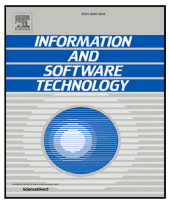
Information and Software Technology 200 (2026) 108302



Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof





Beyond Large Language Models: On the relative value of active learning for code smell detection

Haoxuan Chen [a,b], Chenxuan Hu [a], Chengming Liu [a], Jacky Wai Keung [c], Xiaoxue Ma [d], Wenhua Hu [a,b,*], Chuanxiang Ma [b,e,**]

[a] School of Computer Science and Artificial Intelligence, Wuhan University of Technology, Wuhan, China
[b] Hubei Key Laboratory of Big Data Intelligent Analysis and Application (Hubei University), Wuhan, China
[c] Department of Computer Science, City University of Hong Kong, Hong Kong, China
[d] School of Science and Technology, Hong Kong Metropolitan University, Hong Kong, China
[e] School of Computer Science, Hubei University, Wuhan, Hubei, China

ARTICLE INFO

Keywords:
Code smell detection
Active learning
Large Language Model
Empirical software engineering

ABSTRACT

Context: Machine learning and deep learning approaches for code smell detection typically rely on large-scale labeled datasets. In practice, labeled code smell resources are scarce, while unlabeled code is abundant—making active learning a promising solution to alleviate the data scarcity bottleneck. Prior work in software engineering has demonstrated that active learning can enhance model performance under limited labeling and training budgets, yet its specific effectiveness for code smell detection remains unclear.

Objective: To systematically investigate the impact of active learning (query strategies, classifiers, query rates) on code smell detection and to examine whether the findings are sensitive to the choice of pretrained code representation model.

Method: We conduct a comprehensive empirical study on the MLCQ dataset, which contains 4365 code snippets covering four common code smell types (Data Class, Blob, Long Method, Feature Envy). We first use CodeBERT to extract semantic features of code snippets, then evaluate 5 families of active learning query strategies (instantiated into 11 concrete variants), 8 classifiers, and 10 query rates (ranging from 5% to 50% in 5% increments). We further compare CodeBERT with GraphCodeBERT, CodeT5, and PLBART under the optimal active-learning configuration to assess representation-model robustness. To ensure robust and reliable results, we adopt stratified 10-fold cross-validation, four evaluation metrics (Precision, Recall, F1-score, Matthews Correlation Coefficient (MCC)), and the Scott-Knott Effect Size Difference (ESD) test for statistical significance validation.

Results: (1) Among the 11 query strategies considered, uncertainty-based and committee-based approaches consistently outperform the alternatives, with Vote Entropy showing particularly strong performance; (2) Among the 8 classifiers, Random Forest achieves the best overall results; (3) With respect to the query rate, increasing it from 5% to 30% yields pronounced performance gains, whereas further increasing it from 30% to 50% leads to diminishing improvements and, in some cases, performance degradation; (4) The optimal active learning configuration — combining Random Forest and Vote Entropy — achieves stronger overall performance than the evaluated prompt-based LLM baselines on the MLCQ dataset; (5) In the representation-model robustness analysis, CodeBERT remains the best-performing encoder in terms of F1-score and MCC across all four code smell types.

Conclusion: Active learning effectively enhances code smell detection performance, and practitioners should prioritize pairing CodeBERT feature extraction with Random Forest, alongside the Vote Entropy query strategy, while capping the query rate at 30% to balance detection accuracy and labeling cost. The additional encoder comparison suggests that this conclusion is not narrowly tied to an obsolete representation choice.

* Corresponding author at: School of Computer Science and Artificial Intelligence, Wuhan University of Technology, Wuhan, China.
** Corresponding author at: Hubei Key Laboratory of Big Data Intelligent Analysis and Application (Hubei University), Wuhan, China.
E-mail addresses: haoxuan.chen@whut.edu.cn (H. Chen), chenxuanhu@whut.edu.cn (C. Hu), chengmingliu@whut.edu.cn (C. Liu), jacky.keung@cityu.edu.hk (J.W. Keung), kxma@hkmu.edu.hk (X. Ma), whu1@whut.edu.cn (W. Hu), mxc838@hubu.edu.cn (C. Ma).

https://doi.org/10.1016/j.infsof.2026.108302
Received 7 January 2026; Received in revised form 1 June 2026; Accepted 18 August 2026
Available online 29 August 2026
0950-5849/© 2026 Elsevier B.V. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

1. Introduction

As the scale of software systems continues to expand, software quality has garnered growing attention from both practitioners and researchers in the field of software engineering [1–3]. One critical aspect of maintaining software quality lies in identifying and detecting *code smells* [4–7]. Code smells are structural indicators of design flaws or poor programming practices [8], such as Data Class, Blob, Long Method, and Feature Envy. Although they rarely lead to immediate system failures, code smells progressively degrade the software architecture, compromising overall maintainability and quality. Effective code smell detection, therefore, is an essential strategy for ensuring software quality. By identifying code smells, developers can prioritize refactoring efforts and enhance code maintainability.

Two primary approaches have been proposed for code smell detection: heuristic rule-based methods and machine/deep learning-based techniques. Rule-based methods [9–12] rely on predefined heuristics, where code metrics are compared against manually selected thresholds to detect potential smells. However, the accuracy of this approach is highly dependent on the choice of thresholds, which are often determined in an ad-hoc manner and require subjective expert validation. In contrast, learning-based approaches [13–18] analyze historical data to extract discriminative features and train classifiers that automatically predict smells in new code. By eliminating the need for handcrafted thresholds or rules, these methods have become the prevailing paradigm for code smell detection [19]. Recent research [20–25] has also explored the use of Large Language Models (LLMs) for code smell detection, but preliminary results suggest limited effectiveness (e.g., F1-scores ranging from 0.18 to 0.49 in Blaauwendraad's study [20] using GPT-4o mini with zero-shot) compared to machine/deep learning-based methods.

1.1. Motivation

Despite the substantial promise of machine/deep learning in code smell detection, progress is constrained by a fundamental bottleneck: the scarcity of labeled code smell datasets. Training machine/deep learning models demands large volumes of precisely annotated data, and the absence of such resources directly constrains performance improvements. Moreover, manual labeling is time-consuming and requires substantial expert knowledge. To address data scarcity, researchers in other software engineering domains (e.g., defect prediction [26–28], vulnerability detection [29,30], and requirements engineering [31]) have embraced *active learning*. Specifically, active learning identifies the most information-rich unlabeled samples, engages human experts to label these samples, and iteratively incorporates these high-value samples back into model training. This cyclic process not only reduces reliance on large pre-labeled datasets but also ensures the efficient use of limited annotation budgets—avoiding redundant labeling and maximizing the impact of each labeled example. However, the application of active learning to code smell detection remains largely exploratory, and its effectiveness for this specific task is not yet well understood. Naively transferring strategies that succeed in other domains may not yield comparable benefits for code smell detection and could even degrade performance.

1.2. Our work and contributions

To address this gap, we conduct a large-scale confirmatory and comparative empirical study rather than proposing a new active-learning algorithm. The primary methodological contribution of this study lies in providing a comprehensive empirical evaluation of active learning for code smell detection under a unified and systematically controlled experimental setting, enabling a clearer understanding of how different active learning configurations influence detection effectiveness under limited labeling budgets. Specifically, we systematically examine five families of query strategies — uncertainty sampling, committee-based querying, expected model change, error reduction querying, and density weighted querying — and eleven concrete instantiations: Margin Sampling, Least Confident, Entropy Sampling, Vote Entropy, Batch Mode Active Learning, Expected Error Reduction, Graph Density, Querying Informative and Representative Examples, Self-Paced Active Learning, Ranked Batch Mode Active Learning, and Random Sampling. We carry out this study on the Industry-Relevant Code Smell Data Set (MLCQ) [32], which contains 4365 code snippets covering four common code smell types (Data Class, Blob, Long Method, and Feature Envy) and is widely used in empirical software engineering research [26,33–37]. We first employ CodeBERT to extract semantic features from code snippets in this dataset. Subsequently, we construct code smell detection models by combining these features with eight classifiers, namely Naive Bayes, Logistic Regression, Decision Tree, K-Nearest Neighbors, Multi-Layer Perceptron, Support Vector Machine, Random Forest, and AdaBoost, along with the aforementioned active learning strategies. Moreover, we explore ten query rates, ranging from 5% to 50% in 5% increments, to comprehensively understand the impact of different levels of active data selection on model performance.

RQ1: How do different active learning query strategies affect the performance of code smell detection models?

We evaluate 11 active learning query strategies (including Random Sampling) and a baseline of no active learning ("NONE") across four common code smell types (Data Class, Blob, Long Method, and Feature Envy) using eight classifiers, with the initial labeled data ratio and preset active learning query rate fixed at 5%; we compute the average performance of each query strategy across the eight classifiers for each code smell type and each evaluation metric (Precision, Recall, F1-score, and Matthews Correlation Coefficient (MCC)), and validate the statistical significance of performance differences using the Scott-Knott Effect Size Difference (ESD) test [38]. The results show that across the four code smell types and four evaluation metrics, uncertainty-based query strategies (Margin Sampling, Least Confident, and Entropy Sampling) and the committee-based strategy (Vote Entropy) consistently outperform other strategies, with Vote Entropy exhibiting particularly strong performance. All active learning strategies outperform the "NONE" and Random Sampling baselines in most cases.

RQ2: How does the choice of classifier influence the performance of code smell detection models?

We compare the performance of 8 classifiers across four code smell types and four evaluation metrics, with the initial labeled data ratio and per-iteration query rate fixed at 5%, calculating the average performance of each classifier across the 11 query strategies and validating statistical significance via the Scott-Knott ESD test; the results demonstrate that Random Forest achieves the best overall performance across most code smell types and metrics: it ranks first in F1-score for all four code smell types (0.437 for Data Class, 0.368 for Blob, 0.829 for Long Method, 0.855 for Feature Envy) and leads in MCC for three of the four types (0.425 for Data Class, 0.799 for Long Method, 0.794 for Feature Envy).

RQ3: What is the impact of varying query rates within an active learning framework on the effectiveness of code smell detection?

We set the query rate from 5% to 50% in 5% increments, select the top-performing Vote Entropy and Margin Sampling as the main query strategies (with Random Sampling as the baseline) and Random Forest as the classifier (based on findings from RQ1 and RQ2), and analyze how the performance of code smell detection models changes with increasing query rates across four code smell types and four evaluation metrics. The results indicate that as the query rate increases from 5% to 30%, the Precision, Recall, F1-score, and MCC of models using Vote Entropy and Margin Sampling generally improve, whereas the query rate further increases from 30% to 50%, performance gains become marginal and even decline in some cases.

RQ4: How sensitive is the active learning framework to the choice of pretrained representation model?

We further examine the robustness of the proposed active-learning framework with respect to the choice of pretrained representation model by comparing CodeBERT, GraphCodeBERT, CodeT5, and PLBART under the same optimal Vote Entropy+Random Forest setting with a 30% query rate. The results show that CodeBERT achieves the highest F1-score and MCC across all four code smell types, while alternative encoders remain competitive on selected individual metrics, such as Long Method Recall and Feature Envy Precision. These findings indicate that representation choice affects absolute performance, but CodeBERT provides the strongest and most balanced results under the optimal active-learning configuration.

To underscore active learning's value in code smell detection, we further compare it with a broader set of representative prompt-based LLM baselines, including GPT-family models reported in prior studies and three additional non-GPT models (DeepSeek-V4, GLM-5.1, and Kimi-2.6) evaluated on the same MLCQ dataset. The results show that the optimal active learning configuration (combining Random Forest and Vote Entropy) achieves stronger overall performance across the four smell types, clearly outperforming the evaluated LLM baselines on Blob, Long Method, and Feature Envy, while remaining competitive on Data Class where GPT-4 few-shot obtains the highest F1-score.

This study makes the following contributions.

(1) We provide a large-scale and systematically controlled empirical evaluation for active-learning-based code smell detection that covers five query-strategy families (11 concrete instantiations), eight classifiers, and ten query budgets under a shared CodeBERT representation, enabling controlled and reproducible comparison across the active-learning design space.

(2) We identify the optimal active learning configuration — Vote Entropy as the query strategy, paired with the Random Forest classifier and a 30% query rate — which achieves stronger overall performance than representative prompt-based LLM baselines on the MLCQ dataset under their reported settings, and provide actionable implications for researchers and practitioners.

1.3. Organization

The remainder of this paper is organized as follows. Section 2 reviews the related work. Section 3 presents our application of active learning to code-smell detection, including the model design, query strategies, and stopping criteria. Section 4 describes the experimental setup, and Section 5 reports and analyzes the empirical results. Section 6 first compares the performance of the optimal active learning configuration with LLMs for code smell detection, then discusses the implications of these findings. Section 7 outlines the threats to validity. Finally, Section 8 concludes the paper.

2. Related work

2.1. Heuristic rule-based approaches for code smell detection

Heuristic rule-based approaches to code smell detection typically rely on structural dependencies, software metrics, and expert-crafted rules to flag specific smells. Representative techniques include DECOR [11], which formalized smell characteristics into rule sets for systematic detection; JMove [39], which compared dependency-profile similarity to recommend Move Method refactoring opportunities; and JDeodorant [40], which used Jaccard similarity to detect Feature Envy and, in later versions, recommended refactorings for issues such as Long Methods and Blobs. For dynamic languages, JSNose [41] utilized object-oriented metrics to identify code smells in JavaScript. In addition to individual tools, Fernandes et al. [42] conducted a comprehensive review of the state of practice, highlighting usability issues and offering design guidelines for developers. Collectively, these heuristic methods depend on predefined metrics, rules, and thresholds, which can limit generalizability and lead to false positives, motivating learning-based alternatives.

2.2. Machine/deep learning-based approaches for code smell detection

Machine learning techniques have been extensively applied to the detection of code smells in software systems. Kreimer [43] applied classification and regression trees to find Blob and Long Method smells, demonstrating the promising potential of machine learning-based approaches. Amorim et al. [44] later validated these findings through experiments conducted on four medium-sized open-source systems. Vaucher et al. [45] used a Naive Bayes classifier to examine the evolution of Blob smells over time, whereas Maiga et al. [46] adopted Support Vector Machines to incrementally detect newly introduced Blob instances. Building on this line of work, Vaucher and Khomh [47] explored the use of Bayesian belief networks for handling multiple types of code smells. In a broader comparative study, Fontana et al. [17] evaluated six machine learning classifiers across four categories of smells and found that Random Forest outperformed the other models for most smell types. Nucci et al. [19] replicated and extended this study by merging datasets to enhance the realism and robustness of the evaluation.

In addition, the issue of data imbalance has gained significant attention in code smell detection research. Given the inherent class imbalance in code smell datasets, where instances of code smells are often far fewer than non-smelly code, researchers have explored the potential of imbalance learning methods in improving detection performance. Boutaib et al. [48] proposed ADIODE, an evolutionary search-based engine for detecting eight types of code smells, which incorporates the F-measure as its fitness function to explicitly address the issue of data imbalance. Pecorelli et al. [9] compared machine learning methods (e.g., Naive Bayes) with heuristic rule-based approaches, evaluating five data balancing techniques: oversampling, undersampling, SMOTE, cost-sensitive classifiers, and one-class classifiers. They found that while SMOTE slightly improved performance, other imbalance learning techniques did not significantly enhance detection accuracy. Yedida et al. [49] investigated the combination of fuzzy oversampling and SMOTE, finding it to be effective in addressing the imbalance in code smell detection. Guo et al. [50] introduced STANDER, an oversampling technique that generated synthetic samples based on code entity similarity to address class imbalance in Feature Envy prediction.

Recently, deep learning has gained momentum in code smell detection. Empirical studies show that deep learning models, such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), residual networks, and Transformers often outperform traditional machine learning models. For instance, Sharma et al. [51] trained CNN- and RNN-based detectors for four types of code smells, demonstrating the effectiveness of deep learning and transfer learning for this task. Sharma et al. [51] further explored autoencoder-based variants — AE-Dense, AE-CNN, and AE-LSTM — that compress source-code tokens with an autoencoder and feed the latent representation into a fully connected, one-dimensional convolutional, or LSTM classifier, respectively. Yang et al. [52] proposed a Siamese graph neural network focused on local semantics matching between code graphs, using real-world refactoring data to improve Feature Envy detection. Zhang et al. [53] reviewed data preparation techniques for deep learning-based code smell detection, identifying key challenges in dataset construction and offering recommendations to improve data diversity, standardization, and accessibility for better model performance. More recent efforts further enrich this landscape: Kovačević et al. [54] feed pretrained CuBERT token embeddings into classical machine-learning classifiers to discriminate Long Method and God Class; Ho et al. [55] propose DeepSmells, which routes token-indexing embeddings through a CNN+LSTM cascade to jointly capture local and sequential patterns; and Ho et al. [56] extend this design into EnseSmells, a dual-pathway ensemble that integrates a semantic branch built on pretrained code language models with a structural branch of design metrics.

Although machine learning and deep learning-based approaches have shown significant improvements over heuristic rule-based methods in code smell detection, their performance is still constrained by

the scarcity of labeled datasets. Meanwhile, there exists a large amount of unlabeled data in the field of code smells that could potentially be leveraged for further improvements. Therefore, this paper explores the use of active learning to effectively utilize unlabeled data, aiming to enhance the performance of code smell detection models.

2.3. LLM-based approaches for code smell detection

In recent years, LLMs trained on massive corpora of both natural language text and source code have seen widespread application across software engineering tasks [57–59]. Such progress has also motivated researchers to systematically evaluate and enhance LLM performance for code smell detection tasks [20,21,23,24,60]. For example, Blaauwendraad et al. [20] conducted a comprehensive evaluation of GPT-4o-mini on the MLCQ dataset, using prompt engineering techniques across four types of code smells, covering zero-shot, few-shot, chain-of-thought, and role-driven prompting strategies. The results showed uneven performance across code smells. Specifically, GPT-4o-mini achieves very high Recall on Blob detection, reaching up to 1.00, but its Recall is lower on Long Method and Feature Envy, with maximum values of 0.69 and 0.31, respectively. Precision remains limited across categories, with the highest value reaching 0.44 on Data Class and only 0.17 on Feature Envy. Consequently, its F1-scores remain modest, ranging from 0.18 to 0.55 across the reported settings. In a comparative study, Mesbah et al. [23] compared the performance of GPT-4 and Llama models across four code smell types. Their findings revealed that GPT-4 outperformed Llama in detecting Long Method and Data Class smells under the few-shot setting, with Recall rates of 0.85 and 0.78, respectively. However, GPT-4's performance was significantly weaker for Feature Envy and Blob smells, with Recall rates of just 0.03 and 0.008. On the other hand, Llama showed consistently poor performance across all code smell types, with the highest F1-score only reaching 0.24. Gomes et al. [24] carried out another experiment, using the GPT-4o model to assess the impact of prompt design and model fine-tuning on detection performance. They compared four experimental setups: zero-shot, specialized prompts, specialized prompts combined with evaluation metrics, and model fine-tuning. The results indicated that under the zero-shot condition, both Recall and Precision were generally around 0.5, with F1-scores for some code smells dropping as low as 0.12. However, after optimizing the specialized prompts and fine-tuning the inference parameters, there was a noticeable improvement in the model's detection performance. Overall, existing research demonstrates that while LLMs hold some potential for code smell detection, their overall performance remains relatively modest. There are significant performance biases across different code smell types, and the models show considerable sensitivity to prompt design, leading to fluctuations in the reliability and consistency of detection results.

2.4. Active learning for software engineering

In software engineering, particularly when faced with challenges such as high annotation costs, large datasets, or scarce labeled data, active learning techniques efficiently leverage limited labeling resources. By intelligently selecting and annotating the most informative unlabeled data, active learning not only significantly reduces the annotation workload but also enhances model accuracy and generalization. As a result, it has been widely applied to various software engineering tasks [61]. For example, in vulnerability detection, Yu et al. [29] proposed HARMLESS, which ranked code artifacts for review by their information value and reduced the amount of inspection needed to reach a target Recall. Sun et al. [30] combined active learning with semi-supervised training and used BERT to classify vulnerabilities in smart contracts. In software defect prediction, Li et al. [26] introduced ACo-Forest, an active semi-supervised method that applied disagreement-based sampling to select modules most useful for prediction, and Lu et al. [27] developed TAL, a threshold-based active learning strategy tailored to cross-version defect prediction. Liapis et al. [28] applied active learning methods with ensemble techniques to enhance defect prediction model performance, reduce labeling efforts, and significantly reduce the training set size by up to 75%. Beyond defects, Murukannaiah et al. [62] applied active learning to place recognition in the development of location-aware applications, improving efficiency and accuracy. Despite these advances, the use of active learning for code smell detection remains underexplored and calls for systematic investigation, especially because annotating code smells often requires expert effort.

2.5. Research gap

Overall, the aforementioned existing code smell detection research has mainly focused on heuristic rule-based methods, supervised machine/deep learning approaches, and, more recently, prompt-based LLM techniques. While these approaches have achieved encouraging results, they either rely heavily on handcrafted rules or require large labeled datasets. In addition, existing LLM-based approaches often exhibit unstable performance across different smell types and remain highly sensitive to prompt design and inference settings. Although active learning has been successfully applied to other software engineering tasks such as defect prediction and vulnerability detection, its application to code smell detection remains largely underexplored and lacks systematic empirical evaluation. No prior study has systematically characterized the interaction among query strategies, classifiers, and annotation budgets for code smell detection within a unified experimental setting. To address this gap, our study provides a unified and large-scale investigation of active learning for code smell detection by systematically comparing five query-strategy families, eleven concrete strategies, eight classifiers, ten query rates, and multiple pre-trained code representation models within a controlled experimental framework. Furthermore, by comparing the optimal active learning configuration with recent deep learning approaches and representative LLM-based baselines, our results indicate that active learning constitutes a promising and practical direction for achieving more stable and effective code smell detection while substantially reducing labeling effort.

3. Active learning-based code smell detection

An active learning–driven workflow for code smell detection aims to enhance model performance by strategically selecting high-value unlabeled samples for expert annotation. This process reduces dependence on large labeled datasets and enables more efficient use of limited labeling resources. As illustrated in Fig. 1, the workflow proceeds through the following stages:

1. **Model Initialization:** A base detection model is first trained using a small seed dataset with known code smell labels.
2. **Sample Selection via Query Strategy:** Guided by a chosen active learning query strategy, the system evaluates the unlabeled pool and identifies the most informative code samples for annotation.
3. **Expert Annotation:** Software engineering experts manually review and label the selected code samples.
4. **Model Update:** The newly labeled code samples are incorporated into the training set, and the model is updated through either incremental learning or full retraining to improve its detection capability.
5. **Iterative Refinement:** Steps 2–4 are repeated, continuously selecting, labeling, and integrating new code samples until a stopping condition is satisfied, such as reaching a target performance level or exhausting the query budget.

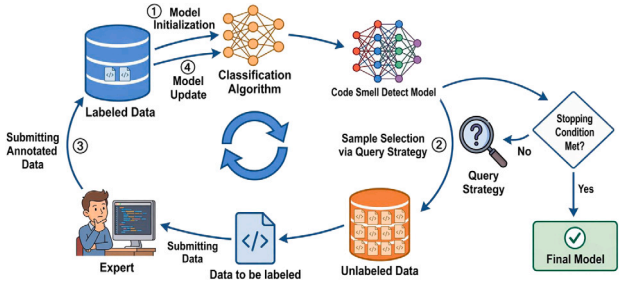


Fig. 1. The overview of the active learning workflow for code smell detection.

Table 1
The categories of classification models for code smell detection.

Category	Classifier
Probabilistic/Statistical models	Naive Bayes, Logistic Regression
Tree-based models	Decision Tree
Instance-based models	K-Nearest Neighbors
Neural models	Multi-Layer Perceptron
Margin-based models	Support Vector Machine
Ensemble models	Random Forest, AdaBoost

3.1. Code smell detection model construction

To effectively detect code smells, we adopt a two-stage architecture in which source code is first mapped into semantic vector representations, followed by downstream classifiers that learn decision boundaries for identifying specific smell types. This design leverages pretrained models to capture rich structural and semantic cues, while allowing traditional classifiers to provide efficient predictions. In this study, we employ CodeBERT [63], a transformer-based model pretrained on both natural language and programming language data. Each code snippet is tokenized and passed through CodeBERT to obtain a 768-dimensional vector embedding, which serves as the feature foundation for building code smell detection models. To ensure a controlled experimental setting, all configurations share the same CodeBERT feature extractor. This design isolates the effects of active learning strategies, classifier choices, and query budgets from those of the underlying code representation, enabling a more rigorous comparison of different active learning configurations. Based on the CodeBERT-generated feature representations, we construct code smell detection models using a variety of machine learning classifiers, as summarized in Table 1:

- **Naive Bayes (NB)** [64] is a probabilistic model built on Bayes' theorem with the simplifying assumption that features are conditionally independent.
- **Logistic Regression (LR)** [65] is a linear classification model that estimates class probabilities through a logistic function.
- **Decision Tree (DT)** [66] constructs a hierarchical set of decision rules by recursively splitting the feature space.
- **K-Nearest Neighbors (KNN)** [67] classifies a sample by examining the labels of its closest neighbors in the feature space.
- **Multi-Layer Perceptron (MLP)** [68] is a feed-forward neural network composed of multiple fully connected layers trained via backpropagation. Its nonlinear activation functions enable it to model complex relationships within the feature space.
- **Support Vector Machine (SVM)** [69] identifies a hyperplane that maximizes the margin between classes. With kernel functions, it can effectively model nonlinear separations in high-dimensional spaces.
- **Random Forest (RF)** [70] is an ensemble of decision trees built from bootstrap samples and random subsets of features. It improves robustness and generalization by aggregating predictions across many diverse trees.

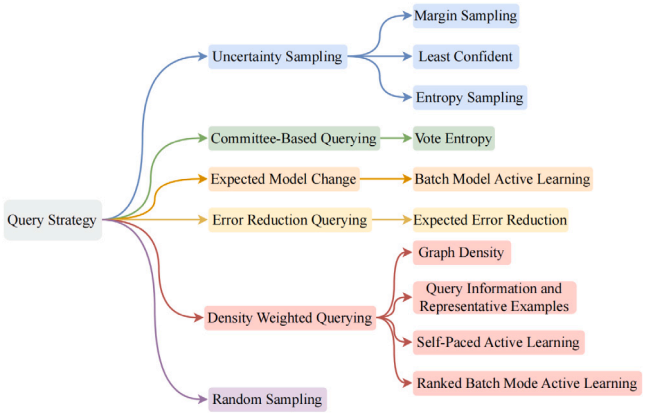


Fig. 2. The eleven query strategies investigated in our study.

- **AdaBoost (ADB)** [71] trains a sequence of weak learners, each focusing on samples misclassified by previous learners. The final model combines these learners into a strong ensemble through weighted voting.

Beyond their widespread use in the software engineering literature [72–74], the selected classifiers were chosen to cover a diverse range of inductive biases, model complexities, interpretability levels, and computational characteristics. This diversity enables us to evaluate active learning across heterogeneous classifier families rather than within a single modeling paradigm. Specifically, the selected classifiers include probabilistic/statistical models (Naive Bayes and Logistic Regression), an instance-based model (K-Nearest Neighbors), a margin-based model (Support Vector Machine), a tree-based model (Decision Tree), ensemble models (Random Forest and AdaBoost), and a shallow neural model (Multi-Layer Perceptron). This selection provides a broad yet controlled evaluation space and is consistent with prior empirical studies of active learning in software engineering [26].

In addition, these classifiers are generally lightweight and efficient to retrain, making them particularly suitable for active learning settings that involve repeated cycles of sample selection, annotation, and model updating. For this reason, and to maintain a controlled experimental setting, we do not include end-to-end deep architectures in the comparison. Such architectures would not only incur substantially higher training costs, but would also simultaneously modify both the representation learning component and the classifier itself, making it difficult to isolate the effects of active learning strategies and classifier choice. By using a shared CodeBERT representation for all classifiers, we ensure a fair and directly comparable evaluation across different classifier families.

3.2. Query strategy

In active learning, query strategies determine which unlabeled code samples to annotate next, aiming to maximize improvements in model performance. Query strategies can be categorized based on their selection mechanisms and the information they exploit, i.e., uncertainty sampling [75], query by committee [76], expected model change [77], expected error reduction [78], density weighted querying [79], and random sampling. To investigate the impact of different strategies on code smell detection, we implement and compare eleven specific methods (including random sampling), as summarized in Fig. 2, with detailed descriptions provided below.

3.2.1. Uncertainty sampling

Uncertainty sampling selects instances for which the model is least certain. We adopt Margin Sampling, Least Confident, and Entropy Sampling as they represent three canonical views of predictive uncertainty — the top-two probability gap, the maximum class probability, and the full predictive distribution, respectively [75,80,81] — and are among the most commonly used uncertainty-based strategies in prior software-engineering active learning studies [34–36]. Being classifier-agnostic and free of method-specific hyper-parameters, they are well suited for fair and reproducible comparison. More elaborate estimators (e.g., Bayesian/MC-dropout posteriors, deep ensembles, learned acquisition functions) are intentionally left out, as they introduce extra modeling assumptions, retraining cost, or hyper-parameters that would reduce comparability under our unified protocol.

- **Margin Sampling (MS)** [80] selects code samples with the smallest difference between the top two predicted class probabilities. A smaller margin indicates higher uncertainty and potentially greater impact on model improvement.
- **Least Confident (LC)** [82] chooses code samples for which the model assigns the lowest confidence to its predicted class. For example, in binary classification, a prediction probability close to 0.5 indicates high uncertainty.
- **Entropy Sampling (ES)** [81] selects code samples with the highest prediction entropy, leveraging an information-theoretic measure to quantify uncertainty. Samples with higher entropy are deemed more informative for labeling.

3.2.2. Committee-based querying

It uses an ensemble of models (committee members) to identify code samples on which the models disagree most. High disagreement indicates that the sample is informative and should be labeled. In our study, we use Vote Entropy as the representative committee-based querying method. We focus on Vote Entropy because it is a widely used and lightweight disagreement-based strategy that can be implemented under the same unified protocol as the other query strategies.

- **Vote Entropy (VE)** [83] measures the degree of disagreement among committee members. In our implementation, the committee is constructed by bagging over different bootstrap samples of the training data, thereby introducing diversity among the base learners and yielding slightly different decision boundaries. Samples with higher disagreement are considered more informative and are prioritized for annotation, as they are more likely to help the model better distinguish ambiguous cases.

3.2.3. Expected model change

This family ranks unlabeled samples by how much their labels are expected to change the model in a useful way. It estimates the size of the update to the parameters or the decision boundary that would follow from acquiring a label, and selects the samples most likely to bring clear performance gains.

- **Batch Mode Active Learning (BMAL)** [84] selects a batch with labels that induce large, beneficial model changes while avoiding redundancy. Within empirical risk minimization, it minimizes a post-labeling risk upper bound (as a surrogate for expected model change), which decomposes into a discriminative term and a representativeness term. The result is an informative, representative batch yielding clear performance gains via substantial, useful updates. Compared with single-instance querying methods, BMAL operates at the batch level and is therefore suitable for fixed per-round query budgets. Its joint consideration of informativeness and representativeness also helps reduce redundant selection among similar code samples.

3.2.4. Error reduction querying

It chooses samples that are anticipated to most reduce the model's expected future error. This approach evaluates unlabeled samples against the model trained on current labeled data to select those that can minimize future prediction risk.

- **Expected Error Reduction (EER)** [85] estimates the current expected risk and the expected post-labeling risk obtained by retraining the model on the candidate paired with each possible label, weighting each outcome by the model's predictive posterior over labels. It then selects the instance that maximizes the estimated risk decrease.

3.2.5. Density weighted querying

Density-weighted querying not only considers the model's uncertainty for each sample but also takes into account the distribution of samples within the feature space. By assigning weights based on the local density of data points, these methods aim to prioritize samples located in sparsely populated regions, which can contribute more significantly to improving the model's generalization performance.

- **Graph Density (GD)** [86] constructs a graph in which nodes represent samples and edge weights reflect the similarity between them. The graph density, defined as the sum of edge weights for each node, serves as a measure of sample representativeness. Samples with the highest graph density are selected, ensuring that the chosen points are both informative and diverse within the sample space.
- **Querying Informative and Representative Examples (QIARE)** [79] uses a min–max optimization approach to select code samples that are both highly informative and representative. This method balances the information content of a sample, which measures its potential to improve the model, with its representativeness, which reflects its coverage within the dataset.
- **Self-Paced Active Learning (SPAL)** [87] optimizes a composite objective function that simultaneously considers sample information density, representativeness, and learnability. SPAL dynamically adjusts the learning pace, ensuring that the model focuses on the most valuable samples at the appropriate time, thereby accelerating convergence.
- **Ranked Batch Mode Active Learning (RBMAL)** [88] selects a batch of samples by ranking unlabeled instances according to their potential value rather than selecting individual samples in each iteration. This ranking mechanism ensures that the selected batch is both informative and representative, improving the efficiency of active learning.

3.2.6. Random sampling

- **Random Sampling (RS)**: As a baseline, code samples are chosen randomly from the unlabeled pool. This approach allows comparison between active learning strategies and traditional passive learning, highlighting the performance gains achievable through informed query selection.

Table 2 summarizes the libraries and key hyper-parameter values used for the eleven query strategies. For methods requiring additional implementation context, BMAL and SPAL use the OSQP quadratic-program solver together with the reported RBF-kernel settings, EER is executed with `p_subsample=1.0` so that no additional candidate subsampling is introduced beyond the library implementation, and the density-based methods differ in how sample similarity is modeled, with GD relying on a Manhattan-distance k-nearest-neighbor graph and QIARE/SPAL using the reported RBF-kernel parameters. We do not perform extensive hyper-parameter tuning for the active learning query strategies for two reasons. First, a small pilot perturbation study on key scalar hyper-parameters across representative strategies shows

Table 2
The implementation configuration of the eleven query strategies (ALiPy 1.2.5/modAL 0.4.2.1).

Strategy	Library	Hyper-parameters (values)
MS	ALiPy	measure='margin'
LC	ALiPy	measure='least_confident'
ES	ALiPy	measure='entropy'
VE	ALiPy	method='query_by_bagging', disagreement='vote_entropy', n_jobs=4, committee size=10
BMAL	ALiPy	beta=1000, gamma=0.1, rho=1, kernel='rbf', gamma_ker=1.0, qp_solver='OSQP'
EER	modAL	loss='binary', p_subsample=1.0, random_tie_break=False
GD	ALiPy	metric='manhattan', k=10, $\gamma = 1/n_{\text{features}}$
QIARE	ALiPy	lambda=1.0, kernel='rbf', gamma=1.0, degree=3, coef0=1.0
SPAL	ALiPy	mu=0.1, gamma=0.1, rho=1, lambda_init=0.1, lambda_pace=0.01, kernel='rbf', gamma_ker=1.0, qp_solver='OSQP'
RBMAL	modAL	n_instances=2, metric='euclidean', n_jobs=None, $\alpha = \lvert U\rvert/(\lvert U\rvert + \lvert D\rvert)$
RS	ALiPy	–

Table 3
The details of our experimental datasets.

Code smell	Description	Number
Data Class	A class that mainly stores data fields and provides little or no behavior, often resulting in excessive reliance from other classes.	451
Blob	A code smell where a single dominant class centralizes most of the system's logic while relying on multiple data-only classes, resulting in an unbalanced and poorly modular design.	256
Long Method	A method that contains an excessively large amount of code, making its logic difficult to understand, maintain, and debug.	119
Feature Envy	A method that interacts with the data of other objects more frequently than its own, indicating misplaced functionality and a violation of encapsulation principles.	291
No Smell	Code snippets that do not exhibit any of the defined code smell types.	3248

no consistent or non-trivial performance changes when varying values around their defaults while keeping other settings fixed. For example, for QIARE, the regularization weight `lambda` is varied around its default value of 1.0 (e.g., 0.5, 0.8, 1.2, and 1.5), without yielding a stable improvement over the default configuration. Second, there is currently no widely accepted or standardized tuning protocol for active learning query strategies, and introducing additional tuning would risk strategy-specific bias and reduce comparability across methods. Accordingly, all strategies are implemented using the official default settings provided by their respective libraries unless otherwise specified.

3.3. Stopping criteria

Active-learning stopping criteria typically fall into two categories: performance thresholds and fixed query rates, with the latter more widely used in practice [89]. Because the final performance of code smell detection models is difficult to predict in advance, we adopt a fixed query-rate stopping rule and study how different rates affect model performance. We consider ten rates from 5% to 50% in 5% increments: the upper bound of 50% is consistent with prior software engineering studies [26], since querying a larger portion of the unlabeled pool would weaken the label-efficiency focus of active learning and approach conventional supervised learning, while the 5% step provides sufficient resolution to observe performance trends at manageable cost. This rule offers a simple and consistent basis for comparing configurations under the same labeling budget; performance-threshold or stability-based criteria may be more appropriate in deployment-oriented settings, and the 5%–50% range here is intended to support a controlled benchmark comparison rather than to represent a universally optimal stopping policy.

Table 4
The confusion matrix.

	Actual smelly	Actual non-smelly
Predicted smelly	TP	FP
Predicted non-smelly	FN	TN

4. Experimental setup

4.1. Datasets

The experiments are conducted on the Industry-Relevant Code Smell Data Set (MLCQ) [32], which is widely used in recent code smell detection research and offers several properties well suited to evaluating active learning. First, it is built from contemporary and actively maintained open-source Java projects, enhancing its practical relevance. Second, its labels come from a committee of 26 industrial developers with domain expertise: each snippet is independently annotated by roughly three reviewers, with disagreements resolved through the cross-checking procedure documented in the original release. This multi-annotator construction improves benchmark credibility while preserving the perceptual subjectivity inherent to code smell identification [90,91]. Third, the dataset spans four representative smells at both class level (Data Class, Blob) and method level (Long Method, Feature Envy), enabling evaluation across structurally different categories while remaining comparable to prior studies [56,73,92]. The characteristics of MLCQ, together with our fixed-budget active learning setting, make redundancy-aware batch selection worth examining. Because samples are queried in batches under predefined annotation budgets, batch-based strategies may otherwise select instances that provide overlapping information. BMAL addresses this issue by jointly considering informativeness and representativeness when constructing a query batch, thereby reducing within-batch redundancy while preserving the potential for substantial model updates. After excluding snippets whose source code could not be retrieved via the provided links, the final dataset contains 4365 snippets.

4.2. Performance measures

In our empirical study, we use three commonly adopted evaluation metrics, namely Precision, Recall, and F1-score, as well as one imbalance-aware correlation-based metric, the Matthews Correlation Coefficient (MCC), to assess the performance of the code smell detection models. In a binary classification setting, all four metrics are computed from a confusion matrix shown in Table 4, which summarizes outcomes in terms of true positives (TP), false positives (FP), false negatives (FN), and true negatives (TN). In the context of code smell detection, TP refers to smelly code samples correctly identified as smelly, FP denotes non-smelly samples incorrectly predicted as smelly, FN represents smelly samples incorrectly classified as non-smelly, and TN corresponds to non-smelly samples correctly recognized as non-smelly.

Precision quantifies the fraction of code samples predicted as smelly that are actually smelly. A higher Precision indicates that the model produces fewer false alarms. In code smell detection, Precision directly reflects the cost of unnecessary refactoring or review effort: each falsely

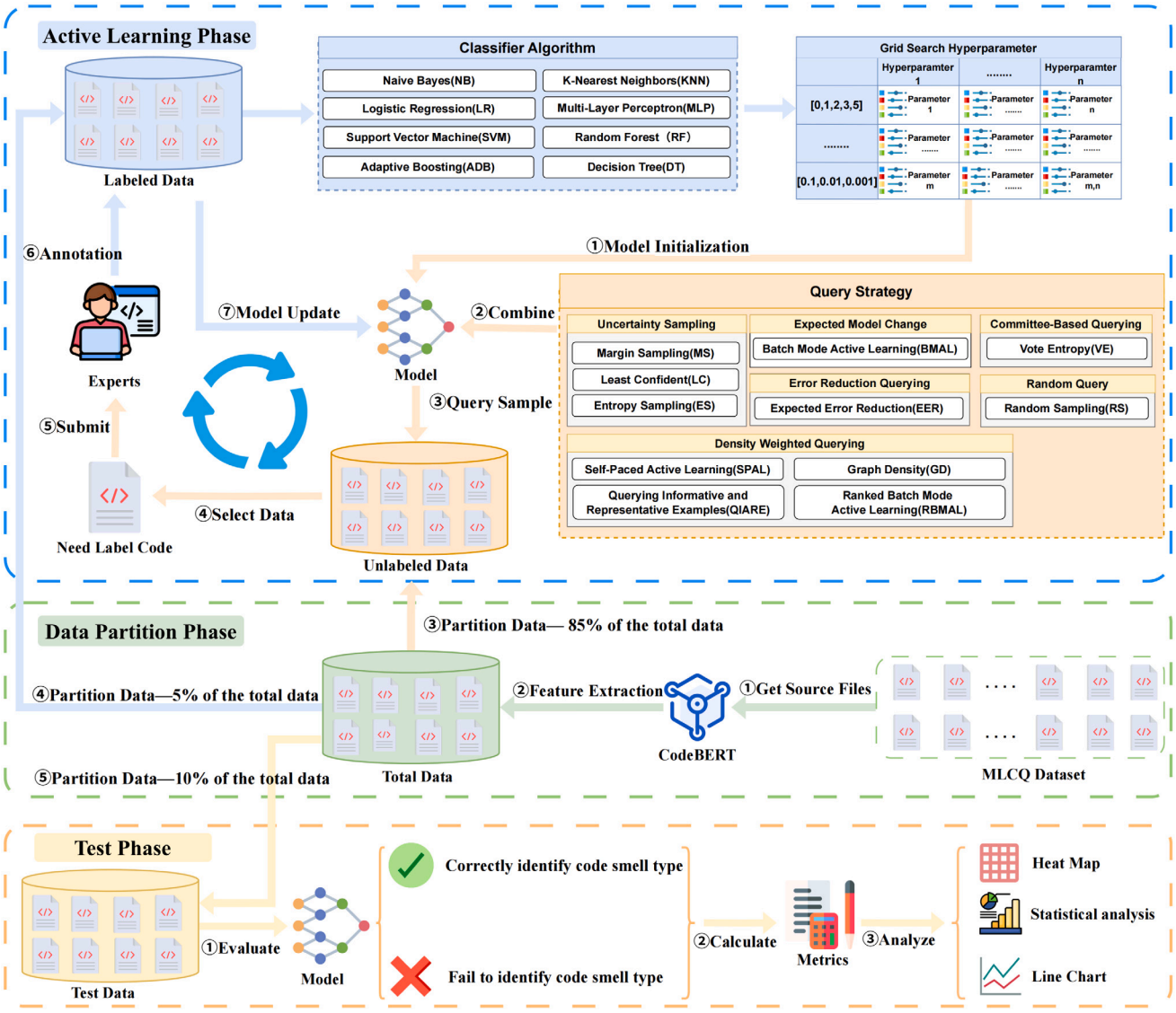


Fig. 3. The overall experimental process of code smell detection based on active learning.

flagged instance may trigger additional inspection, thereby wasting developer time and reducing trust in the detection tool.

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

Recall measures the fraction of actual smelly instances that are correctly detected by the model. Higher Recall reflects the model's ability to identify a larger portion of smelly code in the system. In practice, Recall captures the cost of missed smells: undetected smelly code remains in the codebase and may continue to accumulate technical debt, making Recall especially important in quality assurance and maintainability-oriented scenarios.

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

F1-score combines Precision and Recall into a single metric by calculating their harmonic mean, with values ranging from 0 to 1. A higher F1-score indicates that the model achieves a better balance between identifying smelly instances and avoiding false positives. Since Precision and Recall correspond to two competing risks, namely unnecessary refactoring effort and undetected technical debt, F1-score serves as a balanced summary metric that favors models achieving strong performance in both Precision and Recall simultaneously.

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3}$$

MCC incorporates all four categories of the confusion matrix and is particularly useful for imbalanced datasets. This is highly relevant to MLCQ: as shown in Table 3, non-smelly snippets substantially outnumber each individual smell category, making smelly instances the minority class in every binary detection task. Under such skewed distributions, Accuracy — and in some cases even F1-score — may not fully reflect performance on both classes, whereas MCC yields a high value only when the model performs well on both smelly and non-smelly instances. MCC ranges from −1 (complete disagreement) through 0 (random prediction) to 1 (perfect prediction).

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}} \tag{4}$$

In this study, Precision, Recall, F1-score, and MCC are used because they reflect different but complementary aspects of code smell detection under a fixed decision threshold. Precision and Recall correspond to the two practical risks of false alarms and missed smells, F1-score summarizes the balance between them, and MCC further evaluates overall prediction quality under class imbalance. Alternative metrics,

such as PR-AUC, G-mean, and Balanced Accuracy, may also provide useful perspectives, but the four measures used here are widely adopted in prior code smell detection studies and remain easy to interpret in our setting.

4.3. Experimental procedure

Fig. 3 depicts the experimental workflow, which consists of three phases: dataset partition phase, active learning phase, and test phase.

(1) Dataset partition phase. For fair and reliable model evaluation, we adopt stratified 10-fold cross-validation to split the data. Specifically, the entire dataset is first partitioned into 10 mutually exclusive folds via stratified sampling, where the class distribution is strictly preserved across all folds. In each round of cross-validation, one fold is designated as the stratified test set. From the remaining 9 folds (accounting for 90% of the full dataset), we further select 5% of the entire dataset in a stratified manner to construct the initial labeled subset for model training initialization. The remaining portion (85% of the full dataset, derived from the 9 folds after extracting the initial labeled subset) forms the unlabeled pool for subsequent active learning iterations. To ensure the robustness of experimental results, the entire stratified 10-fold cross-validation procedure is repeated five times independently, resulting in a total of 50 distinct experimental outcomes (10 folds × 5 repetitions). The results presented in the tables and figures are the median values derived from these 50 experiments. We use repeated stratified 10-fold cross-validation to balance estimation stability and experimental cost. Ten-fold cross-validation is a common choice in empirical software engineering, while five repetitions reduce the influence of a particular fold assignment and provide a more stable basis for the Scott-Knott ESD analysis [38].

(2) Active learning phase. Initially, the classifiers described in Section 3.1 are trained on the initial labeled subset, utilizing grid search for hyper-parameter tuning. The specific hyper-parameters and their candidate values are detailed in Table 5. The model achieving the highest F1-score across all configurations is selected as the base learner for the active learning process. Subsequently, the active learning process follows an iterative query strategy. The unlabeled pool consists of the 85% of the dataset (from the 9 non-test folds) defined in the dataset partitioning step. In each round, the model selects the most informative samples from this pool for expert labeling. To simulate real-world labeling, we use the true labels of these samples, pre-annotated in the MLCQ dataset. After each labeling round, the newly labeled samples are added to the labeled subset, and the model is retrained on the updated data. This process (sample selection → expert labeling simulation → model retraining) continues until a preset query rate is reached.

(3) Test phase. Upon completion of the active learning iterations, the final model is evaluated on the stratified test set, which was reserved during the dataset partition phase. To comprehensively assess the model's effectiveness, we calculate four key performance metrics for each experimental run: Precision, Recall, F1-score, and MCC. Furthermore, to rigorously validate the comparative performance and ensure the statistical reliability of our findings, we conduct a statistical significance analysis using the Scott-Knott Effect Size Difference (ESD) test on the collected experimental results.

4.4. Statistical analysis

The Scott-Knott ESD test [38] is a statistical multiple-comparison method that employs hierarchical clustering to group active learning query strategies. It ensures that the performance differences of active learning query strategies between different clusters are statistically significant at the 0.05 significance level, whereas those within the same cluster are not. In this study, we first collect the performance metrics of various active learning query strategies for code smell detection across all datasets. The Scott-Knott ESD test is then applied to these metrics to derive a ranking of the candidate strategies. For instance, Fig. 4 illustrates that Query Strategies 1 and 2 are categorized into the top-ranked cluster, implying that their performance does not differ significantly. By contrast, Query Strategy 8 is assigned to the second-ranked cluster, which indicates that the top-tier strategies exhibit statistically significant outperformance over it.

Table 5
The grid search hyper-parameter configurations for each classifier.

Classifier	Grid search hyper-parameter	Candidate values
NB	C	0.001, 0.01, 0.1, 1
LR	tol	0.1, 0.01, 0.001, 0.0001, 0.00001
DT	min_samples_split	2, 3, 4, 5, 6
KNN	n_neighbors	1, 5, 9, 13, 17
MLP	tol	0.1, 0.01, 0.001, 0.0001, 0.00001
SVM	C	0.001, 0.01, 0.1, 1
RF	n_estimators	10, 20, 30, 40, 50
ADB	n_estimators	50, 100, 150, 200, 250

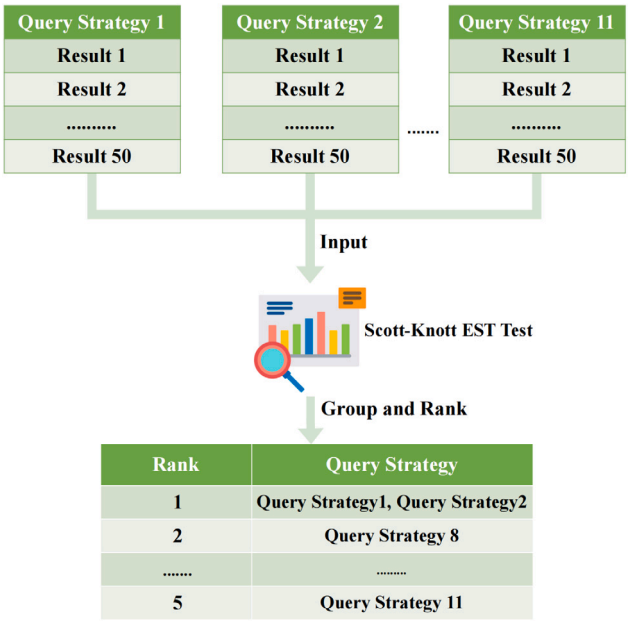


Fig. 4. The example of the Scott-Knott ESD test.

5. Experimental results

5.1. RQ1: How do different active learning query strategies affect the performance of code smell detection models?

Methods: To address RQ1, we evaluate 11 active learning query strategies (including random sampling) alongside a baseline condition of no active learning (denoted as "NONE") across four common code smell types (Data Class, Blob, Long Method, and Feature Envy), using eight classification algorithms as the underlying code smell detection models. For each code smell type and evaluation metric (Precision, Recall, F1-score, and MCC), we compute the average performance score of each query strategy by aggregating the results across the eight classifiers. Consistent with the active learning setup outlined earlier, the initial labeled data ratio and the preset query rate are fixed at 5%. We present the results using heatmaps, where each cell represents the average performance score of a specific query strategy (across the eight classifiers) for a given code smell type and metric, with darker cells indicating better performance. Fig. 5 summarizes these aggregated findings. To further assess the statistical significance of

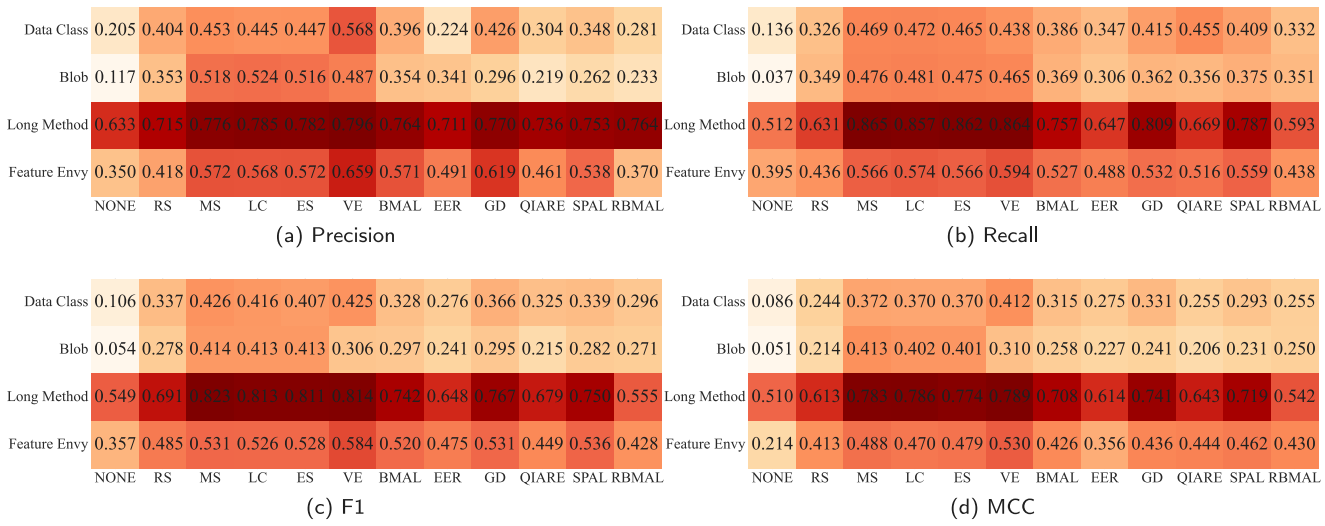


Fig. 5. The average precision, recall, F1-score, and MCC values of the 11 query strategies across code smell types and classifiers.

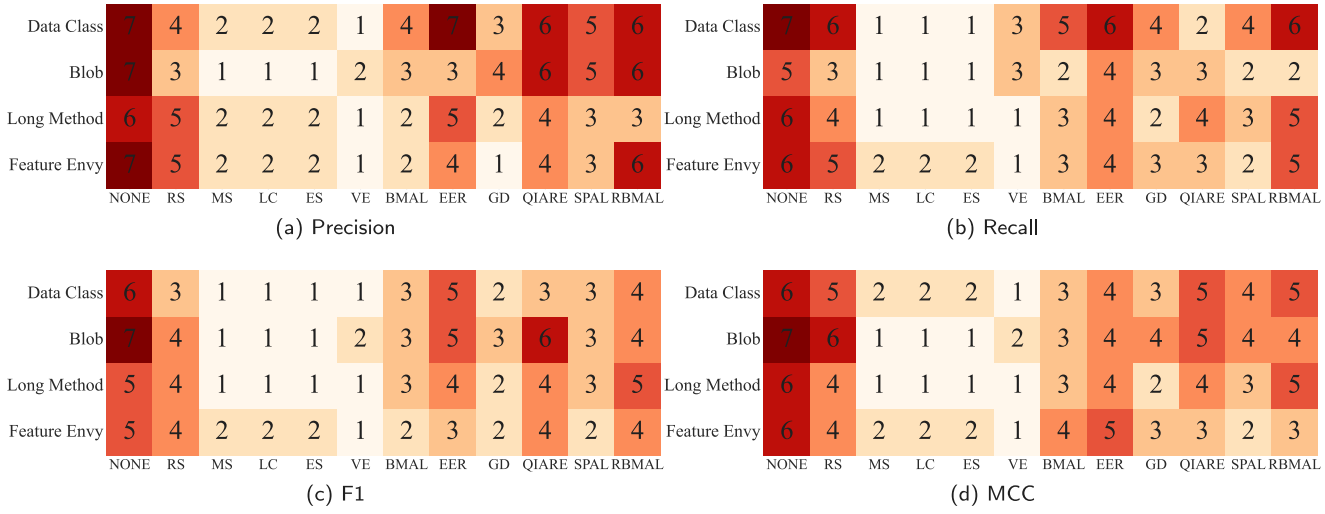


Fig. 6. The Scott-Knott ESD rankings of the 11 query strategies across code smell types and classifiers in terms of precision, recall, F1-score, and MCC.

performance differences among query strategies, we apply the Scott-Knott ESD method. Fig. 6 displays the ranking groups for each metric and code smell type, where lighter cells correspond to higher strategy ranks (i.e., better relative performance).

Results: Across the four code smell categories and 11 query strategies, most strategies outperform two baselines (RS and NONE) across all four metrics (Precision, Recall, F1-score, and MCC). Here, RS refers to randomly selecting 5% of the unlabeled pool for active learning annotation, while NONE denotes training exclusively on the initial 5% labeled subset. Overall, uncertainty-based and committee-based query strategies yield the largest performance gains, with Vote Entropy emerging as the top-performing approach.

(1) **Precision:** As shown in Fig. 5(a), Precision varies across different code smell categories, with four query strategies — Margin Sampling, Least Confident, Entropy Sampling, and Vote Entropy — consistently outperforming others:

- For Data Class, Vote Entropy achieves the highest Precision at 0.568.
- For Blob, Least Confident (0.524) leads, followed closely by Margin Sampling (0.518) and Entropy Sampling (0.516).
- For Long Method, Vote Entropy takes the lead with a Precision of 0.796.
- In Feature Envy, Vote Entropy again performs best with a Precision of 0.659, while Graph Density (0.619) shows competitive performance.

(2) **Recall:** As demonstrated in Fig. 5(b), Recall shows significant variation depending on the query strategy, with some overlap in high-performing approaches across code smell categories:

- For Data Class and Blob, several strategies perform similarly. In Data Class, Margin Sampling (0.469), Least Confident (0.472), Entropy Sampling (0.465), Vote Entropy (0.438), and Querying Informative and Representative Examples (0.455) are notable, with Least Confident achieving the highest Recall (0.472).
- For Blob, Margin Sampling (0.476), Least Confident (0.481), and Entropy Sampling (0.475) show strong results, with Least Confident again leading at 0.481.
- In Long Method, Margin Sampling (0.865), Least Confident (0.857), Entropy Sampling (0.862), Vote Entropy (0.864), and Graph Density (0.809) all exceed a Recall of 0.8, with Margin Sampling achieving the highest at 0.865 and Vote Entropy following closely at 0.864.
- For Feature Envy, Vote Entropy again takes the top spot with a Recall of 0.594.

(3) **F1 and MCC:** As illustrated in Figs. 5(c) and 5(d), F1-score and MCC demonstrate distinct patterns across different code smells:

- For Data Class, Margin Sampling leads in F1-score (0.426), closely followed by Vote Entropy (0.425), Least Confident (0.416), and Entropy Sampling (0.407). However, Vote Entropy excels in MCC with a score of 0.412, with Margin Sampling (0.372), Least Confident (0.370), Entropy Sampling (0.370), and Graph Density (0.331) trailing behind.
- In Blob, Margin Sampling achieves the highest F1-score (0.414) and MCC (0.413), while Vote Entropy performs comparatively poorly in both metrics (F1-score: 0.306, MCC: 0.310), falling behind Least Confident (F1-score: 0.413, MCC: 0.402) and Entropy Sampling (F1-score: 0.413, MCC: 0.401).
- For Long Method, Margin Sampling leads in F1-score (0.823), followed by Vote Entropy (0.814), Least Confident (0.813), and Entropy Sampling (0.811). All four strategies perform strongly in MCC with scores above 0.76 (Vote Entropy: 0.789, Least Confident: 0.786, Margin Sampling: 0.783, Entropy Sampling: 0.774), significantly outperforming Ranked Batch Mode Active Learning (0.542).
- In Feature Envy, Vote Entropy stands out, achieving the highest F1 (0.584) and MCC (0.530), far surpassing Margin Sampling (F1: 0.531, MCC: 0.488), Least Confident (F1: 0.526, MCC: 0.470), and Entropy Sampling (F1: 0.528, MCC: 0.479).

(4) The four evaluation metrics provide complementary perspectives on query-strategy performance. Precision reflects the ability to reduce false positives, whereas F1-score balances Precision and Recall. MCC further evaluates the overall robustness of binary classification by considering all entries of the confusion matrix, making it particularly informative under class imbalance. For example, in the Feature Envy task, Vote Entropy achieves the highest Precision (0.659), while also leading in F1-score (0.584) and MCC (0.530). This indicates that its advantage is not limited to reducing false positives, but also extends to achieving more balanced overall detection performance. A complementary example appears in the Data Class task. Margin Sampling obtains the highest F1-score (0.426), only slightly above Vote Entropy (0.425), but Vote Entropy achieves a substantially higher MCC (0.412 vs. 0.372). This contrast suggests that similar F1-scores may still conceal important differences in prediction balance and classification robustness, which are more clearly revealed by MCC.

(5) Fig. 6 further shows that Vote Entropy generally achieves the strongest overall performance. It belongs to the second Scott-Knott ESD rank group for Precision, F1-score, and MCC in the Blob task and to the third rank group for Blob Recall. For Data Class, Vote Entropy belongs to the first rank group for Precision, F1-score, and MCC, but to the third rank group for Recall. In Long Method and Feature Envy, it remains in the first rank group across all four metrics. Margin Sampling, Least Confident, and Entropy Sampling also belong to the first rank group in Blob detection across all four metrics. In the other three tasks (Data Class, Long Method, and Feature Envy), their Precision and Recall values typically fall into the second rank group, while their F1-score and MCC results remain in the first rank group in selected cases, especially for Data Class and Long Method. One possible explanation for the strong overall performance of Vote Entropy is that, unlike Least Confident, Margin Sampling, and Entropy Sampling, which rely on uncertainty estimates from a single model, Vote Entropy exploits disagreement among multiple committee members. This property may be particularly beneficial for Feature Envy detection, where classification decisions often depend on cross-class interactions and contextual relationships rather than solely on localized structural characteristics. Such cases may exhibit more ambiguous decision boundaries, making committee disagreement a valuable signal for identifying informative samples.

(6) Beyond comparisons among query strategies, the four code smells exhibit notable differences in detection performance. Averaged across the eight classifiers, the best F1-scores on Data Class (Margin Sampling, 0.426) and Blob (Margin Sampling, 0.414) are substantially lower than those on Long Method (Margin Sampling, 0.823) and Feature Envy (Vote Entropy, 0.584), with a comparable gap also observed in MCC. A plausible explanation can be drawn from prior work on the nature of code smells: Data Class and Blob are class-level design flaws whose symptoms emerge from how responsibilities and data are distributed across an entire class, and their detection therefore requires reasoning over the class as a whole rather than over a localized fragment [93]. Empirical studies on developer perception further report that such class-level smells tend to be less consistently recognized even among human reviewers [94]. The lower detection performance observed for Data Class and Blob is consistent with this characterization.

Answer to RQ1

Across four code smell types and four evaluation metrics, uncertainty-based query strategies Margin Sampling, Least Confident, Entropy Sampling and the committee-based strategy Vote Entropy consistently outperform other strategies in most cases.

5.2. RQ2: How does the choice of classifier influence the performance of code smell detection models?

Methods: To address RQ2, we compare eight classifiers across four code smell types using four evaluation metrics. For each code smell type and metric (Precision, Recall, F1-score, and MCC), we calculate the average performance score of each classifier, aggregated across the 11 query strategies. Consistent with RQ1, throughout the active learning experiments, both the initial labeling ratio and per-iteration query rate are fixed at 5%. Fig. 7 presents the results as heatmaps, where each cell denotes the average performance score of a specific classifier (across the 11 query strategies) for a given code smell type and metric—darker cells indicate better performance. To further assess the statistical significance of performance differences among classifiers, we apply the Scott-Knott ESD method. Fig. 8 shows the ranking groups for each metric and code smell type, with lighter cells corresponding to higher classifier ranks (i.e., better relative performance).

Results: Overall, RF leads or ties for the best performance in most smell categories and metrics, and exhibits the most stable behavior. RF shows a consistent advantage in this setting. Below we summarize the main findings by metric; key values are provided for reference.

(1) **Precision:** Fig. 7(a) shows that Random Forest demonstrates strong performance, attaining the top Precision in three of the four code smell types:

- For Data Class, Support Vector Machine achieves the highest Precision (0.583), with Random Forest (0.509) closely following.
- For Blob, Random Forest performs best with a Precision of 0.479.
- For Long Method, Random Forest leads with a Precision of 0.897.
- In Feature Envy, Random Forest (0.928) again leads, while Decision Tree (0.906) and AdaBoost (0.894) deliver competitive Precision.

(2) **Recall:** Fig. 7(b) indicates that Recall performance varies, with Random Forest showing strong results and Naive Bayes also demonstrating high competitiveness in specific categories:

- For Data Class, Naive Bayes achieves the highest Recall (0.773), narrowly outperforming Random Forest (0.762).
- In Blob detection, Random Forest performs best with a Recall of 0.752, followed closely by Naive Bayes (0.742).
- For Long Method, Naive Bayes attains the top Recall of 0.963.
- For Feature Envy, Random Forest yields the highest Recall (0.814), with AdaBoost (0.807) and Decision Tree (0.804) close behind.

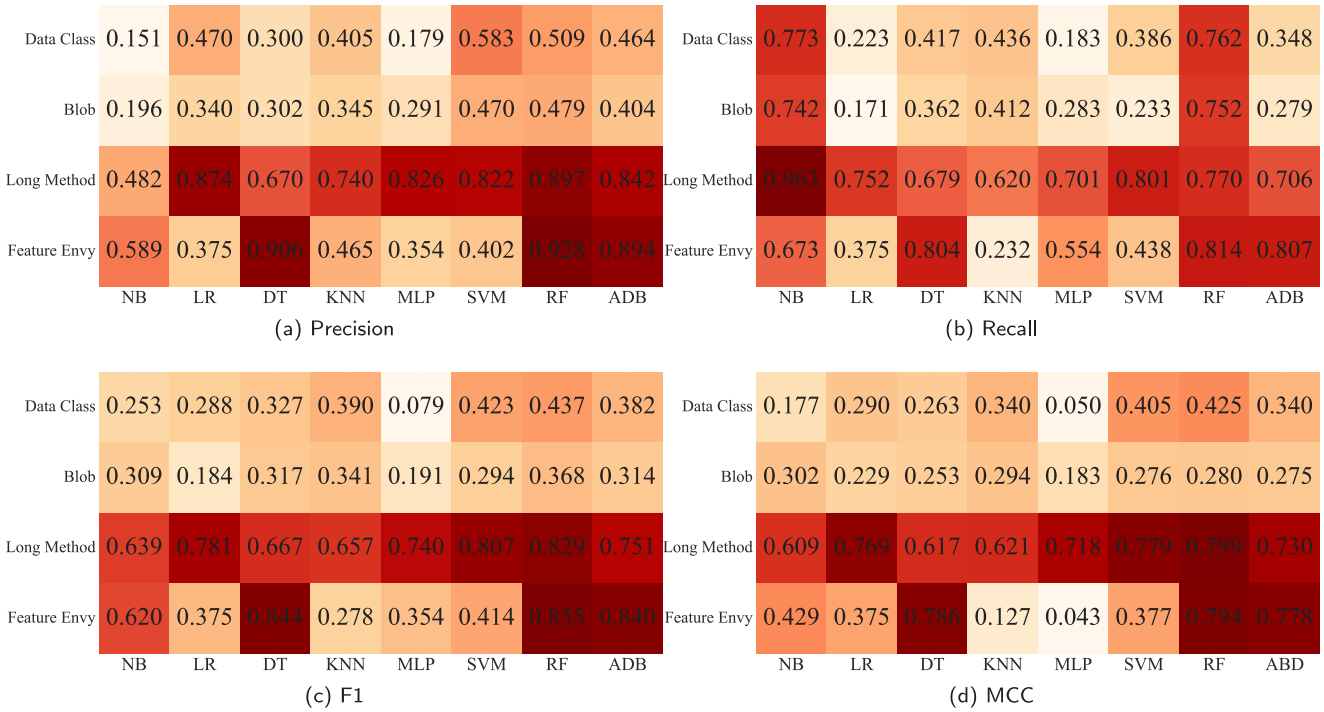


Fig. 7. The average precision, recall, F1-score, and MCC values of the 8 classifiers across code smell types and query strategies.

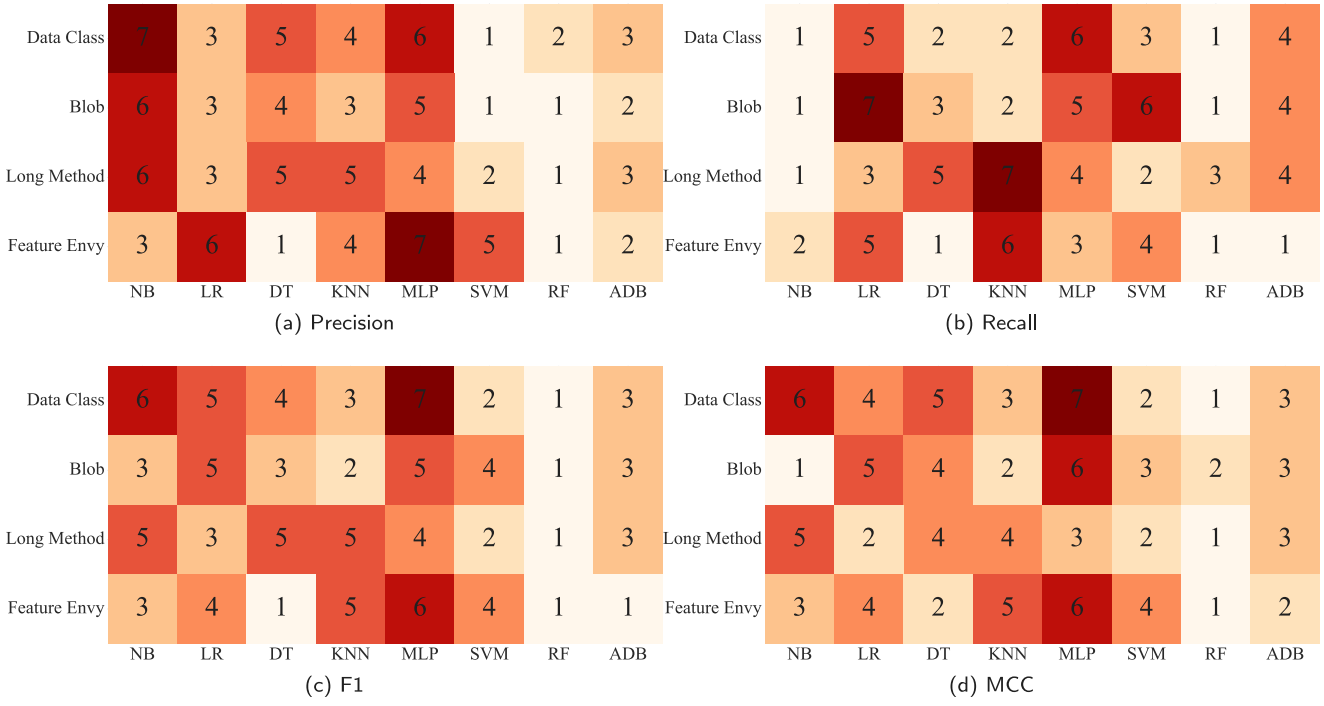


Fig. 8. The Scott-Knott ESD rankings of the 8 classifiers across code smell types and query strategies in terms of precision, recall, F1-score, and MCC.

(3) **F1 and MCC:** Figs. 7(c) and 7(d) further show that Random Forest demonstrates outstanding and consistent performance across both metrics—securing the highest F1-scores in all four code smell categories, and leading in MCC for three out of four:

- For Data Class, Random Forest leads with an F1-score of 0.437 and an MCC score of 0.425.
- For Blob, Random Forest tops the F1-score at 0.368, while Naive Bayes achieves the highest MCC (0.302) followed by K-Nearest Neighbors (0.294), with Random Forest (0.280), Support Vector Machine (0.276), and AdaBoost (0.275) forming a tight second tier.
- For Long Method, Random Forest leads both F1-score (0.829) and MCC (0.799).
- For Feature Envy, Random Forest maintains the top spot in both F1-score (0.855) and MCC (0.794).

(4) The four evaluation metrics provide complementary perspectives on classifier behavior. Precision reflects the ability to reduce false

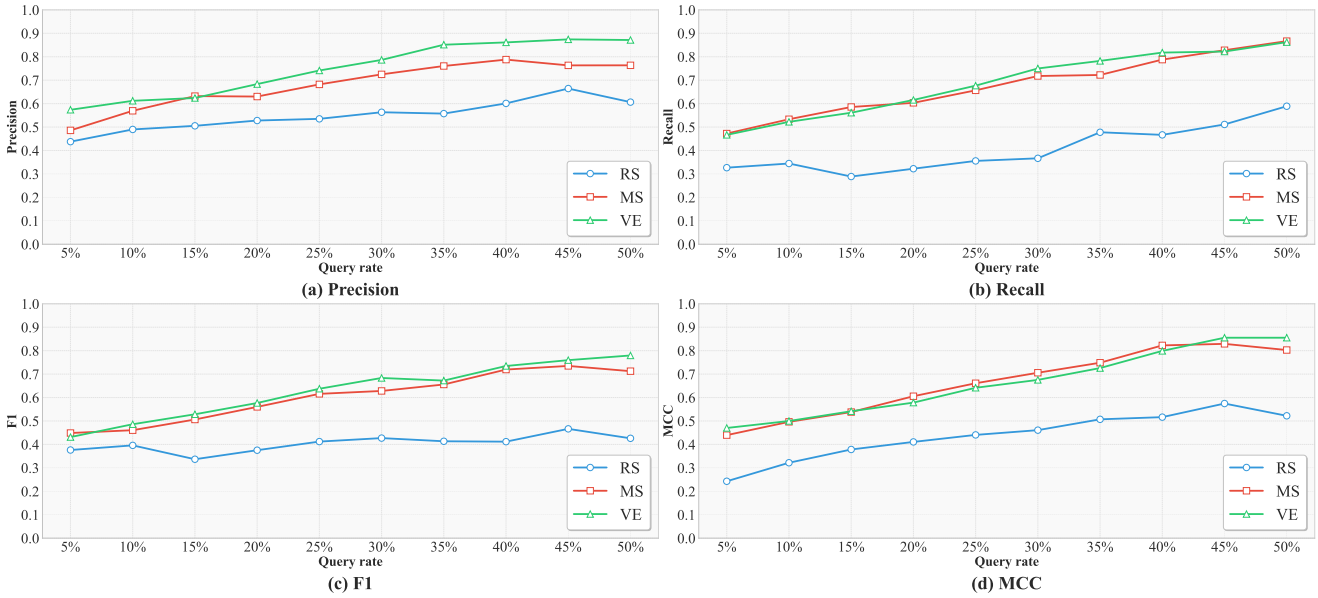


Fig. 9. The performance of code smell detection models with VE, MS, and RS query strategies for Data Class detection (query rate: 5%–50%).

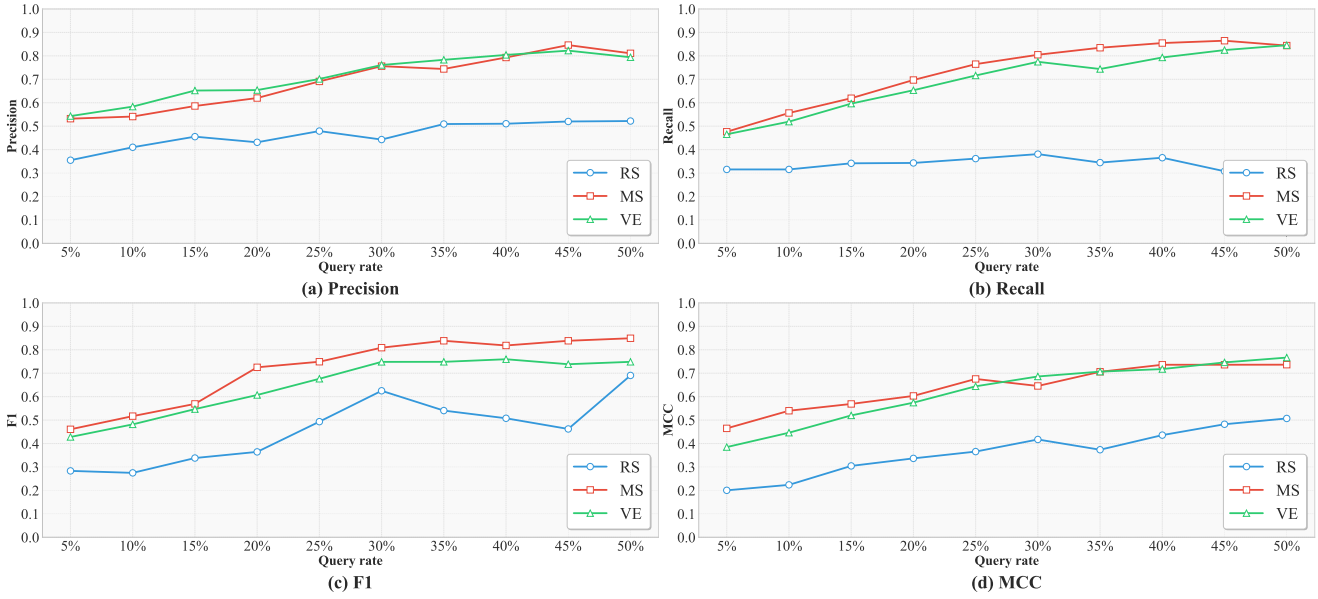


Fig. 10. The performance of code smell detection models with VE, MS, and RS query strategies for Blob detection (query rate: 5%–50%).

positives, whereas Recall emphasizes detecting smelly instances. F1-score balances these two aspects, while MCC further captures overall classification robustness under class imbalance. For example, in the Long Method task, Naive Bayes achieves the highest Recall (0.963), indicating strong capability in detecting smelly instances. However, its relatively lower Precision (0.482) results in substantially lower F1-score (0.639) and MCC (0.609) compared with Random Forest, which achieves a more balanced combination of Precision (0.897) and Recall (0.770), leading to the best F1-score (0.829) and MCC (0.799). This illustrates that high Recall alone does not necessarily translate into strong overall classification quality. Moreover, although F1-score already balances Precision and Recall, MCC further exposes differences in prediction robustness and class balance that may not be fully visible from F1-score alone.

(5) As shown by the Scott-Knott ESD test results in Fig. 8, Random Forest belongs to the second rank group for Precision in Data Class detection, the third rank group for Recall in Long Method detection, and the second rank group for MCC in Blob detection. For all other cases (i.e., across the remaining metrics and code smell types), Random Forest belongs to the first Scott-Knott ESD rank group.

Answer to RQ2

Across four code smell types and four evaluation metrics, Random Forest (RF) as the code smell detection model performs the best in most cases.

5.3. RQ3: What is the impact of varying query rates within an active learning framework on the effectiveness of code smell detection?

Methods: In RQ1 and RQ2, we set both the initial labeling ratio and the query rate to 5%. In RQ3, we aim to investigate the impact of the query rate on the code smell detection model. We set the query rate to values ranging from 5% to 50% in increments of 5%. Based on

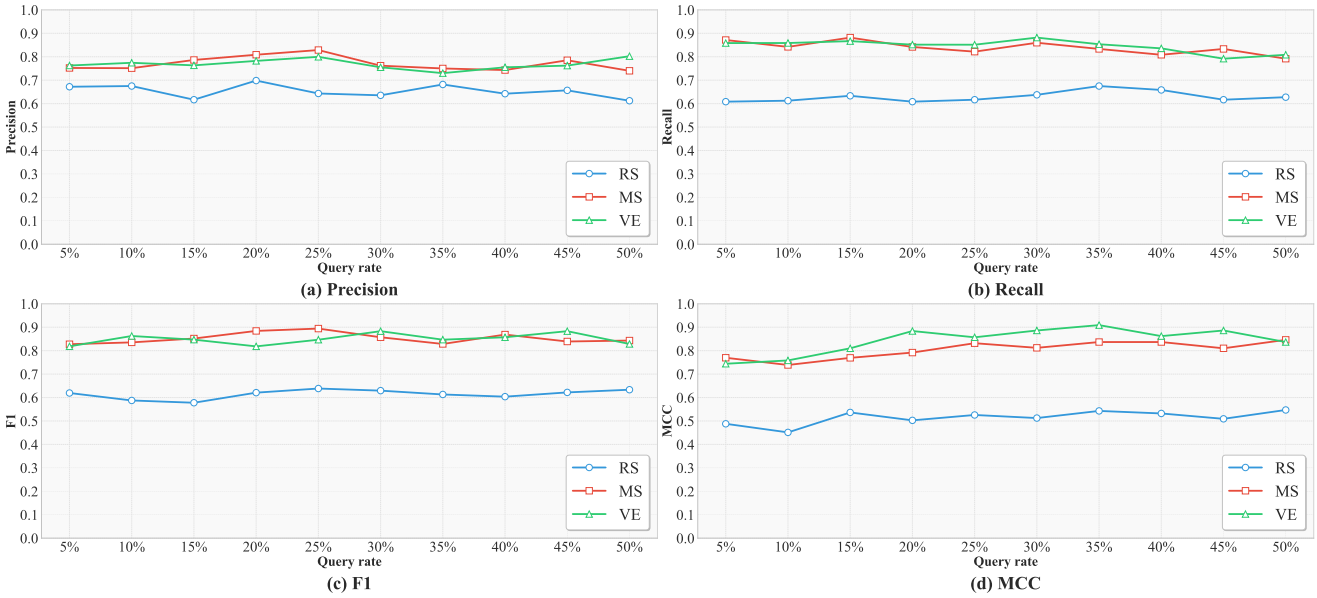


Fig. 11. The performance of code smell detection models with VE, MS, and RS query strategies for Long Method detection (query rate: 5%–50%).

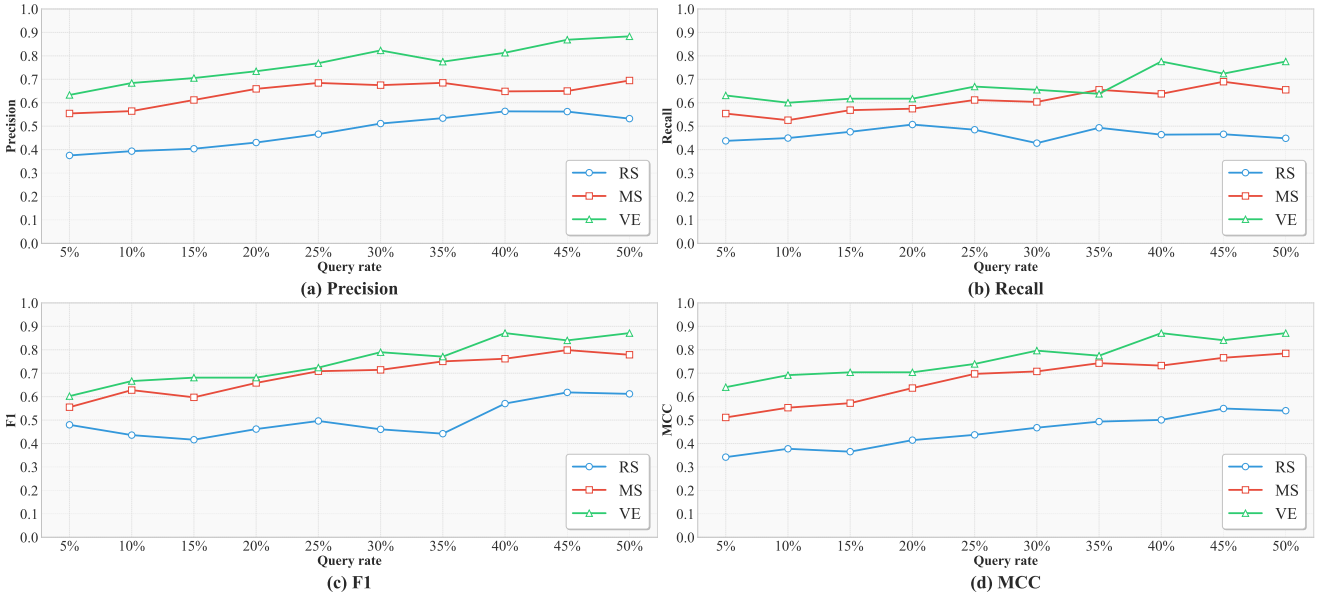


Fig. 12. The performance of code smell detection models with VE, MS, and RS query strategies for Feature Envy detection (query rate: 5%–50%).

the findings from RQ1 and RQ2, we select Vote Entropy and Margin Sampling as the main query strategies because they exhibit superior overall performance, while we select Random Sampling as the baseline method. We also adopt the best-performing Random Forest to construct the code smell detection model.

Results: Figs. 9, 10, 11, and 12 show the Precision, Recall, F1-score, and MCC values of the code smell detection models using Vote Entropy, Margin Sampling, and Random Sampling as query strategies for Data Class, Blob, Long Method, and Feature Envy detection tasks, as the query rate increases from 5% to 50%.

Vote Entropy and Margin Sampling outperform Random Sampling across all four evaluation metrics. However, the performance gap between Vote Entropy/Margin Sampling and Random Sampling narrows as the query rate increases. This is primarily because both active learning strategies (Vote Entropy, Margin Sampling) and the baseline (Random Sampling) accumulate more labeled samples for model training with a higher query rate. At low query rates, Vote Entropy and Margin Sampling can efficiently select high-value samples to drive significant performance gains, but as the training set expands with more labeled data (regardless of the query strategy), the marginal value of active sample selection diminishes, reducing the performance difference between active strategies and Random Sampling.

As the query rate increases from 5% to 30%, the Precision, Recall, F1-score, and MCC values of code smell detection models using Vote Entropy and Margin Sampling as query strategies increase in most cases. However, when the query rate rises from 30% to 50%, the models show only marginal performance improvements with the increasing query rate, and may even decrease occasionally. For example, for Data Class and Blob detection, Vote Entropy and Margin Sampling exhibit only minimal improvements in F1-score and MCC when the query rate increases from 30% to 35%; For Long Method detection, Vote Entropy shows only minimal improvement as the query rate rises from 30% to 35%, while Margin Sampling even experiences a slight decline; For Feature Envy detection, Vote Entropy undergoes a slight decline when

Table 6
The precision, recall, F1-score, and MCC values of different pretrained representation models under the optimal VE+RF setting with the query rate fixed at 30% across four code smell types.

Representation model	Blob				Data Class				Long Method				Feature Envy			
	Pre.	Rec.	F1	MCC	Pre.	Rec.	F1	MCC	Pre.	Rec.	F1	MCC	Pre.	Rec.	F1	MCC
CodeBERT	**0.75**	**0.77**	**0.76**	**0.69**	**0.79**	**0.74**	**0.75**	**0.68**	**0.76**	0.88	**0.88**	**0.89**	0.82	**0.66**	**0.79**	**0.80**
GraphCodeBERT	0.71	0.76	0.73	0.67	0.70	**0.74**	0.72	0.66	0.74	**0.89**	0.81	0.88	**0.83**	0.64	0.73	0.79
CodeT5	0.73	0.76	0.74	0.68	0.72	**0.74**	0.73	0.67	0.75	0.87	0.80	0.87	0.82	0.65	0.72	0.79
PLBART	0.69	0.74	0.72	0.66	0.69	0.72	0.71	0.65	0.73	0.85	0.79	0.85	0.79	0.63	0.70	0.77

the query rate increases from 30% to 35%, whereas Margin Sampling achieves a slight improvement.

Answer to RQ3

When the query rate rises from 5% to 30%, the code smell detection model performance experiences a substantial improvement in most cases. In contrast, as the query rate increases from 30% to 50%, the performance gains are marginal, and occasional declines may occur.

5.4. RQ4. How sensitive is the active learning framework to the choice of pretrained representation model?

Methods: In RQ1–RQ3, we consistently use CodeBERT [63] as the feature extractor for all experiments. This choice is motivated by the fact that CodeBERT is one of the most widely adopted pretrained code models in software engineering and has become a strong and stable baseline for semantic feature extraction in source-code-related tasks [95]. To examine whether our findings depend on this representation choice, and whether other commonly used pretrained code models can provide more effective features than CodeBERT, we extend the analysis in RQ4 to three additional representative pretrained models, namely GraphCodeBERT [96], CodeT5 [97], and PLBART [98]. These models are all widely used in software engineering and code intelligence tasks, while differing in architectural design and pretraining objectives. Specifically, GraphCodeBERT incorporates data-flow information to better capture structural dependencies in code; CodeT5 adopts an encoder–decoder architecture with identifier-aware denoising pretraining; and PLBART is a BART-style sequence-to-sequence model pretrained for program understanding and generation. For a fair comparison, all representation models are evaluated under the same optimal active-learning configuration identified in RQ1–RQ3, namely Vote Entropy as the query strategy and Random Forest (RF) as the classifier, with the query rate fixed at 30%. All other implementation, training, and evaluation procedures are kept the same. We compare their performance using the same four evaluation metrics adopted in the main study, namely Precision, Recall, F1-score, and MCC.

Results: Table 6 reports the encoder comparison results. Overall, CodeBERT achieves the strongest F1-score and MCC performance across all four code smell types under the optimal Vote Entropy+Random Forest setting. Although individual models show slight advantages on specific metrics (e.g., GraphCodeBERT achieves higher Recall for Long Method and higher Precision for Feature Envy), CodeBERT delivers the most consistent overall performance and achieves the best results on all F1-score and MCC metrics. These findings suggest that the active-learning framework remains effective across multiple pretrained representation models, while CodeBERT provides the strongest and most balanced representation in this setting.

Answer to RQ4

The active-learning framework remains effective across pretrained representation models, while CodeBERT delivers the strongest overall performance, achieving the highest F1-score and MCC across all four code smell types under the optimal VE+RF setting.

6. Discussion

6.1. Comparison with non-active-learning code smell detectors

To position our best-performing active-learning configuration relative to prior code smell detectors on MLCQ, we compare Random Forest with Vote Entropy at a 30% query rate against several representative non-active-learning methods. The comparison includes ML_CuBERT [54], which feeds pretrained CuBERT token embeddings into classical machine-learning classifiers to discriminate code smells; three autoencoder-based variants [51], namely AE-Dense, which compresses source-code tokens with an autoencoder and classifies the latent representation via a fully connected network, AE-CNN, which replaces the dense classifier with a one-dimensional convolutional network over the compressed representation, and AE-LSTM, which pairs the autoencoder encoder with an LSTM decoder to capture sequential dependencies; DeepSmells [55], a fusion model that feeds token-indexing embeddings into a CNN+LSTM cascade to jointly capture local convolutional patterns and longer-range sequential context; and EnseSmells [56], a recent extension of DeepSmells that deeply integrates a semantic branch built on pretrained code language models with a structural branch of design metrics through a dual-pathway ensemble. Tables 7 and 8 report the performance results of the methods on MLCQ.

Across the four smell types, Random Forest with Vote Entropy delivers the strongest overall F1 and MCC performance. Compared with EnseSmells, which is the strongest non-active-learning baseline in most cases, the absolute F1 gains are 0.08 on Long Method, 0.39 on Feature Envy, 0.17 on Blob, and 0.11 on Data Class; the corresponding MCC gains are 0.11, 0.41, 0.15, and 0.09, respectively. The comparison also highlights a recurring precision–recall trade-off in the non-active-learning baselines. The autoencoder-based models [51] often achieve high recall at the expense of precision on Data Class and Feature Envy; for example, AE-CNN reaches recall 0.92 but precision 0.15 on Data Class, which keeps its F1 below 0.30. By contrast, Random Forest with Vote Entropy maintains both Precision and Recall above 0.65 on every smell, indicating a more balanced detection profile rather than gains driven by only one side of the trade-off. Overall, Random Forest with Vote Entropy requires labels for only 30% of the training data, as active learning selectively queries the most informative unlabeled instances. Despite this substantially reduced annotation effort, it still outperforms all fully supervised baselines across most smell types and evaluation metrics.

6.2. Comparison between active learning and LLMs

To further validate the effectiveness of active learning, we compare our best-performing configuration, namely Random Forest with the

Table 7
The performance comparison on MLCQ between the configuration (Random Forest + Vote Entropy, 30% query rate) and representative non-active-learning detectors for Long Method and Feature Envy.

Method	Long Method				Feature Envy			
	Precision	Recall	F1	MCC	Precision	Recall	F1	MCC
ML_CuBERT [54]	0.69	0.81	0.75	0.72	0.12	**0.80**	0.21	0.27
AE-Dense [51]	0.74	0.83	0.78	0.75	0.23	0.63	0.34	0.35
AE-CNN [51]	0.74	0.83	0.77	0.75	0.27	0.53	0.35	0.35
AE-LSTM [51]	0.67	0.85	0.75	0.72	0.14	**0.80**	0.23	0.29
DeepSmells [55]	**0.86**	0.75	0.79	0.77	0.59	0.23	0.32	0.33
EnseSmells [56]	0.82	0.79	0.80	0.78	0.52	0.34	0.40	0.39
Active learning	0.76	**0.88**	**0.88**	**0.89**	**0.82**	0.66	**0.79**	**0.80**

Table 8
The performance comparison on MLCQ between the configuration (Random Forest + Vote Entropy, 30% query rate) and representative non-active-learning detectors for Blob and Data Class.

Method	Blob				Data Class			
	Precision	Recall	F1	MCC	Precision	Recall	F1	MCC
ML_CuBERT [54]	0.46	0.58	0.51	0.45	0.50	0.29	0.36	0.31
AE-Dense [51]	0.55	0.64	0.58	0.53	0.16	0.86	0.26	0.11
AE-CNN [51]	0.53	0.66	0.58	0.53	0.15	**0.92**	0.26	0.10
AE-LSTM [51]	0.52	0.65	0.57	0.52	0.15	0.83	0.26	0.09
DeepSmells [55]	0.72	0.46	0.55	0.50	0.69	0.51	0.58	0.52
EnseSmells [56]	0.62	0.59	0.59	0.54	**0.81**	0.53	0.64	0.59
Active learning	**0.75**	**0.77**	**0.76**	**0.69**	0.79	0.74	**0.75**	**0.68**

Vote Entropy strategy at a 30% query rate, against a broader set of LLM baselines on the same MLCQ dataset. As shown in Table 9, the comparison includes GPT-family results reported in prior work, including GPT-4o-mini with zero-shot, few-shot, chain-of-thought (CoT), and role-driven prompting [20], GPT-4 with few-shot prompting [23], and GPT-3.5-Turbo with zero-shot prompting [36]. In addition, we implement and evaluate three representative recent non-GPT models, namely DeepSeek-V4 [99], GLM-5.1 [100], and Kimi-2.6 [101], each under zero-shot, few-shot, and CoT prompting settings. To reduce sampling variance and encourage more stable predictions, all non-GPT models are evaluated with a temperature setting of 0. These models are selected as recent baselines with publicly accessible interfaces and reproducible prompting protocols, and are evaluated on the same stratified test fold and prompting setup as the GPT-family baselines to ensure comparability.

Table 9 shows that the Vote Entropy approach achieves the strongest overall performance among the evaluated methods across the four code smell detection tasks. Specifically, active learning attains F1-scores of 0.76, 0.75, 0.88, and 0.79 for Blob, Data Class, Long Method, and Feature Envy detection, respectively. By comparison, the best F1-scores achieved by the evaluated LLM baselines are 0.52, 0.77, 0.72, and 0.22 for the four tasks, respectively. This indicates that the Vote Entropy active learning clearly outperforms the evaluated LLM baselines on Blob, Long Method, and Feature Envy, while remaining competitive on Data Class, where GPT-4 few-shot achieves the highest F1-score. Overall, these results suggest that the Vote Entropy active learning provides a more effective and stable solution than the evaluated prompt-based LLM baselines for code smell detection on this dataset.

Another notable observation is the substantial variability in LLM performance across different code smell categories and prompting strategies. For example, GPT-4 with few-shot prompting achieves an F1-score of 0.77 on Data Class detection, but its performance drops sharply to 0.02 on Blob and 0.05 on Feature Envy. Similar inconsistencies are also observed among the non-GPT models. For instance, Kimi-2.6 achieves an F1-score of 0.37 on Blob detection under zero-shot prompting, but only 0.08 on Feature Envy. Likewise, GLM-5.1 reaches 0.60 on Data Class detection under CoT prompting, while its F1-scores on Blob, Long Method, and Feature Envy remain substantially lower. In contrast, the Vote Entropy active learning approach maintains consistently strong performance across all four tasks, highlighting the robustness and cross-task stability of traditional machine learning models enhanced with active learning for code smell detection.

It is also worth mentioning that comparison between active learning and LLM-based approaches should be interpreted in the context of their fundamentally different methodological assumptions. Active learning operates within a supervised learning framework, where a task-specific classifier is trained using selectively acquired labeled instances. In contrast, prompt-based LLM approaches perform inference directly with pretrained foundation models, without training on the target dataset. As a result, few-shot and zero-shot prompting are not directly comparable to supervised learning with labeled training data, as they rely on different learning mechanisms. In addition, our evaluation of LLM-based methods is limited to black-box prompting settings, including zero-shot, few-shot, and chain-of-thought (CoT) configurations. Fine-tuned or task-adapted LLMs specifically optimized for code smell detection were not included in this study and may exhibit different performance characteristics. Therefore, our findings should be interpreted as an empirical comparison against the evaluated prompt-based LLM baselines, rather than a comprehensive assessment of all possible LLM-based approaches for code smell detection.

6.3. Practical effectiveness of active learning in code smell detection

To better understand whether the performance gains achieved by active learning are practically meaningful rather than merely statistical, we complement the quantitative evaluation with a qualitative case analysis. Specifically, we compare our best-performing active-learning configuration (Vote Entropy + Random Forest with a 30% query rate) against a non-active-learning baseline that uses random sampling while keeping the classifier and labeling budget unchanged. As shown in RQ1, Vote Entropy consistently outperforms Random Sampling across Precision, Recall, F1-score, and MCC. We therefore examine prediction outcomes that are correctly identified by the Vote Entropy-based model but misclassified by the Random Sampling baseline. The goal is to investigate whether the additional instances detected through active learning correspond to practically meaningful code smells rather than marginal or borderline cases. For each smell type, we manually inspect the source code and present one representative example below.

Blob. Fig. 13 shows `GoIntoAction` from Eclipse Che. The class is only 60 lines long and does not look bloated at first glance. However,

Table 9
The performance comparison between the active learning approach and different LLM baselines.

LLM	Blob			Data Class			Long Method			Feature Envy		
	Pre.	Rec.	F1	Pre.	Rec.	F1	Pre.	Rec.	F1	Pre.	Rec.	F1
GPT-4o-mini [20] (Zero-Shot)	0.26	**1.00**	0.41	0.39	0.68	0.49	0.24	0.54	0.33	0.14	0.25	0.18
GPT-4o-mini [20] (Few-Shot)	0.39	0.81	0.52	0.41	**0.84**	0.55	0.29	0.69	0.41	0.16	0.31	0.21
GPT-4o-mini [20] (CoT)	0.30	0.90	0.45	0.44	0.68	0.53	0.33	0.62	0.43	0.17	0.25	0.20
GPT-4o-mini [20] (Role)	0.25	0.90	0.39	0.37	**0.84**	0.51	0.27	0.62	0.37	0.17	0.31	0.22
GPT-4 [23] (Few-Shot)	0.30	0.01	0.02	0.77	0.78	**0.77**	0.62	0.85	0.72	0.17	0.03	0.05
GPT-3.5-Turbo [36] (Zero-Shot)	0.31	0.14	0.19	0.53	0.67	0.59	0.31	**0.90**	0.46	0.28	0.16	0.21
DeepSeek-V4 [99] (Zero-Shot)	0.37	0.23	0.28	0.58	0.51	0.54	0.18	0.44	0.25	0.10	0.01	0.02
DeepSeek-V4 [99] (Few-Shot)	0.34	0.09	0.14	0.66	0.45	0.54	0.22	0.43	0.29	0.30	0.02	0.04
DeepSeek-V4 [99] (CoT)	0.33	0.19	0.24	0.61	0.48	0.53	0.19	0.41	0.26	0.22	0.02	0.04
GLM-5.1 [100] (Zero-Shot)	0.45	0.20	0.28	0.45	0.73	0.56	0.16	0.64	0.26	0.13	0.11	0.12
GLM-5.1 [100] (Few-Shot)	0.50	0.16	0.24	0.52	0.67	0.59	0.17	0.58	0.26	0.16	0.20	0.18
GLM-5.1 [100] (CoT)	0.41	0.16	0.23	0.56	0.66	0.60	0.14	0.65	0.23	0.14	0.14	0.14
Kimi-2.6 [101] (Zero-Shot)	0.36	0.38	0.37	0.56	0.60	0.58	0.13	**0.90**	0.23	0.12	0.06	0.08
Kimi-2.6 [101] (Few-Shot)	0.34	0.33	0.34	0.56	0.53	0.54	0.13	0.84	0.23	0.10	0.06	0.08
Kimi-2.6 [101] (CoT)	0.30	0.34	0.32	0.61	0.51	0.55	0.12	0.55	0.20	0.07	0.02	0.03
Active learning (VE)	**0.75**	0.77	**0.76**	**0.79**	0.74	0.75	**0.76**	0.88	**0.88**	**0.82**	**0.66**	**0.79**

its methods collectively handle menu-text state, selection parsing, capability checks, navigation, and node inspection. These responsibilities are distributed across several short methods rather than concentrated in one obviously large class.

Data Class. Fig. 14 shows `CassandraIO.Write.Builder` from Apache Beam. The class is annotated with `@AutoValue.Builder` and looks like a routine framework-generated builder. However, all of its members are abstract setters, and no validation or construction logic is present, so the class mainly serves as a container for fields. The smell is therefore masked by a common framework convention.

Long Method. Fig. 15 shows `getVMPassword` from Apache CloudStack. The method is only 25 lines long and does not appear lengthy on the surface. However, within those lines it chains caller resolution, entity lookup, exception construction, authorization, detail loading, and null handling. Multiple concerns are compressed into a short linear flow rather than spread across an obviously long body.

Feature Envy. Fig. 16 shows the factory method `TimeOfYear.fromDateTime` from Google Registry. The method returns an instance of its declaring class, which makes it look like a normal intra-class factory at first glance. However, almost every value used to construct the result is read from the `DateTime` argument. The dependence on the parameter is thus easy to overlook from the method signature alone.

These examples illustrate that many code smell instances are difficult to detect because their problematic characteristics are not immediately visible from superficial structural properties such as class size, method length, or naming conventions. Instead, identifying them often requires recognizing more subtle relationships among responsibilities, dependencies, and contextual interactions within the code.

```java
public class GoIntoAction extends AbstractPerspectiveAction {
    // (1) menu-text state
    public void update(ActionEvent e) { ... }

    // (2) selection parsing + (3) capability check
    public void updateInPerspective(ActionEvent e) { ... }

    // (4) navigation execution
    public void actionPerformed(ActionEvent e) { ... }

    // (5) node-capability inspection
    private boolean canGoInto(ResourceNode node) { ... }
}
```

Fig. 13. The Blob code snippet example (Eclipse Che, `GoIntoAction.java`, L29–88; signatures only).

The qualitative inspection further suggests that the additional instances correctly identified by the Vote Entropy-based model are not merely borderline prediction changes, but correspond to code smells with practically meaningful design and maintainability concerns. In contrast, the Random Sampling baseline tends to miss smell instances whose problematic characteristics are less obvious from surface-level code patterns. Consequently, the observed performance improvements are not merely statistical but are associated with the successful identification of code smell instances that could otherwise remain unnoticed in real-world software projects. Missing such instances may allow maintainability issues, excessive coupling, or responsibility diffusion to persist in the codebase, increasing future maintenance effort and reducing code quality.

```java
@AutoValue.Builder
abstract static class Builder<T> {
    abstract Builder<T> withHosts(List<String> hosts);
    abstract Builder<T> withPort(int port);
    abstract Builder<T> withKeyspace(String keyspace);
    abstract Builder<T> withUsername(String username);
    abstract Builder<T> withPassword(String password);
    abstract Builder<T> withLocalDc(String localDc);
    abstract Builder<T> withConsistencyLevel(String cl);
    abstract Builder<T> withEntity(Class<T> entity);
    abstract Builder<T> withMutationType(MutationType t);
    abstract Write<T> build();
}
```

Fig. 14. The data class code snippet example (Apache Beam, `CassandraIO.java`, L936–957).

```java
@Override
public String getVMPassword(GetVMPasswordCmd cmd) {
    Account caller = CallContext.current().getCallingAccount();
    UserVmVO vm = _userVmDao.findById(cmd.getId());
    if (vm == null) {
        InvalidParameterValueException ex =
            new InvalidParameterValueException("No VM with id");
        ex.addProxyObject(cmd.getId().toString(), "vmId");
        throw ex;
    }
    _accountMgr.checkAccess(caller, null, true, vm);
    _userVmDao.loadDetails(vm);
    String password = vm.getDetail("Encrypted.Password");
    if (password == null || password.equals("")) {
        InvalidParameterValueException ex =
            new InvalidParameterValueException("No password for VM");
        ex.addProxyObject(vm.getUuid(), "vmId");
        throw ex;
    }
    return password;
}
```

Fig. 15. The long method code snippet example (Apache CloudStack, `ManagementServerImpl.java`, L3807–3831).

```java
public static TimeOfYear fromDateTime(DateTime dateTime) {
    DateTime nextYear = dateTime.plusYears(1);
    return create(String.format(
        TIME_OF_YEAR_FORMAT,
        dateTime.getMonthOfYear(),
        dateTime.getDayOfMonth(),
        nextYear.getMillisOfDay() / 1000));
}
```

Fig. 16. The feature envy code snippet example (Google Registry, `TimeOfYear.java`, L61–70).

6.4. Implication

In this subsection, we analyze implications based on our experimental results and provide a few suggestions to practitioners and researchers in this domain.

(1) For practitioners building active-learning-based code smell detection systems, the most effective configuration combines CodeBERT for code feature extraction with a Random Forest (RF) classifier. RF consistently outperforms alternative classifiers such as Naive Bayes and K-Nearest Neighbors across four code smell types (Data Class, Blob, Long Method, Feature Envy) and four key evaluation metrics (Precision, Recall, F1-score, and MCC), making it a robust and reliable core model. Once the base model is established, practitioners should adopt high-yield active learning strategies, especially uncertainty-based sampling (e.g., Margin Sampling) and committee-based querying (e.g., Vote Entropy). These strategies prioritize samples where the model is most uncertain or where multiple models disagree, enabling annotation efforts to focus on the most informative instances and thereby maximizing performance gains while avoiding redundancy. Among these strategies, Vote Entropy is particularly effective for semantically complex smells such as Feature Envy, making it the preferred choice in scenarios where identifying subtle behavioral dependencies is essential. This recommendation is also consistent with broader practitioner-oriented guidance on code smell identification [102], which emphasizes that effective smell detection in practice depends on both reliable detection mechanisms and a clear understanding of the contextual cues that distinguish each smell type.

(2) To balance detection performance and annotation cost, practitioners should limit the total labeling budget to no more than 30% of the unlabeled pool. Experimental results indicate that model performance improves as the query rate increases from 5% to 30%, but further increases in the query rate lead to diminishing or even negative returns. A labeling budget of 5%–30% (validated on the MLCQ dataset) ensures "low-cost, high-performance" results for all four code smell types. For instance, annotating 300 out of 1000 unlabeled code snippets is sufficient to bring the model close to optimal performance, avoiding the inefficiencies of annotating the entire dataset. This low annotation threshold also keeps the approach competitive with few-shot LLMs, as both require minimal labeled data but the active learning setup delivers better overall performance.

(3) For future research, developing adaptive stopping rules based on real-time model performance is crucial. Current fixed query rate rules (e.g., a 50% upper limit) fail to account for early model convergence, where optimal performance may be achieved at a much lower query rate (e.g., 30%). Adaptive rules, such as halting annotation when the validation set's F1-score does not improve for two consecutive iterations, can reduce unnecessary resource consumption and prevent overfitting. This approach addresses the limitations of traditional fixed-ratio stopping criteria, leading to more efficient and effective active learning.

7. Threats of validity

Internal validity relates to the methods and technologies used in our experiments, specifically classification models and active learning query strategies. (1) We employ 5 families of active learning query strategies and 6 families of classifiers, which ensures the diversity and representativeness of research objects and helps enhance the generalizability of experimental results. These strategies are widely used in prior studies [26,33–36]. To mitigate technical errors in our experiments, we implement active learning strategies using the ALiPy [103] and modAL [104] libraries, while classifiers are sourced from Scikit-learn [105]. (2) To ensure fair comparison across strategies, we implement all active-learning methods via the ALiPy [103] and modAL [104] libraries on top of scikit-learn [105] base learners, and adopt each library's official default hyper-parameters (Table 2). While this avoids per-strategy tuning bias, the absence of a consensus tuning protocol for active-learning query strategies remains a limitation. (3) An additional internal-validity concern relates to how labels are consumed in the active learning loop. Following standard practice, each queried sample is assigned the ground-truth label released with MLCQ as the oracle response. This provides a controlled basis for comparing query strategies, but it also means that the learner does not observe annotator-level disagreement during the querying process. Prior work has shown that code smell identification is inherently influenced by developer perception, expertise, and contextual familiarity [90,91]. Consequently, our results evaluate the effectiveness of query strategies under a fixed benchmark labeling scheme rather than under dynamically evolving human labeling behavior. As a result, the reported findings do not capture the potential impact of annotator-level disagreement during the labeling process.

External validity in our work mainly concerns the datasets we study. We conduct experiments on the MLCQ dataset, which focuses on a specific set of software projects and is widely used in code smell detection research [20,23]. The dataset covers four common code smells (Data Class, Blob, Long Method, and Feature Envy) but excludes other relevant smell types such as Duplicate Code, Switch Statement, and Lazy Class, and contains only Java code. Whether the optimal query strategies and classifiers identified in this study generalize to other programming languages or broader smell taxonomies therefore remains uncertain.

Construct validity depends on the evaluation metrics we use to measure the performance of code smell detection models. We adopt three commonly used evaluation metrics (Precision, Recall, and F1-score) and one imbalance-aware correlation-based metric (MCC) to assess model performance. Our evaluation relies on the ground-truth labels provided by MLCQ, which are produced through a multi-annotator process involving industrial developers. Although we manually inspect representative prediction cases, we do not conduct an independent expert validation of the final model outputs. Consequently, the practical relevance of the detected smell instances is assessed through qualitative analysis rather than through a separate human-in-the-loop evaluation procedure. In addition, alternative metrics such as PR-AUC, G-mean, and Balanced Accuracy are not included in the current evaluation, and these may provide further perspectives on model performance under class imbalance.

Conclusion validity refers to the statistical analysis method used in our work. We apply the Scott-Knott ESD test to analyze the significant differences among active learning query strategies, which helps enhance the rigor of our experiments. Notably, this test is widely used in the field of software engineering [38,106–109].

8. Conclusion and future works

Our study presents a comprehensive comparative investigation of active learning for code smell detection. Using the MLCQ dataset, we conduct an extensive analysis that covers five families of query strategies with eleven concrete algorithms, eight classifiers, and ten query rates, in order to examine their specific impacts on model performance. The main findings are as follows: (1) Given the same amount of labeled training data, active-learning-based code smell detectors achieve better overall performance than traditional supervised learning without active sampling, showing clear and consistent gains under benchmark-based evaluation. (2) Among the five families of query strategies, uncertainty sampling and committee-based methods perform best on code smell detection. In particular, Vote Entropy delivers the strongest results in most cases. (3) The Random Forest classifier attains the best overall performance across four standard metrics when paired with the eleven query strategies. (4) When using the query rate as the stopping rule, model performance improves markedly as the query increases from 5% to 30%, yielding strong results at relatively low cost. As the query rate grows from 30% to 50%, the gains become small and may even decline. (5) The optimal active learning setup (Random Forest + Vote Entropy + 30% query rate) achieves stronger benchmark-based performance than representative prompt-based LLM baselines on the MLCQ dataset under their reported settings, offering more balanced and consistent results within this evaluation setting. (6) The representation-model robustness analysis shows that CodeBERT remains the strongest encoder under the optimal Vote Entropy+Random Forest setting, while GraphCodeBERT and CodeT5 provide competitive but not consistently superior results. To support transparency and reproducibility, all source code, experimental artifacts, and datasets used in this study are publicly available [110].

In future work, we plan to evaluate additional code smell types and programming languages to better assess the generalizability of active learning strategies beyond the current Java-based MLCQ dataset. Second, we plan to incorporate a more explicit human-in-the-loop setting, where domain experts participate in the annotation and querying process, to better capture annotator disagreement and its impact on model performance. Third, we plan to investigate automated hyper-parameter tuning for active learning configurations, including query strategy parameters, committee size, and classifier settings, to reduce potential configuration bias and better understand the interaction between tuning choices and query effectiveness. Finally, building on the observed limitation that the current setting relies on a fixed oracle derived from MLCQ ground-truth labels, we plan to explore disagreement-aware active learning scenarios that explicitly model annotator uncertainty and dynamically evolving labeling behavior.

CRediT authorship contribution statement

Haoxuan Chen: Writing – original draft, Validation, Software, Methodology, Investigation, Conceptualization. **Chenxuan Hu:** Software, Methodology, Investigation. **Chengming Liu:** Software, Methodology, Investigation. **Jacky Wai Keung:** Writing – review & editing, Visualization, Validation, Supervision, Formal analysis. **Xiaoxue Ma:** Visualization, Validation, Project administration, Formal analysis, Data curation, Conceptualization. **Wenhua Hu:** Writing – review & editing, Visualization, Validation, Supervision, Funding acquisition, Conceptualization. **Chuanxiang Ma:** Writing – review & editing, Validation, Supervision, Project administration, Funding acquisition, Formal analysis.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the Open Fund of the Hubei Key Laboratory of Big Data Intelligent Analysis and Application (Hubei University) under Grant 2025BDIAA07, the Hubei Province Major Technological Innovation Plan Project under Grant 2024BAA008, and the Hubei Province Key Technological Innovation Plan Project under Grant 2024BAB034.

Data availability

Data will be made available on request.

References

[1] S. Feng, J. Keung, X. Yu, Y. Xiao, K.E. Bennin, M.A. Kabir, M. Zhang, COSTE: Complexity-based OverSampling technique to alleviate the class imbalance problem in software defect prediction, Inf. Softw. Technol. 129 (2021) 106432.
[2] L. Gong, S. Jiang, R. Wang, L. Jiang, Empirical evaluation of the impact of class overlap on software defect prediction, in: 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE, IEEE, 2019, pp. 698–709.
[3] X. Yu, J. Liu, Z. Yang, X. Jia, Q. Ling, S. Ye, Learning from imbalanced data for predicting the number of software defects, in: 2017 IEEE 28th International Symposium on Software Reliability Engineering, ISSRE, IEEE, 2017, pp. 78–89.
[4] F.A. Fontana, P. Braione, M. Zanoni, Automatic detection of bad smells in code: An experimental assessment, J. Object Technol. 11 (2) (2012) 5:1–38.
[5] A. Alazba, H. Aljamaan, Code smell detection using feature selection and stacking ensemble: An empirical investigation, Inf. Softw. Technol. 138 (2021) 106648.
[6] T. Sharma, Detecting and managing code smells: Research and practice, in: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, 2018, pp. 546–547.
[7] Z. Li, X. Zhang, W. Wang, P. Liang, R. Mo, J. Tan, H. Liu, Automated detection of inter-language design smells in multi-language deep learning frameworks, Inf. Softw. Technol. 179 (2025) 107656.
[8] T. Mens, T. Tourwé, A survey of software refactoring, IEEE Trans. Softw. Eng. 30 (2) (2004) 126–139.
[9] F. Pecorelli, F. Palomba, D. Di Nucci, A. De Lucia, Comparing heuristic and machine learning approaches for metric-based code smell detection, in: 2019 IEEE/ACM 27th International Conference on Program Comprehension, ICPC, IEEE, 2019, pp. 93–104.
[10] N. Tsantalis, A. Chatzigeorgiou, Identification of move method refactoring opportunities, IEEE Trans. Softw. Eng. 35 (3) (2009) 347–367.
[11] N. Moha, Y.-G. Guéhéneuc, L. Duchien, A.-F. Le Meur, Decor: A method for the specification and detection of code and design smells, IEEE Trans. Softw. Eng. 36 (1) (2009) 20–36.
[12] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, D. Poshyvanyk, Detecting bad smells in source code using change history information, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE, IEEE, 2013, pp. 268–278.
[13] F. Wang, H. Wu, J. Gao, Code smell detection method based on improved C4. 5 algorithm, Comput. Eng. Des. 42 (04) (2021) 969–975.

[14] F.A. Fontana, M. Zanoni, A. Marino, M.V. Mäntylä, Code smell detection: Towards a machine learning-based approach, in: 2013 IEEE International Conference on Software Maintenance, IEEE, 2013, pp. 396–399.

[15] F. Arcelli Fontana, M.V. Mäntylä, M. Zanoni, A. Marino, Comparing and experimenting machine learning techniques for code smell detection, Empir. Softw. Eng. 21 (3) (2016) 1143–1191.

[16] H. Aljamaan, Voting heterogeneous ensemble for code smell detection, in: 2021 20th IEEE International Conference on Machine Learning and Applications, ICMLA, IEEE, 2021, pp. 897–902.

[17] F.A. Fontana, M. Zanoni, Code smell severity classification using machine learning techniques, Knowl.-Based Syst. 128 (2017) 43–58.

[18] M.I. Azeem, F. Palomba, L. Shi, Q. Wang, Machine learning techniques for code smell detection: A systematic literature review and meta-analysis, Inf. Softw. Technol. 108 (2019) 115–138.

[19] D. Di Nucci, F. Palomba, D.A. Tamburri, A. Serebrenik, A. De Lucia, Detecting code smells using machine learning techniques: Are we there yet? in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering, Saner, IEEE, 2018, pp. 612–621.

[20] R. Blaauwendraad, Evaluating Large Language Models for Code Smell Detection (Ph.D. thesis), Bachelor's thesis, Radboud University, 2025, June 2025. URL: https://www.cs.ru...

[21] K. Lucas, R. Gheyi, E. Soares, M. Ribeiro, I. Machado, Evaluating large language models in detecting test smells, 2024, arXiv preprint arXiv:2407.19261.

[22] B. Zhang, P. Liang, X. Zhou, X. Zhou, D. Lo, Q. Feng, Z. Li, L. Li, A comprehensive evaluation of parameter-efficient fine-tuning on code smell detection, 2024, arXiv preprint arXiv:2412.13801.

[23] D. Mesbah, N. El Madhoun, K. Al Agha, H. Chalouati, Leveraging prompt-based large language models for code smell detection: a comparative study on the MLCQ dataset, in: International Conference on Emerging Internet, Data & Web Technologies, Springer, 2025, pp. 444–454.

[24] A. Gomes, D. Sousa, P. Maia, M. Paixao, Attentionsmelling: Using large language models to identify code smells, in: Simpósio Brasileiro de Engenharia de Software, SBES, SBC, 2025, pp. 271–281.

[25] H.K. Caglar, S. Caglar, E. Tüzün, Automated Inline Comment Smell Detection and Repair with Large Language Models.

[26] M. Li, H. Zhang, R. Wu, Z.-H. Zhou, Sample-based software defect prediction with active and semi-supervised learning, Autom. Softw. Eng. 19 (2) (2012) 201–230.

[27] H. Lu, E. Kocaguneli, B. Cukic, Defect prediction between software versions with active learning and dimensionality reduction, in: 2014 IEEE 25th International Symposium on Software Reliability Engineering, IEEE, 2014, pp. 312–322.

[28] C.M. Liapis, A. Karanikola, S. Kotsiantis, Data-efficient software defect prediction: A comparative analysis of active learning-enhanced models and voting ensembles, Inform. Sci. 676 (2024) 120786.

[29] Z. Yu, C. Theisen, L. Williams, T. Menzies, Improving vulnerability inspection efficiency using active learning, IEEE Trans. Softw. Eng. 47 (11) (2019) 2401–2420.

[30] X. Sun, L. Tu, J. Zhang, J. Cai, B. Li, Y. Wang, ASSBert: Active and semi-supervised bert for smart contract vulnerability detection, J. Inf. Secur. Appl. 73 (2023) 103423.

[31] H. Guan, G. Cai, H. Xu, Automatic requirement dependency extraction based on integrated active learning strategies, Mach. Intell. Res. 21 (5) (2024) 993–1010.

[32] L. Madeyski, T. Lewowski, Mlcq: Industry-relevant code smell data set, in: Proceedings of the 24th International Conference on Evaluation and Assessment in Software Engineering, 2020, pp. 342–347.

[33] H.J. Kang, D. Lo, Active learning of discriminative subgraph patterns for api misuse detection, IEEE Trans. Softw. Eng. 48 (8) (2021) 2761–2783.

[34] V.T. Dhinakaran, R. Pulle, N. Ajmeri, P.K. Murukannaiah, App review analysis via active learning: Reducing supervision effort without compromising classification accuracy, in: 2018 IEEE 26th International Requirements Engineering Conference, RE, IEEE, 2018, pp. 170–181.

[35] C. Magalhães, J. Araujo, A. Sardinha, Mare: An active learning approach for requirements classification, in: 2021 IEEE 29th International Requirements Engineering Conference, RE, IEEE, 2021, pp. 516–521.

[36] L.L. Silva, J.R.d. Silva, J.E. Montandon, M. Andrade, M.T. Valente, Detecting code smells using chatgpt: Initial insights, in: Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2024, pp. 400–406.

[37] F. Li, K. Zou, J.W. Keung, X. Yu, S. Feng, Y. Xiao, On the relative value of imbalanced learning for code smell detection, Softw.: Pr. Exp. 53 (10) (2023) 1902–1927.

[38] C. Tantithamthavorn, S. McIntosh, A.E. Hassan, K. Matsumoto, An empirical comparison of model validation techniques for defect prediction models, IEEE Trans. Softw. Eng. 43 (1) (2016) 1–18.

[39] R. Terra, M.T. Valente, S. Miranda, V. Sales, JMove: A novel heuristic and tool to detect move method refactoring opportunities, J. Syst. Softw. 138 (2018) 19–36.

[40] N. Tsantalis, T. Chaikalis, A. Chatzigeorgiou, JDeodorant: Identification and removal of type-checking bad smells, in: 2008 12th European Conference on Software Maintenance and Reengineering, IEEE, 2008, pp. 329–331.

[41] A.M. Fard, A. Mesbah, Jsnose: Detecting javascript code smells, in: 2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation, SCAM, IEEE, 2013, pp. 116–125.

[42] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, E. Figueiredo, A review-based comparative study of bad smell detection tools, in: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, 2016, pp. 1–12.

[43] J. Kreimer, Adaptive detection of design flaws, Electron. Notes Theor. Comput. Sci. 141 (4) (2005) 117–136.

[44] L. Amorim, E. Costa, N. Antunes, B. Fonseca, M. Ribeiro, Experience report: Evaluating the effectiveness of decision trees for detecting code smells, in: 2015 IEEE 26th International Symposium on Software Reliability Engineering, ISSRE, IEEE, 2015, pp. 261–269.

[45] S. Vaucher, F. Khomh, N. Moha, Y.-G. Guéhéneuc, Tracking design smells: Lessons from a study of god classes, in: 2009 16th Working Conference on Reverse Engineering, IEEE, 2009, pp. 145–154.

[46] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, E. Aimeur, Smurf: A svm-based incremental anti-pattern detection approach, in: 2012 19th Working Conference on Reverse Engineering, IEEE, 2012, pp. 466–475.

[47] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, H. Sahraoui, BDTEX: A GQM-based Bayesian approach for the detection of antipatterns, J. Syst. Softw. 84 (4) (2011) 559–572.

[48] S. Boutaib, S. Bechikh, F. Palomba, M. Elarbi, M. Makhlouf, L.B. Said, Code smell detection and identification in imbalanced environments, Expert Syst. Appl. 166 (2021) 114076.

[49] R. Yedida, T. Menzies, How to improve deep learning for software analytics: (A case study with code smell detection), in: Proceedings of the 19th International Conference on Mining Software Repositories, 2022, pp. 156–166.

[50] J. Guo, Y. Zhao, T. Zheng, Z. Chen, M. Jiang, Z. Ding, Alleviating class imbalance in feature envy prediction: An oversampling technique based on code entity attributes, Inf. Softw. Technol. 180 (2025) 107673.

[51] T. Sharma, V. Efstathiou, P. Louridas, D. Spinellis, Code smell detection by deep direct-learning and transfer-learning, J. Syst. Softw. 176 (2021) 110936.

[52] Q. Yang, D. Yu, X. Chen, Y. Xu, W. Yan, B. Hu, Feature envy detection based on cross-graph local semantics matching, Inf. Softw. Technol. 174 (2024) 107515.

[53] F. Zhang, Z. Zhang, J.W. Keung, X. Tang, Z. Yang, X. Yu, W. Hu, Data preparation for deep learning based code smell detection: A systematic literature review, J. Syst. Softw. 216 (2024) 112131.

[54] A. Kovačević, J. Slivka, D. Vidaković, K.-G. Grujić, N. Luburić, S. Prokić, G. Sladić, Automatic detection of long method and god class code smells through neural source code embeddings, Expert Syst. Appl. 204 (2022) 117607.

[55] A. Ho, A.M. Bui, P.T. Nguyen, A. Di Salle, Fusion of deep convolutional and LSTM recurrent neural networks for automated detection of code smells, in: Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering, 2023, pp. 229–234.

[56] A. Ho, A.M. Bui, P.T. Nguyen, A. Di Salle, B. Le, EnseSmells: Deep ensemble and programming language models for automated code smells detection, J. Syst. Softw. 224 (2025) 112375.

[57] X. Yu, Z. Zhang, F. Niu, X. Hu, X. Xia, J. Grundy, What makes a high-quality training dataset for large language models: A practitioners' perspective, in: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 2024, pp. 656–668.

[58] Z. Yang, F. Liu, Z. Yu, J.W. Keung, J. Li, S. Liu, Y. Hong, X. Ma, Z. Jin, G. Li, Exploring and unleashing the power of large language models in automated code translation, Proc. ACM Softw. Eng. 1 (FSE) (2024) 1585–1608.

[59] X. Yu, L. Liu, X. Hu, J.W. Keung, J. Liu, X. Xia, Fight fire with fire: How much can we trust chatgpt on source code-related tasks? IEEE Trans. Softw. Eng. 50 (12) (2024) 3435–3453.

[60] A.R. Sadik, S. Govind, Benchmarking LLM for code smells detection: Openai GPT-4.0 vs DeepSeek-V3, 2025, arXiv preprint arXiv:2504.16027.

[61] J.I. Irabedra, M. Solari, G. Mousqués, Active learning in software engineering: A rapid review, in: 2024 L Latin American Computer Conference, CLEI, IEEE, 2024, pp. 1–8.

[62] P.K. Murukannaiah, M.P. Singh, Platys: An active learning framework for place-aware application development and its evaluation, ACM Trans. Softw. Eng. Methodol. (TOSEM) 24 (3) (2015) 1–32.

[63] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., Codebert: A pre-trained model for programming and natural languages, 2020, arXiv preprint arXiv:2002.08155.

[64] H. Zhang, The optimality of naive Bayes, Aa 1 (2) (2004) 3.

[65] D.W. Hosmer Jr., S. Lemeshow, R.X. Sturdivant, Applied Logistic Regression, John Wiley & Sons, 2013.

[66] J.R. Quinlan, C4. 5: Programs for Machine Learning, Elsevier, 2014.

[67] L.E. Peterson, K-nearest neighbor, Scholarpedia 4 (2) (2009) 1883.

[68] A. Pinkus, Approximation theory of the MLP model in neural networks, Acta Numer. 8 (1999) 143–195.

[69] C. Cortes, V. Vapnik, Support-vector networks, Mach. Learn. 20 (3) (1995) 273–297.

[70] A. Liaw, M. Wiener, et al., Classification and regression by randomforest, R News 2 (3) (2002) 18–22.

[71] Y. Freund, R.E. Schapire, A decision-theoretic generalization of on-line learning and an application to boosting, J. Comput. System Sci. 55 (1) (1997) 119–139.

[72] R.S. Rao, S. Dewangan, A. Mishra, M. Gupta, A study of dealing class imbalance problem with machine learning methods for code smell severity detection using PCA-based feature selection technique, Sci. Rep. 13 (1) (2023) 16245.

[73] M. Škipina, J. Slivka, N. Luburić, A. Kovačević, Automatic detection of feature envy and data class code smells using machine learning, Expert Syst. Appl. 243 (2024) 122855.

[74] S. Mcmurray, A.H. Sodhro, A study on ML-based software defect detection for security traceability in smart healthcare applications, Sensors 23 (7) (2023) 3470.

[75] D.D. Lewis, J. Catlett, Heterogeneous uncertainty sampling for supervised learning, in: Machine Learning Proceedings 1994, Elsevier, 1994, pp. 148–156.

[76] H.S. Seung, M. Opper, H. Sompolinsky, Query by committee, in: Proceedings of the Fifth Annual Workshop on Computational Learning Theory, 1992, pp. 287–294.

[77] D. Cohn, L. Atlas, R. Ladner, Improving generalization with active learning, Mach. Learn. 15 (2) (1994) 201–221.

[78] N. Roy, A. McCallum, Toward optimal active learning through sampling estimation of error reduction, in: Proc. 18th International Conf. on Machine Learning, 2001.

[79] S.-J. Huang, R. Jin, Z.-H. Zhou, Active learning by querying informative and representative examples, Adv. Neural Inf. Process. Syst. 23 (2010).

[80] T. Scheffer, C. Decomain, S. Wrobel, Active hidden markov models for information extraction, in: International Symposium on Intelligent Data Analysis, Springer, 2001, pp. 309–318.

[81] F. Jing, M. Li, H.-J. Zhang, B. Zhang, Entropy-based active learning with support vector machines for content-based image retrieval, in: 2004 IEEE International Conference on Multimedia and Expo (ICME)(IEEE Cat. No. 04TH8763), Vol. 1, IEEE, 2004, pp. 85–88.

[82] A. Culotta, A. McCallum, Reducing labeling effort for structured prediction tasks, in: AAAI, Vol. 5, 2005, pp. 746–751.

[83] S. Argamon-Engelson, I. Dagan, Committee-based sample selection for probabilistic classifiers, J. Artificial Intelligence Res. 11 (1999) 335–360.

[84] Z. Wang, J. Ye, Querying discriminative and representative samples for batch mode active learning, ACM Trans. Knowl. Discov. Data (TKDD) 9 (3) (2015) 1–23.

[85] D.A. Cohn, Z. Ghahramani, M.I. Jordan, Active learning with statistical models, J. Artificial Intelligence Res. 4 (1996) 129–145.

[86] S. Ebert, M. Fritz, B. Schiele, Ralf: A reinforced active learning formulation for object class recognition, in: 2012 IEEE Conference on Computer Vision and Pattern Recognition, IEEE, 2012, pp. 3626–3633.

[87] Y.-P. Tang, S.-J. Huang, Self-paced active learning: Query the right thing at the right time, in: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 33, 2019, pp. 5117–5124.

[88] T.N. Cardoso, R.M. Silva, S. Canuto, M.M. Moro, M.A. Gonçalves, Ranked batch-mode active learning, Inform. Sci. 379 (2017) 313–337.

[89] W. Mi, Y. Li, S. Wang, Empirical evaluation of the active learning strategies on software defects prediction, in: 2020 6th International Symposium on System and Software Reliability, ISSSR, IEEE, 2020, pp. 83–89.

[90] R.M. de Mello, R. Oliveira, A. Garcia, On the influence of human factors for identifying code smells: A multi-trial empirical study, in: 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM, IEEE, 2017, pp. 68–77.

[91] M. Hozano, A. Garcia, N. Antunes, B. Fonseca, E. Costa, Smells are sensitive to developers! on the efficiency of (un)guided customized detection, in: Proceedings of the 25th International Conference on Program Comprehension, ICPC, IEEE, 2017, pp. 388–398.

[92] L. Madeyski, T. Lewowski, Detecting code smells using industry-relevant data, Inf. Softw. Technol. 155 (2023) 107112.

[93] R. Marinescu, Detection strategies: Metrics-based rules for detecting design flaws, in: 20th IEEE International Conference on Software Maintenance, 2004. Proceedings, IEEE, 2004, pp. 350–359.

[94] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, Do they really smell bad? A study on developers' perception of bad code smells, in: 2014 IEEE International Conference on Software Maintenance and Evolution, IEEE, 2014, pp. 101–110.

[95] C. Ni, W. Wang, K. Yang, X. Xia, K. Liu, D. Lo, The best of both worlds: Integrating semantic features with expert features for defect prediction and localization, in: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2022, pp. 672–683.

[96] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al., Graphcodebert: Pre-training code representations with data flow, 2020, arXiv preprint arXiv:2009.08366.

[97] Y. Wang, W. Wang, S. Joty, S.C. Hoi, Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, in: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, 2021, pp. 8696–8708.

[98] W. Ahmad, S. Chakraborty, B. Ray, K.-W. Chang, Unified pre-training for program understanding and generation, in: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2021, pp. 2655–2668.

[99] DeepSeek-AI, DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026.

[100] GLM-5-Team, A. Zeng, X. Lv, Z. Hou, Z. Du, Q. Zheng, B. Chen, D. Yin, C. Ge, C. Huang, C. Xie, C. Zhu, C. Yin, C. Wang, G. Pan, H. Zeng, H. Zhang, H. Wang, H. Chen, J. Zhang, J. Jiao, J. Guo, J. Wang, J. Du, J. Wu, K. Wang, L. Li, L. Fan, L. Zhong, M. Liu, M. Zhao, P. Du, Q. Dong, R. Lu, Shuang-Li, S. Cao, S. Liu, T. Jiang, X. Chen, X. Zhang, X. Huang, X. Dong, Y. Xu, Y. Wei, Y. An, Y. Niu, Y. Zhu, Y. Wen, Y. Cen, Y. Bai, Z. Qiao, Z. Wang, Z. Wang, Z. Zhu, Z. Liu, Z. Li, B. Wang, B. Wen, C. Huang, C. Cai, C. Yu, C. Li, C. Hu, C. Zhang, D. Zhang, D. Lin, D. Yang, D. Wang, D. Ai, E. Zhu, F. Yi, F. Chen, G. Wen, H. Sun, H. Zhao, H. Hu, H. Zhang, H. Liu, H. Zhang, H. Peng, H. Tai, H. Zhang, H. Liu, H. Wang, H. Yan, H. Ge, H. Liu, H. Chu, J. Zhao, J. Wang, J. Zhao, J. Ren, J. Wang, J. Zhang, J. Gui, J. Zhao, J. Li, J. An, J. Li, J. Yuan, J. Du, J. Liu, J. Zhi, J. Duan, K. Zhou, K. Wei, K. Wang, K. Luo, L. Zhang, L. Sha, L. Xu, L. Wu, L. Ding, L. Chen, M. Li, N. Lin, P. Ta, Q. Zou, R. Song, R. Yang, S. Tu, S. Yang, S. Wu, S. Zhang, S. Li, S. Li, S. Fan, W. Qin, W. Tian, W. Zhang, W. Yu, W. Liang, X. Kuang, X. Cheng, X. Li, X. Yan, X. Hu, X. Ling, X. Fan, X. Xia, X. Zhang, X. Zhang, X. Pan, X. Zou, X. Zhang, Y. Liu, Y. Wu, Y. Li, Y. Wang, Y. Zhu, Y. Tan, Y. Zhou, Y. Pan, Y. Zhang, Y. Su, Y. Geng, Y. Yan, Y. Tan, Y. Bi, Y. Shen, Y. Yang, Y. Li, Y. Liu, Y. Wang, Y. Li, Y. Wu, Y. Zhang, Y. Duan, Y. Zhang, Z. Liu, Z. Jiang, Z. Yan, Z. Zhang, Z. Wei, Z. Chen, Z. Feng, Z. Yao, Z. Chai, Z. Wang, Z. Zhang, B. Xu, M. Huang, H. Wang, J. Li, Y. Dong, J. Tang, GLM-5: From vibe coding to agentic engineering, 2026, arXiv:2602.15763. URL: https://arxiv.org/abs/2602.15763.

[101] M. AI, Kimi-K2.6, 2026, https://huggingface.co/moonshotai/Kimi-K2.6.

[102] R. de Mello, R. Oliveira, A. Uchôa, W. Oizumi, A. Garcia, B. Fonseca, F. de Mello, Recommendations for developers identifying code smells, IEEE Softw. 40 (2) (2022) 90–98.

[103] Y.-P. Tang, G.-X. Li, S.-J. Huang, ALiPy: Active learning in python, 2019, arXiv preprint arXiv:1901.03802.

[104] T. Danka, P. Horvath, modAL: A modular active learning framework for python, 2018, arXiv preprint arXiv:1805.00979.

[105] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al., Scikit-learn: Machine learning in python, J. Mach. Learn. Res. 12 (2011) 2825–2830.

[106] D. Lin, C. Tantithamthavorn, A.E. Hassan, The impact of data merging on the interpretation of cross-project just-in-time defect models, IEEE Trans. Softw. Eng. 48 (8) (2021) 2969–2986.

[107] M. Kondo, D.M. German, O. Mizuno, E.-H. Choi, The impact of context metrics on just-in-time defect prediction, Empir. Softw. Eng. 25 (1) (2020) 890–939.

[108] S. Qiu, H. Xu, J. Deng, S. Jiang, L. Lu, Transfer convolutional neural network for cross-project defect prediction, Appl. Sci. 9 (13) (2019) 2660.

[109] P. Yang, L. Zhu, Y. Zhang, C. Ma, L. Liu, X. Yu, W. Hu, On the relative value of clustering techniques for unsupervised effort-aware defect prediction, Expert Syst. Appl. 245 (2024) 123041.

[110] H. Chen, C. Hu, C. Liu, J.W. Keung, W. Hu, C. Ma, X. Ma, ALCSD: Active learning for code smell detection — Replication package, 2026, https://github.com/somethingpluto/active-learning-code-smell. GitHub repository.