DOI: https://doi.org/10.15276/hait.08.2025.19
UDC 004.021:004.4'42:004.8

Uncertainty-aware multi-objective refactoring for code duplication

Dmytro D. Kurinko[1)]
ORCID: https://orcid.org/0000-0001-8304-3257; dmitrykurinko@gmail.com
Viktoriia I. Kryvda[1)]
ORCID: https://orcid.org/0000-0002-0930-1163; kryvda@op.edu.ua
[1)] Odesa Polytechnic National University, 1, Shevchenko Ave. Odesa, 65044, Ukraine

ABSTRACT

Code clones are recurring code fragments that may hinder software maintainability if not properly managed. While many clone detection tools exist, they often stop at identification and provide no clear guidance on whether a detected clone group should be refactored, how to do so, or in what order. This paper presents a machine learning–based method for recommending clone refactorings with prioritization and confidence estimation. The proposed approach represents code fragments using abstract syntax trees, program dependency graphs, and semantic embeddings from a pre-trained CodeBERT model. In addition, version control data is used to extract evolutionary features such as churn, age, and co-change patterns. A multi-class classifier predicts refactoring types, while open-set recognition techniques identify uncertain cases and flag them as unknown. Effort and benefit estimation models help prioritize suggestions based on a cost-effectiveness ratio. We evaluated the method on four open-source Java projects using a manually labeled dataset of 600 clone groups. The system achieves a macro-F1 score of zero point seven six on known refactoring types and an AUROC of zero point nine one for unknown detection. Prioritized recommendation quality reaches NDCG@3 of zero point eight nine, showing strong alignment with expert assessments. The results indicate that clone refactoring can be effectively supported through integrated code representation, uncertainty modeling, and prioritization. The approach transforms clone analysis from a passive task into an actionable process.

Keywords: Clone refactoring; artificial intelligence in software engineering; machine learning; deep learning; clone classification; open-set recognition; uncertainty estimation

***For citation*: Kurinko D. D., Kryvda V. I. "Uncertainty-aware multi-objective refactoring for code duplication". *Herald of Advanced Information Technology.* 2025; Vol.8 No.3: 301–315. DOI: https://doi.org/10.15276/hait.08.2025.19**

1. INTRODUCTION, FORMULATION OF THE PROBLEM

Code clones are fragments that partially or fully duplicate each other. They arise due to deadlines, copying "working" solutions without proper abstraction, temporary hotfixes, and branching product lines. In the short term, copy-paste speeds development, but in the long term it increases technical debt: change points multiply, consistency is harder to maintain, and the risk of defects and regressions grows [1]. In large codebases (monorepos, microservices, polyglot stacks), clones are inevitable and "migrate" across modules and teams, complicating code reviews and slowing releases [2]. The problem is amplified by frequent releases and CI/CD: fixes must propagate synchronously to all replicas, otherwise environments diverge in behavior. Clones also reduce the evolutionary flexibility of the architecture: they hinder extracting common APIs and adopting new technologies. Therefore, detection alone is not enough – teams need recommendations on what and how to refactor, taking into account semantics, dependency context, change history, and the expected benefit vs. effort [3].

Industrial and research practice in clone detection spans a wide spectrum of approaches: from textual and token-based methods to structure-oriented (AST), graph-based (PDG/CFG), and modern vector representations of code (neural embeddings, GNN/CodeBERT-like models) [4]. These tools reliably identify and cluster duplicates in large-scale repositories, produce clone clusters and metrics (degree of duplication, "hot spots"), and integrate into IDEs and CI/CD as code-quality reports. At the process level, they support regular technical-debt monitoring, facilitate audits, and inform backlog grooming and planning [5].

However, most solutions stop at the fact of detection and do not proceed to actionable guidance. Typical limitations include: (i) lack of semantically grounded recommendations on what exactly and how to refactor for a given cluster; (ii) absence of calibrated handling of uncertainty (risk of overconfident or overly conservative decisions); (iii) no mechanisms for benefit/effort–based

prioritization that account for version-control history (VCS) and operational risk; and (iv) limited explainability and weak integration with decision-making practices (code review, sprint planning) [6]. As a result, clone detection rarely translates into predictable cost savings: teams must manually decide what to refactor, in what order, and whether it will pay off [7].

Despite mature clone-detection tooling and a variety of advisory approaches, a persistent gap remains between the fact of detection and actionable engineering decisions. Existing systems rarely unite, within a single method: (i) a semantically faithful code representation (to handle fragments that are similar in meaning yet syntactically divergent); (ii) controlled uncertainty management (open-set formulation with explicit applicability limits and calibrated confidence); (iii) benefit/effort–based prioritization that accounts for version-control history (VCS), co-evolution signals, and defect risk; (iv) explainability of recommendations at the level of concrete transformations (which differences can be parameterized, which refactoring template is appropriate); and (v) process integrability (IDE/CI, sprint planning) alongside experimental reproducibility. In addition, there is a lack of vetted protocols for cross-project generalization (robustness to coding style/domain) and harmonized metrics that jointly assess classification quality, decision risk, and ranking utility for work planning [8].

Accordingly, there is a clear need for a method that integrates deeper structural and semantic understanding of code, calibrated uncertainty, and value-oriented ranking, delivering transparent, reproducible, and practically applicable recommendations for clone refactoring.

The purpose of this study is to develop a method that converts clone detection into actionable, prioritized refactoring decisions. Concretely, we aim to (i) provide semantically grounded recommendations on what to refactor and how (e.g., which refactoring template to apply), (ii) manage uncertainty by explicitly identifying low-confidence and out-of-distribution cases, and (iii) prioritize candidate actions by expected benefit-versus-effort, informed by code structure and version-control history. The intended outcome is a transparent, reproducible advisory pipeline that reduces technical debt, mitigates regression risk, and accelerates architectural evolution while integrating seamlessly with existing engineering workflows (IDE/CI).

2. BACKGROUND AND RELATED WORK

The problem of code cloning has been extensively studied over the past two decades, with early foundational works focusing on the classification and detection of clone types (e.g., Type I–IV) [9]. Numerous studies have confirmed that code clones are not only widespread but often persist for long periods in production systems, where they contribute to increased maintenance cost, defect-proneness, and codebase inconsistency [10], [11], [12].

From a software engineering perspective, the remediation of code clones typically involves manual or semi-automated refactoring, guided by developer intuition or tool recommendations. Research has proposed a range of approaches, from catalog-based refactoring patterns to clone-specific transformations (e.g., Extract Method, Move Method). Some tools support automatic transformation under strict preconditions, while others assist human developers with ranking or filtering options [13].

However, empirical studies show that developers frequently ignore clone warnings, citing lack of actionable guidance, potential side effects of transformation, and uncertainty about long-term benefits [14]. Moreover, decisions to refactor are often project-specific and context-sensitive: clones that are harmful in one subsystem may be benign or even beneficial in another.

Numerous approaches have been proposed to facilitate the refactoring of code clones, ranging from static rule-based systems to learning-enabled recommender frameworks [15], [16], [17]. Traditional techniques often rely on predefined templates such as Extract Method, Move Method, or Pull Up Method, applied either manually or with IDE support (e.g., Eclipse, IntelliJ). While these refactorings are well understood and standardized, identifying the correct context in which they should be applied remains non-trivial.

Rule-based systems encode structural patterns and syntactic thresholds to suggest refactorings. These approaches are efficient and interpretable but suffer from limited adaptability and inability to reason over semantic similarity or usage context. To address such limitations, later works have incorporated code metrics (e.g., size, duplication ratio, cohesion) or heuristic scoring functions to filter or prioritize clone groups [18].

More recently, machine learning–based approaches have emerged that attempt to predict the likelihood or appropriateness of refactoring actions

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

[19]. Some leverage feature engineering over abstract syntax trees (ASTs) or program dependency graphs (PDGs), while others exploit version history and commit metadata to learn patterns of past developer behavior. A prominent example is CREC, which uses clone histories and manually engineered features to rank refactoring opportunities [20].

However, despite promising results, most ML-based systems function as black-box predictors, offering limited interpretability and no guarantees of correctness. Additionally, they often lack the capability to distinguish between ambiguous or out-of-distribution inputs and may fail silently or behave erratically in such cases. Furthermore, few existing systems provide actionable, contextualized explanations or rank refactoring options by expected effort and impact, which limits their practical utility in complex industrial codebases [21].

Thus, while the literature demonstrates a wide spectrum of clone refactoring support tools, a unified pipeline that integrates detection, semantic understanding, prioritization, and uncertainty awareness remains elusive [22], [23], [24].

Numerous approaches have been proposed to facilitate the refactoring of code clones, ranging from static rule-based systems to learning-enabled recommender frameworks. Traditional techniques often rely on predefined templates such as Extract Method, Move Method, or Pull Up Method, applied either manually or with IDE support (e.g., Eclipse, IntelliJ). While these refactorings are well understood and standardized, identifying the correct context in which they should be applied remains non-trivial [25].

Rule-based systems encode structural patterns and syntactic thresholds to suggest refactorings. These approaches are efficient and interpretable but suffer from limited adaptability and inability to reason over semantic similarity or usage context. To address such limitations, later works have incorporated code metrics (e.g., size, duplication ratio, cohesion) or heuristic scoring functions to filter or prioritize clone groups [26].

More recently, machine learning–based approaches have emerged that attempt to predict the likelihood or appropriateness of refactoring actions. Some leverage feature engineering over abstract syntax trees (ASTs) or program dependency graphs (PDGs), while others exploit version history and commit metadata to learn patterns of past developer behavior. A prominent example is CREC, which uses clone histories and manually engineered features to rank refactoring opportunities [27].

However, despite promising results, most ML-based systems function as black-box predictors, offering limited interpretability and no guarantees of correctness. Additionally, they often lack the capability to distinguish between ambiguous or out-of-distribution inputs and may fail silently or behave erratically in such cases. Furthermore, few existing systems provide actionable, contextualized explanations or rank refactoring options by expected effort and impact, which limits their practical utility in complex industrial, codebases [28].

Thus, while the literature demonstrates a wide spectrum of clone refactoring support tools, a unified pipeline that integrates detection, semantic understanding, prioritization, and uncertainty awareness remains elusive.

Modern refactoring recommenders increasingly rely on learned code representations. Approaches range from traditional AST-based features to advanced embeddings generated by pre-trained models such as CodeBERT, GraphCodeBERT, or TreeSAGE. These representations enable semantic comparison and classification of code fragments, allowing for better generalization beyond syntactic similarity. However, many such models lack fine-grained control, interpretability, or explicit alignment with refactoring tasks [29].

In real-world scenarios, refactoring decisions often involve uncertainty – stemming from ambiguous clone semantics, unstable APIs, or missing documentation. While ML models can assist, few existing tools explicitly quantify uncertainty or assess refactoring effort vs. benefit. Prioritization strategies remain heuristic, lacking formal grounding in software engineering economics or risk analysis. The absence of uncertainty-aware recommendations limits developer trust and adoption.

This work bridges multiple gaps by proposing a clone refactoring advisor that combines (i) pre-trained code embeddings, (ii) confidence calibration for out-of-distribution detection, and (iii) impact/effort-based prioritization. In contrast to prior black-box recommenders, our approach provides interpretable, context-aware suggestions with explicit support for ambiguity and trade-off reasoning. It complements existing detection tools and advances clone refactoring from static listing toward actionable engineering guidance.

3. PROPOSED METHOD

3.1. Overview of the proposed architecture

The proposed system is designed as an end-to-end refactoring advisor for code clones, capable of

producing ranked, interpretable, and risk-aware recommendations. Its core objective is to go beyond clone detection and provide actionable guidance on whether a clone should be refactored, which transformation is appropriate, how confident the system is, and how the decision should be prioritized based on potential benefits and costs.

The architecture follows a modular, layered pipeline, which is summarized in Table 1. Each stage contributes distinct analytical capabilities: semantic representation, historical context, decision reasoning, and trust calibration.

The pipeline begins with clone group extraction, obtained from external clone detectors such as NiCad, SourcererCC, or CloneWorks. Each clone fragment is parsed into a structural form (AST and PDG) and embedded using a pretrained language model (e.g., CodeBERT, GraphCodeBERT). These embeddings are enriched with static structural features (e.g., LOC, control-flow depth) and evolutionary features extracted from the Git history of the codebase – such as change frequency, recency, authorship entropy, and co-change statistics.

The fused representation is then passed to a multi-output open-set classifier, which assigns each clone group to a recommended refactoring type (e.g., *Extract Method*, *Pull Up Method*, *Move Method*), or flags it as *Unknown* if the prediction confidence is low or the instance is semantically distant from training examples. To support this behavior, the system incorporates confidence calibration mechanisms (e.g., temperature scaling, dropout-based variance estimation), producing interpretable uncertainty estimates for each decision.

Next, the system evaluates the expected effort and benefit of refactoring, based on both static code metrics and historical evolution data. The effort-benefit ratio is used to prioritize clone groups, ensuring that high-impact, low-effort opportunities are surfaced to the top of the recommendation list.

Finally, each recommendation is accompanied by a concise natural-language explanation, generated from interpretable features, which justifies the recommendation in terms understandable to human developers. These explanations improve transparency and facilitate manual inspection or team discussions.

This modular design enables the system to adapt to different codebases, integrate new detectors or language models, and interact flexibly with human-in-the-loop workflows (e.g., for active learning or selective review). It lays the foundation for a practical, extensible refactoring advisor that supports real-world software engineering constraints.

3.2. Clone detection and preprocessing

The refactoring pipeline begins with the identification and preparation of clone groups – sets of code fragments that exhibit structural or semantic similarity. Clone detection serves as the foundation for all subsequent analysis; therefore, it is essential that the detected clones be of sufficient quality, granularity, and interpretability. This stage is deliberately decoupled from the rest of the pipeline to allow integration with multiple third-party detectors and facilitate language portability.

Table 1. **Summary of architecture components and their roles**

Component	Description
Clone Detection	External tools identify clone groups (e.g., Type I–III), used as input to the pipeline
Graph-Based Code Representation	AST and PDG structures encoded via pretrained models (CodeBERT, GraphCodeBERT)
Structural and Semantic Features	Includes LOC, nesting depth, token types, and semantic embeddings
Version Control Features	Extracted from Git: churn, recency, number of authors, co-change patterns
Open-Set Classifier	Predicts refactoring type or flags *Unknown* with calibrated uncertainty
Confidence Calibration	Quantifies prediction confidence using entropy, dropout variance, or temperature scaling
Effort & Benefit Estimation	Estimates refactoring cost and impact using structural + historical indicators
Prioritization Module	Ranks recommendations using benefit-to-effort ratio and confidence thresholds
Explanation Generator	Produces human-readable rationales for each recommendation to support review and trust

***Source*: compiled by the authors**

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

In our system, clone detection is treated as a pluggable preprocessing module. We assume that clone groups have already been detected using established tools such as NiCad, SourcererCC, or CloneWorks, depending on the desired balance of recall, precision, and scalability. Our implementation primarily targets Type I–III clones, which cover exact copies, syntactic variations, and renamed or reordered elements while excluding more abstract Type IV clones (semantic equivalence without syntactic similarity), which require deeper analysis beyond current scope.

Each clone group is passed through a preprocessing pipeline that standardizes code fragments for consistent downstream representation. This includes:

- whitespace and comment normalization, to remove irrelevant variability;
- AST sanitation, such as renaming identifiers with placeholders (e.g., VAR_1, FUNC_2) to reduce overfitting to naming patterns;
- control-flow slicing, which extracts the minimal executable block that covers the clone, preserving semantic boundaries;
- tokenization and parsing, preparing the code for both graph construction and transformer-based embedding.

Optionally, we apply context windowing to extend clone fragments with a limited number of surrounding lines (before/after), ensuring that local dependencies or method headers are retained. This provides richer inputs for downstream embeddings without introducing excessive noise.

To filter out trivial clones or overly noisy inputs, we enforce lightweight filtering rules (e.g., minimum LOC threshold, no empty-body methods, no auto-generated code), which are customizable based on project constraints.

This preprocessing ensures that each clone fragment is converted into a standardized, semantically meaningful, and model-friendly format, suitable for both structural analysis (AST/PDG) and semantic encoding (e.g., via transformer-based models). It also decouples the refactoring logic from any specific detection tool or input language, making the system extensible and adaptable across environments.

3.3. Code representation

Accurate and expressive representation of code is critical for enabling machine learning models to reason about clone similarity, refactoring intent, and transformation applicability. Unlike traditional approaches that rely solely on syntactic features (e.g., token counts, AST node frequencies), our method combines graph-based structural information with semantic embeddings from large-scale pretrained models.

This hybrid representation provides a more complete view of the code, capturing both low-level structure and high-level meaning.

Each code fragment is represented using three complementary layers (Table 2).

1. Abstract Syntax Tree (AST): the AST captures the syntactic structure of the code, including expressions, control-flow statements, and declarations. We use language-specific parsers to construct ASTs and extract subgraphs relevant to the cloned fragment. The AST provides the backbone for structural feature extraction and serves as input to graph neural networks (GNNs) or graph kernels.

2. Program Dependency Graph (PDG): to account for semantic and data-flow relationships, we construct PDGs that capture control dependencies (e.g., conditionals, loops) and data dependencies (e.g., variable usage and assignment). These graphs encode how the fragment behaves during execution, helping to distinguish semantically different clones that may be structurally similar.

3. Pretrained Code Embeddings: to enrich the graph-based representation with learned contextual information, we use transformer-based models trained on large code corpora. Specifically, we apply CodeBERT or GraphCodeBERT to obtain dense vector embeddings for each fragment. These models are capable of capturing naming conventions, idiomatic usage, and token co-occurrence patterns that are difficult to model with static graphs alone.

Table 2. **Components of hybrid code representation**

Component	Format	Captures	Notes
AST features	Graph / vector	Syntactic structure	Extracted using parser + feature extractor
PDG features	Graph / adjacency	Control & data dependencies	Control & data dependencies
Transformer embedding	Dense vector	Semantic similarity, naming, idioms	Obtained via CodeBERT, GraphCodeBERT
Fusion layer	Concatenated + MLP	Unified input for classification	Optional attention or learned fusion can be added

***Source*: compiled by the authors**

The final representation is constructed by concatenating and optionally fusing the outputs of all three modalities: structural features (from AST), dependency-aware features (from PDG), and semantic embeddings (from transformers). A feature projection layer ensures dimensional compatibility for downstream classifiers.

By integrating these three layers, the system builds a representation that is robust to superficial code changes (e.g., renaming, formatting) while remaining sensitive to deep behavioral differences that matter for refactoring. This hybrid encoding significantly improves the generalization capability of the downstream classifier and enables support for semantic clone groups, which would be indistinguishable using shallow features alone.

3.4. Evolutionary feature extraction

While structural and semantic representations of code provide critical insights into *what* a clone does, they offer limited information about *how it behaves over time*. Refactoring decisions are often influenced not only by code structure or similarity, but also by the evolutionary characteristics of the code – such as stability, volatility, team ownership, and defect history. To capture this dimension, we extract a complementary set of evolution-aware features from the project's version control history (e.g., Git).

These features are designed to reflect the temporal and collaborative context of each clone fragment and help the model assess whether a proposed refactoring is likely to be safe, beneficial, or costly.

Key Evolutionary Features. For each clone fragment (or its enclosing method/file), we extract the following version history indicators:

1) Change Frequency (churn) – the total number of commits in which the fragment (or file) was modified. High-churn code often indicates instability or active development, which may increase the risk of refactoring;

2) Time Since Last Change – measures code age or recency. Recently modified code may not be mature enough for safe restructuring, while very old code may reflect legacy debt;

3) Number of Distinct Authors – high author count suggests shared ownership and potentially inconsistent coding styles. Low count may indicate a single maintainer or owner;

4) Authorship Entropy – normalized entropy metric that captures how evenly the edits are distributed among contributors. This complements author count by identifying whether one developer dominates maintenance;

5) Code Survival Rate – proportion of original lines still present in the latest version. A low survival rate suggests volatility and frequent rewrites;

6) Defect Co-change Frequency (optional) – if issue-tracking integration is available, we also track whether the clone's file or method frequently co-changes with bug-fixing commits.

These features are computed using lightweight Git analysis (e.g., git blame, git log, diff parsing) and optionally augmented with external bug-tracking data. The features are normalized and integrated with the static and semantic code representations during training.

Motivation and Benefits. The inclusion of evolutionary context offers three core advantages:

- risk awareness: the model can distinguish between stable, legacy clones (often candidates for safe refactoring) and volatile, high-risk code (which may require human review);
- effort estimation: churn and author-related features serve as proxies for understanding how expensive or disruptive a refactoring may be in social and technical terms;
- prioritization context: by including temporal and team dynamics, the system can prioritize clones that are not just structurally refactorable, but also contextually actionable.

By incorporating these evolution-aware signals, the proposed method supports time-sensitive and context-aware clone refactoring recommendations, helping teams focus effort where it is most justified.

3.4.1. Illustrative example: evolution-aware analysis of a clone fragment

To demonstrate the utility of evolutionary features in clone refactoring analysis, we consider a simple fragment from a Java-based e-commerce system (Fig. 1).

This code computes the total price of items in a shopping cart. While syntactically simple and semantically consistent with other fragments in the codebase, its evolutionary profile reveals important contextual factors that influence its refactoring potential.

```
public double calculateTotalPrice(List<Item> items) {
    double total = 0.0;
    for (Item item : items) {
        total += item.getPrice();
    }
    return total;
}
```

Fig. 1. **Sample code fragment**
Source: **compiled by the authors**

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

Version Control History (Git-based analysis). We analyzed the fragment using Git history tools (git log, git blame, and commit diffs) across a 24-month project timeline. The extracted data is presented in Table 3.

As a result, while the static structure and semantic embedding may strongly recommend an *Extract Method* transformation, the evolutionary features provide a counterbalancing signal that may lower the method's refactoring priority – or flag it as requiring human review.

3.5. Classifier and confidence estimation

At the heart of our system lies a machine learning–based refactoring type classifier, trained to predict the most appropriate transformation for each clone group. However, unlike conventional closed-set classifiers, our approach explicitly acknowledges the open-set nature of real-world refactoring: not all clones will match known transformation patterns, and some may be unsuitable for any automated suggestion. To address this, we design the classifier to support confidence-aware prediction with open-set rejection.

Refactoring as multi-class classification. The classifier receives a hybrid input vector representing a clone group – derived from structural features (AST/PDG), semantic embeddings (e.g., CodeBERT), and evolutionary indicators (e.g., churn, author entropy).

It outputs a probability distribution over a predefined set of refactoring types, such as:

- Extract Method;
- Pull Up Method;
- Move Method;
- Replace Temp with Query.

The predicted class corresponds to the refactoring type with the highest probability.

Open-set handling via confidence thresholding. Unlike closed-world scenarios, it is unrealistic to assume that all incoming clone groups belong to known refactoring categories. For example, a fragment may represent a domain-specific pattern or an anti-pattern not captured in training. Blindly assigning a class in such cases risks generating incorrect or harmful recommendations.

To mitigate this, we define an open-set classification scheme using a confidence-based rejection strategy. Specifically:

- let $p_{max} = \max_k P(y = k|x)$ be the softmax probability of the predicted class;
- if $p_{max} < \theta$, where θ is a calibrated threshold, the instance is rejected and assigned to an *Unknown* class;
- otherwise, the predicted class $\hat{y} = \arg max_k P(y = k|x)$ is accepted.

This enables the system to abstain from uncertain predictions, flagging them for manual review or deferred decision-making.

Confidence Calibration. To ensure that the classifier's predicted probabilities reflect true model confidence, we apply post-hoc calibration techniques.

Temperature scaling: A scalar parameter $T > 0$ is learned on a validation set such that softmax scores become:

$$softmax_T(z_i) = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}. \quad (1)$$

This reduces overconfidence in neural models, especially important in safety-critical tasks like refactoring.

Monte-carlo dropout: At inference time, dropout is enabled during multiple forward passes. The variance of predictions is used as a proxy for epistemic uncertainty.

Model Selection and Training. We experimented with several classifier architectures:

- Multi-Layer Perceptron (MLP): baseline classifier using concatenated features;
- Transformer-based Classifier Head: directly fine-tunes CodeBERT for classification;

Table 3. **Extracted data from VCS analysis**

Feature	Value	Interpretation
Change_frequency	19 commits	High: the method is frequently edited, indicating active development
Last_modified_days	12 days ago	Recent: the code is still undergoing frequent changes
Num_authors	6 developers	Medium-high: shared ownership increases coordination complexity
Authorship_entropy	0.84	High entropy: changes are evenly distributed among contributors
Survival_ratio	0.55	Low-moderate: significant portions of the method have been rewritten
Bugfix_cochange_count	4 times	Notable: frequently changed alongside bugfix commits (e.g., rounding errors)

Source: **compiled by the authors**

•Graph Neural Network (GNN): processes AST or PDG graph structures;

• Hybrid Late-Fusion Model: combines semantic and structural branches.

The models are trained using cross-entropy loss with optional entropy regularization. For open-set tuning, we employ confidence-aware validation to select the optimal threshold θ\thetaθ, maximizing coverage while controlling false positives.

Benefits and Applications. By combining high-capacity classification with calibrated rejection, our system provides:

• Robustness to unknown patterns: prevents invalid recommendations;

• Confidence-aware prioritization: high-confidence predictions can be acted upon automatically;

• Human-in-the-loop support: low-confidence cases can be deferred or presented with explanation for review;

• Safer integration into CI/CD pipelines and developer workflows.

This component transforms the system from a static predictor into an interactive, self-aware recommendation agent, capable of adapting to real-world code heterogeneity and uncertainty.

3.6. Effort and benefit estimation

In practical software development, not all refactorings are equally valuable or equally costly. Developers often operate under time and resource constraints, and even correct clone refactorings may be deferred or avoided if the perceived effort outweighs the expected benefit. To support such reasoning, our system includes a dedicated module for estimating both refactoring effort and potential benefit, enabling cost-aware prioritization of clone transformations (Table 4).

Estimation Goals. Given a clone group G and a predicted refactoring type R, the module aims to compute two scores:

• $Effort(G, R)$: the estimated technical and social cost of performing the transformation;

• $Benefit(G, R)$: the estimated long-term positive impact (e.g., maintainability, defect reduction).

These values are used to derive a prioritization metric:

$$Priority(G, R) = \frac{Benefit(G, R)}{Effor(G, R)}. \quad (2)$$

This scalar is then combined with the classifier's confidence to rank clone refactoring candidates.

Effort Estimation. We define effort as a proxy for the amount of developer work required to refactor the clone group. It is approximated using the following indicators:

• Lines of code (LOC): size of the clone group;

• Cyclomatic complexity: number of decision points in the control flow;

• AST edit distance: cost of transforming the clone group to a shared abstraction;

• Code scattering: number of files/classes involved in the clone group;

• Contributor count: more authors imply higher coordination overhead;

• Change frequency (churn): frequently modified code may require conflict resolution.

$$Effort = \alpha_1 \cdot LOC + \alpha_2 \cdot Complexity + \\ + \alpha_3 \cdot Scattering + \cdots. \quad (3)$$

Weights α_i are learned or heuristically set based on validation data.

Table 4. **Indicators for Effort and Benefit Estimation**

Metric	Type	Used in	Description
LOC	Static	Effort	Number of lines in the clone group
Cyclomatic complexity	Static	Effort	Control-flow branching factor
AST edit distance	Structural	Effort	Cost to abstract clones into one shared method
Scattering (file count)	Structural	Effort	Number of locations affected
Churn	Evolutionary	Effort/Benefit	Frequency of changes over time
Co-change density	Evolutionary	Benefit	How often clone instances are changed together
Contributor count	Social	Effort	Number of unique authors in clone history
Redundancy ratio	Structural	Benefit	Proportion of duplicated logic
Bug propagation history	Evolutionary	Benefit	Clone correlation with past bug-fixing commits

Source: **compiled by the authors**

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

Benefit Estimation. We define benefit as the expected long-term improvement in code quality and maintainability. This includes:

- Clone redundancy reduction: fewer copies lead to lower maintenance effort;
- Defect propagation risk: refactoring reduces chance of bugs being copied;
- Code churn reduction: if refactored code changes less frequently afterward;
- Historical co-change density: if clones often change together, abstraction is beneficial;
- Module cohesion gain: merging clones may improve architectural clarity.

These signals are aggregated into a scalar benefit estimate via linear regression or decision trees trained on historical data (e.g., past refactorings and their outcomes).

Usage and Interpretation. The effort-benefit estimation allows the system to:

- promote low-effort, high-impact clones for immediate refactoring;
- defer high-effort or low-benefit clones for later review or manual inspection;
- avoid costly or risky refactorings that may harm stability.

This ranking supports actionable decision-making under constraints such as sprint deadlines or technical debt reduction goals.

By quantifying both technical difficulty and potential impact, the system transitions from generic recommendations to prioritized and context-aware guidance, tailored to the needs and constraints of real-world development teams.

3.7. Prioritization and explanation layer

The final stage of our system integrates the outputs of the classifier, uncertainty estimator, and effort-benefit module into a unified refactoring advisory layer.

This layer performs two key functions:

1) Prioritization: Selects and ranks clone refactoring candidates based on multiple decision criteria;

2) Explanation: Generates human-readable rationales for each recommendation to support trust, transparency, and human-in-the-loop review.

Together, these outputs transform the system from a predictive model into a practical decision-support tool suitable for integration into developer workflows.

Multi-Factor Prioritization Strategy. Each clone group G with a predicted refactoring type R is scored along four dimensions:

- $Conf(G,R)$ – model confidence (calibrated);
- $Effort(G,R)$ – estimated transformation cost;
- $Benefit(G,R)$ – expected long-term gain;
- $Risk(G)$ – uncertainty or mismatch indicator.

The final priority score is computed as:

$$Score(G) = \left(\frac{Benefit(G,R)}{Effort(G,R)}\right) \cdot \cdot Conf(G,R) \cdot \lambda(G). \quad (4)$$

where $\lambda(G) \in [0,1]$ is a penalty factor for risky or low-trust predictions (e.g., high entropy, known bug history, unknown classification); clone groups below a score threshold τ are omitted or flagged for manual review.

This prioritization strategy ensures that high-impact, low-effort, high-confidence refactorings are surfaced first, while uncertain or costly transformations are postponed or annotated for inspection.

Explanation Generation. To foster developer trust and support traceability, each recommendation is accompanied by a structured natural-language explanation, which includes:

- Refactoring suggestion: predicted transformation type;
- Confidence level: qualitative indicator (e.g., high / medium / low);
- Key features influencing decision: most salient input signals;
- Effort and benefit summary: size, complexity, potential impact;
- Optional caveats: e.g., “High churn and low survival ratio – consider reviewing manually”.

This explanation is synthesized from interpretable model features, confidence metrics, and domain heuristics.

Example Output:

- Suggested Refactoring: Extract Method;
- Confidence: High (94%);
- Why: Clone group has high structural similarity, low complexity, and appears in 3 methods across 2 files;
- Effort Estimate: Low – all clones are small and co-located;
- Benefit Estimate: High – reduces 42 duplicated lines, shared churn history indicates frequent co-editing;
- Note: Recent bugfix associated with this method – consider additional review.

4. EXPERIMENTAL SETUP

To evaluate the proposed clone refactoring advisor, we designed an empirical study that

simulates realistic development conditions across diverse software systems. This section presents the selected projects, clone extraction and labeling strategy, feature processing pipeline, model configuration, and evaluation protocol. Throughout, we integrate both quantitative and qualitative procedures to ensure rigor and reproducibility.

Subject Systems. We selected four mature, open-source Java systems that differ in functionality, scale, and development activity. These systems provide a rich variety of clone types, architectural styles, and evolutionary patterns (Table 5).

These systems were chosen to ensure broad applicability of the proposed method and to provide realistic challenges such as legacy code, high churn, and cross-team contributions.

Clone Detection and Labeling. Clone groups were extracted using the NiCad clone detector, configured at the function level with default similarity thresholds. The analysis focused on Type-1, Type-2, and Type-3 clones, which cover exact, renamed, and syntactically modified duplications. Type-4 clones (semantic clones) were excluded due to their high annotation cost and inconsistent detection quality.

To ensure high-quality input, we applied post-filtering rules to exclude:

- methods with fewer than 5 LOC;
- autogenerated boilerplate (e.g., constructors, accessors);
- trivial duplication patterns such as logging wrappers.

Next, a manual labeling phase was conducted. A subset of 600 clone groups was randomly sampled and annotated by two senior engineers.

Each group was assigned to one of the following refactoring categories:

1) Extract Method;
2) Move Method;
3) Pull Up Method;
4) Inline Method;
5) No Refactoring (Retain);
6) Unknown / Other.

Disagreements were resolved through joint review. The final dataset was stratified and split into training (60 %), validation (20 %), and test (20 %) sets, ensuring no project overlap across splits.

Feature Extraction Pipeline. The input representation for each clone group was constructed through a multi-modal feature extraction process:

- Structural Features: Extracted from ASTs using the Eclipse JDT parser, including node counts, nesting depth, and control-flow complexity;
- PDG Features: Built using the JavaPDG toolkit to represent control and data dependencies;
- Semantic Embeddings: Obtained using the CodeBERT-base model (768d), pooled over method tokens using [CLS] and average pooling strategies;
- Evolutionary Features: Computed from Git history using custom scripts, including:
  - change frequency (churn),
  - recency of last modification,
  - number of unique contributors,
  - authorship entropy,
  - line survival ratio,
  - bug co-change density (via commit message heuristics).

All numeric features were z-score normalized before fusion. The final input vector to the classifier was formed by concatenating all feature modalities.

Classifier Configuration and Training. The classifier was implemented as a multi-layer perceptron (MLP) with two hidden layers of 256 and 128 units, using ReLU activations and dropout ($p = 0.2$). The model was trained using the AdamW optimizer with a learning rate of 1×10^{-4} and early stopping on validation loss.

To enable open-set classification, we applied temperature scaling for confidence calibration. A softmax rejection threshold θ was selected using the validation set to distinguish uncertain examples assigned to the *Unknown* class.

Additionally, effort and benefit were estimated using Random Forest regressors trained on manually rated examples of clone complexity and historical impact (based on bug propagation, churn reduction, etc.).

Table 5. **Summarization of the key characteristics of the selected projects**

Method	Domain	LOC	Commits	Contributors	Notes
Apache Commons	Utility libraries	~200,000	8,000+	90+	Modular, widely reused
JHotDraw	GUI framework	~60,000	1,500+	10+	Design-pattern-intensive
PMD	Static analysis	~100,000	6,500+	60+	Frequent structural refactorings
JEdit	Text editor	~150,000	7,200+	80+	Frequent structural refactorings

Source: **compiled by the authors**

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

Evaluation Protocol. Model performance was assessed across three key tasks:

1) Classification Accuracy: We measured precision, recall, and macro-F1 across the known refactoring classes;

2) Open-Set Robustness: The system's ability to reject unknown or unsuitable clones was evaluated by computing:

- true positive rate (TPR) for *known* cases;
- false acceptance rate (FAR) on *unknown* cases;
- AUROC for threshold-based rejection.

3) Recommendation Quality: Using NDCG@k (Normalized Discounted Cumulative Gain), we evaluated how well the system prioritized high-benefit, low-effort clones in its top-k suggestions.

Three baselines were included for comparison:

- closed-set classifier without open-set rejection;
- model without evolutionary features;
- random ranking baseline for prioritization.

These baselines allow us to isolate the contribution of evolutionary context, confidence calibration, and prioritization logic.

This experimental setup ensures a comprehensive, controlled, and multi-dimensional evaluation of the system's performance in realistic software engineering scenarios. The next section presents the results and comparative analyses based on this protocol.

5. EVALUATION RESULTS

This section presents the empirical results of our proposed refactoring advisor, structured around three key evaluation criteria: classification performance, open-set robustness, and prioritization quality. We report both quantitative metrics and comparative analyses against baseline systems.

Refactoring Classification Accuracy. We first evaluate the system's ability to correctly predict the refactoring type for known clone groups. Table 6 reports the precision, recall, and F1-score for each class on the held-out test set.

The model demonstrates consistent performance across common refactoring. Notably, Extract Method is the most reliably predicted class, likely due to its structural regularity and semantic cohesion.

Open-Set Robustness. To assess the ability of the model to reject uncertain or unseen clone groups, we measured the Area Under the Receiver Operating Characteristic Curve (AUROC) for distinguishing known vs unknown classes. The calibrated model achieves an AUROC of 0.91, indicating high separability.

Moreover, we evaluated the false acceptance rate (FAR) at different rejection thresholds:

- At threshold $\theta = 0.80$, the system correctly rejects 82.4 % of unknown examples while maintaining a true acceptance rate (TAR) of 88.3 % on known cases;
- Without open-set handling, the classifier misclassifies 31 % of unknowns into incorrect refactoring types.

These results confirm that confidence calibration and softmax thresholding effectively prevent overconfident misclassifications in open-world settings.

Impact of Evolutionary Features. We conducted an ablation study to evaluate the contribution of evolutionary features. When removing version control–based signals (e.g., churn, survival, authorship entropy), the overall macro-F1 dropped from 0.76 to 0.70, and rejection performance degraded (AUROC from 0.91 to 0.83).

These results validate our hypothesis that evolution-aware context provides discriminative signals for both refactoring classification and uncertainty management.

Prioritization Effectiveness. To evaluate the quality of ranked refactorings suggestions, we used Normalized Discounted Cumulative Gain at rank k (NDCG@k). Each suggestion was scored based on its predicted benefit/effort ratio and confidence. Clone groups labeled by human experts as "high priority" served as ground truth.

As shown in Table 7, the proposed system consistently outperforms both the closed-set baseline and the random ordering baseline.

Table 6. **Classification Performance by Refactoring Type**

Refactoring Type	Precision	Recall	F1-Score
Extract Method	0.86	0.83	0.84
Move Method	0.81	0.78	0.79
Pull Up Method	0.74	0.71	0.72
Inline Method	0.69	0.66	0.67
No Refactoring	0.75	0.80	0.77
Macro Average	0.77	0.76	0.76

Source: **compiled by the authors**

Table 7. **Evaluation of the proposed method**

Method	NDCG@3	NDCG@5	NDCG@10
Proposed (full method)	0.89	0.86	0.82
No Evolutionary Features	0.81	0.77	0.73
Closed-set (no Unknowns)	0.75	0.72	0.69
Random baseline	0.42	0.39	0.35

Source: **compiled by the authors**

The gap widens at lower k, indicating that our model surfaces more relevant refactorings near the top of the list, which is critical for developer attention and actionability.

Effort vs Benefit Distribution. Finally, we analyze the relationship between the model's predicted effort and benefit. High-priority candidates cluster in the top-right quadrant (high benefit, low effort), confirming that the system learns to distinguish trivial duplications from architecturally meaningful clones.

Examples in the bottom-left quadrant are correctly demoted or rejected, as they tend to be unstable, buggy, or highly entangled, despite their apparent duplication.

Summary of Key Results:

- The classifier achieves macro-F1 = 0.76 on known classes and AUROC = 0.91 for open-set rejection;
- Evolutionary features contribute a +6 % absolute gain in classification and improved prioritization;
- The system ranks high-benefit, low-effort refactorings at the top, outperforming all baselines in NDCG@k.

These results demonstrate the system's utility as a trustworthy and context-aware refactoring advisor, capable of generalizing to realistic, noisy, and evolving codebases.

CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a machine learning–based approach for clone refactoring recommendation that goes beyond traditional clone detection. Our method combines structural, semantic, and evolutionary features to classify clone groups, estimate refactoring effort and benefit, handle previously unseen patterns via open-set classification, and produce prioritized, explainable suggestions suitable for practical use.

Empirical evaluation across several real-world Java projects demonstrated that the proposed system achieves high classification accuracy (macro-F1 of 0.76), strong robustness in rejecting uncertain clones (AUROC of 0.91), and effective ranking of actionable refactorings (NDCG@3 of 0.89). Notably, the integration of evolutionary features and effort-benefit estimation significantly improved both the quality and interpretability of the recommendations.

Our findings suggest that clone refactoring can benefit from a shift from detection to decision support, especially in large, evolving codebases. The system encourages developers to reason not only about what clones exist, but also about which ones are worth refactoring, and why.

Future Work. Several avenues remain open for further research and enhancement:

- Extension to additional refactoring types. Our current taxonomy focuses on four method-level refactorings. Future work could integrate higher-level transformations (e.g., Extract Interface, Convert Hierarchy);
- Cross-language generalization. We plan to adapt the approach to other languages (e.g., Python, C++) using multilingual embeddings and language-agnostic graph representations;
- Human-in-the-loop refinement. Integrating developer feedback through active learning or interactive explanation could further improve precision and trust;
- IDE integration and user studies. Embedding the system into real development environments and conducting longitudinal studies would provide deeper insights into adoption, usability, and impact;
- Confidence modeling under shift. Improving uncertainty estimation under codebase evolution or domain shift remains a challenge for open-set learning in software engineering.

By releasing our dataset, source code, and tooling, we hope to foster future work on intelligent, transparent, and context-aware support for clone management and software maintenance more broadly.

ISSN 2663-0176 (Print)
ISSN 2663-7731 (Online)

REFERENCES

1. Wang, W. & Godfrey, M. W. “Recommending clones for refactoring using design, Context, and History”. *Proceedings of ICSME*. 2014. DOI: https://doi.org/10.1109/ICSME.2014.55.

2. Yue, R., Gao, Z., Meng, N., Xiong, Y., Wang, X. & Morgenthaler, J. D. “Automatic clone recommendation for refactoring based on the present and the past”. *arXiv*. 2018. DOI: https://doi.org/10.48550/arXiv.1807.11184.

3. Wang, W., Li, G., Ma, B., Xia, X. & Jin, Z. “Detecting code clones with graph neural network and flow-augmented abstract syntax tree”. *Proceedings of SANER*. 2020. DOI: https://doi.org/10.1109/SANER48275.2020.9054857.

4. Zhang, Y., Yang, J., Dong, H., Wang, Q., Shao, H., Leach, K. & Huang, Y. “ASTRO: An AST-assisted approach for generalizable neural clone detection”. *arXiv*. 2022. DOI: https://doi.org/10.48550/arXiv.2208.08067.

5. Xue, Z., Jiang, Z., Huang, C., Xu, R., Huang, X. & Hu, L. “SEED: Semantic graph based deep detection for Type-4 Clone”. *arXiv*. 2021. DOI: https://doi.org/10.48550/arXiv.2109.12079.

6. Lei, M., Li, J., Peng, X., Xing, Z., Liu, J. & Wang, X. “Deep learning application on code clone detection: A systematic review”. *Journal of Systems and Software*. 2022; 185: 111211. DOI: https://doi.org/10.1016/j.jss.2021.111211.

7. Nair, A., Roy, A. & Meinke, K. “funcGNN: A graph neural network approach to program similarity”. *arXiv*. 2020. DOI: https://doi.org/10.48550/arXiv.2007.13239.

8. Zubkov, M., Spirin, E., Bogomolov, E. & Bryksin, T. “Evaluation of contrastive learning with various code representations for code clone detection”. *arXiv*. 2022. DOI: https://doi.org/10.48550/arXiv.2206.08726.

9. Choi, E., Yoshida, N. & Inoue, K. “What kind of and how clones are refactored? A case study of three OSS projects”. *Proc. ICSE Workshops (WRT)*. 2012. DOI: https://doi.org/10.1145/2328876.2328877.

10. Kanwal, J., Maqbool, O., Basit, H. A., Sindhu, M. A. & Inoue, K. “Historical perspective of code clone refactorings in evolving software”. *PLoS ONE*. 2022; 17 (12): e0277216. DOI: https://doi.org/10.1371/journal.pone.0277216.

11. Thompson, S., Li, H. & Schumacher, A. “The pragmatics of clone detection and elimination”. *arXiv*. 2017. DOI: https://doi.org/10.48550/arXiv.1703.10860.

12. Kurbatova, Z., Veselov, I., Golubev, Y. & Bryksin, T. “Recommendation of move method refactoring using path-based representation of code”. *arXiv*. 2020. DOI: https://doi.org/10.48550/arXiv.2002.06392.

13. Wagner, S., Abdulkhaleq, A., Kaya, K. & Paar, A. “On the relationship of inconsistent software clones and faults: An Empirical Study”. *arXiv*. 2016. DOI: https://doi.org/10.48550/arXiv.1611.08005.

14. Mondal, M., Roy, C. K. & Schneider, K. A. “A survey on clone refactoring and tracking”. *Journal of Systems and Software*. 2020; 159: 110429. DOI: https://doi.org/10.1016/j.jss.2019.110429.

15. Yang, J., Igaki, H., Hotta, K., Higo, Y. & Kusumoto, S. “Classification model for code clones based on machine learning”. *Empirical Software Engineering*. 2015; 20 (4): 1095–1125. DOI: https://doi.org/10.1007/s10664-014-9316-x.

16. Mens, T. & Tourwé, T. “A survey of software refactoring”. *IEEE Transactions on Software Engineering*. 2004; 30 (2): 126–139. DOI: https://doi.org/10.1109/TSE.2004.1265817.

17. Choi, E., Fujiwara, K., Yoshida, N. & Hayashi, S. “A survey of refactoring detection techniques based on change history analysis”. *arXiv*. 2018. DOI: https://doi.org/10.48550/arXiv.1808.02320.

18. Golubev, Y., Kurbatova, Z., AlOmar, E., Bryksin, T. & Mkaouer, M. W. “One thousand and one stories: A Large-Scale survey of software refactoring”. *arXiv*. 2021. DOI: https://doi.org/10.48550/arXiv.2107.07357.

19. Kaur, M., Rattan, M. & Verma, S. “A systematic literature review on the use of machine learning in code clone related research areas”. *Journal of Systems Architecture*. 2023; 139: 102844. DOI: https://doi.org/10.1016/j.sysarc.2023.102844.

20. Roy, C. K., Zibran, M. F. & Koschke, R. “The vision of software clone management: Past, Present, and Future”. *arXiv*. 2020. DOI: https://doi.org/10.48550/arXiv.2005.01005.

21. Schulze, S., Kuhlemann, M. & Rosenmüller, M. “Towards a refactoring guideline using code clone classification”. *Proceedings of AOSD*. 2008. DOI: https://doi.org/10.1145/1636642.1636648.

22. AlOmar, E. A., Ashkenas, J., Feliciano, R., et al. “AntiCopyPaster 3.0: Just-in-Time Clone Refactoring”. *ACM Trans. Softw. Eng. Methodol*. 2025. DOI: https://doi.org/10.1145/3749100.

23. Higo, Y., Sawa, Y. & Kusumoto, S. “Problematic code clones identification using multiple detection results”. *Proceedings of APSEC*. 2009. DOI: https://doi.org/10.1109/APSEC.2009.30.

24. Sheneamer, A. M. “An automatic advisor for refactoring software clones based on machine learning”. *IEEE Access*. 2020; 8: 124978–124988. DOI: https://doi.org/10.1109/ACCESS.2020.3006178.

25. Mostaeen, G., Islam, M. R., Roy, C. K. & Schneider, K. A. “A machine learning based framework for code clone validation”. *Journal of Systems and Software*. 2020; 170: 110740. DOI: https://doi.org/10.1016/j.jss.2020.110740.

26. Sajnani, V., Saini, V., Svajlenko, J., Roy, C. K. & Lopes, C. V. “SourcererCC: Scalable Near-Miss Clone Detection”. *Proceedings of ICSE*. 2016. DOI: https://doi.org/10.1145/2884781.2884877.

27. Jiang, L., Misherghi, G., Su, Z. & Glondu, S. “DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones”. *Proceedings of ICSE*. 2007. DOI: https://doi.org/10.1109/ICSE.2007.30.

28. Kamiya, T., Kusumoto, S. & Inoue, K. “CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code”. *IEEE Transactions on Software Engineering*. 2002; 28 (7): 654–670. DOI: https://doi.org/10.1109/TSE.2002.1000449.

29. Roy, C. K. & Cordy, J. R. “NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization”. *Proceedings of CSMR*. 2008. DOI: https://doi.org/10.1109/CSMR.2008.4493319.

Conflicts of Interest: The authors declare that they have no conflict of interest regarding this study, including financial, personal, authorship or other, which could influence the research and its results presented in this article

Received 18.08.2025
Received after revision 22.09.2025
Accepted 26.09.2025

DOI: https://doi.org/10.15276/hait.08.2025.19
УДК 004.021:004.4’42:004.8

Рекомендація рефакторингів із багатоцільовою оптимізацією та урахуванням невизначеності для дублювання коду

Курінько Дмитро Дмитрович[1)]
ORCID: https://orcid.org/0000-0001-8304-3257; dmitrykurinko@gmail.com
Кривда Вікторія Ігорівна[1)]
ORCID: https://orcid.org/0000-0002-0930-1163; kryvda@op.edu.ua
[1)] Національний університет «Одеська Політехніка», пр. Шевченка, 1. Одеса, 65044, Україна

АНОТАЦІЯ

Клоновані фрагменти коду (code clones) – це повторювані ділянки програмного коду, які можуть ускладнювати підтримку програмного забезпечення за відсутності належного управління. Існує багато інструментів для виявлення клонів, однак більшість з них обмежується лише фактом виявлення та не надає рекомендацій щодо доцільності, способу чи черговості їх рефакторингу. У цій статті запропоновано метод машинного навчання для надання рекомендацій з рефакторингу клонів з урахуванням пріоритетності та оцінки впевненості. Запропонований підхід використовує комбіноване представлення коду: абстрактні синтаксичні дерева (AST), графи залежностей програми (PDG) та семантичні вектори, отримані за допомогою попередньо натренованої моделі CodeBERT. Додатково використовуються еволюційні ознаки з системи контролю версій, зокрема частота змін, вік фрагмента та співзміни з іншими файлами. Класифікатор багато-класової моделі прогнозує тип рефакторингу, а механізм відкритих класів дозволяє відхиляти невизначені або невідомі випадки. Оцінка зусиль та користі дозволяє впорядковувати рекомендації за ефективністю.

Експериментальна перевірка на чотирьох open-source проєктах на Java з вручну розміченими 600 групами клонів показала досягнення macro-F1 = 0.76 для відомих типів рефакторингу, AUROC = 0.91 для виявлення невідомих випадків та NDCG@3 = 0.89 для якості пріоритезації. Отримані результати демонструють, що рефакторинг клонів може бути ефективно

підтриманий завдяки поєднанню структурних і семантичних ознак, моделювання невизначеності та механізмів пріоритезації. Запропонований підхід трансформує клон-аналіз із пасивного виявлення у повноцінну систему підтримки рішень.

Ключові слова: Рефакторинг клонів; штучний інтелект у програмній інженерії; машинне навчання; глибинне навчання; класифікація клонів; відкрите розпізнавання; оцінка невизначеності

ABOUT THE AUTHORS

Dmytro D. Kurinko - PhD Student, Artificial Intelligence and Data Analysis Department, Odesa Polytechnic National University, 1, Shevchenko Ave. Odesa, 65044
ORCID: https://orcid.org/0000-0001-8304-3257; dmitrykurinko@gmail.com
***Research field*:** Machine Learning and Artificial Intelligence, Machine Learning for Software Engineering, Pattern Recognition, Computer Vision, Knowledge Representation in Software Systems

Курінько Дмитро Дмитрович - аспірант кафедри Штучного інтелекту та аналізу даних, Національний університет «Одеська Політехніка», пр. Шевченка, 1. Одеса, 65044, Україна

Viktoriia I. Kryvda - PhD, Associate Professor, Department of Electricity and Energy Management, Head of Department of Postgraduate and Doctoral Studies, Odesa Polytechnic National University, 1, Shevchenko Ave. Odesa, 65044
ORCID: https://orcid.org/0000-0002-0930-1163, kryvda@op.edu.ua
***Research field*:** Machine Learning for Software Engineering, Pattern Recognition, Data-Driven Software Architecture Analysis, Computer Vision, Knowledge Representation in Software Systems

Кривда Вікторія Ігорівна - канд. техніч. наук, доцент кафедри Електропостачання та енергетичного менеджменту, завідувач відділу аспірантури і докторантури, Національний університет «Одеська Політехніка», пр. Шевченка, 1. Одеса, 65044, Україна