Code Smell Localization with Ensemble Feature Selection

Pratik Tripathy, Mellacheruvu Ramarakesh, and Dr. Amrita Chaturvedi

Indian Institute of Technology (BHU) Varanasi, Varanasi, U.P., India

Abstract. Code smells are indicators of underlying structural problems that can adversely affect the maintainability and readability of software. Previous methodologies that utilized datasets such as the Qualitas Corpus and deep learning frameworks encountered several obstacles, including data imbalance, synthetic oversampling, and limitations in real-time application.This study aims to address these issues by compiling a balanced dataset collected from open-source Java sources, with code smell samples manually annotated. To ensure data consistency and efficiently handle outliers, we employ median imputation and robust scaling approaches. The Fisher Score is paired with Sequential Forward Selection to create a concise and clear set of features. A Random Forest classifier trained on this curated dataset outperforms baseline models, as indicated by higher AUC and F1 scores, indicating better generalization without risk of overfitting. Furthermore, we have created a Visual Studio Code extension that incorporates our trained model along with the CK metrics tool, facilitating real-time detection and localization of code smells.

Keywords: code smell · software maintainability · ensemble

1 Introduction

Maintaining code quality is critical in the complex systems that many software engineers construct. A code smell is defined as a symptom within the source code that indicates a potentially deeper problem related to design, maintainability, or readability. Finding code smells is one of the most difficult tasks since they are structural faults in code that can impair readability and maintainability even if they do not directly result in errors. Examples are God Class, Feature Envy, Long Methods, and Long Parameter Lists. These smells impede code comprehension, debugging, and maintenance, lowering efficiency and increasing costs. Manual classification is time-consuming, hence machine learning and deep learning are being utilized more frequently for detection.

The contributions of this study are fivefold:

1. **Balanced Dataset Creation:** To address the inadequacies of current datasets such as MLCQ, a fresh dataset of Java code is sourced from GitHub.
2. **Improved Preprocessing:** Because the data is balanced, median imputation and robust scaling are employed to manage outliers more efficiently.

3. **Effective Feature Selection:** To reduce computational overhead, an ensemble approach that combines Fisher Score and Sequential Forward Selection is used.
4. **Improved Model Performance:** A Random Forest classifier trained on the data has better F1 and AUC scores, which suggests better generalization and less overfitting.
5. **Tool Development:** A VS Code extension is created to identify, locate, and show code smells in projects.

2 Literature review

Nandini *et al.* proposed an ensemble feature selection method that addressed dataset dimensionality and imbalance, achieving over 95% accuracy with six classifiers on six code smells [1]. Quanxin Yang *et al.* developed a fusion framework combining AST metrics with semantic embeddings, outperforming traditional techniques on benchmark datasets [2]. Milica ˇSkipina *et al.* compared pretrained code embeddings (CodeT5, CuBERT) to traditional metrics, finding no clear advantage, but highlighting their scalability and adaptability [3]. Amal Alazba *et al.* contrasted pretrained neural code embeddings with conventional code metrics for code smell detection [4]. Yang Zhang *et al.* proposed DeleSmell, integrating semantic and structural features processed through a CNN-GRU-attention model [5]. Dewangan *et al.* presented a machine learning-based technique with PCA for dimensionality reduction, applying four PCA classifiers to code smell detection [6]. Aleksandar Kovačević *et al.* compared traditional heuristics with ML models for detecting God Class and Long Method smells, showcasing transfer learning [7]. Kaur *et al.* optimized code smell prediction using ensemble learning and feature selection strategies on Java datasets, with Random Forest and CFS proving to be the most effective [8]

3 Proposed Methodology

The proposed approach is encompasses the systematic collection of a relevant code corpus, the creation of a high-quality, manually annotated dataset serving as ground truth, the development and evaluation of machine learning models tailored for each code smell, and finally, the practical deployment of these models within Visual Studio Code via a dedicated extension. The six code smells targeted in this study are God Class, Refused Bequest, Shotgun Surgery, Feature Envy, Long Method, and Long Parameter List.

3.1 Data Acquisition

To address the limitations of publicly available code smell detection datasets and to build a less synthetic, more balanced dataset, this method starts with scraping custom Java code corpus from GitHub using GitHub personal tokens/API. Because of their maturity and relevance to daily processes, repositories with more

than 100 ratings are given preference. All .java files from these repositories are gathered recursively by a script and stored locally. The total number of files is regulated for effective processing in order to control scale. The manual annotation step is based on this structured dataset, which guarantees a large and varied collection for training precise code scent detection algorithms.

3.2 Manual Annotation

To provide the ground truth dataset needed to train machine learning algorithms, the human annotation stage is essential. We employed source file analysis to find six predefined code smells in the Java code corpus that we scraped from GitHub. Annotation uses a binary labeling technique, giving a code element a "1" if it has a particular code smell and a "0" otherwise. The annotated data, which comprises the code element identifier, file path, code smell, and label, serves as the ground truth for evaluating automated detection models. The correctness of annotations has a direct effect on trained algorithms' capacity to identify code smells in fresh code. The balanced nature of the dataset is evident by the Table 1.

Table 1. Manual Annotation Results

Code Smell type	**true label (in %)**
Feature Envy	51.38
God Class	57.12
Long Method	51.74
Long Parameter	60.37
Refused Bequest	50.18
Shotgun Surgery	52.14

3.3 Metric Extraction

In this stage, the CK tool (Chidamber and Kemerer metric suite) is used to extract pertinent code metrics from manually annotated Java files. It computes a complete set of object-oriented metrics at the class level, and for each annotated code element (Java classes), metrics are computed and used as feature vectors to describe it's characteristics as given in Table 2. The metrics are subsequently integrated with binary labels derived from manual annotations to form a structured dataset, in which each instance corresponds to a code element defined by its metrics and a label that signifies the presence or absence of a specific code smell. This dataset is then utilized for preprocessing and training the model.

3.4 Data Preprocessing

This phase is crucial for ensuring the reliability and accuracy of the subsequent detection models.

Table 2. Some CK metrics

Metric name	description
WMC	Total complexity of all methods in a class
CBO	Number of other classes a class is coupled with
DIT	Levels of inheritance from the class to the root
NOC	Count of immediate subclasses inheriting the class
LCOM	Measures dissimilarity of methods in a class
RFC	Number of methods executed in response to a message

1. **Handling Missing Values:** Median imutation is used to handle missing values obtained due to poor code structure or parsing errors considering its resilience towards outliers over mean imputation.

$$\hat{x}_{ij} = \begin{cases} x_{ij}, & \text{if } x_{ij} \text{ is observed} \\ \text{median}(\{x_{1j}, x_{2j}, \ldots, x_{nj}\}), & \text{if } x_{ij} \text{ is missing} \end{cases}$$

2. **Feature Scaling:** Given that measurements differ in scale, RobustScaler normalizes the data by subtracting the median and scaling it by the interquartile range (IQR), thus ensuring robustness against outliers.

$$x'_{ij} = \frac{x_{ij} - \text{median}(x_j)}{\text{IQR}(x_j)}$$

$$\text{where IQR}(x_j) = Q_3(x_j) - Q_1(x_j)$$

These procedures are executed with precision before dividing the dataset to provide clean and uniformly scaled input for model training.

3.5 Ensemble Feature Selection

To reduce dimensionality and enhance code smell detection, a two-stage ensemble feature selection strategy is applied:

1. **Initial Filtering with Fisher Score:** Fisher Score ranks features based on their discriminative power by comparing between-class and within-class variance. Features with higher scores are more effective in distinguishing smelly from non-smelly instances. A top percentage is selected to retain the most relevant features. The Fisher Score F_j for the j^{th} feature is defined as:

$$F_j = \frac{\sum_{c=1}^{C} n_c (\mu_j^{(c)} - \mu_j)^2}{\sum_{c=1}^{C} n_c (\sigma_j^{(c)})^2}$$

where:
- C is the total number of classes,
- n_c is the number of samples in class c,
- $\mu_j^{(c)}$ is the mean of feature j for class c,

Table 3. Metrics selected

Code smell type	Final CK metrics selected
Feature Envy	cbo, protectedFieldsQty, finalFieldsQty, assignmentsQty, variablesQty, innerClassesQty, uniqueWordsQty, logStatementsQty
God Class	cbo, wmc, totalMethodsQty, defaultMethodsQty, totalFieldsQty, finalFieldsQty
Long Method	cbo, lcom*, loc, returnQty, loopQty, comparisonsQty, numbersQty, mathOperationsQty, variablesQty, logStatementsQty
Long Parameter	parenthesizedExpsQty, variablesQty, maxNestedBlocksQty
Refused Bequest	dit, lcom, lcom*, tcc, lcc, totalMethodsQty, publicMethodsQty, privateMethodsQty, returnQty, assignmentsQty, modifiers
Shotgun Surgery	cbo, cboModified, fanin, fanout, noc, rfc, lcom*, staticMethodsQty, privateMethodsQty, protectedMethodsQty, totalFieldsQty, privateFieldsQty, protectedFieldsQty, finalFieldsQty, nosi, numbersQty, variablesQty, maxNestedBlocksQty, lambdasQty, uniqueWordsQty, logStatementsQty

- μ_j is the overall mean of feature j across all classes,
- $\sigma_j^{(c)}$ is the standard deviation of feature j for class c.

2. **Refinement with Sequential Forward Selection:** Sequential Forward Selection (SFS) is a greedy wrapper method applied to the Fisher-selected features. It starts with an empty set and iteratively adds features that improve classifier performance (e.g., logistic regression with cross-validation). This helps identify effective combinations of features, enhancing performance and interpretability.

By combining Fisher Score filtering with SFS, the ensemble approach balances computational efficiency with the ability to find highly effective feature combinations for each specific code smell and results in reduced features as given in Table 3.

3.6 Model Training

After ensemble feature selection on the preprocessed dataset, it is split into training and testing sets using a 70-30 stratified split. For each of the six code smells, a Random Forest Classifier is trained due to its effectiveness and robustness in handling non-linear data. Only the chosen features are used to train the model. Following training, the joblib package is used to serialize the model, preprocessing pipeline (median imputation and robust scaling), and chosen features for usage on unseen code in the future without requiring retraining. The pipeline is as Figure 1.

3.7 Development of a VS Code Extension

The final stage of the methodology involves the practical deployment of the trained code smell detection models through the development of a dedicated extension for Visual Studio Code.

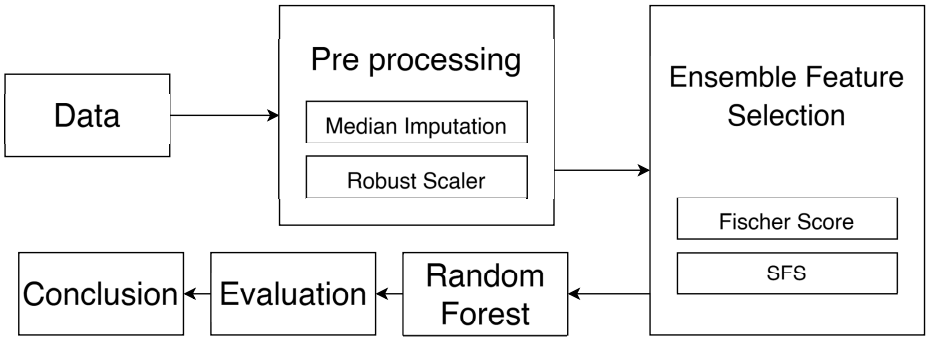

Fig. 1. The Random Forest model training pipeline

1. **IDE Integration and UI:** The VS Code extension adds a custom icon to the activity bar, opening a sidebar panel that serves as the main interface. Users can access core functions like initiating code analysis from this panel.
2. **Code Analysis Workflow:** Users can run code smell analysis on the open project via a button or command. The extension gathers metrics from Java source files, loads trained models for six types of code smells, and preserves the necessary preprocessing activities.
3. **Detection and Visualization:** The models use processed metrics to predict the presence of code smells. The results are presented in an easy-to-read list format, sorted by type of smell and with hyperlinks to the relevant code components.
4. **Localization and Navigation:** By clicking on a detected smell, the user can quickly evaluate and refactor code, jumping right to the affected line or block.
5. **Coverage and UX Features:** The plugin analyzes both classes and interfaces, gives log access for further understanding, and provides visual feedback (such as loading indicators and notifications). It also allows asynchronous processing, which helps the IDE run faster.

4 Evaluation Setup

The Visual Studio Code extension was built on version 1.99.3, and the training and evaluation of the machine learning models were implemented using Python libraries, including scikit-learn, pandas, and numpy, within a standard Python environment, on an Intel Core-i7 12700H processor with Windows 11 operating system. The following metrics are used in the performance comparison along with the parameters given in Table 4:

1. **Accuracy:** Accuracy is defined as the ratio of correctly classified instances, including both true positives and true negatives, to the total number of instances in the test set.

Table 4. Parameters used

Parameter	**value**
n_classes	2246
n_metrics	50
k_fisher	0.6
k_sfs	best
test_fraction	0.3
n_estimators (in RF)	300

2. **F1 score:** The F1 score is the harmonic mean of precision and recall, offering a single, balanced metric that accounts for both false positives and false negatives.
3. **Area Under the ROC Curve (AUC-ROC):** The Area Under the Receiver Operating Characteristic Curve (AUC-ROC) quantifies the model's ability to distinguish between positive and negative classes across various classification thresholds.

5 Results

This section presents the empirical results from evaluating the Random Forest model for detecting six targeted code smells using the annotated dataset. The model was assessed on a held-out test set using standard classification metrics and compared against a baseline, demonstrating strong performance as summarized in Table 5.

1. **Accuracy:** The model achieved high accuracy scores across all code smells, indicating effective classification of code elements as smelly or non-smelly.
2. **F1 score:** Consistently high F1 scores reflect a solid balance between precision and recall, confirming the model's robustness in detecting various code smells.
3. **Area Under the ROC Curve (AUC-ROC):** High AUC-ROC values suggest strong discriminative power, showing the model's ability to distinguish between smelly and non-smelly instances effectively.

6 Conclusion

This research developed a Java code smell detection methodology using a custom GitHub dataset, robust preprocessing, and ensemble feature selection along with VS Code extension. Future work includes integrating PMD for severity assessment, developing LLM-based report generation and sharing features, adding CI/CD integration via GitHub Actions, extending to other languages (Go, Python), and conducting historical code quality analysis.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

Table 5. Performance Comparison of Classifiers

Code Smell type	**Metric**	**performance (Baseline RF / Our RF)**
God Class	Accuracy	**98.6** / 96.13
	AUC-ROC	87.1 / **96.4**
	F1	92.7 / **96.5**
Refused Bequest	Accuracy	96.2 / **98.5**
	AUC-ROC	83.4 / **98.5**
	F1	91.2 / **98.5**
Shotgun Surgery	Accuracy	94.8 / **98.7**
	AUC-ROC	88.9 / **98.6**
	F1	89.3 / **99.6**
Feature Envy	Accuracy	97.6 / **99.9**
	AUC-ROC	91.4 / **99.9**
	F1	88.7 / **99.9**
Long Method	Accuracy	98.0 / **99.9**
	AUC-ROC	91.2 / **99.8**
	F1	92.6 / **99.8**
Long Parameter	Accuracy	**99.2** / 98.9
	AUC-ROC	94.2 / **98.8**
	F1	84.2 / **98.7**

References

1. Abhishilpa Nandini, Randeep Singh: Improving Code Smell Detection by Reducing Dimensionality Using Ensemble Feature Selection and Machine Learning, SN Computer Science
2. Quanxin Yang, Dongjin Yu, Sixuan Wang, Yihang Xu, Xin Chen, Jie Chen, Bin Hu: Enhancing structural knowledge in code smell identification: A fusion learning framework combining AST-based metrics with semantic embeddings, Expert Systems with Applications
3. Milica Škipina, Jelena Slivka, Nikola Luburić, Aleksandar Kovačević: Automatic detection of Feature Envy and Data Class code smells using machine learning, Expert Systems with Applications
4. Amal Alazba, Hamoud Aljamaan, Mohammad Alshayeb: CoRT: Transformer-based code representations with self-supervision by predicting reserved words for code smell detection, Empirical Software Engineering
5. Yang Zhang, Chuyan Ge, Shuai Hong, Ruili Tian, Chunhao Dong, Jingjing Liu: DeleSmell: Code smell detection based on deep learning and latent semantic analysis, Knowledge-Based Systems
6. Seema Dewangan, Rajwant Singh Rao, Pravin Singh Yadav: Dimensionally Reduction based Machine Learning Approaches for Code smells Detection, 2022 International Conference on Intelligent Controller and Computing for Smart Power
7. Aleksandar Kovačevi, Jelena Slivka, Dragan Vidaković, Katarina-Glorija Grujić, Nikola Luburić, Simona Prokić, Goran Sladić: Automatic detection of Long Method and God Class code smells through neural source code embeddings, Expert Systems with Applications
8. Inderpreet Kaur, Arvinder Kaur: A Novel Four-Way Approach Designed With Ensemble Feature Selection for Code Smell Detection, IEEE Access