

## Article

# Deep Active Learning for Label-Efficient Refactoring Prediction

Abdulmajeed Alameer <sup>1,\*</sup>  and Amal Alazba <sup>2</sup> 

<sup>1</sup> Department of Computer Science, College of Computer and Information Sciences, King Saud University, Riyadh 11451, Saudi Arabia

<sup>2</sup> Department of Information Systems, College of Computer and Information Sciences, King Saud University, Riyadh 11451, Saudi Arabia; aalazba@ksu.edu.sa

\* Correspondence: abalameer@ksu.edu.sa

## Abstract

Software refactoring improves the maintainability of code and reduces technical debt, but making the construction of a labeled refactoring dataset is a costly and labor-intensive process. To make refactoring prediction more deployable under limited annotation budgets, this paper introduces a Deep Active Learning (DAL) pipeline that iteratively trains a deep neural classifier on software-metric representations and selectively queries labels for the most informative unlabeled entities. Our proposed approach is evaluated in a pool-based setting across class-, method-, and variable-level refactoring datasets (multiple refactoring types) using a consistent training protocol and a broad set of query strategies. Results show that DAL can recover near full-data effectiveness with substantially fewer labels: on average, reaching the target performance requires 11.4% labeled data for class-level, 25.0% for method-level, and 20.0% for variable-level refactorings—corresponding to roughly 75–89% labeling savings, demonstrating improved data efficiency for refactoring prediction. Moreover, uncertainty-based and dropout-enhanced strategies were the most consistently effective query strategies across refactoring types and labeling budgets.

**Keywords:** software refactoring; refactoring prediction; active learning; deep active learning; uncertainty sampling; MC-dropout; software metrics; machine learning for software engineering

## 1. Introduction

Refactoring is an essential practice in software engineering that restructures source code to improve its internal design while preserving the system's externally visible behavior. By applying transformations such as renaming identifiers, extracting or inlining methods, and relocating or reorganizing classes, developers can enhance comprehensibility and maintainability and mitigate the accumulation of technical debt as software evolves.

A broad range of empirical studies report that refactoring is associated with improvements in code-quality indicators, including lower complexity and higher maintainability [1–4]. These improvements are not merely cosmetic: they can reduce the effort required for understanding and debugging, support safer evolution, and accelerate the implementation of new features. For long-lived systems, sustained refactoring is often necessary to prevent gradual design erosion and to keep the codebase manageable.

Despite these benefits, refactoring remains challenging to execute consistently and at scale. Developers must decide which parts of the codebase deserve attention, choose appropriate refactoring operations, and ensure that changes do not introduce faults or



Academic Editors: Yunwei Dong,  
Guanjun Liu and Xiaomin Wei

Received: 2 March 2026

Revised: 21 April 2026

Accepted: 23 April 2026

Published: 27 April 2026

**Copyright:** © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and

conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

unintended side effects. In large and fast-changing projects, manual identification of refactoring candidates can be time-consuming, making tool support essential for prioritizing opportunities and guiding developers toward effective interventions.

To assist with this decision-making, recent research has explored automated refactoring opportunity prediction using machine learning [5–10]. Supervised models trained on historical changes, code metrics, or structural representations can learn patterns that correlate with refactoring events and then recommend likely candidates. For instance, Sagar et al. formulated method-level refactoring type prediction as a multi-class classification problem over metric variations extracted from commits, finding that Random Forest classifiers trained on code metrics achieve the best average accuracy of 75% across six refactoring types [11]. Jesse et al. trained RefactorBERT, a transformer-based model, on over a million refactoring commits to derive a RefactorScore that identifies refactor-prone code at the token level, demonstrating that learned representations from commit histories can scale to industrial-size repositories [12]. Büyük and Nizam proposed a deep learning approach that combines class-level abstract syntax tree embeddings with recurrent neural networks over code histories to detect code modification requirements, reporting average accuracy of 83–86% and showing that structural representations outperform text- and method-level embeddings [13]. However, building such predictors typically assumes the availability of sufficiently large labeled datasets [6]. In practice, software repositories may provide abundant unlabeled code entities and change histories, while reliable labels (e.g., validated refactoring instances or confirmed refactoring needs) remain scarce and costly to obtain. Consequently, practitioners face a dual challenge: (i) the computational cost of training high-capacity models and (ii) the human or tooling effort required to annotate enough samples to achieve strong generalization, especially when migrating the model across projects or evolving codebases.

This paper proposes the use of Deep Active Learning (DAL) in software refactoring prediction to address both challenges. Rather than labeling a large dataset upfront, we train a deep model iteratively and use an active learning strategy to select which instances should be labeled next. This setting is particularly well-suited to refactoring prediction, where large pools of unlabeled candidates are readily available but annotation budgets are limited. By querying only the most informative samples—those expected to yield the largest performance gain for the current model—DAL aims to maximize predictive improvement per annotated instance while simultaneously reducing labeling cost. In addition, selecting smaller, high-value labeled subsets can shorten training cycles and enable faster iteration without sacrificing accuracy.

To the best of our knowledge, this work is the first to study DAL as a practical mechanism for making refactoring prediction more efficient and easier to adopt. While the underlying deep neural classifier utilizes standard architectural components to ensure reproducibility and stability, the novelty of this work lies in the optimization of the DAL pipeline specifically for the multi-level (class, method, and variable) refactoring domain, where we evaluate which query strategies most effectively navigate the imbalanced nature of software metrics. More specifically, our main contributions are:

1. A DAL technique for refactoring opportunity prediction that explicitly targets label-efficiency by prioritizing samples with high expected utility.
2. A comprehensive empirical evaluation that compares various DAL query strategies for refactoring prediction across class-, method-, and variable-level refactoring types and quantifies their effectiveness and labeling efficiency at multiple labeling budgets relative to passive-learning (PL).
3. A publicly available replication package containing source code, experimental settings, and processed datasets to support reproducibility and enable future extensions.

The remainder of this paper is organized as follows. Section 2 gives brief background information about refactoring prediction and DAL. Section 3 reviews prior research on refactoring prediction and active learning in software engineering. Section 4 describes the proposed DAL approach, including model design and query strategies. Section 5 details the empirical evaluation, stating the study objectives and research questions, describing the refactoring datasets and data processing steps, presenting the performance metrics, and introducing the replication package. Section 6 presents the experimental setup and results, followed by a discussion of implications. Threats to validity are examined in Section 8, and Section 9 concludes the paper and outlines future directions.

## 2. Background Information

This section provides the background information needed for the rest of the paper. We first summarize the purpose of software refactoring and illustrate common refactoring operations at different program granularities. Then, we give an overview of DAL that motivates our efficiency-oriented approach.

### 2.1. Software Refactoring

Software refactoring refers to semantics-preserving transformations that improve the internal structure of a program without changing its externally observable behavior [1,14]. Refactorings may be applied at multiple granularities (e.g., variable, method, and class) with goals such as improving readability, reducing complexity, increasing modularity, and lowering future maintenance effort.

Refactoring catalogs commonly describe operations at different levels of abstraction [14,15]. Below we list representative examples that are frequently supported by modern IDEs. We also provide three concrete before/after examples in Java (class-, method-, and variable-level), which are among the most common refactorings in practice.

- Class-level refactorings (structural changes around types and responsibilities):
  - Extract Class: Split an overgrown class by moving a coherent subset of fields/methods into a new class.
  - Move Class: Relocate a class to a more appropriate package/module (often to reduce inappropriate coupling).
  - Pull Up/Push Down: Move common members to a superclass (pull up) or specialized members to subclasses (push down).
  - Rename Class: Improve semantic clarity when the current name no longer matches the class responsibility.
- Method-level refactorings (improving method structure, naming, and responsibilities):
  - Extract Method: Create a new method for a cohesive block of code to reduce method size and improve readability.
  - Inline Method: Replace calls to a trivial method with its body to remove unnecessary indirection.
  - Move Method: Move a method to the class where the data it primarily uses resides (often to increase cohesion).
  - Rename Method: Align a method name with its behavior after functionality changes.
- Variable-level refactorings (improving local clarity, state management, and encapsulation):
  - Rename Variable/Field: Use descriptive identifiers that reduce cognitive load.
  - Encapsulate Field: Restrict direct field access by introducing getters/setters (or properties) and updating references.

- Split Variable: Replace a variable that holds multiple conceptual values over time with distinct variables.
- Inline Temp: Remove a short-lived temporary variable when it does not add clarity.

### Illustrative Refactoring Examples in Java

#### (1) Class-level example: Extract Class.

A common motivation for Extract Class is an overgrown class that mixes multiple responsibilities (e.g., customer identity and address handling). The refactoring separates a cohesive subset of fields and methods into a new class. Figure 1 shows an example of an Extract Class refactoring operation.

##### Before (mixed responsibilities)

```
public class Customer {
    private String name;
    private String street;
    private String city;
    private String zip;

    public Customer(String name, String
        street,
            String city, String zip) {
        this.name = name;
        this.street = street;
        this.city = city;
        this.zip = zip;
    }

    public String shippingLabel() {
        return name + "\n" + street + "\n"
            + city + "\n" + zip;
    }
}
```

##### After (Extract Class: Address)

```
public class Customer {
    private String name;
    private Address address;

    public Customer(String name, Address
        address) {
        this.name = name;
        this.address = address;
    }

    public String shippingLabel() {
        return name + "\n" + address.format()
            ;
    }
}

public class Address {
    private String street;
    private String city;
    private String zip;

    public Address(String street, String
        city,
            String zip) {
        this.street = street;
        this.city = city;
        this.zip = zip;
    }

    public String format() {
        return street + "\n" + city + "\n" +
            zip;
    }
}
```

**Figure 1.** Class-level refactoring example (Extract Class): address-related state and formatting behavior are moved into a dedicated Address class.

#### (2) Method-level example: Extract Method.

The Extract Method is one of the most common method-level refactorings used to decompose long methods into smaller, well-named units. Figure 2 shows an example of an Extract Method refactoring operation.

**Before (single long method)**

```
public class OrderService {
    public double total(Order order) {
        double subtotal = 0.0;
        for (Item i : order.getItems()) {
            subtotal += i.getPrice() * i.
                getQty();
        }
        double discount = 0.0;
        if (subtotal > 100.0) {
            discount = subtotal * 0.10;
        }
        return subtotal - discount;
    }
}
```

**After (Extract Method)**

```
public class OrderService {
    public double total(Order order) {
        double subtotal = computeSubtotal(
            order);
        double discount = computeDiscount(
            subtotal);
        return subtotal - discount;
    }

    private double computeSubtotal(Order
        order) {
        double subtotal = 0.0;
        for (Item i : order.getItems()) {
            subtotal += i.getPrice() * i.
                getQty();
        }
        return subtotal;
    }

    private double computeDiscount(double
        subtotal) {
        if (subtotal > 100.0) {
            return subtotal * 0.10;
        }
        return 0.0;
    }
}
```

**Figure 2.** Method-level refactoring example (Extract Method): cohesive steps are decomposed into named helper methods to improve readability and testability.

(3) Variable-level example: Rename Variable.

Rename Variable is a frequent local refactoring that improves code readability by making intent explicit. Figure 3 shows an example of a Rename Variable refactoring operation.

**Before (unclear name)**

```
public class Checkout {
    public double finalPrice(double price) {
        double d = 0.0;
        if (price > 100.0) {
            d = price * 0.10;
        }
        return price - d;
    }
}
```

**After (Rename Variable)**

```
public class Checkout {
    public double finalPrice(double price) {
        double discount = 0.0;
        if (price > 100.0) {
            discount = price * 0.10;
        }
        return price - discount;
    }
}
```

**Figure 3.** Variable-level refactoring example (Rename Variable): renaming improves clarity without changing behavior.

## 2.2. Active Learning

Active learning (AL) seeks to reduce labeling effort by iteratively selecting the most informative unlabeled instances for annotation under a fixed budget. In the common pool-based setting, AL starts from a small labeled set and iteratively selects additional unlabeled instances to be annotated by an oracle (e.g., a developer or domain expert), aiming to maximize performance gains per label queried. DAL adapts these ideas to deep neural networks and is especially relevant when labeling and training costs are a bottleneck.

Compared to classical AL with shallow models, DAL introduces practical and methodological challenges. Deep models typically require more data and computation to converge, making frequent retraining costly.

### 3. Related Work

Research on refactoring has expanded along two complementary directions: (i) mining and detecting refactoring activities from software histories, and (ii) predicting refactoring opportunities to support developers before changes are made. In parallel, the broader software engineering (SE) community has increasingly investigated learning strategies that reduce the cost of building predictive models, especially when labeled data are limited. Since our work focuses on Deep Active Learning for refactoring prediction, we review prior studies on refactoring detection and prediction, followed by learning approaches that explicitly target label efficiency.

#### 3.1. Refactoring Detection and Mining from Software Repositories

A number of studies aim to identify refactoring events after they occur by analyzing commits and version-control traces. AlOmar et al. introduced Self-Affirmed Refactoring (SAR), where developers explicitly describe refactoring in commit messages, and proposed a learning-based pipeline to recognize such commits and categorize their intent [16]. Their results show that textual and commit-level signals can be leveraged to find refactoring-related changes at scale. Similarly, Noei et al. proposed MLRefScanner for identifying refactoring commits in Python-based machine learning projects, using change features that capture both general and ML-specific refactoring patterns [17]. These works demonstrate the value of learning-based refactoring mining, but their primary goal is retrospective identification of refactoring activities. In contrast, our work targets prospective prediction: given code entities (e.g., classes, methods, or variables), we aim to predict refactoring opportunities before a refactoring is performed.

#### 3.2. Refactoring Opportunity Prediction

Refactoring opportunity prediction is commonly formulated as a supervised classification problem, where models learn from labeled examples derived from project histories and/or refactoring detectors. Several studies have utilized static code metrics to predict refactoring opportunities at various granularities. At the class level, research has explored the predictive utility of metrics [18,19] and the use of dimensionality reduction like PCA to manage high-dimensional data [20]. Similarly, traditional classifiers have been evaluated for method-level prediction [21,22], with the results suggesting that structural signals typically outperform textual signals [11]. While these works establish the learnability of refactoring, they largely assume the availability of large labeled datasets.

Prior research has extensively explored refactoring prediction across various granularities. Notably, Aniche et al. [5] conducted a systematic evaluation of supervised learning models at the class, method, and variable levels, demonstrating that robust performance can be achieved using conventional classifiers. Their study also provided a rigorous analysis of feature importance, specifically identifying the software metrics that most effectively predict various refactoring operations. Building on these established foundations regarding feature relevance, our work shifts the focus toward the challenge of maximizing label efficiency through DAL. Nyamawe et al. proposed a recommendation approach that leverages histories of feature requests and previously applied refactorings to predict refactorings needed for new feature work [23]. Alameer and Alazba proposed a two-level tree-based stacking ensemble to predict refactoring opportunities across three granularities—class, method, and variable levels. They explored multiple stacking configurations (varying base and meta learners) and applied two feature selection techniques to identify the most informative software metrics for prediction. The approach was evaluated on a publicly available dataset with over two million refactoring instances and was benchmarked against

nine traditional ML models and seven tree-based ensemble methods; the results showed that tree-based stacking consistently outperforms traditional machine learning models [6].

Beyond metric-based approaches, representation learning has also been adopted: Kurbatova et al. proposed a recommender for Move Method refactoring that uses code2vec embeddings to capture syntactic/semantic properties of code fragments, demonstrating the promise of learned code representations for refactoring recommendations [24].

Collectively, these works show that refactoring opportunities are learnable from historical data and code characteristics. However, most approaches assume that sufficiently large labeled datasets are available and typically optimize for predictive effectiveness rather than *label efficiency* and *training efficiency*.

### 3.3. Label-Efficient Learning and Active Learning for Software Engineering

A practical barrier for deploying refactoring predictors is the cost of obtaining labels (e.g., validated refactoring instances or high-confidence refactoring needs) and the cost of training and iterating on complex models. Active learning addresses this by selecting, from a large unlabeled pool, the instances whose labels are expected to yield the largest improvement in the model [25]. DAL extends AL to deep models and is often motivated by scenarios where unlabeled data are abundant, but annotation is expensive, and model retraining is costly [26–30].

For example, Liapis et al. investigated data-efficient software defect prediction by comparatively evaluating (i) active learning-enhanced variants of traditional classifiers and (ii) voting ensemble schemes using common query strategies such as uncertainty, margin, and entropy sampling [31]. The results of this work show that active learning methods frequently match baseline performance while substantially reducing labeling effort (e.g., at least 75% training-set reduction in most of the cases) [31]. While their target is defect proneness rather than refactoring need, their results reinforce the broader SE motivation for AL: competitive performance can often be achieved under tight labeling budgets.

Active learning has also been applied to vulnerability inspection. For instance, Yu et al. [32] proposed the HARMLESS framework, which uses a staged querying policy and error-correction mechanisms to reduce the human effort required for inspecting large codebases. While this study demonstrates the power of AL in SE, our work is the first to systematically investigate DAL specifically for the multi-level refactoring prediction domain.

More recently, several studies have broadened the application of AL across diverse SE tasks. Feyzi and Daneshdoost [33] studied the effectiveness of deep active learning specifically for software defect prediction, proposing deep learning classifiers trained with pool-based AL and evaluating multiple sampling strategies; their results show improvements in F-measure and AUC compared to traditional ML baselines while substantially reducing the annotation burden. Wang et al. extended AL to the cross-project just-in-time (JIT) defect prediction setting, where labeled data from the target project are scarce, and demonstrated that querying uncertain target-project samples with an active deep learning strategy improves cross-project prediction performance over passive labeling [34]. In the vulnerability detection domain, Sun et al. combined pool-based active learning with semi-supervised pseudo-labeling in ASSBert, a BERT-based framework for smart contract vulnerability detection; by selecting highly uncertain Solidity files for manual annotation, their approach achieves strong detection performance with only a small initial labeled set [35]. Finally, Hu et al. provided the first systematic benchmark of AL acquisition functions for training code models (Active Code Learning), evaluating eleven strategies across code-understanding tasks such as clone detection and vulnerability detection in IEEE TSE; their results highlight that representation choice strongly governs AL effectiveness

and that output-vector-based sample selection consistently outperforms other acquisition functions [36].

### 3.4. Summary of Differences from Prior Work

In summary, prior research has (i) detected and characterized refactorings in commit histories [16,17], (ii) predicted refactoring opportunities using supervised learning with metrics and learned representations [5,18,19,24], and (iii) explored various modeling improvements to enhance predictive performance. In contrast, our paper investigates Deep Active Learning as a mechanism to reduce labeling and training costs while maintaining competitive accuracy, making refactoring prediction more deployable in settings with large unlabeled repositories and limited annotation budgets.

## 4. Methodology

In this section, we present the methodology used to utilize DAL to predict refactoring opportunities. We discuss the problem formulation, general framework, refactoring datasets, data preprocessing, deep learning model architecture, DAL protocol, query strategies, and implementation.

### 4.1. Problem Formulation

We formulate refactoring opportunity prediction as a supervised binary classification problem. Given a software entity  $x$  (e.g., class or method) represented by a vector of software metrics, the goal is to predict whether a refactoring operation should be applied ( $y = 1$ ) or not ( $y = 0$ ). Since labeling refactoring opportunities requires expert knowledge and is costly, we adopt a pool-based DAL setting to reduce the annotation effort while maintaining predictive performance [25,28,37].

Let  $\mathcal{P}$  denote the training pool,  $\mathcal{L} \subset \mathcal{P}$  the labeled subset,  $\mathcal{U} = \mathcal{P} \setminus \mathcal{L}$  the unlabeled pool, and  $\mathcal{T}$  a fixed test set. At each DAL round, the learner selects informative instances from  $\mathcal{U}$  for labeling by an oracle, expanding  $\mathcal{L}$  and retraining the model.

### 4.2. Overall Framework

Figure 4 summarizes our DAL approach for refactoring opportunity prediction. The pipeline starts from the refactoring datasets and proceeds through Data preprocessing to construct a fixed test set  $\mathcal{T}$  and a training pool  $\mathcal{P}$ . The core DAL loop then iteratively trains a deep model on the current labeled set  $\mathcal{L}_r$ , scores the unlabeled pool  $\mathcal{U}_r$ , and applies Query Strategies to select a batch  $Q$  for oracle labeling and subsequent set updates, repeating until a stopping criterion is satisfied. Finally, the model is evaluated on  $\mathcal{T}$  to report predictive performance and produce learning curves and runtime summaries aligned with the research questions. Each component is described in detail in the subsequent sections.

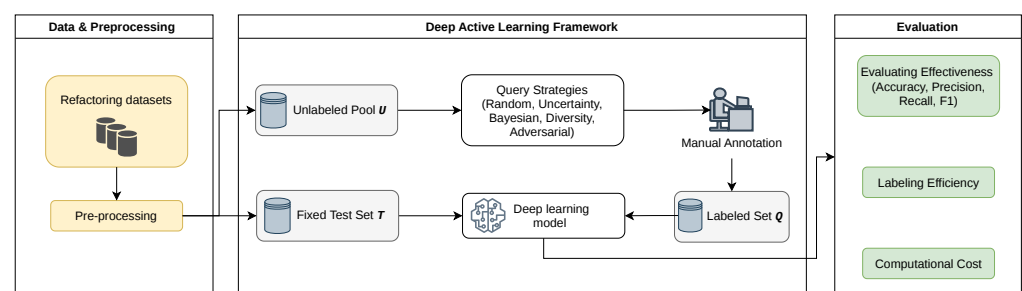


Figure 4. DAL methodology.

### 4.3. Refactoring Datasets

We use the refactoring datasets by Aniche et al. [5], spanning three granularity levels: class-level, method-level, and variable-level. Each dataset corresponds to a refactoring operation (e.g., Extract Class (EC), Extract Method (EM), or Rename Variable (RV)) and includes refactored and non-refactored instances. Table 1 reports the dataset sizes.

The experiments were conducted on multiple refactoring datasets mined from real-world software systems. Each dataset corresponds to a specific refactoring operation (e.g., Extract Class or Move Method) and follows established refactoring detection and dataset construction practices reported in prior work [5]. Each instance represents a software entity described using a set of static software metrics, including structural, complexity, coupling, and cohesion measures. Class-level refactorings include a total of 46 features, with 37 source code metrics (e.g., coupling between objects (CBO), weighted methods per class (WMC), number of methods, and lines of code), 5 process-related metrics (e.g., number of commits, bug fixes, and previous refactoring operations), and 4 ownership-based metrics (e.g., author ownership and number of major/minor contributors). Method-level refactorings incorporate an additional 20 method-specific source code metrics, including number of parameters, loops, assignments, and mathematical operations, resulting in a total of 57 features. Variable-level refactorings introduce one additional metric that captures the number of times a variable is used, leading to a final feature count of 58. Prior empirical studies have shown that software engineering datasets are often heavily skewed, with minority events occurring infrequently; our refactoring datasets follow the same pattern, exhibiting severe class imbalance [38,39].

The data was partitioned into an initial labeled set  $\mathcal{L}_0$ , a large unlabeled pool  $\mathcal{U}$ , and a fixed test set  $\mathcal{T}$ . Table 1 summarizes the original dataset sizes as reported by Aniche et al. The first two numeric columns (Refactored and Non-refactored) reflect the true, highly imbalanced distributions in the data.

**Table 1.** Aniche et al. datasets [5].

| Refactoring Operation                  | Refactored Instances | Non-Refactored Instances |
|----------------------------------------|----------------------|--------------------------|
| Class-Level                            |                      |                          |
| Extract Class (EC)                     | 41,191               | 10,692                   |
| Extract Interface (EI)                 | 10,495               | 10,692                   |
| Extract Subclass (ESC)                 | 6436                 | 10,692                   |
| Extract Superclass (ESP)               | 26,814               | 10,692                   |
| Move and Rename Class (MARC)           | 654                  | 10,692                   |
| Move Class (MC)                        | 49,815               | 10,692                   |
| Rename Class (RC)                      | 3991                 | 10,692                   |
| Total (Class-Level)                    | 139,396              | 10,692                   |
| Method-Level                           |                      |                          |
| Extract and Move Method (EMM)          | 9723                 | 293,467                  |
| Extract Method (EM)                    | 327,493              | 293,467                  |
| Inline Method (IM)                     | 53,827               | 293,467                  |
| Move Method (MM)                       | 163,078              | 293,467                  |
| Pull Up Method (PUM)                   | 155,076              | 293,467                  |
| Push Down Method (PDM)                 | 62,630               | 293,467                  |
| Rename Method (RM)                     | 427,935              | 293,467                  |
| Total (Method-Level)                   | 1,199,762            | 293,467                  |
| Variable-Level                         |                      |                          |
| Extract Variable (EV)                  | 6709                 | 702,494                  |
| Inline Variable (IV)                   | 30,894               | 702,494                  |
| Parameterize Variable (PV)             | 22,537               | 702,494                  |
| Rename Parameter (RP)                  | 33,675               | 702,494                  |
| Rename Variable (RV)                   | 324,955              | 702,494                  |
| Replace Variable With Attribute (RVWA) | 25,894               | 702,494                  |
| Total (Variable-Level)                 | 444,664              | 702,494                  |
| <b>Overall Total</b>                   | <b>1,783,822</b>     | <b>1,006,653</b>         |

#### 4.4. Data Preprocessing

For each refactoring dataset, we standardized all numeric features using z-score normalization. We then performed a stratified train–test split with 80% of the data used as the training pool  $\mathcal{P}$  and the remaining 20% as the test set  $\mathcal{T}$ , using a fixed random seed for reproducibility. The test set was held out and never queried during DAL. Within  $\mathcal{P}$ , labels were initially hidden except for a small seed set  $\mathcal{L}_0$ , and DAL progressively revealed additional labels subject to a labeling budget.

To mitigate the extreme class imbalance during training, we applied random under-sampling on the training pool only, after the train–test split. For each refactoring dataset, we retained all minority (refactored) instances in  $\mathcal{P}$  and uniformly sampled the same number of majority (non-refactored) instances from  $\mathcal{P}$  without replacement, thereby constructing a balanced training subset for model learning.

Random under-sampling was applied only to the training set after the train/test split, whereas the test set was preserved in its original imbalanced distribution. This decision was motivated by two considerations. First, it reduced majority-class dominance during training and improves learning stability. Second, the original refactoring datasets were very large, and running multi-round DAL experiments on the PL training pools was computationally inapplicable given our available hardware resources. Therefore, under-sampling was necessary to make the experiments feasible. This design is also broadly consistent with prior work on refactoring prediction datasets, such as that of Aniche et al., who also applied under-sampling, although before the train/test split. In our case, performing under-sampling only on the training data preserves a more realistic evaluation setting. All DAL strategies operated on this (under-sampled) training pool, while performance was consistently reported on the unchanged test set. It is worth noting that because random under-sampling was applied after the train/test split, the reported DAL behavior and labeling-efficiency results should be interpreted as specific to an under-sampled training setting and may differ with the original class imbalance.

Following data preprocessing, the prepared training pool was used as input to the DAL framework. The next section presents the proposed DAL framework, including the prediction model and the adopted query strategies.

#### 4.5. Deep Learning Model

Deep learning has shown strong effectiveness in software engineering prediction tasks, particularly when learning complex feature interactions [40]. We employed a deep neural network as the base learner due to its ability to capture non-linear relationships among software metrics. We utilized a feed-forward neural network. The model consists of two fully connected hidden layers of width 128 with ReLU activations and dropout regularization ( $p = 0.5$ ), followed by a two-unit output layer with softmax activation. The representation produced by the second hidden layer was further used as a latent embedding for embedding-based query strategies. The feed-forward neural network was selected as the common base learner to provide a unified evaluation setting for all DAL strategies. The goal of the study is to compare query strategies that have the same predictive backbone, rather than to optimize over multiple model families. Evaluating DAL against strong tree-based and ensemble baselines remains an important direction for future work.

Given an input  $x$ , the network produces class probabilities  $p_\theta(y = c \mid x)$  for  $c \in \{0, 1\}$  parameterized by  $\theta$ . Model parameters are optimized using the cross-entropy loss [41]

$$\mathcal{L}_{\text{CE}}(\theta) = -\frac{1}{|\mathcal{L}|} \sum_{(x,y) \in \mathcal{L}} \sum_{c \in \{0,1\}} \mathbb{I}[y = c] \log p_\theta(y = c \mid x), \quad (1)$$

with the Adam optimizer, learning rate  $10^{-3}$ , batch size 64, and 15 epochs per DAL round.

To ensure fair comparison, the same architecture and hyperparameters were used for all query strategies. Dropout was retained at inference time for Bayesian DAL strategies based on Monte Carlo (MC) sampling [28].

#### 4.6. Deep Active Learning Framework

We adopted a pool-based DAL framework, where the deep model iteratively selects the most informative unlabeled instances for expert annotation [28,30]. At each DAL round, the model was trained on the current labeled set, used to evaluate the unlabeled pool, and guided by an acquisition function to select a batch of informative instances. The newly labeled instances were added to the training set, and the process repeated until the labeling budget was exhausted.

Algorithm 1 summarizes the proposed DAL process for refactoring opportunity prediction. We initialized a small labeled subset  $\mathcal{L}_0$  by uniform random sampling from the training pool  $\mathcal{P}$ . At each round, we trained the model on  $\mathcal{L}$ , computed an acquisition score for each  $x \in \mathcal{U}$ , selected a batch of instances with highest scores, queried their labels, and added them to  $\mathcal{L}$ .

---

#### Algorithm 1 Deep Active Learning for refactoring prediction

---

**Require:** Training pool  $\mathcal{P}$ , test set  $\mathcal{T}$ , initial fraction  $p_{\text{init}}$ , query fraction  $p_q$ , label cap  $p_{\text{max}}$ , max rounds  $R_{\text{max}}$ , acquisition function  $a(\cdot)$

- 1: Initialize  $\mathcal{L} \subset \mathcal{P}$  by sampling  $|\mathcal{L}| = \max(20, \lfloor p_{\text{init}} |\mathcal{P}| \rfloor)$  uniformly at random
- 2:  $\mathcal{U} \leftarrow \mathcal{P} \setminus \mathcal{L}$
- 3: **for**  $r = 0$  to  $R_{\text{max}}$  **do**
- 4:   Train deep model  $f_\theta$  on  $\mathcal{L}$  (fixed epochs and hyperparameters)
- 5:   Evaluate  $f_\theta$  on  $\mathcal{T}$  and compute metrics
- 6:   **if**  $\frac{|\mathcal{L}|}{|\mathcal{P}|} \geq p_{\text{max}}$  **then**
- 7:     **break** {label budget reached (e.g., 50% of  $\mathcal{P}$ )}
- 8:   **end if**
- 9:    $B \leftarrow \max(20, \lfloor p_q |\mathcal{P}| \rfloor)$
- 10:   Compute  $a(x)$  for all  $x \in \mathcal{U}$
- 11:   Select  $Q \subseteq \mathcal{U}$  with  $|Q| = B$  having the highest acquisition scores
- 12:   Query oracle labels for  $Q$
- 13:    $\mathcal{L} \leftarrow \mathcal{L} \cup Q, \quad \mathcal{U} \leftarrow \mathcal{U} \setminus Q$
- 14: **end for**

---

Percent-based labeling budget.

To enable consistent comparison across datasets of different sizes, we express labeling in terms of the fraction of the training pool. We initialized with  $p_{\text{init}} = 5\%$  of  $\mathcal{P}$  labeled (with a minimum of 20 labeled instances), and at each round we queried an additional  $p_q = 2\%$  of  $\mathcal{P}$  (minimum 20 instances). The cumulative labeling budget was capped at  $p_{\text{max}} = 30\%$  of  $\mathcal{P}$ .

Training and convergence.

At each round  $r$ , the model  $f_\theta$  was trained from scratch on the current labeled set  $\mathcal{L}_r$  with fixed hyperparameters. We then evaluated  $f_\theta$  on the held-out test set  $\mathcal{T}$  and used the F1-score (Section 5.2) as the primary metric  $M_r$ . The DAL process ran for at most  $R_{\text{max}} = 30$  rounds and stopped when the fraction of labeled instances in the training pool reached a predefined label cap  $p_{\text{max}}$  (set to 50% in our experiments), i.e., when

$$\frac{|\mathcal{L}_r|}{|\mathcal{P}|} \geq p_{\text{max}}.$$

Thus, all strategies were compared under the same maximum labeling budget.

#### 4.7. Query Strategies

For each unlabeled instance  $x \in \mathcal{U}_r$ , an acquisition function  $a(x)$  quantified its informativeness. At round  $r$ , we selected the top- $B$  instances with the highest acquisition scores for labeling, where  $B$  is determined by the query fraction  $p_q$ .

Let the model produce class probabilities  $p_\theta(y = c | x)$  for  $c \in \{0, 1\}$ . We computed an acquisition score  $a(x)$  for each unlabeled instance and queried the top- $B$  instances utilizing the following groups of strategies:

Entropy sampling [25].

$$a_{\text{ES}}(x) = - \sum_{c \in \{0,1\}} p_\theta(y = c | x) \log(p_\theta(y = c | x) + \epsilon), \quad (2)$$

where  $\epsilon$  is a small constant for numerical stability.

Least-confidence sampling [42].

$$a_{\text{LC}}(x) = 1 - \max_{c \in \{0,1\}} p_\theta(y = c | x). \quad (3)$$

Margin sampling [43].

Let  $p_{(1)}(x)$  and  $p_{(2)}(x)$  denote the largest and second-largest class probabilities.

$$a_{\text{MS}}(x) = 1 - (p_{(1)}(x) - p_{(2)}(x)). \quad (4)$$

MC-dropout uncertainty (entropy with dropout) [28].

We performed  $K$  stochastic forward passes (dropout enabled) to obtain  $p_\theta^{(k)}(y = c | x)$  and used the mean predictive distribution  $\bar{p}(y = c | x) = \frac{1}{K} \sum_{k=1}^K p_\theta^{(k)}(y = c | x)$ . Entropy was then computed using  $\bar{p}$  as in Equation (2). We also considered entropy, least-confidence, and margin sampling computed on  $\bar{p}(y | x)$ .

BALD (Bayesian Active Learning by Disagreement) [28].

BALD scores instances by the mutual information between predictions and model parameters:

$$a_{\text{BALD}}(x) = H[\bar{p}(y | x)] - \frac{1}{K} \sum_{k=1}^K H[p_\theta^{(k)}(y | x)], \quad (5)$$

where  $H[\cdot]$  is entropy.

Cluster-based selection [30].

We also evaluated cluster-based strategies that operate on learned embeddings from the penultimate layer. KMeans selects points closest to cluster centroids that approximate core-set selection by maximizing the minimum distance to the labeled set.

Adversarial acquisition [44].

We included adversarial query strategies (BIM, DeepFool) that prioritize instances requiring minimal perturbations to change the predicted class, thus selecting points near the decision boundary.

Random sampling.

Random sampling serves as a baseline for assessing the added value of DAL strategies.

#### 4.8. Implementation Details

This methodology integrates deep learning with active learning to predict refactoring opportunities under limited labeling budgets. By selecting informative instances using

uncertainty-, diversity-, and adversarial-based acquisition functions, the proposed DAL framework aims to achieve strong predictive performance while reducing annotation effort.

Our experiments were conducted using the DeepAL framework for DAL in Python [45], available as an open-source library. (<https://github.com/ej0cl6/deep-active-learning>) (accessed on 26 March 2026). DeepAL provides a generic implementation of pool-based active learning with a variety of query strategies, implemented in Python 3.9.0 on top of PyTorch 2.7.1 (for model definition and training) and scikit-learn (for preprocessing and clustering), with additional support from NumPy and pandas for data handling, tqdm for progress reporting, and matplotlib for visualization. In this work, we adapted DeepAL to the refactoring prediction setting. Specifically, we replaced image-based models with a feed-forward neural network for tabular software metrics. In the DAL setup, 5% of the training pool was initially labeled, an additional 5% was queried in each round, and training stopped when performance plateaued or the labeling budget reached 50% of the training pool.

We used DeepAL's existing implementations of uncertainty-based, Bayesian (MC-dropout), diversity-based (KMeans), and adversarial (BIM, DeepFool) strategies while modifying the experimental driver to (i) express the initial labeled size and per-round query size as fixed percentages of the training pool with minimum counts, (ii) enforce a maximum labeling budget of 30% of the training pool, (iii) add a convergence-based early stopping criterion based on F1-score improvement (minimum improvement threshold and patience), and (iv) log round-level metrics and runtime to CSV files for all datasets and strategies. These adaptations allow us to apply DeepAL directly to large, imbalanced refactoring datasets and to generate the learning curves and comparative analyses reported in this paper.

## 5. Empirical Evaluation

In this section, we describe in detail our empirical evaluation. The objective of the empirical study and the research questions are clearly stated. We then describe the performance metrics used to validate the proposed methodology. Finally, the replication package is presented.

### 5.1. Research Questions

This study investigates the effectiveness and efficiency of DAL for refactoring opportunity prediction under limited labeling budgets. Specifically, we address the following research questions:

RQ1: Effectiveness.

How effective are different DAL query strategies for refactoring prediction compared with Passive Learning (PL)?

This research question evaluates the predictive effectiveness of different DAL query strategies for refactoring prediction by comparing them against a PL model trained on all labeled instances.

RQ2: Labeling Efficiency.

To what extent can DAL reduce the labeling effort required to achieve competitive performance?

Here, we analyze learning curves as a function of the labeled fraction, assessing how quickly DAL strategies approach their peak performance relative to the PL labeled dataset.

RQ3: Computational Efficiency.

What is the computational cost of query strategies, and how do they compare in terms of runtime efficiency under equivalent labeling budgets?

This research question analyzes per-round runtime and overall execution cost to assess the practical feasibility of the evaluated strategies.

### 5.2. Evaluation Metrics

We assess the effectiveness and efficiency of DAL strategies on the fixed test set after each DAL round. We report standard classification metrics, including precision, recall, and F1-score. Let TP, FP, TN, and FN denote the confusion matrix entries. We computed:

Accuracy.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}. \quad (6)$$

Precision and recall.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}, \quad \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}. \quad (7)$$

F1-score.

$$\text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}. \quad (8)$$

Given the severe class imbalance in the refactoring datasets (Table 1), we use F1 as the primary metric, both for reporting and for convergence-based stopping, and report accuracy, precision, and recall for completeness.

Labeling efficiency.

To study labeling efficiency and strategy comparison, we analyze learning curves  $\{(\ell_r, \text{F1}_r)\}_r$ , where

$$\ell_r = \frac{|\mathcal{L}_r|}{|\mathcal{P}|} \quad (9)$$

is the labeled fraction at round  $r$ .

Computational efficiency.

For computational efficiency, we recorded per-round runtime (seconds) including training, query, and evaluation, and compared strategies under equivalent labeling budgets.

To complement the runtime analysis in RQ3, we computed this metric at 50% labeled because it represents a realistic mid-budget setting where strategies have had enough iterations to learn but labeling is still limited. For each dataset and strategy, we define

$$\Delta \text{F1} = \text{F1}_{\text{strategy}} - \text{F1}_{\text{PL}}, \quad \Delta T = T_{\text{strategy}} - T_{\text{PL}},$$

where F1 is the per-round F1-score and  $T$  is the per-round runtime including training/evaluation and query selection.

### 5.3. Replication Package

To support transparency and reproducibility, we provide a publicly available replication package. The package includes (i) the refactoring datasets used in this study and (ii) the complete implementation of the DAL framework and query strategies, along with configurations specifying all hyperparameters and labeling budgets. All experiments can be executed using a single entry-point script and produce round-level results, including performance metrics, labeling progress, and runtime information. The package is available online (<https://github.com/amalazba/Deep-Active-Learning-for-Label-Efficient-Refactoring-Prediction>) (accessed on 26 March 2026).

## 6. Results and Discussion

This section presents the empirical results of applying DAL to refactoring opportunity prediction under constrained labeling budgets. We discuss predictive effectiveness (RQ1), labeling efficiency (RQ2), and computational efficiency (RQ3). Throughout the analysis, we compare strategies under various labeling budgets. Given the strong class imbalance typical in the refactoring prediction, we emphasize the F1-score as the primary metric of predictive effectiveness.

### 6.1. Predictive Effectiveness

As shown in Tables 2–4, the effectiveness of DAL strategies was evaluated at labeling budgets of 10%, 25%, and 50% relative to the PL baseline. We evaluated labeling budgets from 5% to 50% of the training data; however, the 50% budget is included only as an upper-bound reference point to illustrate the progression toward the PL baseline. The reported values are  $\Delta F1$  with respect to the PL model, such that values closer to zero indicate better recovery of baseline performance, while positive values indicate that the DAL strategy outperformed the PL model. Across the three refactoring granularities (class-, method-, and variable-level), the tables reveal a consistent budget-dependent trend: the gap between DAL-based models and the PL baseline decreases as the labeling budget increases. In Table 2, most entries are negative, indicating expected degradation under an insufficient labeling budget; in Table 3, many strategies move close to zero, and several become positive; and by Table 4, many types of refactoring match or exceed the PL baseline performance.

**Table 2.** RQ1 (Effectiveness): F1-scores across refactoring types @10 labeling budgets.

| Type           | PL(F1) | RS     | LC            | MS     | ES            | LCD           | MSD           | ESD           | BALD   | KM     | BIM    | DF            |
|----------------|--------|--------|---------------|--------|---------------|---------------|---------------|---------------|--------|--------|--------|---------------|
| Class-level    |        |        |               |        |               |               |               |               |        |        |        |               |
| EC             | 0.938  | −0.004 | −0.008        | −0.007 | −0.001        | −0.007        | −0.005        | −0.004        | −0.007 | −0.007 | −0.012 | −0.015        |
| EI             | 0.917  | −0.027 | −0.021        | −0.016 | −0.021        | −0.019        | −0.024        | −0.019        | −0.032 | −0.033 | −0.018 | <b>−0.015</b> |
| ESP            | 0.945  | −0.017 | <b>−0.001</b> | −0.011 | −0.005        | −0.006        | −0.008        | −0.010        | −0.013 | −0.008 | −0.007 | −0.006        |
| ESC            | 0.900  | −0.021 | −0.022        | −0.019 | −0.020        | <b>−0.012</b> | −0.019        | −0.014        | −0.043 | −0.041 | −0.016 | −0.018        |
| MARC           | 0.556  | −0.137 | +0.017        | −0.021 | +0.028        | −0.178        | <b>+0.076</b> | −0.056        | −0.139 | −0.037 | −0.039 | −0.053        |
| MC             | 0.982  | −0.010 | <b>−0.006</b> | −0.011 | −0.009        | −0.009        | −0.010        | −0.007        | −0.011 | −0.016 | −0.009 | −0.009        |
| RC             | 0.891  | −0.069 | −0.051        | −0.060 | −0.046        | <b>−0.042</b> | −0.047        | −0.043        | −0.077 | −0.073 | −0.044 | −0.048        |
| Method-level   |        |        |               |        |               |               |               |               |        |        |        |               |
| EM             | 0.891  | −0.027 | −0.016        | −0.013 | −0.014        | −0.015        | −0.018        | <b>−0.013</b> | −0.028 | −0.019 | −0.017 | −0.017        |
| EMM            | 0.216  | −0.059 | −0.047        | −0.049 | −0.037        | −0.042        | −0.055        | <b>−0.033</b> | −0.060 | −0.046 | −0.040 | −0.049        |
| IM             | 0.606  | −0.050 | −0.046        | −0.059 | −0.034        | −0.062        | <b>−0.031</b> | −0.045        | −0.089 | −0.066 | −0.064 | −0.057        |
| MM             | 0.788  | −0.047 | −0.033        | −0.033 | −0.034        | −0.036        | −0.034        | <b>−0.032</b> | −0.057 | −0.048 | −0.040 | −0.037        |
| PDM            | 0.715  | −0.069 | −0.059        | −0.064 | −0.039        | −0.051        | −0.054        | <b>−0.038</b> | −0.095 | −0.068 | −0.069 | −0.051        |
| PUM            | 0.840  | −0.037 | −0.025        | −0.028 | −0.028        | <b>−0.023</b> | −0.027        | −0.028        | −0.051 | −0.043 | −0.040 | −0.026        |
| RM             | 0.881  | −0.021 | −0.012        | −0.020 | −0.016        | −0.017        | −0.017        | <b>−0.011</b> | −0.024 | −0.018 | −0.015 | −0.019        |
| Variable-level |        |        |               |        |               |               |               |               |        |        |        |               |
| EV             | 0.090  | −0.014 | −0.015        | −0.007 | <b>+0.002</b> | −0.016        | −0.008        | −0.005        | −0.018 | −0.015 | −0.007 | −0.010        |
| IV             | 0.297  | −0.058 | −0.031        | −0.023 | −0.040        | <b>−0.004</b> | −0.021        | −0.026        | −0.041 | −0.036 | −0.027 | −0.035        |
| PV             | 0.246  | −0.064 | −0.048        | −0.054 | <b>−0.023</b> | −0.030        | −0.038        | −0.030        | −0.077 | −0.049 | −0.034 | −0.047        |
| RP             | 0.823  | −0.024 | −0.018        | −0.016 | −0.016        | <b>−0.013</b> | −0.016        | −0.016        | −0.033 | −0.027 | −0.020 | −0.017        |
| RV             | 0.812  | −0.038 | −0.030        | −0.026 | −0.023        | −0.022        | <b>−0.019</b> | −0.020        | −0.045 | −0.034 | −0.031 | −0.025        |
| RVWA           | 0.295  | −0.055 | −0.026        | −0.026 | −0.024        | −0.039        | <b>−0.021</b> | −0.031        | −0.071 | −0.045 | −0.021 | −0.029        |

**Table 3.** RQ1 (Effectiveness): F1-scores across refactoring types @25 labeling budgets.

| Type           | PL(F1) | RS     | LC            | MS     | ES            | LCD           | MSD           | ESD           | BALD   | KM     | BIM           | DF            |
|----------------|--------|--------|---------------|--------|---------------|---------------|---------------|---------------|--------|--------|---------------|---------------|
| Class-level    |        |        |               |        |               |               |               |               |        |        |               |               |
| EC             | 0.938  | −0.001 | −0.001        | −0.000 | −0.001        | +0.002        | −0.002        | −0.006        | −0.005 | −0.003 | <b>+0.002</b> | −0.001        |
| EI             | 0.917  | −0.010 | −0.000        | −0.005 | −0.003        | +0.001        | −0.007        | −0.001        | −0.036 | −0.017 | −0.003        | −0.004        |
| ESP            | 0.945  | −0.005 | −0.006        | −0.004 | −0.002        | <b>−0.000</b> | −0.003        | −0.002        | −0.012 | −0.007 | −0.004        | −0.000        |
| ESC            | 0.900  | −0.011 | −0.004        | −0.003 | −0.006        | −0.004        | −0.004        | <b>−0.001</b> | −0.037 | −0.011 | −0.008        | −0.002        |
| MARC           | 0.556  | −0.108 | −0.048        | −0.068 | −0.051        | <b>−0.019</b> | −0.063        | −0.064        | −0.058 | −0.089 | −0.061        | −0.039        |
| MC             | 0.982  | −0.008 | −0.002        | −0.002 | −0.003        | −0.003        | −0.004        | <b>−0.001</b> | −0.013 | −0.010 | −0.006        | −0.005        |
| RC             | 0.891  | −0.039 | −0.008        | −0.003 | −0.013        | −0.010        | −0.011        | <b>−0.003</b> | −0.070 | −0.029 | −0.009        | −0.021        |
| Method-level   |        |        |               |        |               |               |               |               |        |        |               |               |
| EM             | 0.891  | −0.016 | −0.003        | −0.000 | −0.001        | −0.001        | <b>+0.001</b> | −0.003        | −0.024 | −0.014 | −0.007        | −0.007        |
| EMM            | 0.216  | −0.025 | <b>−0.009</b> | −0.013 | −0.029        | −0.023        | −0.033        | −0.022        | −0.055 | −0.052 | −0.030        | −0.025        |
| IM             | 0.606  | −0.033 | −0.032        | −0.027 | −0.025        | <b>−0.004</b> | −0.016        | −0.010        | −0.046 | −0.023 | −0.037        | −0.025        |
| MM             | 0.788  | −0.034 | −0.007        | −0.011 | −0.015        | −0.007        | −0.010        | <b>−0.006</b> | −0.049 | −0.026 | −0.026        | −0.010        |
| PDM            | 0.715  | −0.052 | −0.014        | −0.012 | −0.015        | <b>−0.003</b> | −0.014        | −0.015        | −0.095 | −0.044 | −0.036        | −0.015        |
| PUM            | 0.840  | −0.026 | +0.001        | −0.003 | −0.001        | +0.002        | <b>+0.003</b> | −0.001        | −0.046 | −0.023 | −0.015        | −0.005        |
| RM             | 0.881  | −0.014 | +0.001        | −0.001 | <b>+0.002</b> | −0.002        | −0.002        | −0.004        | −0.021 | −0.019 | −0.007        | −0.004        |
| Variable-level |        |        |               |        |               |               |               |               |        |        |               |               |
| EV             | 0.090  | −0.016 | −0.010        | −0.002 | +0.001        | <b>+0.012</b> | −0.007        | +0.005        | −0.011 | −0.014 | −0.002        | +0.011        |
| IV             | 0.297  | −0.006 | −0.003        | −0.034 | +0.001        | +0.007        | +0.006        | −0.009        | −0.054 | −0.015 | −0.000        | <b>+0.031</b> |
| PV             | 0.246  | −0.037 | <b>−0.005</b> | −0.031 | −0.031        | −0.031        | −0.024        | −0.020        | −0.063 | −0.039 | −0.026        | −0.018        |
| RP             | 0.823  | −0.018 | +0.001        | +0.002 | −0.000        | <b>+0.003</b> | +0.002        | +0.001        | −0.032 | −0.016 | −0.008        | −0.005        |
| RV             | 0.812  | −0.021 | +0.001        | −0.001 | +0.001        | +0.003        | +0.002        | <b>+0.003</b> | −0.039 | −0.019 | −0.013        | −0.016        |
| RVWA           | 0.295  | −0.032 | +0.010        | +0.026 | −0.002        | −0.008        | <b>+0.036</b> | +0.006        | −0.021 | +0.006 | −0.034        | +0.017        |

**Table 4.** RQ1 (Effectiveness): F1-scores across refactoring types @50 labeling budgets.

| Type           | PL(F1) | RS            | LC            | MS            | ES            | LCD           | MSD           | ESD           | BALD   | KM     | BIM           | DF            |
|----------------|--------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|--------|--------|---------------|---------------|
| Class-level    |        |               |               |               |               |               |               |               |        |        |               |               |
| EC             | 0.938  | +0.002        | −0.002        | −0.001        | −0.000        | +0.001        | −0.001        | −0.003        | +0.001 | −0.002 | +0.001        | +0.001        |
| EI             | 0.917  | −0.005        | −0.000        | −0.002        | −0.005        | −0.001        | −0.004        | −0.003        | −0.022 | −0.009 | −0.003        | <b>+0.002</b> |
| ESP            | 0.945  | −0.007        | −0.004        | <b>+0.001</b> | −0.001        | −0.001        | −0.003        | −0.000        | −0.005 | −0.002 | −0.003        | −0.006        |
| ESC            | 0.900  | −0.005        | −0.003        | <b>+0.001</b> | −0.004        | −0.003        | −0.002        | −0.004        | −0.018 | −0.002 | −0.006        | −0.005        |
| MARC           | 0.556  | −0.016        | −0.014        | −0.004        | <b>+0.010</b> | −0.021        | −0.013        | −0.024        | −0.068 | −0.075 | +0.002        | +0.007        |
| MC             | 0.982  | −0.006        | −0.004        | −0.002        | −0.001        | −0.001        | −0.003        | −0.002        | −0.008 | −0.003 | −0.002        | <b>−0.001</b> |
| RC             | 0.891  | <b>−0.001</b> | −0.002        | −0.004        | −0.003        | −0.012        | −0.006        | −0.003        | −0.043 | −0.005 | −0.009        | <b>−0.001</b> |
| Method-level   |        |               |               |               |               |               |               |               |        |        |               |               |
| EM             | 0.891  | −0.009        | −0.001        | +0.001        | <b>+0.002</b> | −0.000        | +0.000        | −0.001        | −0.020 | −0.006 | +0.000        | −0.000        |
| EMM            | 0.216  | −0.025        | −0.020        | −0.022        | −0.008        | −0.022        | −0.017        | −0.016        | −0.033 | −0.024 | −0.009        | <b>+0.005</b> |
| IM             | 0.606  | −0.025        | −0.005        | +0.004        | <b>+0.014</b> | −0.017        | −0.025        | −0.008        | −0.046 | −0.025 | −0.019        | +0.003        |
| MM             | 0.788  | −0.019        | −0.002        | +0.002        | <b>+0.003</b> | −0.001        | +0.000        | +0.002        | −0.044 | −0.011 | −0.005        | −0.003        |
| PDM            | 0.715  | −0.018        | −0.017        | <b>+0.006</b> | +0.003        | −0.008        | +0.003        | +0.001        | −0.102 | −0.028 | −0.019        | −0.017        |
| PUM            | 0.840  | −0.015        | +0.004        | +0.007        | +0.003        | +0.006        | +0.007        | <b>+0.008</b> | −0.041 | −0.007 | −0.004        | +0.001        |
| RM             | 0.881  | −0.005        | +0.004        | +0.002        | −0.000        | <b>+0.004</b> | +0.002        | +0.003        | −0.022 | −0.007 | −0.001        | +0.003        |
| Variable-level |        |               |               |               |               |               |               |               |        |        |               |               |
| EV             | 0.090  | −0.004        | −0.004        | −0.003        | +0.005        | +0.004        | <b>+0.008</b> | −0.002        | −0.010 | −0.006 | +0.001        | −0.007        |
| IV             | 0.297  | −0.005        | +0.014        | −0.010        | +0.006        | +0.012        | +0.014        | <b>+0.014</b> | −0.038 | +0.009 | +0.006        | −0.001        |
| PV             | 0.246  | −0.031        | −0.026        | −0.021        | −0.011        | −0.009        | −0.010        | −0.031        | −0.017 | −0.015 | <b>−0.006</b> | −0.013        |
| RP             | 0.823  | −0.007        | <b>+0.003</b> | +0.001        | +0.002        | +0.003        | −0.001        | +0.002        | −0.029 | −0.004 | −0.003        | +0.000        |
| RV             | 0.812  | −0.009        | +0.004        | +0.004        | <b>+0.006</b> | +0.003        | +0.003        | +0.004        | −0.032 | −0.009 | −0.009        | +0.000        |
| RVWA           | 0.295  | +0.016        | +0.005        | +0.017        | <b>+0.036</b> | +0.001        | −0.007        | +0.011        | −0.046 | +0.004 | −0.026        | −0.003        |

At the 10% labeling budget (Table 2), the observed performance varies across refactoring types and query strategies. For class-level refactorings, several uncertainty-based and dropout-based variants are among the closest to the baseline. For instance, ES yields the smallest observed degradation for the EC (−0.001), while LC is closest to the baseline for both the ESP (−0.001) and MC (−0.006). For the RC, LCD shows the smallest observed degradation among the compared strategies (−0.042). The MARC exhibits larger numerical

variation across strategies: RS and LCD show larger degradations ( $-0.137$  and  $-0.178$ , respectively), whereas MSD and ES exceed the baseline ( $+0.076$  and  $+0.028$ , respectively). A similar trend can be observed for method-level refactorings. ESD is often among the stronger strategies, with the smallest observed degradation for the EM ( $-0.013$ ), EMM ( $-0.033$ ), MM ( $-0.032$ ), PDM ( $-0.038$ ), and RM ( $-0.011$ ). Variable-level refactorings show that ES slightly exceeds the PL baseline for the EV ( $+0.002$ ), while LCD and MSD are among the closest to the baseline for the IV, RP, RV, and RVWA. In contrast, BALD and KM are often associated with larger degradations across the variable-level rows, consistent with their numerical behavior in the class- and method-level sections of the same table.

At the 25% labeling budget (Table 3), the overall gap to the PL baseline narrows substantially. For class-level refactorings, the table shows several near-zero or positive results, indicating that DAL can closely recover baseline performance with one-quarter of the labels. The EC is slightly improved under both LCD and BIM ( $+0.002$ ), the EI has its closest observed value under LCD ( $+0.001$ ), and ES is approximately recovered under LCD and DF. ESD is among the closest to the baseline for the ESC ( $-0.001$ ), MC ( $-0.001$ ), and RC ( $-0.003$ ). Although the MARC remains challenging, its closest observed value under LCD ( $-0.019$ ) is much closer to the baseline than most values observed at the 10% budget in Table 2, indicating improved stability as more labels become available. Method-level refactorings in Table 3 also exhibit stronger recovery and reduced sensitivity to strategy choice. MSD slightly exceeds the baseline for the EM ( $+0.001$ ) and also has the closest observed value for the PUM ( $+0.003$ ), while ES is closest to the baseline for the RM ( $+0.002$ ). For the IM and PDM, LCD gives the closest recovery to the PL baseline ( $-0.004$  and  $-0.003$ , respectively), and ESD remains among the closest for the MM ( $-0.006$ ). The strongest numerical improvements at this budget appear in the variable-level section of Table 3. DF improves the IV by  $+0.031$ , LCD improves the EV by  $+0.012$  (with DF close at  $+0.011$ ), and MSD achieves a gain for the RVWA ( $+0.036$ ), with MS also showing a positive result ( $+0.026$ ).

At the 50% labeling budget (Table 4), many DAL strategies are close to or slightly exceed the PL baseline, and positive  $\Delta F1$ -score values become more common across class-, method-, and variable-level refactorings. At the class level, performance differences are generally small (often within  $\pm 0.005$ ), with several gains above baseline (e.g., EC under RS/BALD/BIM/DF, EI under DF), while ES shows the highest observed value for the MARC ( $+0.010$ ), indicating that strategy choice may still matter for harder tasks but less so for easier ones. A similar trend appears at the method level: the highest observed values are obtained by the ES for the EM ( $+0.002$ ), IM ( $+0.014$ ), and MM ( $+0.003$ ); ESD for the PUM ( $+0.008$ ); LCD for the RM ( $+0.004$ ); and MS for the PDM ( $+0.006$ ), while the EMM remains difficult but is slightly improved under DF ( $+0.005$ ). Variable-level refactorings also show multiple gains (e.g., EV with MSD  $+0.008$ , IV with LC/MSD/ESD  $+0.014$ , RV with ES  $+0.006$ , and RVWA with ES  $+0.036$ ), but the PV remains below baseline for all strategies (best observed value: BIM at  $-0.006$ ), suggesting persistent task-specific difficulty that may require improvements beyond sample selection.

Overall, the results in Tables 2–4 suggest that uncertainty-based strategies (ES, LC, and MS) and their dropout-enhanced variants (LCD, MSD, and ESD) tend to show relatively consistent numerical performance across budgets and refactoring granularities, although these trends are descriptive and were not established as statistically significant across the full acquisition process. Moreover, the results suggest that DAL can serve as a practical strategy for reducing labeling cost in refactoring prediction while maintaining predictive effectiveness, with some cases showing slight numerical improvements.

Table 5 summarizes, for each refactoring type, the DAL strategy with the highest observed performance (BestDAL) at the 10%, 25%, and 50% labeling budgets. In the table,

PL (F1) denotes the PL baseline, and F1@10/F1@25/F1@50 denote the actual F1 values achieved by BestDAL at the 10%, 25%, and 50% labeling budgets, respectively. Overall, the table shows a clear numerical trend of improvement as the labeling budget increases, with many refactoring types approaching or, in some cases, slightly exceeding the PL baseline by 25–50%. At 10%, most best observed results remain below the PL baseline, although some refactoring types already show encouraging early performance; difficult cases such as the MARC, however, remain relatively low despite improvement. By 25%, many best observed F1 values move closer to, and in some cases slightly exceed, the PL baseline, especially for several variable-level refactorings. At 50%, many class- and method-level tasks reach F1 values that are close to or slightly above the PL baseline, suggesting that half-budget labeling can often provide comparable effectiveness. Across refactoring types, uncertainty-based and dropout-enhanced strategies (ES, LCD, MSD, and ESD) appear as frequently as BestDAL, while DF is also competitive in selected cases, indicating that these strategies tend to show favorable numerical performance overall.

**Table 5.** RQ1 (Effectiveness): Grand summary of the best-performing DAL strategy (BestDAL) across refactoring types at labeling budgets of 10%, 25%, and 50% in term of F1-score.

| Refactoring Type | PL (F1) | F1@10        | BestDAL@10 | F1@25        | BestDAL@25 | F1@50        | BestDAL@50 |
|------------------|---------|--------------|------------|--------------|------------|--------------|------------|
| Class-level      |         |              |            |              |            |              |            |
| EC               | 0.938   | 0.937        | ES         | 0.940        | BIM        | 0.940        | RS         |
| EI               | 0.917   | 0.902        | DF         | 0.918        | LCD        | <b>0.919</b> | DF         |
| ES               | 0.945   | 0.944        | LC         | 0.945        | LCD        | <b>0.946</b> | MS         |
| ESC              | 0.900   | 0.888        | LCD        | 0.899        | ESD        | <b>0.901</b> | MS         |
| MARC             | 0.556   | <b>0.632</b> | MSD        | 0.537        | LCD        | 0.566        | ES         |
| MC               | 0.982   | 0.977        | LC         | <b>0.982</b> | ESD        | 0.981        | DF         |
| RC               | 0.891   | 0.850        | LCD        | 0.888        | ESD        | <b>0.890</b> | DF         |
| Method-level     |         |              |            |              |            |              |            |
| EM               | 0.891   | 0.878        | ESD        | 0.892        | MSD        | <b>0.892</b> | ES         |
| EMM              | 0.216   | 0.183        | ESD        | 0.207        | LC         | <b>0.221</b> | DF         |
| IM               | 0.606   | 0.575        | MSD        | 0.602        | LCD        | <b>0.620</b> | ES         |
| MM               | 0.788   | 0.756        | ESD        | 0.782        | ESD        | <b>0.792</b> | ES         |
| PDM              | 0.715   | 0.676        | ESD        | 0.712        | LCD        | <b>0.721</b> | MS         |
| PUM              | 0.840   | 0.817        | LCD        | 0.843        | MSD        | <b>0.849</b> | ESD        |
| RM               | 0.881   | 0.871        | ESD        | 0.883        | ES         | <b>0.886</b> | LCD        |
| Variable-level   |         |              |            |              |            |              |            |
| EV               | 0.090   | 0.092        | ES         | <b>0.102</b> | LCD        | 0.098        | MSD        |
| IV               | 0.297   | 0.293        | LCD        | <b>0.328</b> | DF         | 0.311        | ESD        |
| PV               | 0.246   | 0.223        | ES         | <b>0.241</b> | LC         | 0.240        | BIM        |
| RP               | 0.823   | 0.810        | LCD        | 0.827        | LCD        | <b>0.827</b> | LC         |
| RV               | 0.812   | 0.793        | MSD        | 0.816        | ESD        | <b>0.819</b> | ES         |
| RVWA             | 0.295   | 0.274        | MSD        | 0.331        | MSD        | 0.331        | ES         |

We performed a ScottKnott ESD [46,47] analysis to statistically compare the performance of the evaluated DAL query strategies, and the results assigned all strategies to the same rank (1.0). This indicates that at the labeling budgets considered, no statistically significant performance differences were detected among the compared approaches. We note, however, that this analysis compares strategies at specific operating points rather than across the full acquisition trajectory. From the perspective of this study, the ScottKnott ESD outcome is nonetheless favorable as it suggests that the use of DAL can reduce the labeling effort without causing a statistically detectable degradation in predictive performance at the evaluated budgets. Accordingly, differences observed among strategies in Tables 2–4 should be interpreted as descriptive numerical trends rather than as evidence of statistically significant superiority of any particular strategy across all budgets.

## 6.2. Labeling Efficiency

Table 6 reports the earliest labeling budget at which any DAL strategy reaches at least 99% of the PL F1-score for each refactoring type. This criterion is defined using the F1-score and provides a practically meaningful target corresponding to near-PL predictive effectiveness. Overall, the results show substantial reductions in labeling effort across all three refactoring granularities. For class-level refactorings, the 99% threshold is reached with very small budgets in most cases, including the EC (5%, ESD), ES (5%, KM), MARC (10%, MSD), MC (10%, LC), and EI/ESC (15%, MS/MSD), corresponding to labeling-effort reductions of 85–95%; only the RC requires a somewhat larger budget (20%, MSD). Method-level refactorings are generally more demanding, but DAL still reaches the 99% threshold with 15% labeled data for the EM and RM (ESD/LCD), 20% for PUM (LCD), and 25% for the IM, MM, and PDM (LCD/ESD/LCD), while the EMM remains the most difficult case, requiring 50% labeled data (DF). Variable-level refactorings show more mixed behavior, with the EV, IV, RP, and RVWA reaching the threshold with only 10–15% labeled data, RV at 20%, and PV requiring 45% (ES). In several cases (e.g., MARC, EMM, EV, IV, PV, and RVWA), the achieved F1-score at the reported labeling budget slightly exceeds the PL baseline, indicating that near-PL or slightly higher predictive effectiveness can sometimes be obtained with substantially fewer labeled instances. The most practically relevant results occur at very low labeling budgets. Across many refactoring types, DAL reaches near-PL performance using only 5–15% labeled data. These results correspond to labeling-effort reductions of 85–95%, demonstrating that DAL provides its greatest value in the low-budget regime where manual annotation is most costly.

**Table 6.** RQ2 (Labeling Efficiency): Labeling requirements to reach 99% of PL performance. We report the earliest labeling point at which any DAL query strategy matches or exceeds the PL F1-score, along with the achieved F1-score at that point.

| Refactoring Type | PL (F1-Score) | % Labeled to Reach 99% of PL (Random Sampling) | Best Strategy | % Labeled to Reach 99% of PL (Best Strategy) | Achieved F1-Score | Saved Labeling Effort |
|------------------|---------------|------------------------------------------------|---------------|----------------------------------------------|-------------------|-----------------------|
| Class-level      |               |                                                |               |                                              |                   |                       |
| EC               | 0.938         | 10                                             | ESD           | 5                                            | 0.931             | 95%                   |
| EI               | 0.917         | 30                                             | MS            | 15                                           | 0.911             | 85%                   |
| ESC              | 0.900         | 30                                             | MSD           | 15                                           | 0.897             | 85%                   |
| ES               | 0.945         | 20                                             | KM            | 5                                            | 0.938             | 95%                   |
| MARC             | 0.556         | NA                                             | MSD           | 10                                           | 0.632             | 90%                   |
| MC               | 0.982         | 20                                             | LC            | 10                                           | 0.977             | 90%                   |
| RC               | 0.891         | 50                                             | MSD           | 20                                           | 0.888             | 80%                   |
| Method-level     |               |                                                |               |                                              |                   |                       |
| EMM              | 0.216         | NA                                             | DF            | 50                                           | 0.221             | 50%                   |
| EM               | 0.891         | 40                                             | ESD           | 15                                           | 0.883             | 85%                   |
| IM               | 0.606         | 45                                             | LCD           | 25                                           | 0.602             | 75%                   |
| MM               | 0.788         | NA                                             | ESD           | 25                                           | 0.782             | 75%                   |
| PUM              | 0.840         | NA                                             | LCD           | 20                                           | 0.836             | 80%                   |
| PDM              | 0.715         | NA                                             | LCD           | 25                                           | 0.712             | 75%                   |
| RM               | 0.881         | 35                                             | LCD           | 15                                           | 0.876             | 85%                   |
| Variable-level   |               |                                                |               |                                              |                   |                       |
| EV               | 0.090         | NA                                             | ES            | 10                                           | 0.092             | 90%                   |
| IV               | 0.297         | 30                                             | MS            | 15                                           | 0.311             | 85%                   |
| PV               | 0.246         | NA                                             | ES            | 45                                           | 0.252             | 55%                   |
| RP               | 0.823         | 50                                             | ESD           | 15                                           | 0.817             | 85%                   |
| RV               | 0.812         | NA                                             | MS            | 20                                           | 0.810             | 80%                   |
| RVWA             | 0.295         | 30                                             | LC            | 15                                           | 0.321             | 85%                   |

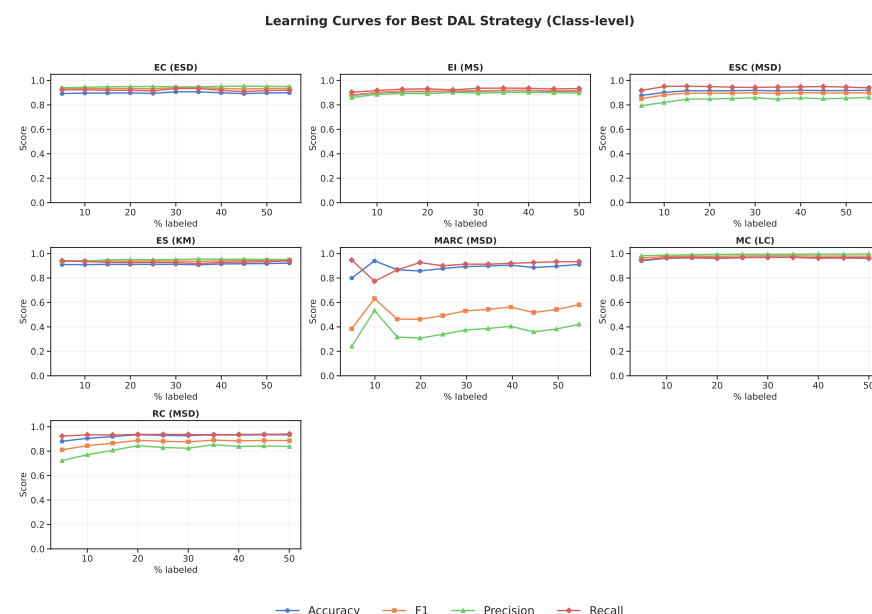
The granularity-level summary in Table 7 further confirms the label-efficiency advantage of DAL by aggregating the labeling needed to reach the target performance across refactoring levels. Class-level refactorings are the most label-efficient on average, requiring only  $11.4 \pm 5.6\%$  labeled data ( $88.6 \pm 5.6\%$  savings), followed by variable-level refactorings at  $20.0 \pm 12.6\%$  ( $80.0 \pm 12.6\%$  savings), while method-level refactorings require the largest labeling budget at  $25.0 \pm 11.9\%$  on average ( $75.0 \pm 11.9\%$  savings).

**Table 7.** RQ2 (Labeling Efficiency): Aggregate labeling requirements to reach 99% of PL performance, summarized by refactoring granularity. Values are mean  $\pm$  std across refactoring types in each level.

| Granularity    | Needed% (Mean $\pm$ Std) | Savings% (Mean $\pm$ Std) | <i>n</i> |
|----------------|--------------------------|---------------------------|----------|
| Class-level    | 11.4 $\pm$ 5.6%          | 88.6 $\pm$ 5.6%           | 7        |
| Method-level   | 25.0 $\pm$ 11.9%         | 75.0 $\pm$ 11.9%          | 7        |
| Variable-level | 20.0 $\pm$ 12.6%         | 80.0 $\pm$ 12.6%          | 6        |

Overall, the results indicate that the performance gains are primarily driven by the query strategy's ability to select 'hard' boundary cases. For instance, while the deep learning model provides a high-dimensional representation of software metrics, the 11.4% to 25.0% label efficiency reached is a direct result of the DAL selection process as the baseline required significantly more data to achieve comparable F1-scores.

We further analyzed the learning curve that complements the results in Table 6 by showing how the selected best DAL strategy evolves across the labeling process and across various metrics, including accuracy, F1, precision, and recall. As shown in Figures 5–7, the best DAL strategy generally yields learning curves that improve steadily as the labeling budget increases, although the degree of agreement among the evaluation metrics differs across datasets. In several cases, the four metrics are closely aligned, indicating stable and balanced predictive behavior; this pattern is clearly visible for datasets such as the EC, EM and MC, where accuracy, F1, precision, and recall have similar performance. A similar consistent trend can also be observed in some other datasets where the curves rise smoothly with limited separation between metrics. In contrast, other datasets exhibit noticeable variation among the metrics, suggesting a less balanced classification behavior. This is particularly evident for the MARC, EMM, EV, PV, IV and RVWA, where accuracy and recall tend to remain stronger while precision and F1 lag behind, producing wider gaps among the curves. Such divergence indicates that for these datasets, good overall classification coverage does not necessarily translate into equally strong positive-class discrimination. Overall, the figures show that the effectiveness of the best DAL strategy is dataset-dependent: some refactoring types exhibit uniformly strong and coherent metric behavior, whereas others reveal clear trade-offs between precision, recall, and F1.

**Figure 5.** Learning curves for the best-performing DAL strategy on class-level refactoring datasets. Each subplot corresponds to one dataset, and each curve shows the variation in accuracy, F1, precision, and recall across labeling percentages.

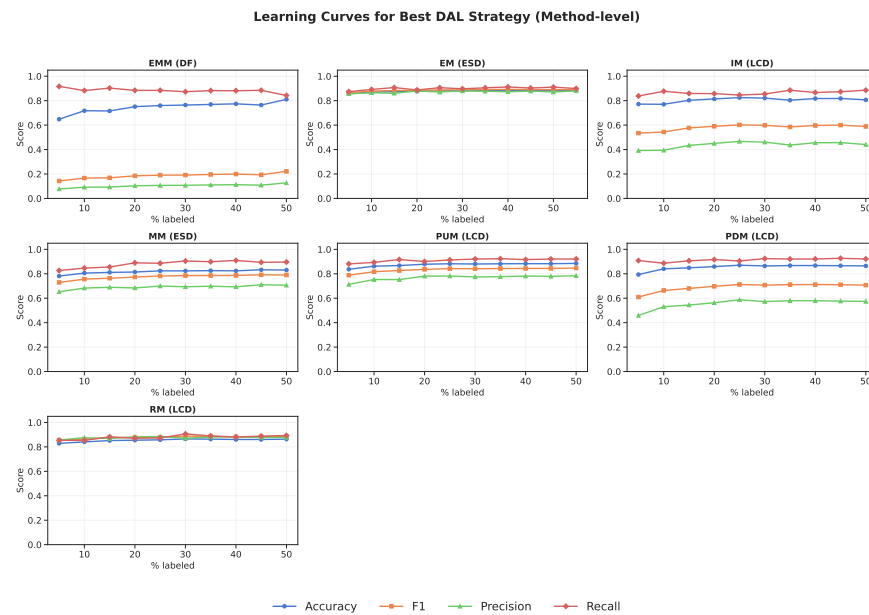


**Figure 6.** Learning curves for the best-performing DAL strategy on method-level refactoring datasets. Each subplot corresponds to one dataset, and each curve shows the variation in accuracy, F1, precision, and recall across labeling percentages.

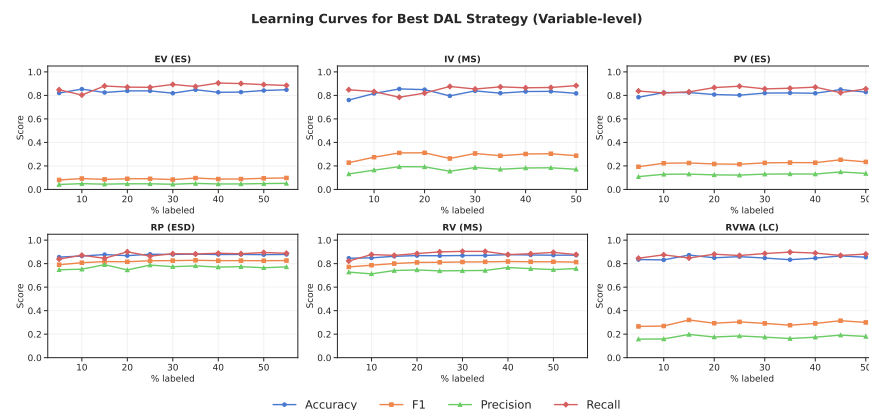


**Figure 7.** Learning curves for the best-performing DAL strategy on variable-level refactoring datasets. Each subplot corresponds to one dataset, and each curve shows the variation in accuracy, F1, precision, and recall across labeling percentages.

### 6.3. Computational Efficiency

Figure 8 shows the distribution of per-round query time for all evaluated strategies. The boxplots summarize the median, interquartile range (IQR), whiskers, and outliers for each strategy. Lower values indicate faster sample selection. The figure highlights substantial differences in query-time overhead across strategies, with uncertainty-based methods and random sampling showing low overhead, while clustering- and adversarial-based methods exhibit much larger variability and extreme outliers.

A clear pattern emerges: query-time overhead varies substantially across strategies, with differences spanning multiple orders of magnitude. Random sampling (RS) exhibits the lowest overhead and remains close to zero across rounds, as expected. Uncertainty-based strategies (LC, MS, and ES) and their dropout-enhanced variants (LCD, MSD, and ESD) also maintain comparatively low query-time distributions, with compact boxplots and relatively small spreads, indicating stable and computationally practical behavior. BALD appears more expensive than standard uncertainty sampling, but its query times remain far below the most computationally intensive methods. In contrast, clustering- and adversarial-

based strategies (particularly KM, BIM, and DF) show larger query costs, wide dispersion, and numerous extreme outliers, indicating that their runtime can become highly variable and occasionally very large depending on the dataset size. When considered together with the effectiveness results in RQ1, the boxplot supports the conclusion that LC/ES/MS and their dropout variants (LCD/MSD/ESD) offer the strongest overall trade-off between predictive performance and computational efficiency, whereas high-cost strategies may only be justified in specific cases where their accuracy gains are consistently substantial.

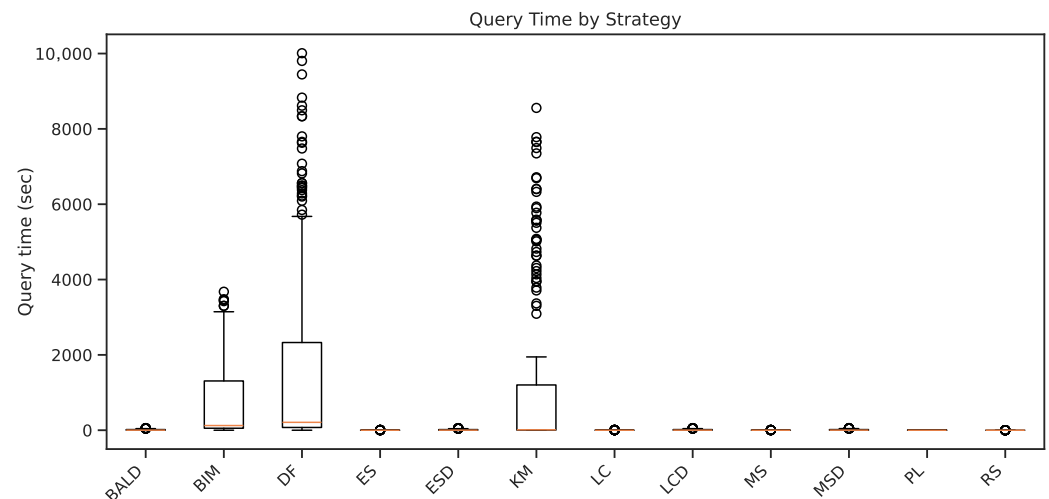


**Figure 8.** RQ3 (Computational Efficiency): Distribution of per-round query time across DAL strategies over all refactoring datasets and labeling rounds.

Figure 9 presents the mean query time of the DAL strategies across all refactoring datasets and reveals clear variability across both strategies and refactoring levels. In general, class-level datasets tend to exhibit lower and more stable query times, whereas method- and variable-level datasets show higher and more uneven costs. This pattern is consistent with the differences in dataset scale reported in Table 1, where method- and variable-level refactorings involve substantially larger instance spaces than class-level refactorings. The figure also indicates that query time is not determined by dataset size alone but also by the underlying DAL mechanism. We notice that some strategies remain relatively stable across most datasets, while others show larger costs on large datasets such as the EM, RM, and RV at method- and variable-level refactorings. Overall, the heatmap highlights that computational efficiency is both strategy- and dataset-dependent, and therefore the predictive gains of DAL should be interpreted together with the query-time overhead required to obtain them.

Table 8 provides a direct cost-effectiveness view at the 50% labeling budget by summarizing, across refactoring types,  $\Delta F1$  and  $\Delta Time$  (minutes) denote the difference between each DAL strategy and the PL model ( $DAL - PL$ ), where positive  $\Delta F1$  indicates better predictive performance and negative  $\Delta Time$  indicates faster execution. Median and interquartile range (IQR) are reported to summarize central tendency and variability across refactoring types. The F1 Status and Time Status columns indicate whether the median performance is better/worse and faster/slower than the PL model, respectively.

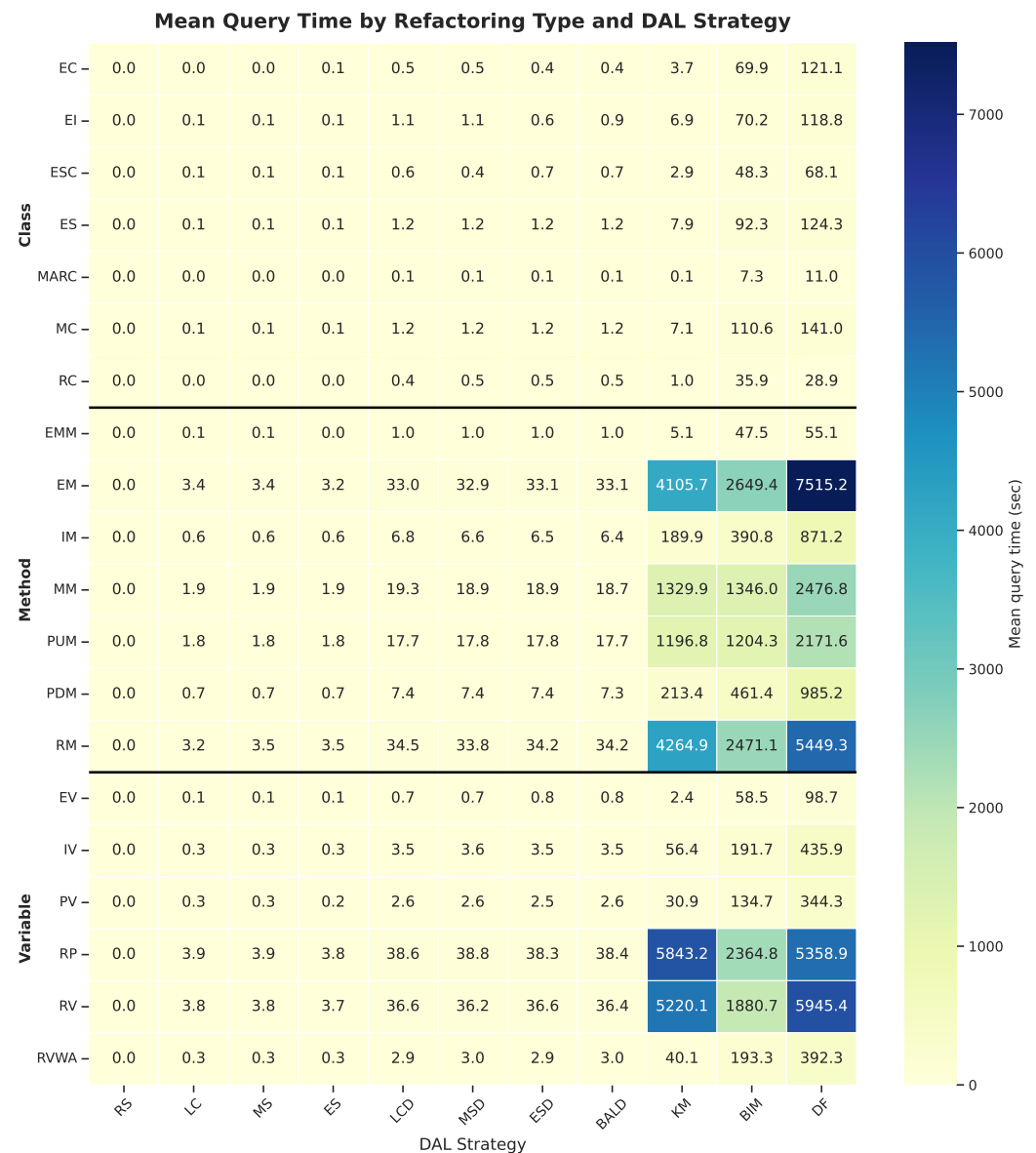


**Figure 9.** RQ3 (Computational Efficiency): Mean query time (in seconds) for each DAL strategy across refactoring datasets.

**Table 8.** RQ3 (Computational Efficiency): Effectiveness–runtime trade-off of DAL strategies at the 50% labeling budget, summarized across refactoring types and measured relative to the PL model trained on 100% labeled data.

| Strategy | Median $\Delta F1$ | IQR $\Delta F1$ | Median $\Delta Time$ (min) | IQR $\Delta Time$ (min) | F1 Status | Time Status | <i>n</i> |
|----------|--------------------|-----------------|----------------------------|-------------------------|-----------|-------------|----------|
| ES       | 0.0019             | 0.0070          | −0.0763                    | 0.6500                  | Better    | Faster      | 20       |
| MS       | 0.0007             | 0.0057          | −0.0750                    | 0.6673                  | Better    | Faster      | 20       |
| DF       | −0.0006            | 0.0051          | 0.0072                     | 2.3038                  | Worse     | Slower      | 20       |
| LCD      | −0.0011            | 0.0106          | −0.0835                    | 0.6874                  | Worse     | Faster      | 20       |
| ESD      | −0.0013            | 0.0056          | −0.0658                    | 0.6717                  | Worse     | Faster      | 20       |
| MSD      | −0.0018            | 0.0079          | −0.0645                    | 0.6705                  | Worse     | Faster      | 20       |
| LC       | −0.0022            | 0.0080          | −0.0758                    | 0.6792                  | Worse     | Faster      | 20       |
| BIM      | −0.0037            | 0.0081          | 0.0063                     | 1.0997                  | Worse     | Slower      | 20       |
| KM       | −0.0063            | 0.0093          | −0.0094                    | 0.2544                  | Worse     | Faster      | 20       |
| RS       | −0.0070            | 0.0118          | −0.0785                    | 0.6443                  | Worse     | Faster      | 20       |
| BALD     | −0.0306            | 0.0252          | −0.0658                    | 0.6487                  | Worse     | Faster      | 20       |

The table shows that ES and MS are the only strategies with positive median  $\Delta F1$  (+0.0019 and +0.0007, respectively) while also being faster than the PL model (median  $\Delta Time$  of −0.0763 and −0.0750 min), making them the strongest Pareto-like choices in this comparison. Although several other strategies are also faster than PL (e.g., LCD, ESD, MSD,

LC, KM, RS, and BALD), their median  $\Delta F1$  values are negative, indicating a trade-off in which runtime savings come at some loss in predictive performance; among these, LCD, ESD, and MSD remain relatively competitive with small median F1 degradation (from  $-0.0011$  to  $-0.0018$ ). In contrast, DF and BIM are both slower and worse on median, with positive median runtime overhead and negative median  $\Delta F1$ , making them less attractive under equal-budget conditions.

These results suggest that standard uncertainty-based strategies (ES and MS) provide the most favorable balance, while dropout-based uncertainty variants remain strong alternatives when a slight median F1-score loss is acceptable in exchange for faster runtime. The table also shows that BALD has the largest median F1 degradation despite being faster and that RS, although fast, is less competitive in effectiveness than the best uncertainty-based methods. Overall, the evidence indicates that in large-scale refactoring prediction, uncertainty-based DAL is the most cost-effective choice at 50% labeling.

## 7. Practical Implications

The results of this study have direct implications for the practical deployment of refactoring prediction in real software development workflows. Although the proposed DAL framework is primarily evaluated as a predictive modeling approach, its intended use in practice is as a human-in-the-loop decision-support tool rather than a fully autonomous refactoring mechanism. In this role, the system helps developers identify code entities that are more likely to require refactoring while reducing the labeling burden needed to maintain and improve the model over time.

From a scalability perspective, the proposed framework can be deployed incrementally on large real-world repositories by prioritizing recently changed components, pull requests, or selected submodules. In an integrated development environment (IDE), the proposed approach can be incorporated as a lightweight recommendation component. Software metrics for classes, methods, or variables can be extracted from the current project state or from recently modified entities, and the trained model can use these metrics to estimate the likelihood that a given entity requires refactoring. The output can then be presented to developers as ranked recommendations, for example, through an inspection panel, warning list, or code-quality assistant. In this setting, the tool does not automatically modify source code; instead, it highlights candidates that deserve developer attention and leaves the final decision to the programmer. This integration style is consistent with the practical goal of supporting refactoring awareness without interrupting normal development activities.

A similar integration can be achieved in continuous integration (CI) pipelines. During commit validation, pull-request analysis, or scheduled quality checks, the same metric extraction process can be applied to newly added or modified code entities. The trained model can then produce a ranked report of components that are likely to benefit from refactoring. Such reports may be attached to CI artifacts, code review dashboards, or quality gates, enabling maintainers to inspect potential refactoring needs before technical debt accumulates further. In this context, the system acts as an auxiliary quality-analysis layer that complements existing static analysis and code review practices.

An important implication of the proposed framework is the use of developer feedback as a source of labels. Because DAL focuses on selecting the most informative instances, developers do not need to label large volumes of data. Instead, the system can query only uncertain, borderline, or representative entities and ask developers or maintainers to confirm whether a suggested refactoring is appropriate. These confirmations or rejections can then be stored as newly labeled examples and incorporated into subsequent training rounds. This creates a continuous human-in-the-loop workflow in which the model improves in-

crementally through targeted expert input. Such a mechanism is especially attractive in industrial settings, where expert time is limited and annotation cost is a major constraint.

Another practical consideration is that the current framework assumes ideal developer feedback, where queried instances receive immediate, accurate, and consistent labels. In real deployment settings, however, annotations may be delayed, uncertain, or inconsistent across developers, especially when refactoring decisions are subjective or depend on project-specific conventions. Such label noise may reduce the effectiveness of the active learning process and could influence which query strategies remain most effective in practice.

In practice, false positives can be reduced in several ways: by using confidence-based ranking, by setting decision thresholds to favor higher precision, by limiting recommendations to the top-ranked candidates, and by requiring developer verification before any refactoring action is considered. These safeguards help ensure that the system supports developers without overwhelming them with noisy alerts or unnecessary interventions.

## 8. Threats to Validity

We discuss potential threats to the validity of our study following standard empirical software engineering guidelines.

### 8.1. Internal Validity

Internal validity concerns whether the observed effects can be attributed to the proposed DAL framework rather than confounding factors. To mitigate this threat, we fixed all model hyperparameters, training epochs, and preprocessing steps across query strategies. All strategies operated on the same initial labeled set and test split, and differences arose solely from the query mechanism. Another threat to validity arises from the use of a fixed parameter configuration across all datasets and DAL strategies. Although this supports a controlled comparison, it does not establish whether the reported results are robust to alternative settings of key learning parameters, such as learning rate, dropout probability, hidden dimension, or number of epochs. Accordingly, the findings of this study should be interpreted with respect to the selected configuration rather than as evidence of parameter-invariant behavior.

### 8.2. Construct Validity

Construct validity relates to whether the chosen metrics and experimental setup adequately capture the concept of refactoring prediction performance. We evaluated models using precision, recall, and F1-score, with F1-score selected as the primary metric due to the severe class imbalance in refactoring datasets.

### 8.3. Conclusion Validity

Conclusion validity concerns the reliability of the observed results. To reduce random effects, we used a fixed test set that remained untouched throughout the DAL process and applied consistent evaluation after each round. Performance trends were analyzed via learning curves rather than single-point estimates, reducing sensitivity to stochastic fluctuations in early rounds. A threat to conclusion validity arises from the fact that the reported results are based on a single experimental run with a fixed random seed. Although the replication package supports reproducibility of the implementation, it does not by itself demonstrate robustness against stochastic variation in initialization, data partitioning, or DAL sampling. Therefore, the observed differences among strategies should be interpreted with caution as their stability across repeated runs was not explicitly assessed. Future work should repeat the experiments with multiple random seeds and report summary statistics, such as mean and standard deviation, to provide a stronger robustness analysis.

Another threat to conclusion validity arises from the sensitivity of active learning to the initial labeled set or model initialization. To partially mitigate this threat, all DAL strategies were evaluated using the same train/test split and initial labeled set, ensuring that differences among strategies are attributable only to the query mechanism. However, the experiments were performed using a single run, and future work should repeat the experiments with multiple runs to ensure the robustness of the results.

#### 8.4. External Validity

A primary threat to the external validity of this study is the exclusive use of Java projects. While Java is a dominant language in software engineering research and provides a mature ecosystem for refactoring tools, our findings may not fully generalize to languages with different paradigms, such as Python or C++. Although the datasets used in our evaluation cover multiple granularities, the specific coding styles and project characteristics in Java codebases may influence the performance of the DAL strategies. Consequently, the results should be interpreted as a demonstration of DAL's potential within the Java ecosystem rather than a universal performance guarantee across all development environments.

Another threat to external validity is that the study relies only on static software metrics and does not consider richer code representations, such as AST-based features, code embeddings, or historical change patterns. Therefore, the findings should be interpreted as applying to metric-based refactoring prediction, and the relative effectiveness of DAL strategies may differ when using richer semantic or higher-dimensional representations.

A potential threat to external validity arises from dataset bias in refactoring prediction since the instances may reflect project-specific characteristics, coding conventions, and historical development patterns. In this study, evaluation was conducted using an instance-level 80/20 train–test split rather than a strict cross-project protocol. Therefore, the reported results demonstrate effectiveness within the observed data distribution, but they do not fully establish generalization to entirely unseen projects. Future work should investigate cross-project settings, such as leave-one-project-out evaluation, to better assess robustness and generalizability. Moreover, because DAL was evaluated on an under-sampled training pool, the observed label-efficiency and relative strategy behavior may not fully generalize to the original imbalanced setting.

Another threat to external validity is that all DAL strategies were evaluated using the same feed-forward neural network. Although this supports a controlled comparison of the query strategies, the reported behavior should be interpreted as specific to this neural tabular setting and may differ for other model families, such as tree-based, graph-based, or transformer-based approaches.

## 9. Conclusions and Future Work

Refactoring recommendation and opportunity prediction can support maintenance planning and technical debt management, yet adoption of refactoring recommendation systems in practice is often constrained by the cost of obtaining high-quality labeled data. In this paper, we investigated a DAL pipeline that combines a deep neural classifier with pool-based query selection to prioritize the annotation of the most informative software entities. By iteratively training the model and acquiring labels in a targeted manner, the proposed approach aims to substantially reduce the labeling effort needed to reach strong predictive performance.

The experimental results across multiple refactoring datasets and granularities indicate that DAL is an effective strategy for label-efficient refactoring prediction. In particular, we observed that competitive performance can be reached with a small fraction of labeled data, requiring, on average, only 11.4% labeled instances for class-level prediction, 25.0%

for method-level prediction, and 20.0% for variable-level prediction. These findings translate into large reductions in annotation demand (approximately 75–89% labeling savings, depending on granularity) while in several settings achieving performance comparable to, or occasionally exceeding, the fully supervised baseline trained on the complete labeled set. Overall, these outcomes suggest that DAL can enable practical deployment scenarios where extensive manual labeling is infeasible.

From a practical standpoint, our evaluation also highlights an important effectiveness–efficiency trade-off among acquisition functions. Uncertainty-driven strategies (e.g., least-confidence, margin, and entropy) and their Bayesian approximations based on MC-dropout were among the most favorable choices across the evaluated budgets, and they tend to incur modest query-time overheads. In contrast, diversity- and adversarial-oriented strategies may impose substantially higher computational costs during selection, which can limit their practicality in time- or resource-constrained settings. These observations imply that for many real-world scenarios, uncertainty-based DAL constitutes a strong default configuration.

Despite these promising results, several limitations should be considered. Our study relies on datasets derived exclusively from Java projects and utilizes static software metrics as input features; consequently, the observed performance and sampling behaviors may not directly generalize to other programming languages. Additionally, our evaluation focuses on pool-based DAL; alternative configurations, such as stream-based selection or human-in-the-loop workflows with richer feedback mechanisms, remain to be explored. Moreover, all DAL strategies in this study were evaluated using the same feed-forward neural network as the common base learner; as such, the reported findings characterize DAL behavior within this neural tabular setup and may not generalize to other strong model families.

A key direction for future research involves validating these results using cross-language datasets to confirm whether the query strategies identified as most effective in this work maintain their performance across different programming ecosystems. Future efforts should involve evaluating the DAL pipeline on datasets from diverse sources and languages, such as C++ or Python, to assess its robustness against varying software metric distributions. Furthermore, examining projects across multiple programming languages will help generalize our findings regarding the application of the DAL framework. Another promising avenue is the integration of these models into developer-facing refactoring tools to study human factors, trust, and end-to-end productivity impacts. In addition, future work should extend the empirical comparison beyond the feed-forward neural backbone used here by evaluating the same DAL query strategies on other strong model families, such as tree-based ensembles, graph-based models, and transformer-based approaches, to determine whether the effectiveness–efficiency patterns observed in this study generalize across different predictive backbones. To facilitate reproducibility and follow-on research, we have provided a replication package including the datasets, implementations, and scripts necessary to reproduce the experiments presented in this paper.

**Author Contributions:** Conceptualization, A.A. (Abdulmajeed Alameer) and A.A. (Amal Alazba); methodology, A.A. (Abdulmajeed Alameer) and A.A. (Amal Alazba); software, A.A. (Abdulmajeed Alameer) and A.A. (Amal Alazba); validation, A.A. (Abdulmajeed Alameer) and A.A. (Amal Alazba); formal analysis, A.A. (Abdulmajeed Alameer) and A.A. (Amal Alazba); investigation, A.A. (Abdulmajeed Alameer) and A.A. (Amal Alazba); resources, A.A. (Abdulmajeed Alameer) and A.A. (Amal Alazba); data curation, A.A. (Abdulmajeed Alameer) and A.A. (Amal Alazba); writing—original draft preparation, A.A. (Abdulmajeed Alameer) and A.A. (Amal Alazba); writing—review and editing, A.A. (Abdulmajeed Alameer); visualization, A.A. (Abdulmajeed Alameer) and A.A. (Amal Alazba); supervision, A.A. (Abdulmajeed Alameer); project adminis-

tration, A.A. (Abdulmajeed Alameer); funding acquisition, A.A. (Abdulmajeed Alameer). All authors have read and agreed to the published version of the manuscript.

**Funding:** This research project was supported by a grant from the Ongoing Research Funding program, (ORF-2026-1537), King Saud University, Riyadh, Saudi Arabia.

**Data Availability Statement:** The datasets and source code used in this study are publicly available at <https://github.com/amalazba/Deep-Active-Learning-for-Label-Efficient-Refactoring-Prediction> (accessed on 26 March 2026).

**Conflicts of Interest:** The authors declare no conflicts of interest.

## References

1. Mens, T.; Tourwé, T. A Survey of Software Refactoring. *IEEE Trans. Softw. Eng.* **2004**, *30*, 126–139. [CrossRef]
2. Kim, M.; Zimmermann, T.; Nagappan, N. An Empirical Study of Refactoring Challenges and Benefits at Microsoft. *IEEE Trans. Softw. Eng.* **2014**, *40*, 633–649. [CrossRef]
3. Kim, M.; Zimmermann, T.; Nagappan, N. A field study of refactoring challenges and benefits. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, New York, NY, USA, 11–16 November 2012; FSE '12. [CrossRef]
4. Silva, D.; Tsantalis, N.; Valente, M.T. Why we refactor? confessions of GitHub contributors. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, NY, USA, 13–18 November 2016; FSE 2016; pp. 858–870. [CrossRef]
5. Aniche, M.; Maziero, E.; Durelli, R.S.; Durelli, V.H.S. The Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring. *IEEE Trans. Softw. Eng.* **2022**, *48*, 1432–1450. [CrossRef]
6. Alameer, A.; Alazba, A. Predicting software refactoring opportunities using two-level TB-Stacking ensemble models. *Autom. Softw. Eng.* **2026**, *33*, 32. [CrossRef]
7. Dallal, J.A. Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics. *Inf. Softw. Technol.* **2012**, *54*, 1125–1141. [CrossRef]
8. Pantuchina, J.; Bavota, G.; Tufano, M.; Poshyanyk, D. Towards just-in-time refactoring recommenders. In Proceedings of the 26th Conference on Program Comprehension (ICPC '18), Gothenburg, Sweden, 28–29 May 2018; ACM: New York, NY, USA, 2018; pp. 312–315. [CrossRef]
9. Köşker, O.; Turhan, B.; Bener, A.B. An expert system for determining candidate software classes for refactoring. *Expert Syst. Appl.* **2009**, *36*, 10000–10003. [CrossRef]
10. Cui, D.; Wang, J.; Wang, Q.; Ji, P.; Qiao, M.; Zhao, Y.; Hu, J.; Wang, L.; Li, Q. Three Heads Are Better Than One: Suggesting Move Method Refactoring Opportunities with Inter-class Code Entity Dependency Enhanced Hybrid Hypergraph Neural Network. In Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, 27 October–1 November 2024; ACM: New York, NY, USA, 2024; pp. 745–757. [CrossRef]
11. Sagar, P.S.; AlOmar, E.A.; Mkaouer, M.W.; Ouni, A.; Newman, C.D. Comparing Commit Messages and Source Code Metrics for the Prediction of Refactoring Activities. *Algorithms* **2021**, *14*, 289. [CrossRef]
12. Jesse, K.; Kuhmuench, C.; Sawant, A. RefactorScore: Evaluating Refactor Prone Code. *IEEE Trans. Softw. Eng.* **2023**, *49*, 5008–5026. [CrossRef]
13. Büyük, O.; Nizam, A. Deep learning with class-level abstract syntax tree and code histories for detecting code modification requirements. *J. Syst. Softw.* **2023**, *206*, 111851. [CrossRef]
14. Fowler, M. *Refactoring: Improving the Design of Existing Code*, 2nd ed.; Addison-Wesley Professional: Boston, MA, USA, 2018.
15. Opdyke, W.F. Refactoring Object-Oriented Frameworks. Ph.D. Thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992.
16. AlOmar, E.A.; Mkaouer, M.W.; Ouni, A. Toward the automatic classification of Self-Affirmed Refactoring. *J. Syst. Softw.* **2021**, *171*, 110821. [CrossRef]
17. Noei, S.; Li, H.; Zou, Y. Detecting Refactoring Commits in Machine Learning Python Projects: A Machine Learning-Based Approach. *ACM Trans. Softw. Eng. Methodol.* **2025**, *34*, 1–25. [CrossRef]
18. Patnaik, A.; Panigrahi, R.; Padhy, N. Prediction of Accuracy on Open Source Java Projects Using Class Level Refactoring. In Proceedings of the 2020 International Conference on Computer Science, Engineering and Applications (ICCSEA), Gunupur, India, 13–14 March 2020; pp. 1–6. [CrossRef]
19. Aribandi, V.K.; Kumar, L.; Bhanu Murthy Neti, L.; Krishna, A. Prediction of Refactoring-Prone Classes Using Ensemble Learning. In *Proceedings of the Neural Information Processing*; Gedeon, T., Wong, K.W., Lee, M., Eds.; Springer: Cham, Switzerland, 2019; pp. 242–250. [CrossRef]

20. Kumar, L.; Sureka, A. Application of LSSVM and SMOTE on Seven Open Source Projects for Predicting Refactoring at Class Level. In Proceedings of the 2017 24th Asia-Pacific Software Engineering Conference (APSEC), Nanjing, China, 4–8 December 2017; pp. 90–99. [\[CrossRef\]](#)
21. Kumar, L.; Satapathy, S.M.; Murthy, L.B. Method Level Refactoring Prediction on Five Open Source Java Projects using Machine Learning Techniques. In Proceedings of the 12th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference), Pune, India, 14–16 February 2019; ISEC '19; pp. 1–10. [\[CrossRef\]](#)
22. Panigrahi, R.; Kuanar, S.K.; Kumar, L. An Empirical Study for Method-Level Refactoring Prediction by Ensemble Technique and SMOTE to Improve Its Efficiency. *Int. J. Open Source Softw. Process. (IJOSSP)* **2021**, *12*, 19–36. [\[CrossRef\]](#)
23. Nyamawe, A.S.; Liu, H.; Niu, N.; Umer, Q.; Niu, Z. Automated Recommendation of Software Refactorings Based on Feature Requests. In Proceedings of the 2019 IEEE 27th International Requirements Engineering Conference (RE), Jeju, Republic of Korea, 23–27 September 2019; pp. 187–198; ISSN 2332-6441. [\[CrossRef\]](#)
24. Kurbatova, Z.; Veselov, I.; Golubev, Y.; Bryksin, T. Recommendation of Move Method Refactoring Using Path-Based Representation of Code. In Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, Seoul, Republic of Korea, 27 June–19 July 2020; pp. 315–322. [\[CrossRef\]](#)
25. Settles, B. *Active Learning Literature Survey*; Technical Report 1648; University of Wisconsin–Madison: Madison, WI, USA, 2009.
26. Ren, P.; Xiao, Y.; Chang, X.; Huang, P.-Y.; Li, Z.; Gupta, B.B.; Chen, X.; Wang, X. A Survey of Deep Active Learning. *ACM Comput. Surv.* **2021**, *54*, 1–40. [\[CrossRef\]](#)
27. Gal, Y.; Ghahramani, Z. Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning. In Proceedings of the 33rd International Conference on Machine Learning (ICML), New York City, NY, USA, 19–24 June 2016.
28. Gal, Y.; Islam, R.; Ghahramani, Z. Deep Bayesian Active Learning with Image Data. In Proceedings of the 34th International Conference on Machine Learning (ICML), Sydney, NSW, Australia, 6–11 August 2017; pp. 1183–1192.
29. Houthby, N.; Huszár, F.; Ghahramani, Z.; Lengyel, M. Bayesian Active Learning for Classification and Preference Learning. *arXiv* **2011**, arXiv:1112.5745. [\[CrossRef\]](#)
30. Sener, O.; Savarese, S. Active Learning for Convolutional Neural Networks: A Core-Set Approach. In Proceedings of the International Conference on Learning Representations (ICLR), Vancouver, BC, Canada, 30 April–3 May 2018.
31. Liapis, C.M.; Karanikola, A.; Kotsiantis, S. Data-efficient software defect prediction: A comparative analysis of active learning-enhanced models and voting ensembles. *Inf. Sci.* **2024**, *676*, 120786. [\[CrossRef\]](#)
32. Yu, Z.; Theisen, C.; Williams, L.A.; Menzies, T. Improving Vulnerability Inspection Efficiency Using Active Learning. *IEEE Trans. Softw. Eng.* **2021**, *47*, 2401–2420. [\[CrossRef\]](#)
33. Feyzi, F.; Daneshdoost, A. Studying the effectiveness of deep active learning in software defect prediction. *Int. J. Comput. Appl.* **2023**, *45*, 534–552. [\[CrossRef\]](#)
34. Wang, Y.; Li, Y.; Ren, Y.; Yu, J. Enhancing Cross-Project Just-In-Time Defect Prediction with Active Deep Learning. In Proceedings of the 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C), Chiang Mai, Thailand, 22–26 October 2023; pp. 93–102. [\[CrossRef\]](#)
35. Sun, X.; Tu, L.; Zhang, J.; Cai, J.; Li, B.; Wang, Y. ASSBERT: Active and semi-supervised BERT for smart contract vulnerability detection. *J. Inf. Secur. Appl.* **2023**, *73*, 103423. [\[CrossRef\]](#)
36. Hu, Q.; Guo, Y.; Xie, X.; Cordy, M.; Ma, L.; Papadakis, M.; Traon, Y.L. Active Code Learning: Benchmarking Sample-Efficient Training of Code Models. *IEEE Trans. Softw. Eng.* **2024**, *50*, 1080–1095. [\[CrossRef\]](#)
37. Alturayef, N.; Ahmad, I. EASE: An enhanced active learning framework for aspect-based sentiment analysis based on sample diversity and data augmentation. *Expert Syst. Appl.* **2025**, *261*, 125525. [\[CrossRef\]](#)
38. Kamei, Y.; Shihab, E.; Adams, B.; Hassan, A.E.; Mockus, A.; Sinha, A.; Ubayashi, N. A Large-Scale Empirical Study of Just-In-Time Quality Assurance. *IEEE Trans. Softw. Eng.* **2013**, *39*, 757–773. [\[CrossRef\]](#)
39. Aburas, A. Handling Imbalanced Datasets in Software Refactoring Prediction. In *Proceedings of the International Conference for Information and Communication Technologies*; Springer: Cham, Switzerland, 2023; pp. 145–158.
40. Wang, S.; Liu, T.; Nam, J.; Tan, L. Automatically Learning Semantic Features for Defect Prediction. *IEEE Trans. Softw. Eng.* **2017**, *43*, 297–321.
41. Shannon, C.E. A mathematical theory of communication. *Bell Syst. Tech. J.* **1948**, *27*, 379–423. [\[CrossRef\]](#)
42. Lewis, D.D. A sequential algorithm for training text classifiers: Corrigendum and additional data. In *Proceedings of the AcM Sigir Forum*; ACM: New York, NY, USA, 1995; Volume 29, pp. 13–19.
43. Scheffer, T.; Decomain, C.; Wrobel, S. Active hidden markov models for information extraction. In *Proceedings of the International Symposium on Intelligent Data Analysis*; Springer: Cham, Switzerland, 2001; pp. 309–318.
44. Ducoffe, M.; Precioso, F. Adversarial active learning for deep networks: A margin based approach. *arXiv* **2018**, arXiv:1802.09841. [\[CrossRef\]](#)
45. Huang, K.H. DeepAL: Deep Active Learning in Python. *arXiv* **2021**, arXiv:2111.15258. [\[CrossRef\]](#)

46. Tantithamthavorn, C.; McIntosh, S.; Hassan, A.E.; Matsumoto, K. The Impact of Automated Parameter Optimization for Defect Prediction Models. *IEEE Trans. Softw. Eng.* **2019**, *45*, 683–711. [[CrossRef](#)]
47. Tantithamthavorn, C.; McIntosh, S.; Hassan, A.E.; Matsumoto, K. An Empirical Comparison of Model Validation Techniques for Defect Prediction Models. *IEEE Trans. Softw. Eng.* **2017**, *43*, 1–18. [[CrossRef](#)]

**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.