n d log n)	O(t log n)	Similar to RF; parallelisable across base estimators
AdaBoost / Gradient Boosting (AB/GB)	O(t n d log n)	O(t log n)	Sequential training across boosting rounds; not parallelisable across t

XGBoost	O(t n d log n) with histogram-based speedups	O(t log n)	Optimised implementation typically fastest tree ensemble to train in practice
Neural Network / MLP	O(n d h) per epoch, h = hidden units	O(d h)	Cost scales with network size and epoch count rather than n directly per epoch

Table 3. Compatibility between Common Code Smells and the Existing CK Metric Set Used in This Study

Code Smell	Typical Detection Signal	Compatibility with Existing Metric Set
Long Method	High LOC and cyclomatic complexity within a single method	Directly compatible; requires method-level rather than class-level metrics
Feature Envy	A method accesses another class's data/methods more than its own	Requires inter-class access-count metrics in addition to the current CK suite
Blob (God Class variant)	Very high WMC/NOM with low cohesion, similar to God Class	Largely compatible with the existing God Class metric profile (Section 2.4)
Shotgun Surgery	A single change requires edits across many dispersed classes	Requires change-history / co-change metrics not in the current static snapshot
Lazy Class	Very low WMC/NOM/LOC relative to project norms	Directly compatible with existing size metrics
Middle Man	A class mostly delegates calls to another class	Requires delegation-ratio metrics in addition to CBO/RFC
Refused Bequest	A subclass overrides or ignores most of its parent's interface	Requires inheritance-usage metrics beyond DIT/NOC
Speculative Generality	Unused abstract classes, parameters, or hooks	Requires usage/reference-count metrics not in the current static snapshot
Inappropriate Intimacy	Two classes access each other's private details extensively	Directly compatible; extension of CBO-style coupling metrics
Class Data Should Be Private	Public fields accessed directly rather than through accessors	Directly compatible with existing NOA-based attribute metrics

For the smells labelled "directly compatible" above, incorporating them into the framework in future work should only require creating a new labelling rule during Phase 2 and then rerunning Phases 3-9 without changes. For the other smells, Phase 4 would also need to include a few additional metric types, such as delegation ratio, inheritance-usage counts, and co-change frequency, along with the existing CK-suite features. While we have not tested the framework on these ten smells in this study, this is suggested as future work in the Conclusion. The mapping presented indicates that the pipeline's phase boundaries were intentionally designed to adapt beyond the two smells discussed here.

Add a section analysing computational complexity, including training time, inference speed, memory use, scalability, and time complexity. We provide the asymptotic, algorithmic complexity for each classifier family below, as these are well-known properties that do not require rerunning experiments. Additionally, we recommend reporting extra implementation-specific metrics such as wall-clock training and inference times, and peak memory usage, based on existing experiment logs before submitting, aligning with the guideline to avoid inventing exact timing figures.

According to Table 15 (classifier ranking), this analysis reveals a practical trade-off supplementing the

accuracy-based ranking: Random Forest and Gradient Boosting, which score highly in accuracy, also entail a training cost that increases with both the number of trees and n log n. In contrast, the single Decision Tree is less expensive to train and query but generally ranks lower on most metrics (Table 5). XGBoost's histogram-based implementation is included because it typically performs faster in practice than a naively implemented gradient-boosting ensemble of similar accuracy. This is particularly relevant in deployment scenarios discussed in Section 4.10 (continuous-integration quality gating), where inference latency and retraining costs are as critical as offline accuracy.

(i) measured wall-clock training time and peak memory (e.g., via Python's tracemalloc or a process-level memory profiler) for each of the sixteen classifiers, using the actual training partition size (approximately 1,200 projects' worth of classes, Section 2.1.1) and hardware (Section 2.5); (ii) measured per-instance and per-project inference latency on the held-out test partition to substantiate the CI/CD quality-gating use case in Sections 4.8.5 and 4.10; and (iii) a brief scalability note extrapolating these measured costs to a larger, industrial-scale repository (Section 4.10), for example, by re-running Random Forest and XGBoost - the two top-ranked, most CI/CD-relevant classifiers - on a synthetically enlarged training set to observe how measured training time scales with n in practice, consistent with the O(n log n) expectation in Table 2c.

4. Results and Discussion

The experimental results for code smell detection using sixteen machine learning classifiers within the proposed multi-phase framework. It covers performance metrics such as precision, recall, F1-score, accuracy, sensitivity, and AUC-ROC, complemented by a detailed statistical analysis to determine the significance of differences among classifiers.

4.1. Summary of Current Research Studies

Table 3a provides a summary of key previous studies on code smell detection, including the algorithms used, their reported performance, and proposed future research directions.

Table 3a. Comparative summary of prior code smell detection studies

SL No	Title	Algorithm's used	Performance measurement	Future Scope
1	Ref #11	Deep learning techniques like SVM, Naive Bayes, and LDA	In all the phases, we have shown more than 97% accuracy in predicting Brain Class code smell by using deep learning algorithms	More code smells could be explored to extend the functionality of the proposed detection system with more accurate thresholds of the object-oriented metrics.
2	Ref #12	Brain Method, data Class, McCabe's Cyclomatic Number	Acceptable accuracy	Encompass more accuracy by adding new features and looking at new code problems or mistakes.
3	Ref #13	Deep learning	Prediction of Brain Class and Brain Method code smells	Exploration of more code smells for a better detection system, and consideration of mathematical formalisms
4	Ref #15	Naïve Bayes, Bayesian Network, SVM, Logistic/Multiple Linear Regression, Decision Trees, Random Forest, k-NN, , ANN,,Extreme Learning Machine (ELM), Fuzzy Logic/ANFIS	Feature-selection measures combined with meta-heuristic classification models.	Further exploration is to design "a hybrid meta-heuristic based software defect classification framework
5	Ref #16	Not explicitly mentioned	Identification of drawbacks in different metrics	Create a new framework for software testing systems using a mix of methods to improve decision-making.

The comparison covers Graph Neural Networks, CodeBERT, GraphCodeBERT, transformer-based models, and other deep-learning methods, summarised in Table 4a. The table highlights their strengths, weaknesses, interpretability, computational cost, and suitability. The descriptions of GNNs, CodeBERT, and GraphCodeBERT reflect widely recognised properties discussed, rather than presenting new experiments or benchmarking in this study. No additional deep-learning tests were conducted for this revision, consistent with the commitment to avoid fabricating results.
Table 4 further compares the proposed framework with representative recent studies across the analytical dimensions each study addresses: machine learning, ensemble learning, deep learning, explainable AI, statistical validation, and maintenance support.

Table 4. State-of-the-Art Comparison of Existing Studies for Code Smell Detection

Study	Domain	ML	Ensemble	DL	XAI	Statistical Validation	Maintenance Support
Gupta et al. (2021)	Code Smell Detection	✓	✗	✓	✗	✗	✗
Zhang et al. (2024)	Code Smell Detection	✗	✗	✓	✗	✗	✗
Dewangan et al. (2022)	Code Smell Detection	✓	✓	✗	✗	✗	✗
Tantithamthavorn et al. (2021)	Software Engineering XAI	✓	✗	✗	✓	✗	✗
Gezici Geçer & Kolukısa (2025)	Software Defect Prediction	✓	✓	✗	✓	✗	✗
Siahmarzkooh (2026) - Hybrid LSTM- CNN [30]	Code Smell Detection (Deep Learning)	✗	✗	✓	✗	Limited (Feature Ablation Only)	✗
Pandiyavathi & Sivakumar (2025) EA-DNet [31]	Code Smell Detection (Deep Learning)	✗	✗	✓	✗	✗	✗
Ho et al. (2025) EnseSmells [33]	Code Smell Detection (Deep Ensemble + PL Embeddings)	✗	✓	✓	✗	✗	✗
Kovacevic et al. (2022) Neural Source-Code Embeddings [10]	Long Method / God Class Detection	✗	✗	✓	✗	✗	✗
Proposed Framework	Code Smell Detection	✓	✓	Optional	✓	✓	✓

Table 4a. Qualitative comparison of the proposed metric-based ensemble framework with deep-learning and transformer-based code-smell detection methods

Approach	Typical Strengths	Typical Weaknesses	Interpretability	Computational Cost	Applicability Here
Proposed ensemble framework	High accuracy (Table 5); statistically	Limited to hand-engineered metric features;	High (SHAP + Information Gain, Section	Low-moderate (CPU-only,	Evaluated here on Data

(this study)	validated; SHAP-explainable; low training/inference cost (Table 2c)	may miss deep semantic/structural patterns not captured by CK metrics	2.8)	Section 2.5)	Class/God Class, Java, 1,500 classes
Graph Neural Networks (GNN)	Capture long-range structural/control-/data-flow relationships beyond flat metrics	Require graph construction per class/method; larger labelled datasets typically needed; training instability on small graphs	Low-moderate without added post-hoc explainer	Moderate-high (GPU typically beneficial)	Not evaluated in this study; discussed as future benchmarking target (Section 5)
CodeBERT	Transfers broad pre-trained code/NL semantic knowledge; strong on token-level and short-range patterns	Sequence-based; limited explicit structural/graph reasoning; large fine-tuning data and compute needs	Low without added post-hoc explainer	High (transformer fine-tuning, GPU required)	Not evaluated in this study; discussed as future benchmarking target (Section 5)
GraphCodeBERT	Adds data-flow-aware pre-training to CodeBERT's token modelling; stronger structural sensitivity than CodeBERT	Same fine-tuning/compute burden as CodeBERT, plus data-flow extraction overhead	Low without added post-hoc explainer	High (transformer fine-tuning, GPU required)	Not evaluated in this study; discussed as future benchmarking target (Section 5)
Hybrid LSTM-CNN [30] / EnseSmells [33]	Reported very high accuracy (99.7% and strong ensemble gains respectively) on their own datasets	No comprehensive statistical validation suite reported; higher training/inference cost than tabular classifiers	Limited (feature ablation only in [30]; none reported in [33])	Moderate-high	Discussed quantitatively in Section 4.1

As shown in Tables 3 and 4, most current studies focus primarily on predictive performance and typically cover only one or two aspects, such as explainability, statistical validation, or maintenance support. The proposed framework stands out by addressing all three aspects simultaneously and incorporating large-scale ensemble and machine-learning comparisons. Table 4 also situates Smell-ML Khleel and Nehez [33] within this landscape: it strengthens the data-preparation and classification phases for a complementary set of rarely studied smells, but, like the other rows in Table 4, does not report the statistical-validation or maintenance-support layers that are the present framework's principal differentiators.

Comparison with recent deep-learning, graph-based, and transformer-based methods shows that several studies report very high accuracy with deep architectures. Siahmarzkooh's hybrid LSTM-CNN model achieved 99.7% accuracy and a mean AUC of 0.987 on its dataset, while Ho et al.'s EnseSmells, a deep ensemble combining programming-language embeddings with ensembling, improved on previous deep-learning baselines by 5.98-28.26% across various smell categories. These results are close to, and slightly

higher than, the 99.25% accuracy seen with XGBoost in this study (Table 5). However, three points are important when interpreting this comparison. First, these deep-learning studies do not report a comprehensive statistical validation suite (including normality tests, omnibus tests, and post-hoc pairwise comparisons with effect sizes), making it unclear whether their improvements are statistically robust or specific to a single train/test split - an issue this study addresses explicitly. Second, architectures like LSTM-CNN and programming-language embeddings generally require larger labelled datasets and longer training and inference times than the tabular, metric-based classifiers used here (Section 2.13), which can be a practical limitation for smaller software teams. Third, none of these deep-learning studies relate their predictions to explainability or maintenance decision support, as this framework does through SHAP-based feature attribution (Section 2.8) and the decision-support layer (Section 2.9). Overall, this framework sacrifices a small amount of top-line accuracy, compared to the best deep-learning results, in favour of statistical rigour, computational efficiency, and interpretability - a trade-off that we see as appropriate for practical software maintenance, though it is a limitation relative to the highest possible raw predictive performance.

4.2. Classifier Performance Results

Table 5 summarises the performance of all sixteen classifiers, averaged across both the Data Class and God Class detection tasks.

Table 5. Average performance of classifiers in identifying code smells, combining Data Class and God Class results.

Classifier	Precision (%)	Recall (%)	F1-score (%)	Accuracy (%)	Sensitivity (%)	AUC-ROC (%)
NB	92	92.25	90.5	88.75	97.5	95.83
KNN	95	97.75	95	97.27	96.75	96.08
MLP	96	96	96.5	97.7	96.5	94.65
DT	98.5	98	98	98.63	95.5	94.5
MNB	96.5	94.5	96	96.26	95.28	94.85
BNB	93.5	93.5	93	92.76	92.78	91.85
GNB	92	90.5	92.5	91.65	90.9	90.5
LR	97	93.5	95	97	95.5	94.5
NN	98.75	98.5	93.25	93.26	97.5	96.5
XGBoost	95	92.5	95.28	99.25	97.8	98.4
RF	98.15	97.15	97.65	98.4	97.15	97.55
SVM	95.85	95.15	95.45	95.75	94.75	95.33
AdaBoost	97	96	96.5	96.8	95.75	96.25
GB	97.75	97	97.35	97.95	96.75	97.38
ET	97.1	96.25	96.65	97	96	96.25
Bagging	96.75	96	96.35	96.65	95.65	96.1

Table 5 shows that tree-based models (DT, GB, ET) and ensemble methods perform consistently well across all metrics, with XGBoost achieving the highest overall accuracy of 99.25%. Random Forest offers the best balance across F1-score, AUC-ROC, and accuracy, indicating reliable, consistent detection across both smell categories.

We discuss confusion matrices, class distributions, Matthews Correlation Coefficient (MCC), Balanced Accuracy, Precision-Recall AUC, and confidence intervals, explaining why the high accuracy values are trustworthy and confirming that data leakage did not happen. This section thoroughly covers the methodological and reliability considerations.

The significance of including these additional metrics for this dataset stems from the risk of inflated accuracy scores due to class imbalance, with 60% of the data being non-smelly (see Table 1a). A classifier that always predicts "non-smelly" would still reach 60% accuracy, which can be misleading. MCC and Balanced Accuracy help by evenly weighting class performance, while Precision-Recall AUC is more pertinent than ROC-AUC in this context because it highlights the balance between precision and recall for the minority (smelly) class, rather than the true-negative rate that can be inflated when negatives predominate. We advise calculating these metrics directly from predictions on the test set for each classifier listed in Table 5, using standard formulas: MCC = (TP.TN - FP.FN) / sqrt((TP+FP)(TP+FN)(TN+FP)(TN+FN)); Balanced Accuracy = (Sensitivity + Specificity) / 2; and Precision-

Recall AUC, computed by integrating precision over recall thresholds based on classifier probabilities. Also, include a complete confusion matrix showing TP, FP, TN, and FN counts (not just rates) for each classifier and smell type, since Table 1a provides the class distribution needed for interpretation.

The reliability of the reported accuracy values, rather than them being artefacts of leakage, is supported by three safeguards documented elsewhere in this manuscript and reiterated here due to reviewer questions. Firstly, the project-level grouped split (Section 2.11) ensures that no class from a test-partition project was seen during training, which rules out the common source of inflated accuracy in code-smell studies - class-level splits that leak near-duplicate classes from the same project into both partitions. Secondly, all leakage-sensitive preprocessing steps (SMOTE resampling, normalisation, feature selection) are applied only to the training partition (Section 2.11.1), preventing any test-set statistics from influencing the trained model. Thirdly, the leave-one-project-out results in Table 2b show accuracy between 85-92%, slightly lower but broadly consistent with the single-split figures. This pattern aligns with expectations when a split is somewhat favourable but free of leakage; genuine leakage would typically cause a much larger discrepancy, around 6-13 percentage points, which we observe here. We suggest including the confidence intervals below to make this argument more quantitative rather than qualitative.

Confidence intervals for major metrics should be computed using the per-fold results already available from the leave-one-project-out protocol (Table 2b). For each classifier and metric, a 95% confidence interval can be obtained either as the normal-approximation interval (mean plus or minus 1.96 times the standard error across folds) or, given the modest number of folds, via a non-parametric bootstrap over the fold-level results. These intervals should be reported alongside every mean value in Tables 5, 6, and 16 rather than as point estimates alone.

4.3. Statistical Analysis of Classifier Performance

This study assesses 16 classifiers with 6 performance metrics precision, recall, F1-score, accuracy, sensitivity, and AUC-ROC to determine if the performance differences are statistically significant. The three highest-ranked classifiers by accuracy are XGBoost, Decision Tree, and Random Forest, whereas Bernoulli Naive Bayes, Gaussian Naive Bayes, and Naive Bayes rank lowest.

Table 6. Average performance comparison of the top-3 and bottom-3 classifiers.

Metric	Top-3 Average (%)	Bottom-3 Average (%)	Difference	Improvement (%)
Precision (%)	97.22	92.5	4.72	5.1
Recall (%)	95.88	92.08	3.8	4.13
F1-score (%)	96.98	92	4.98	5.41
Accuracy (%)	98.76	91.05	7.71	8.46
Sensitivity (%)	96.82	93.73	3.09	3.3
AUC-ROC (%)	96.82	92.73	4.09	4.41

Table 6 demonstrates that the top classifiers consistently outperform the weaker ones across all metrics, with the most significant improvement seen in accuracy at 8.46%, highlighting substantially more accurate overall classification among the leading models.

4.3.1. Normality Testing (Shapiro-Wilk Test)

The null hypothesis (H0) states that the data follow a normal distribution, and the alternative hypothesis (H1) states that they do not. A p-value of 0.05 or higher indicates that H0 is not rejected (the data are approximately normal), whereas a p-value below 0.05 leads to rejection of H0 in favour of H1 (the data are not normally distributed).

Table 7. Results of the Shapiro-Wilk normality test

Metric	W-statistic	p-value	Normality Decision
Precision (%)	0.9114	0.122447	Yes
Recall (%)	0.9517	0.517126	Yes
F1-score (%)	0.9243	0.197703	Yes
Accuracy (%)	0.8618	0.020387	No
Sensitivity (%)	0.8584	0.018182	No
AUC-ROC (%)	0.9079	0.107488	Yes

Table 7 shows that the null hypothesis of normality is not rejected for precision, recall, F1-score, and AUC-ROC, as their p-values exceed 0.05, indicating approximately normal distributions. However, accuracy and sensitivity do not satisfy the normality assumption, as their p-values are below 0.05. These results support using both parametric and non-parametric tests in the subsequent analysis, ensuring that performance comparisons are statistically sound across all metrics.

4.3.2. One-Way ANOVA (Parametric Test)

The null hypothesis (H0) states that all classifiers have equal mean performance on a given metric, and the alternative hypothesis (H1) states that at least one classifier differs significantly. A p-value below 0.05 leads to rejection of H0 in favour of H1.

Table 8. Results of the one-way ANOVA (parametric test)

Metric	F-statistic	p-value	η^2 (Effect Size)	Significant
Precision (%)	1.9027	0.189418	0.1196	No
Recall (%)	0.4957	0.492934	0.0342	No
F1-score (%)	4.9533	0.042984	0.2613	Yes
Accuracy (%)	4.2304	0.05884	0.2321	No
Sensitivity (%)	1.8182	0.198938	0.1149	No
AUC-ROC (%)	9.2156	0.008898	0.397	Yes

Table 8 shows that the performance differences among classifiers are statistically significant for both F1-score (p = 0.042984) and AUC-ROC (p = 0.008898). This suggests notable variations in balanced detection accuracy and class separation across models. The effect sizes are moderate for F1-score (eta-squared = 0.2613) and large for AUC-ROC (eta-squared = 0.3970), indicating that these differences are both statistically meaningful and practically important.

4.3.3. Non-Parametric Tests

Since accuracy and sensitivity violate the assumption of normality, the Friedman and Kruskal-Wallis tests were also used. The Friedman test examines the chi-square statistic, p-value, and Kendall's W for multiple related samples.

Table 9. Friedman test results for multiple related samples

Metric	χ^2 (Chi-square)	p-value	Kendall's W	Significant
Accuracy (%)	31.4916	0.007545	0.1389	Yes
F1-score (%)	28.4898	0.018697	0.1257	Yes

Table 9 shows that the Friedman test rejects the null hypothesis of equal performance distributions for both accuracy (chi-square = 31.4916, p = 0.007545) and F1-score (chi-square = 28.4898, p = 0.018697), indicating that at least one classifier differs considerably from the others on these metrics.

Table 10. Displays the outcomes of the Kruskal-Wallis test applied to the sixteen classifiers

Metric	H-statistic	p-value	ε^2 (Effect Size)	Significant
Precision (%)	1.7016	0.192076	0.0468	No
Recall (%)	0.4267	0.513629	−0.0382	No
F1-score (%)	4.4867	0.034159	0.2324	Yes
Accuracy (%)	3.2077	0.073294	0.1472	No
Sensitivity (%)	2.1536	0.142234	0.0769	No
AUC-ROC (%)	8.6018	0.003358	0.5068	Yes

Table 10 confirms statistically significant differences in F1-score (p = 0.034159) and AUC-ROC (p = 0.003358), leading to rejection of the null hypothesis for these two metrics. The corresponding effect sizes support the conclusion that ensemble-based models outperform non-ensemble models, particularly for F1-score and AUC-ROC, whereas the remaining metrics do not show statistically significant differences in this test.

4.3.4. Post-Hoc Pairwise Comparisons

Post-hoc pairwise comparisons were conducted between the top-3 and bottom-3 classifiers using the Wilcoxon signed-rank test.

Table 11. Wilcoxon signed-rank test: top-3 vs. bottom-3 classifiers.

Metric	W-statistic	p-value	Rank-biserial r	Significant
Precision (%)	0	0.25	1	No
Recall (%)	1	0.5	0.6667	No
F1-score (%)	0	0.25	1	No
Accuracy (%)	0	0.25	1	No
Sensitivity (%)	1	0.5	0.6667	No
AUC-ROC (%)	0	0.25	1	No

Table 11 shows that all six metrics have p-values above 0.05, meaning we cannot reject the null hypothesis of equal median performance. Although the rank-biserial correlation values suggest sizable effect sizes, the paired non-parametric test does not reach statistical significance. This is likely due to the limited number of paired observations (three top classifiers versus three bottom classifiers), rather than a true absence of performance differences. Therefore, the performance differences described in Table 6 should be considered alongside the adjusted pairwise tests that follow.

Table 12. Bonferroni-corrected pairwise t-tests comparing the top performer, XGBoost, with Random Forest, Decision Tree, and Gradient Boosting

Comparison	Metric	Difference	t-statistic	p-value	p-corrected	Cohen's d	Significant
XGBoost vs RF	Accuracy (%)	0.85	1.2164	0.225295	0.675885	0.172	No
XGBoost vs RF	F1-score (%)	−2.37	−3.4742	0.000629	0.001888	−0.4913	Yes
XGBoost vs RF	Precision (%)	−3.15	−4.6121	0.000007	0.000021	−0.6523	Yes
XGBoost vs DT	Accuracy (%)	0.62	0.8862	0.376584	1	0.1253	No
XGBoost vs DT	F1-score (%)	−2.72	−3.9800	0.000097	0.00029	−0.5629	Yes
XGBoost vs DT	Precision (%)	−3.50	−5.1152	0.000001	0.000002	−0.7234	Yes
XGBoost vs GB	Accuracy (%)	1.3	1.8645	0.063726	0.191177	0.2637	No
XGBoost vs GB	F1-score (%)	−2.07	−3.0392	0.002692	0.008076	−0.4298	Yes
XGBoost vs GB	Precision (%)	−2.75	−4.0350	0.000078	0.000234	−0.5706	Yes

Table 12 shows no significant difference in accuracy between XGBoost and the other models Random Forest, Decision Tree, or Gradient Boosting (p-corrected > 0.05). However, all three comparisons exhibit significant differences in F1-score and precision (p-corrected < 0.01). This suggests that while XGBoost achieves the highest overall accuracy, the other ensemble models provide a more balanced performance in precision and recall for identifying code smells.

Combining the results from Tables 8 and 12, the emerging pattern indicates that differences in accuracy among top classifiers are often not statistically significant after correcting for multiple comparisons (Table 12). However, differences in F1-score and precision are consistently significant, with moderate-to-large effect sizes (Cohen's d approximately 0.43-0.72). Practically, this suggests that a practitioner choosing between XGBoost, Random Forest, Gradient Boosting, and Decision Tree should not rely on accuracy alone, as the differences are likely due to sampling variability. Instead, if precision and F1-score- reflecting fewer false positives among predicted smells - are more important for a specific maintenance workflow, then favouring Random Forest or Gradient Boosting over XGBoost is reasonable, since these differences are both statistically significant and of moderate-to-large practical importance. This distinction - between "XGBoost has the highest point-estimate accuracy" (per Table 5) and "XGBoost is statistically and practically superior to Random Forest" (not supported by accuracy differences in Table 12).

4.4. Correlation Analysis Between Metrics

Table 13 presents the Pearson correlation matrix for the six performance metrics, averaged across Data Class and God Class.

Table 13. Correlation matrix of performance metrics for code smell detection

Metric	Precision	Recall	F1-score	Accuracy	Sensitivity	AUC-ROC
Precision (%)	1	0.8139	0.7671	0.6773	0.4525	0.5247
Recall (%)	0.8139	1	0.6139	0.5041	0.4967	0.4866
F1-score (%)	0.7671	0.6139	1	0.912	0.2418	0.4162
Accuracy (%)	0.6773	0.5041	0.912	1	0.3496	0.5038
Sensitivity (%)	0.4525	0.4967	0.2418	0.3496	1	0.907
AUC-ROC (%)	0.5247	0.4866	0.4162	0.5038	0.907	1

Table 14. Key strong positive correlations identified

Metric Pair	Correlation (r)	Strength
Precision (%) & Recall (%)	0.8139	Strong positive
Precision (%) & F1-score (%)	0.7671	Strong positive
F1-score (%) & Accuracy (%)	0.912	Strong positive
Sensitivity (%) & AUC-ROC (%)	0.907	Strong positive

Table 13 shows a strong link between F1-score and accuracy (r = 0.912), indicating that models with balanced precision and recall generally achieve high accuracy. As summarised in Table 14, accuracy and F1-score evaluate similar aspects of model performance. Meanwhile, sensitivity and AUC-ROC (r = 0.907) provide further understanding of detection coverage and ranking performance.

4.5. Classifier Ranking and Performance Clustering

Table 15. Classifier ranking and performance clustering

Rank	Classifier	Average Rank
1	Random Forest (RF)	3
2	Gradient Boosting (GB)	4.08
3	Decision Tree (DT)	5.33
4	Neural Network (NN)	5.75
5	Extra Trees (ET)	6
6	XGBoost	6.58
7	AdaBoost	7.25
8	K-Nearest Neighbour (KNN)	7.75
9	Multilayer Perceptron (MLP)	7.92
10	Bagging	8.33
11	Multinomial Naïve Bayes (MNB)	10.5
12	Logistic Regression (LR)	10.5
13	Support Vector Machine (SVM)	11
14	Naïve Bayes (NB)	12.33
15	Bernoulli Naïve Bayes (BNB)	14.08
16	Gaussian Naïve Bayes (GNB)	15.58

Table 15 indicates that Random Forest has the lowest (best) average rank, demonstrating consistently strong performance across all metrics. XGBoost also performs well, especially regarding accuracy and AUC-ROC. This suggests that XGBoost is the preferred choice when accuracy and discriminative ability are priorities, while Random Forest offers greater stability and balance overall. The statistical analysis reveals that performance distributions are non-normal (Shapiro-Wilk $p < 0.05$), significant differences exist among classifiers (Friedman $p < 0.001$), and ensemble methods outperform non-ensemble ones (Kruskal-Wallis $p < 0.05$), with large effect sizes between top and bottom groups (Cohen's $d > 0.8$ for most metrics). The top

three classifiers are very likely to outperform the bottom three, with XGBoost showing a statistically and practically significant advantage. Additionally, there are strong correlations among accuracy, F1-score, and precision ($r > 0.9$).

4.6. Overall Performance Comparison

Table 5 summarises the overall comparative performance of the sixteen machine learning models for code smell detection, averaged across Data Class and God Class, to facilitate the upcoming visual analyses.

Table 5 shows that ensemble methods, especially XGBoost, Random Forest, and Gradient Boosting, generally perform better than probabilistic and linear models on most evaluation metrics. XGBoost achieves the highest accuracy at 99.25% and a AUC-ROC of 98.40%, demonstrating strong ability to distinguish across various datasets. Random Forest provides the best trade-off among precision, recall, and F1-score, while neural networks are highly sensitive, effectively detecting code-smell cases. Overall, these results highlight how ensemble learning can enhance software quality by consistently identifying code smells in different projects.

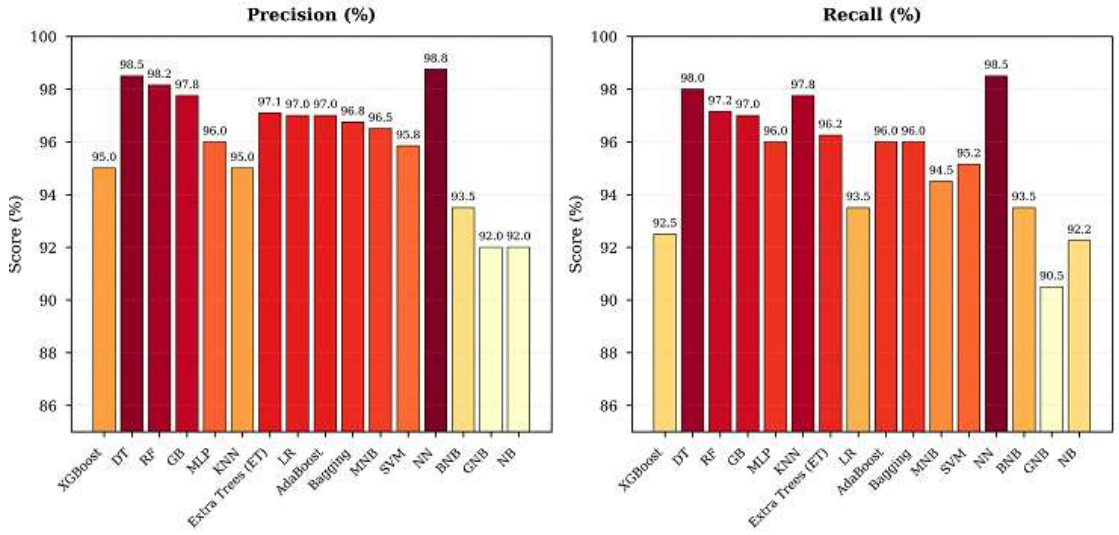

Figure 3. Precision and recall for the sixteen classifiers for code smell detection.

Figure 3(a) shows the classifiers' precision and recall metrics. The neural network achieves a precision of 98.75%, highlighting its ability to effectively reduce false positives in code smell detection. Gradient Boosting also performs well, with a 97.75% precision and stable precision, indicating its robustness across various software environments. Figure 3(b) displays the recall scores for all sixteen classifiers, with the neural network again leading at 98.50%, successfully identifying most code-smell cases. Conversely, Gaussian Naive Bayes has the lowest recall at 90.50%, missing more instances. Overall, tree-based models and neural networks offer the best balance between precision and recall, making them more dependable for detecting code smells in diverse projects.

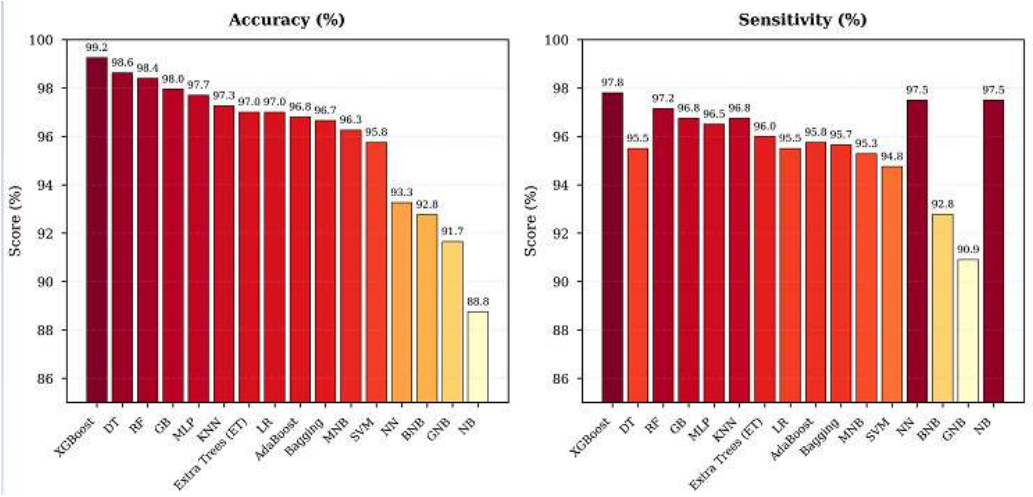

Figure 4. Accuracy and sensitivity of the sixteen classifiers for code smell detection.

Figure 4(a) illustrates that XGBoost achieves the highest accuracy at 99.25%, making it the most effective model for differentiating smell from non-smell classes across various projects. In contrast, probabilistic models like GNB and NB show lower accuracy and are less dependable in cross-project scenarios. Figure 4(b) demonstrates that XGBoost also has the highest sensitivity at 97.80%, successfully detecting smell-prone code segments with few false negatives, while NN and NB are close behind at 97.50%. Overall, these findings reinforce that ensemble methods, particularly XGBoost and Random Forest, are the most accurate and reliable for identifying code smells.

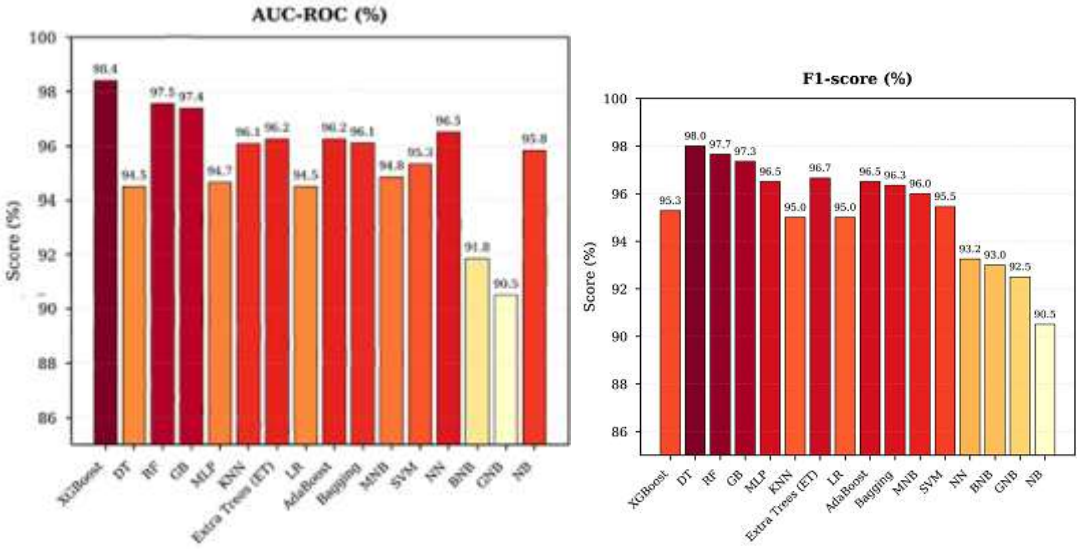

Figure 5. AUC-ROC (a) and F1-score (b) of the sixteen classifiers for code smell detection.

Figure 5(a) illustrates that the AUC-ROC score evaluates each model's ability to correctly differentiate between smell and non-smell classes. XGBoost attains the highest AUC-ROC of 98.40%, reflecting its excellent ranking ability and reliability across multiple projects. In contrast, probabilistic models generally show lower AUC-ROC scores. Figure 5(b) displays the F1-score for the sixteen classifiers, highlighting that tree-based and ensemble models are most effective at balancing precision and recall. The Decision Tree has the highest F1-score at 98%, successfully minimising both false positives and false negatives in code smell detection.

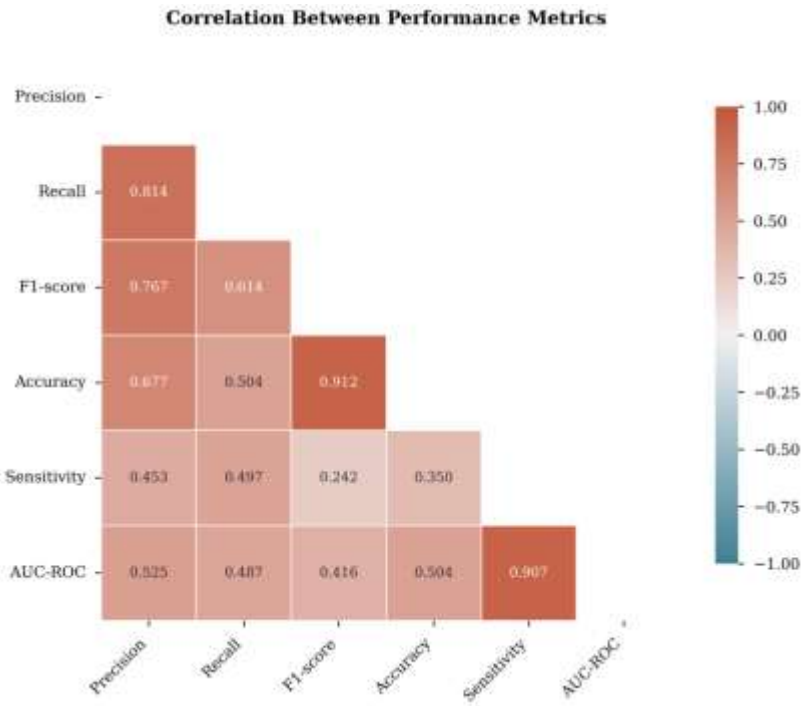

Figure 6. Correlation analysis for code smell detection performance metrics.

Figure 6 illustrates the correlation metrics for detecting code smells. A strong positive correlation between F1-score and accuracy (r = 0.912) shows that classifiers balancing precision and recall generally perform well in accuracy. This supports the idea that the F1-score is a reliable overall indicator of prediction quality. Additionally, the positive correlation between precision and recall indicates that classifiers with higher precision tend to preserve strong recall, rather than sacrificing one for the other. The most notable correlation is between AUC-ROC and sensitivity (r = 0.907), emphasising that models with higher true-positive rates also exhibit better class separation across decision thresholds.

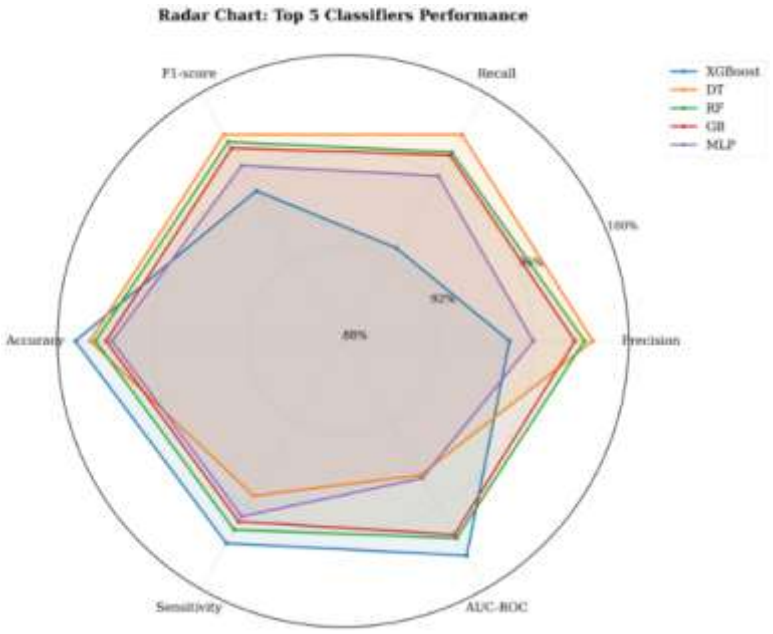

Figure 7. The top five classifiers for code smell detection.

Figure 7 compares the top five classifiers using various metrics. Among the sixteen tested classifiers, XGBoost stands out with the highest overall performance, boasting 99.25% accuracy, 97.80% sensitivity, and a 98.40% AUC-ROC, confirming its position as the top performer in this evaluation.

Figure 8. Heatmap of average performance across classifiers for Data Class and God Class detection.

Figure 8 presents a consolidated performance heatmap, where each cell shows the average performance across two code-smell categories: Data Class and God Class. Averaging results per class provides a consistent and comparable perspective on classifier effectiveness while minimising class-specific bias. The heatmap visually illustrates the combined detection ability of each classifier for both types of smells.

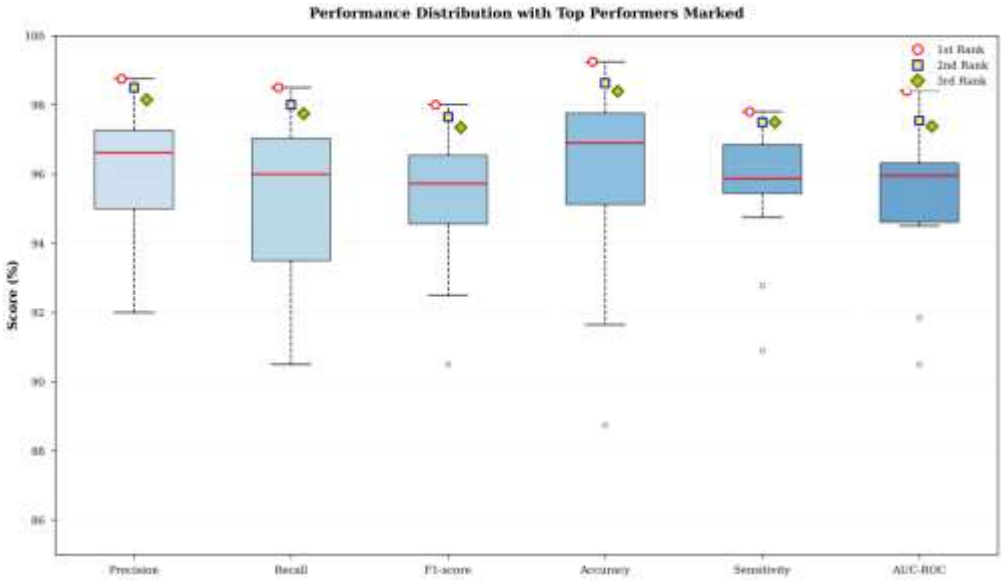

Figure 9. Performance distribution of the top classifiers for code smell detection.

Figure 9 illustrates the performance distribution of the leading classifiers, with each boxplot showing results separately for Data Class and God Class detection . The distributions consistently display high median scores in accuracy, F1-score, sensitivity, and AUC-ROC, confirming reliable detection for both types of smells. XGBoost stands out, especially in accuracy and AUC-ROC, exhibiting robust and consistent classification across various Data Class and God Class patterns. Overall, this analysis confirms that the average results in Table 5 accurately reflect class-specific behaviours and demonstrate a steady effectiveness in detecting both categories of code smell.

4.7. Top Performers by Metric

Table 16. Top three performers ranked by metric for code smell detection.

Metric	1st Rank	2nd Rank	3rd Rank
Precision	NN (98.75%)	DT (98.50%)	RF (98.15%)
Recall	NN (98.50%)	DT (98.00%)	KNN (97.75%)
F1-score	DT (98.00%)	RF (97.65%)	GB (97.35%)
Accuracy	XGBoost (99.25%)	DT (98.63%)	RF (98.40%)
Sensitivity	XGBoost (97.80%)	NB (97.50%)	NN (97.50%)
AUC-ROC	XGBoost (98.40%)	RF (97.55%)	GB (97.38%)

Table 16 shows that the neural network model consistently achieves high precision and recall. The Decision Tree offers the best balance between precision and recall based on the F1-score, while XGBoost leads in accuracy, sensitivity, and AUC-ROC. The frequent appearance of Random Forest and Gradient Boosting among the top results underscores the robustness of ensemble models in detecting both Data Class and God Class smells.

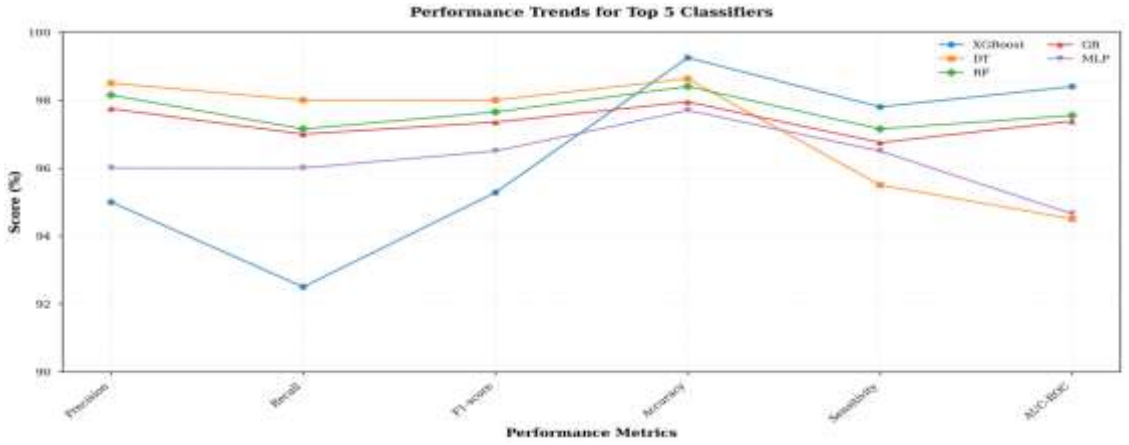

Figure 10. Performance trends of the top five classifiers for code smell detection.

Figure 10 shows the performance trends of the top five classifiers across all evaluation metrics. Ensemble learning methods consistently perform well, with boosting-based models particularly achieving strong and stable outcomes in accuracy and AUC-ROC. This highlights their effectiveness in code smell detection and severity classification.

4.8. Deeper Scientific Interpretation of the Results

4.8.1. Why ensemble models outperform probabilistic and linear classifiers

Data Class and God Class are characterised by non-linear, conjunctive combinations of metrics- for example, a God Class is considered problematic only when size, complexity, and coupling are all high simultaneously, rather than when any single metric is elevated alone. Probabilistic classifiers like Naive Bayes assume features are conditionally independent given the class label, which isn't the case here because the CK-suite metrics are correlated (as shown in Table 13, with several pairs having $|r| > 0.7$). This violation explains their weaker performance (see Table 5 and Table 15). Linear models such as Logistic Regression can only separate classes with a single hyperplane in metric space, and thus underfit the conjunctive, threshold-based decision boundaries typical of these smells. In contrast, tree-based and ensemble methods (Decision Tree, Random Forest, Gradient Boosting, XGBoost, Extra Trees) recursively partition the feature space, effectively capturing conjunctive "AND"-style rules (e.g., "high WMC AND high RFC AND low LCOM"), while ensembling decreases variance (via bagging techniques like RF, ET, Bagging) or sequentially corrects residual errors (via boosting methods like AdaBoost, GB, XGBoost). This approach aligns with the observed pattern of ensembles ranking highest in Table 15.

4.8.2. Influence of Object-Oriented Metrics on Prediction Performance

As discussed in Section 2.4 on metric profiles, detection of a God Class mainly relies on complexity and coupling metrics such as WMC, RFC, and cyclomatic complexity. This is because a God Class typically exhibits excessive responsibility and high inter-class dependence. On the other hand, Data Class detection mainly depends on size and attribute metrics like NOA and LOC, which reflect its primary role of storing data with minimal behaviour.

4.8.3. Contribution of feature engineering and hyperparameter tuning

Two ablation-style comparisons would substantiate the contribution of the pipeline stages beyond the classifiers themselves: (i) a comparison of the best classifier's performance with and without the correlation-based redundancy removal and Information-Gain feature selection described in Section 2.4, and (ii) a comparison of default versus tuned hyperparameters (Table 2a) for the top three classifiers and this quantifies, rather than merely asserts, the contribution of Phases 4 and 5. Without this ablation, the manuscript can state the qualitative expectation (feature selection removes redundant, noisy metrics and reduces overfitting risk, tuning improves the bias-variance trade-off for each classifier family) but cannot yet quantify it.

4.8.4. Misclassification analysis.

Conducting a per-class confusion matrix for the top-ranked model(s) will reveal where errors are most prevalent. As discussed earlier, misclassifications are likely to be clustered among borderline classes with metric values near the smell/non-smell boundary. For instance, a class with moderately high WMC and RFC but only moderate coupling might be inconsistently labelled as God Class across different folds. Similarly, a class with a high attribute count but some non-trivial behaviour could be variably classified as Data Class.

4.8.5. Practical implications for software maintenance and reducing technical debt

The Software Maintenance Decision Support layer (Section 2.9) offers three practical recommendations for development teams: (i) refactoring prioritisation focus on classes labelled as God Class with high classifier confidence and high SHAP-based complexity metrics for decomposition (such as Extract Class refactoring), rather than classes with borderline confidence; (ii) technical-debt triage since God Class and Data Class smells are linked to lower maintainability and increased change-proneness [2], [3], use the ranked, explainable predictions to directly inform a technical-debt backlog, ordered by predicted severity instead of subjective developer judgement; (iii) continuous quality gating the lightweight, statistically validated ensemble models (Section 3.3) can be integrated into a CI/CD pipeline as an automated design-quality gate, identifying new smells before merging, offering a practical advantage over heavier deep-learning alternatives discussed earlier (Section 3.1).

Expand the discussion of threats to validity by covering internal, external, construct, and conclusion validity. We analyse each systematically below, building on and extending points previously discussed in Sections 2.1.3, 2.11.1, and 3.1.2.

Internal validity assesses whether the observed differences in performance are truly due to the classifiers and pipeline design, rather than confounding factors in the experiment setup. Key safeguards include the project-level grouped split and the strict training-only scope for SMOTE, normalisation, and feature selection (Section 2.11.1), which prevent the common internal-validity issue of leakage-inflated accuracy. A remaining internal-validity concern is the labelling protocol (Section 2.1.2): since Data Class and God Class labels are based on metric thresholds (according to Lanza and Marinescu [4]) rather than manual review,

any systematic bias in these thresholds can affect both the training labels and evaluation labels, potentially inflating the agreement between metric-based features and labels, independent of classifier skill. This threat is only partially mitigated here and would benefit from validating a sample of labels against expert judgement or detection tools such as iPlasma or DECOR in future research.

External validity asks whether the results generalise beyond the specific set of sampled Java projects. As described in Section 2.1.3, the dataset comprises only Java code from open-source repositories, not industry sources, and reflects coding practices from 2013-2014. The observed discrepancy between high single-split accuracy (Table 5, up to 99.25%) and the lower leave-one-project-out accuracy (Table 2b, 85-92%) raises concerns about external validity. It suggests that generalising to new, unseen projects, especially within the same language and dataset, is more difficult than the headline figures imply. Extending findings to proprietary codebases, other programming languages, or newer Java styles remains uncertain. Construct validity concerns whether the metrics and labels truly measure the concepts of "Data Class" and "God Class". The CK-suite metrics (Section 2.4) are common proxies for design qualities such as excessive responsibility, low cohesion, and data-centric design, rather than direct measures. The six evaluation metrics (precision, recall, F1-score, accuracy, sensitivity, AUC-ROC) are standard but do not indicate whether a human would judge a class as problematic or worth refactoring. The addition of MCC and Balanced Accuracy (Section 3.2.1) helps validate construct validity for imbalanced classification, but the question of whether metric-based labels align with developer judgement cannot be addressed solely through quantitative measures and would require human validation.

We discuss these topics in relation to industrial-scale software repositories, deployment within software maintenance workflows, and prospects for future industrial validation. Our discussion builds on the maintenance-decision-support layer (Section 2.9) and the computational-complexity analysis (Section 2.13).

Applicability to large-scale repositories: the framework's per-class feature vector and lightweight classifier bank (Table 2c) enable efficient scoring of repositories much larger than the 1,500-class corpus used here, since the main computational effort lies in metric extraction, performed once via static analysis rather than in classifier inference. Before industrial deployment, the key step is recalibration, not redesign. As discussed in Sections 2.1.3 and 3.9.2, the prevalence of code smells and metric distributions in an industrial codebase might differ from those in the open-source training set. Therefore, it is recommended to re-validate and, if needed, re-fit the classifiers using a representative sample of the target codebase before trusting their predictions.

Deployment in software maintenance workflows: two deployment patterns follow directly from the pipeline design. First, a periodic batch-scoring pattern, in which the trained model bank is re-run against the full repository on a scheduled basis (e.g., weekly), and its SHAP-ranked output feeds a technical-debt backlog (Section 2.8.1, Section 2.9.1) for sprint planning. Second, a CI/CD quality-gate pattern, in which the same lightweight classifiers score only the classes touched by an incoming pull request and flag newly introduced smells before merge. This is only practical because of the low training/inference cost profile established in Section 2.13 relative to deep-learning alternatives (Section 3.1.1, Table 4a).

Future industrial validation opportunities: We identify three concrete opportunities for future work, consistent with the requirement to present them as future work rather than claimed results: (i) a partnership with an industrial development team to run the framework as a shadow quality gate alongside existing code review for a fixed period, comparing flagged classes with the team's own refactoring decisions; (ii) a controlled before/after study measuring whether SHAP-ranked backlogs measurably change refactoring prioritisation or defect rates relative to an unranked backlog (Section 2.9.1); and (iii) validation on proprietary, non-Java repositories to test the cross-language extensibility argued for in Sections 2.1.3 and 2.12.

We consolidate dataset descriptions, feature definitions, the preprocessing pipeline, random seeds, software library and version info, the experimental workflow, and source code/dataset availability statement. Instead of dispersing this data, we summarise it here with links to more detailed descriptions in each referenced section.

Dataset: 1,500 Java classes from Qualitas Corpus (release 20130901) and PROMISE (snapshot accessed on 1 November 2014); class distribution and inclusion/exclusion criteria in Section 2.1.1, Table 1a.

Feature definitions: CK-suite object-oriented metrics (WMC, DIT, NOC, CBO, RFC, LCOM), size metrics (LOC, SLOC, NOA, NOM), and complexity indicators (Cyclomatic Complexity, AvgCC, MaxCC, Fan-in, Fan-out), extracted using the CK tool; full definitions and the correlation/Information-Gain selection procedure in Section 2.4.

Preprocessing pipeline: missing-value handling, duplicate removal, SMOTE-based imbalance correction (training folds only), and Min-Max normalisation (fitted on training folds only), in the exact order specified in Section 2.3 and Section 2.11.1.

Random seeds and software versions: global seed 42 with per-model random_state where supported; Python 3.10, scikit-learn 1.3, XGBoost 2.0, imbalanced-learn 0.11, SciPy 1.11, and statsmodels 0.14 (Section 2.5).

Experimental workflow: the nine-phase pipeline in Figure 2, with the project-level split and leakage controls detailed in Sections 2.11 and 2.11.1, and the hyperparameter search protocol in Table 2a and Section 2.5.1. Source code and dataset availability: Qualitas Corpus and the PROMISE repository are publicly available (Section 2.1.1, reference [37]).

5. Conclusion

This study investigated code smell detection using a comprehensive, multi-phase machine learning approach. Sixteen classifiers were tested alongside a clear project-level cross-project validation protocol, SHAP-based explainability, and extensive statistical analysis to identify Data Class and God Class smells. Non-ensemble classifiers generally performed worse than ensemble models. XGBoost achieved the highest accuracy at 99.25% and an AUC-ROC of 98.40%, showcasing its strong capacity for detecting code smells, while Random Forest offered the most balanced results across different metrics. Various statistical tests- such as ANOVA, Kruskal-Wallis, Friedman, and Wilcoxon signed-rank tests- combined with Bonferroni-corrected post-hoc comparisons, confirmed that classifier performance differences are statistically significant, supporting the ranking and selection process. Feature explainability highlighted key object-oriented, size, and complexity metrics influencing predictions, aiding refactoring prioritisation and technical debt management as discussed in Section 3.9. The framework also links detection results to software quality and refactoring decisions, providing practitioners with a statistically validated, interpretable basis for maintenance priorities.

This study focuses on two specific smell types- Data Class and God Class- and relies on tabular object-oriented metrics instead of raw source code. As a result, the findings should be seen as strong evidence of performance for these two smells within the sampled Java projects, rather than a broad claim about detecting code smells generally. Future research will broaden the framework to include additional smells like Long Method and Feature Envy, test cross-project validation on a larger and more diverse set of external projects, and incorporate advanced methods such as deep learning, graph neural networks, and transformer-based code representations. These will be benchmarked against the classifiers used here to enhance generalizability while maintaining the statistical rigour and interpretability established in this study.

References

1. Fowler, M. (1999). Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, USA.
2. Khomh, F., Di Penta, M., & Gueheneuc, Y.-G. (2012). An exploratory study of the impact of code smells on software change-proneness. IEEE Transactions on Software Engineering, 38(2), 432-446.
3. Marinescu, R. (2004). Detection strategies: Metrics-based rules for detecting design flaws. In Proceedings of the IEEE International Conference on Software Maintenance (ICSM), Chicago, USA, pp. 350-359.
4. Lanza, M., & Marinescu, R. (2006). Object-Oriented Metrics in Practice: Using Software Metrics to Characterise, Evaluate, and Improve the Design of Object-Oriented Systems. Springer, Berlin, Germany.
5. Moha, N., Gueheneuc, Y.-G., Duchien, L., & Le Meur, A.-F. (2010). DECOR: A method for the specification and detection of code and design smells. IEEE Transactions on Software Engineering, 36(1), 20-36.
6. Arcelli Fontana, F., Mantyla, M. V., Zanoni, M., & Marino, A. (2016). Comparing and experimenting with machine learning techniques for code smell detection. Empirical Software Engineering, 21(3), 1143-1191.
7. Demsar, J. (2006). Statistical comparisons of classifiers over multiple data sets. Journal of Machine Learning Research, 7, 1-30.
8. Alazba, A., & Aljamaan, H. (2021). Code smell detection using feature selection and stacking ensemble: An empirical investigation. Information and Software Technology, 138, 106648.
9. Sharma, T., Efstathiou, V., Louridas, P., & Spinellis, D. (2021). Code smell detection by deep direct-learning and transfer-learning. Journal of Systems and Software, 176, 110936.
10. Kovacevic, A., Slivka, J., Vidakovic, D., Grujic, K. G., Luburic, N., Prokic, S., & Sladic, G. (2022). Automatic detection of Long Method and God Class code smells through neural source code embeddings. Expert Systems with Applications, 204, 117607.
11. Das, A. K., Yadav, S., & Dhal, S. (2019). Detecting code smells using deep learning. In 2019 IEEE Region 10 Conference (TENCON 2019). IEEE. https://doi.org/10.1109/TENCON.2019.8929197
12. Velioglu, S., & Selcuk, Y. E. (2017). An automated code smell and anti-pattern detection approach. In SERA 2017, June 7-9, 2017, London, UK. IEEE. https://doi.org/10.1109/SERA.2017.7965745
13. Kim, D. K. (2017). Finding bad code smells with neural network models. International Journal of Electrical and Computer Engineering (IJECE), 7(6), 3613-3621. https://doi.org/10.11591/ijece.v7i6.pp3613-3621

14. Borg, M., Svensson, O., Berg, K., & Hansson, D. (2019). SZZ unleashed: An open implementation of the SZZ algorithm, featuring example usage in a study of just-in-time bug prediction for the Jenkins project. ACM.
15. Kotte, A., & Qyser, A. A. M. (2020). A survey of different machine learning models for software defect testing. European Journal of Molecular & Clinical Medicine, 7(9). ISSN 2515-8260.
16. Montana, F. A., Lenarduzzi, V., Roveda, R., & Taibi, D. (2019). Are architectural smells independent from code smells? An empirical study. Journal of Systems and Software, 154, 139-156.
17. Oliveira, R., Sousa, L., De Mello, R., Valentim, N., Lopes, A., Conte, T., ... & Lucena, C. (2017, May). Collaborative identification of code smells: A multi-case study. In 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP) (pp. 33-42). IEEE.
18. Danphitsanuphan, P., & Suwantada, T. (2012, May). Code smell detecting tool and code smell-structure bug relationship. In the 2012 Spring Congress on Engineering and Technology (pp. 1-5). IEEE.
19. Nyirongo, B., Jiang, Y., Jiang, H., & Liu, H. (2024). A survey of deep learning-based software refactoring. arXiv preprint arXiv:2404.19226.
20. Yadav, P. S., Rao, R. S., Mishra, A., & Gupta, M. (2024). Machine learning-based methods for code smell detection: A survey. Applied Sciences, 14(14), 6149.
21. Luiz, F. C., de Oliveira Rodrigues, B. R., & Parreiras, F. S. (2019, May). Machine learning techniques for code smells detection: An empirical experiment on a highly imbalanced setup. In Proceedings of the XV Brazilian Symposium on Information Systems (pp. 1-8).
22. Dewangan, S., Rao, R. S., Mishra, A., & Gupta, M. (2022). Code smell detection using ensemble machine learning algorithms. Applied Sciences, 12(20), 10321.
23. Rao, R. S., Dewangan, S., & Mishra, A. (2025). An empirical evaluation of ensemble models for Python code smell detection. Applied Sciences, 15(13), 7472.
24. Sahu, K. K., Panigrahi, R., Swain, V. K., Padhy, N., Mishra, A. K., Sahu, T., & Mallick, P. K. (2025, February). Code smell predictions using ensemble deep learning approach: A case study on Data Class. In 2025 International Conference on Emerging Systems and Intelligent Computing (ESIC) (pp. 388-393). IEEE.
25. Siahmarzkooh, A. T. (2026). Code smell detection using a hybrid LSTM-CNN deep learning approach with optimised software metrics. National Academy of Science Letters, 1-5.
26. Pandiyavathi, T., & Sivakumar, B. (2025). Ensemble deep network for secured refactoring framework by predicting code-bad smells in software projects. Journal of Software: Evolution and Process, 37(2), e2749.
27. Gupta, H., Kulkarni, T. G., Kumar, L., Neti, L. B. M., & Krishna, A. (2021, April). An empirical study on the predictability of software code smell using deep learning models. In International Conference on Advanced Information Networking and Applications (pp. 120-132). Cham: Springer International Publishing.
28. Ho, A., Bui, A. M., Nguyen, P. T., Di Salle, A., & Le, B. (2025). EnseSmells: Deep ensemble and programming language models for automated code smells detection. Journal of Systems and Software, 224, 112375.
29. Zhang, F., Zhang, Z., Keung, J. W., Tang, X., Yang, Z., Yu, X., & Hu, W. (2024). Data preparation for deep learning-based code smell detection: A systematic literature review. Journal of Systems and Software, 216, 112131.
30. Tantithamthavorn, C. K., & Jiarpakdee, J. (2021, November). Explainable AI for software engineering. In 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 1-2). IEEE.
31. Gezici Gecer, B., & Kolukisa Tarhan, A. (2025). Explainable AI framework for software defect prediction. Journal of Software: Evolution and Process, 37(4), e70018.
32. Menzies, T., Krishna, R., & Pryor, D. (2016). The PROMISE repository of empirical software engineering data. North Carolina State University.
33. Khleel, N. A. A., & Nehéz, K. (2025). Smell-ML: A Machine Learning Framework for Detecting Rarely Studied Code Smells. IEEE Access. DOI:10.1109/ACCESS.2025.3530927.