Semi-Supervised Code Clustering Using Embedding Representations

Kuang Ren
Department of Mechanical & Industrial Engineering
University of Toronto

MIE8888 Project Report*

Abstract

Labeling source code for machine learning tasks is expensive and time-consuming, creating a significant barrier to supervised learning approaches. This paper investigates semi-supervised learning (SSL) techniques for clustering code by algorithm, presenting a comparative analysis of their performance when combined with modern code embeddings. We evaluate two graph-based SSL methods—Label Propagation and Label Spreading—within an active learning framework that employs multiple query strategies (random, entropy-based, class-uniform) to minimize labeling effort. Our experiments are conducted using five state-of-the-art code embedding models (CodeBERT, GraphCodeBERT, UniXcoder, CodeT5, InCoder) on a dataset of over 12,000 Java and C/C++ functions. The results demonstrate that the effectiveness of active learning is critically dependent on the choice of code embedding. Structurally-aware models (GraphCodeBERT, UniXcoder) consistently benefit from uncertainty-based sampling, whereas for sequential models (CodeBERT, CodeT5, InCoder), random sampling surprisingly outperforms active learning. Furthermore, we identify that enforcing class balance, deferring the onset of active learning, and using Label Propagation are key to optimal performance. Our findings provide crucial insights for building efficient, low-resource code understanding systems and challenge the universal applicability of active learning in semantic code spaces.

*I am deeply grateful to my supervisor, Professor Eldan Cohen, for his invaluable guidance, unwavering patience, and insightful feedback throughout this project.

1 Introduction

The rapid evolution of software systems has generated vast repositories of source code, creating unprecedented opportunities for applying machine learning to tasks like code classification, summarization, and clone detection. A fundamental prerequisite for these tasks is the ability to group code snippets by their functionality, a process known as code clustering. However, a major bottleneck in applying supervised learning to this problem is the prohibitive cost and expertise required to create large, accurately labeled datasets. This challenge is particularly acute in programming languages, where semantic labeling demands specialized knowledge.

Semi-supervised learning (SSL) offers a promising pathway to leverage the abundance of unlabeled code alongside a small set of expensive labeled examples. Among SSL paradigms, active learning is especially appealing for code clustering. It iteratively selects the most informative samples for a human to label, aiming to achieve high accuracy with minimal annotation cost. Concurrently, the field of code representation learning has been revolutionized by pre-trained embedding models that encode syntactic and semantic information into dense vectors. Models like CodeBERT and GraphCodeBERT have shown remarkable success by treating code as a form of language, either as a flat token sequence or by incorporating structural elements like Abstract Syntax Trees (ASTs) and Data Flow Graphs (DFGs).

Despite these advancements, the synergistic effect of combining different code embeddings with various SSL and active learning strategies remains poorly understood. It is often assumed that uncertainty-based active learning will invariably outperform random sampling, and that more advanced embeddings will linearly improve performance. However, our investigation reveals a more nuanced reality. The performance of the entire system is governed by a complex interaction between the embedding's architectural biases (structural vs. sequential), the graph-based propagation algorithm, and the sampling strategy.

This paper makes the following contributions: (1) We present a large-scale empirical evaluation of SSL for code clustering across five diverse embedding models and two programming languages. (2) We demonstrate that the superiority of active learning is not guaranteed; it critically depends on the embedding model, with random sampling outperforming entropy-based methods for several prominent models. (3) We identify and analyze key factors for success, including the necessity of class-balanced sampling, the optimal timing for initiating active learning, and the consistent advantage of Label Propagation over Label Spreading. (4) We provide evidence that Java's structural regularity leads to significantly better clustering performance compared to C, highlighting the impact of language-level features on representation learning. Our work offers empirical insights and practical guidance for researchers exploring semi-supervised learning approaches to code understanding in low-resource settings.

2 Background

2.1 Semi-Supervised Learning Methods

Semi-supervised learning (SSL) methods aim to leverage both a small set of labeled data and a large set of unlabeled data to improve model performance. In the context of function clustering from code embeddings, SSL is particularly valuable because labeling source code is expensive and time-consuming, while obtaining raw code is relatively easy. This section describes three core SSL approaches used in our study: Label Propagation, Label Spreading, and Active Learning.

2.1.1 Active Learning

Active learning (AL) is a semi-supervised machine learning paradigm in which the model iteratively selects the most informative samples from an unlabeled pool for annotation, with the goal of achieving high accuracy using fewer labeled instances. Starting with a limited set of labeled examples, the model identifies and queries labels for the samples deemed most informative. These newly annotated instances are incorporated into the training set, and the model is retrained, progressively enhancing its predictive performance across successive iterations[3]. As Burr Settles (2009) puts it, AL is useful when "unlabeled data is abundant, but labels are expensive to obtain" [21]. In this project, we apply active learning to improve clustering performance in a semi-supervised setting. Our approach begins with a small set of labeled examples per class, and at each iteration, the model queries additional samples according to a specific selection strategy (e.g., entropy-based, class-uniform sampling, pair-wise distance). This process continues until the target number of labeled samples is reached, and evaluation is performed using metrics such as accuracy, F1-score, ARI, and NMI.

2.1.2 Label Propagation

Label Propagation (LP) [26] is a classical graph-based semi-supervised learning (SSL) method that leverages the manifold assumption: samples that are close to each other in the feature space, or strongly connected in a similarity graph, are likely to share the same label.

Graph construction. The algorithm begins by representing the dataset as an undirected weighted graph $G = (V, E)$, where each node $v_i \in V$ corresponds to a sample x_i. The edges E encode pairwise similarities between samples, with weights given by an affinity matrix $W \in \mathbb{R}^{n \times n}$. A common choice is the Radial Basis Function (RBF) kernel:

$$w_{ij} = \exp\left(-\sum_{d=1}^{m} \frac{(x_{id} - x_{jd})^2}{\sigma_d^2}\right)$$

or cosine similarity for embedding vectors. The degree matrix D is diagonal with $D_{ii} = \sum_j W_{ij}$.

Label propagation process. Let $Y \in \mathbb{R}^{n \times c}$ be the label matrix, where n is the number of samples and c is the number of classes. For labeled samples, $Y_{ik} = 1$ if sample i belongs to

class k and 0 otherwise. For unlabeled samples, all entries are initially zero. The algorithm maintains a label distribution matrix F (initially set to Y) and iteratively updates it by averaging labels from neighbors:

$$F^{(t+1)} = D^{-1}WF^{(t)}$$

After each update, the rows of F corresponding to labeled samples are reset to their original one-hot label vectors (the *clamping step*). This ensures that labeled data remain fixed "sources" of label information during propagation.

Convergence, output, and trade-offs Label Propagation is a simple, non-parametric method that effectively exploits the geometry of the data manifold. The iterative process continues until F converges (changes between iterations fall below a threshold) or a maximum number of iterations is reached, and the final predicted labels for each node are obtained by taking the class with the highest probability in F. However, its reliance on clamping means that it *fully trusts* the initial labeled set: if a labeled sample is incorrect, the error will propagate through the graph and cannot be corrected, making the method sensitive to label noise and potentially degrading performance in noisy or weakly labeled settings.

2.1.3 Label Spreading

Label Spreading (LS) [25] is a graph-based semi-supervised learning method closely related to Label Propagation, but with a key modification: instead of keeping the labels of the initially labeled points fixed, it allows them to be updated slightly during the iterations. This makes the method more robust to label noise, as incorrect initial labels can be gradually corrected.

Graph construction. As in Label Propagation, the dataset is represented as an undirected weighted graph $G = (V, E)$, with nodes corresponding to samples and edges encoding similarities. The affinity matrix W can be computed using the RBF kernel:

$$w_{ij} = \exp\left(-\sum_{d=1}^{m} \frac{(x_{id} - x_{jd})^2}{\sigma_d^2}\right)$$

or alternative similarity measures such as cosine similarity. The degree matrix D is diagonal with $D_{ii} = \sum_j W_{ij}$, and W is normalized to produce a probability transition matrix S.

Label spreading process. Let $Y \in \mathbb{R}^{n \times c}$ be the initial label matrix, defined as in Label Propagation. LS updates the label distribution matrix F using:

$$F^{(t+1)} = \alpha SF^{(t)} + (1 - \alpha)Y,$$

where $\alpha \in (0, 1)$ controls the trade-off between label propagation from neighbors (α term) and retention of the original labels ($1 - \alpha$ term). Unlike LP, the clamping step is removed — initial labeled points are not forced to remain fixed, enabling correction of mislabeled data.

Convergence, output, and trade-offs Label Spreading iteratively updates the label distribution F until convergence or a maximum number of iterations is reached, with final predicted labels obtained by taking the class with the maximum value in each row of F. This method retains the manifold-exploiting benefits of Label Propagation while adding robustness to label noise by relaxing the fixed-label constraint; however, this flexibility can slightly reduce the influence of perfectly correct labeled data, as their label distributions are allowed to drift during updates.

2.2 Code Embeddings

Code embeddings are dense vector representations of source code that encode both syntactic and semantic information. Such representations make it possible to apply clustering, classification, and retrieval methods on code in a manner analogous to natural language processing. In our workflow, individual functions are first transformed into fixed-length embeddings and then grouped based on similarity in the embedding space. We extract embeddings using the `[CLS]` token representation unless otherwise noted, ensuring that each code snippet is represented by a single holistic vector.

To ensure robustness, we employ five state-of-the-art embedding models, each capturing different aspects of code semantics and structure: CodeBERT, GraphCodeBERT, UniXcoder, CodeT5, and InCoder. These models differ in architecture, training objectives, and how they incorporate structural information such as Abstract Syntax Trees (ASTs) or Data Flow Graphs (DFGs). Below, we provide a concise yet detailed overview of each.

2.2.1 CodeBERT

CodeBERT [9] is a bimodal pre-trained language model designed to jointly understand natural language (NL) and programming language (PL) representations. Architecturally, it adopts the same 12-layer bidirectional Transformer structure as RoBERTa-base [17], with approximately 125 million parameters. The model is trained on both NL–code pairs (bimodal data) and standalone code or NL text (unimodal data) collected from high-quality, non-fork GitHub repositories across six programming languages: Python, Java, JavaScript, PHP, Ruby, and Go, using the CodeSearchNet dataset [15].

In the pre-training phase, the input is constructed as the concatenation of two segments with a special separator token:

$$[\text{CLS}], w_1, w_2, \ldots, w_n, [\text{SEP}], c_1, c_2, \ldots, c_m, [\text{EOS}],$$

where one segment is natural language text and the other is source code. The `[CLS]` token, placed at the beginning, is used as an aggregated sequence representation for classification or ranking tasks. The output of CodeBERT consists of: (1) contextual vector representations for each token in both NL and PL sequences, and (2) the final hidden state of the `[CLS]` token, which serves as the aggregate representation.

The pre-training process combines two objectives: (i) **Masked Language Modeling (MLM)**, where 15% of tokens in both NL and PL segments are masked and predicted from context;

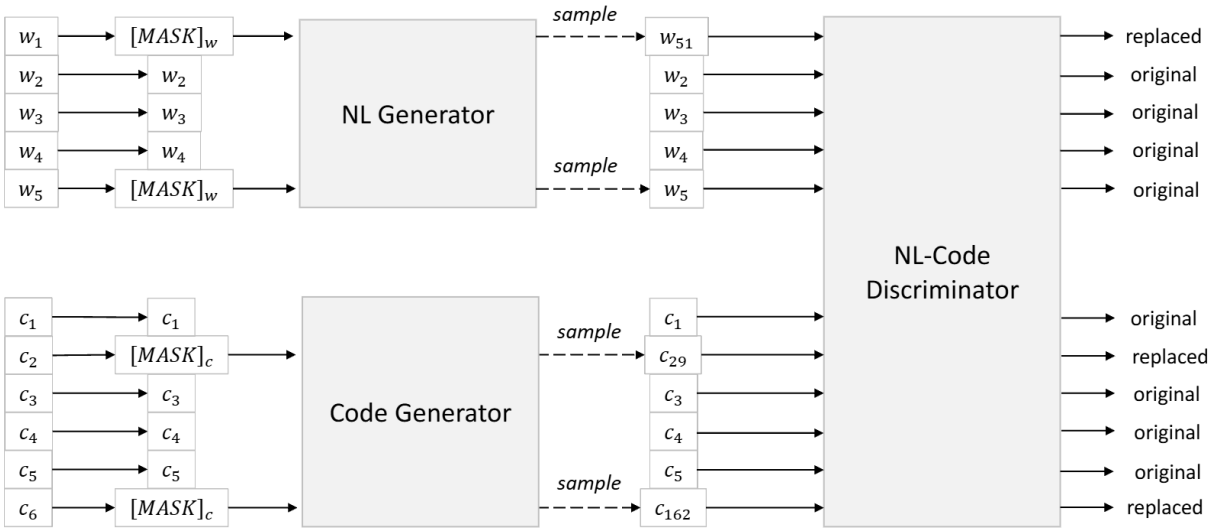


Figure 1: CodeBERT Architecture

and (ii) **Replaced Token Detection (RTD)**, where selected tokens are replaced with plausible alternatives generated by language models, and the task is to classify each token as original or replaced. These objectives enable the model to leverage both bimodal and large-scale unimodal datasets effectively.

2.2.2 GraphCodeBERT

GraphCodeBERT [14] extends CodeBERT by incorporating explicit program structure through **data flow graphs (DFGs)**. Unlike CodeBERT, which treats code as a flat token sequence, GraphCodeBERT augments the Transformer encoder with edges capturing variable definitions, usages, and updates, enabling better modeling of semantic relationships in code.

The model uses a 12-layer bidirectional Transformer (768 hidden dimensions, 12 attention heads) with an *edge-aware* self-attention mask that connects tokens both sequentially and via their DFG links. To construct the DFG, a source code snippet $C = \{c_1, c_2, \ldots, c_n\}$ is parsed into an **abstract syntax tree (AST)** using a compiler tool. From the AST, the *variable sequence* $V = \{v_1, v_2, \ldots, v_k\}$ is extracted, where each variable is a graph node. A directed edge $\varepsilon = \langle v_i, v_j \rangle$ indicates that v_j's value depends on v_i; for example, in `x = expr`, edges are added from all variables in `expr` to `x`.

The pre-training dataset (CodeSearchNet) and input are adopted following the setup of CodeBERT:

$$[\text{CLS}], w_1, w_2, \ldots, w_n, [\text{SEP}], c_1, c_2, \ldots, c_m, [\text{EOS}],$$

with the code sequence paired with its DFG. Pre-training combines: (i) **Masked Language Modeling (MLM)** for token prediction, (ii) **Edge Prediction** to recover missing graph edges, and (iii) **Node Alignment** between code tokens and graph nodes. This joint modeling of lexical and structural information produces embeddings that capture both syntax and semantics of programs.

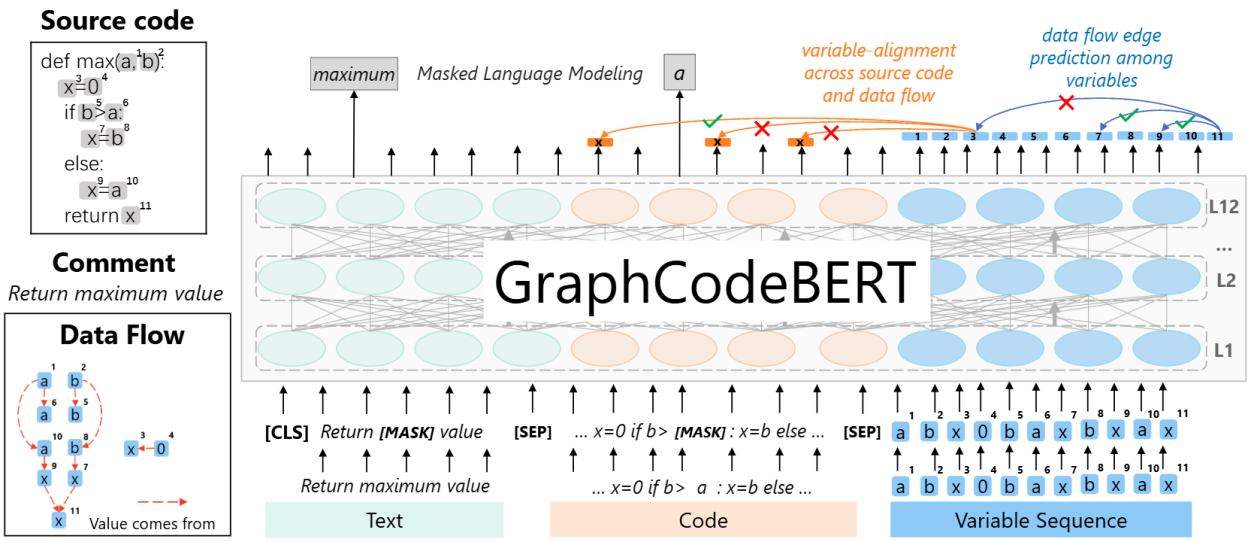


Figure 2: GraphCodeBERT Architecture

2.2.3 UniXcoder

UniXcoder [13] is a unified cross-modal pre-trained model designed to represent code by jointly leveraging multiple modalities, namely code comments and **abstract syntax trees (ASTs)**. Architecturally, UniXcoder is based on the Transformer and adopts a 12-layer encoder–decoder framework with 768-dimensional hidden states and 12 attention heads. It employs mask attention matrices [6] together with prefix adapters to control the behavior of the model.

A key innovation of UniXcoder lies in its **AST mapping function** F (Algorithm 1), which transforms a tree-structured AST into a flattened token sequence that preserves the complete syntactic structure through a one-to-one mapping. For example, the AST subtree "parameters $\rightarrow$ (data)" is mapped to the sequence "`<parameters,left>` (data) `<parameters,right>`".

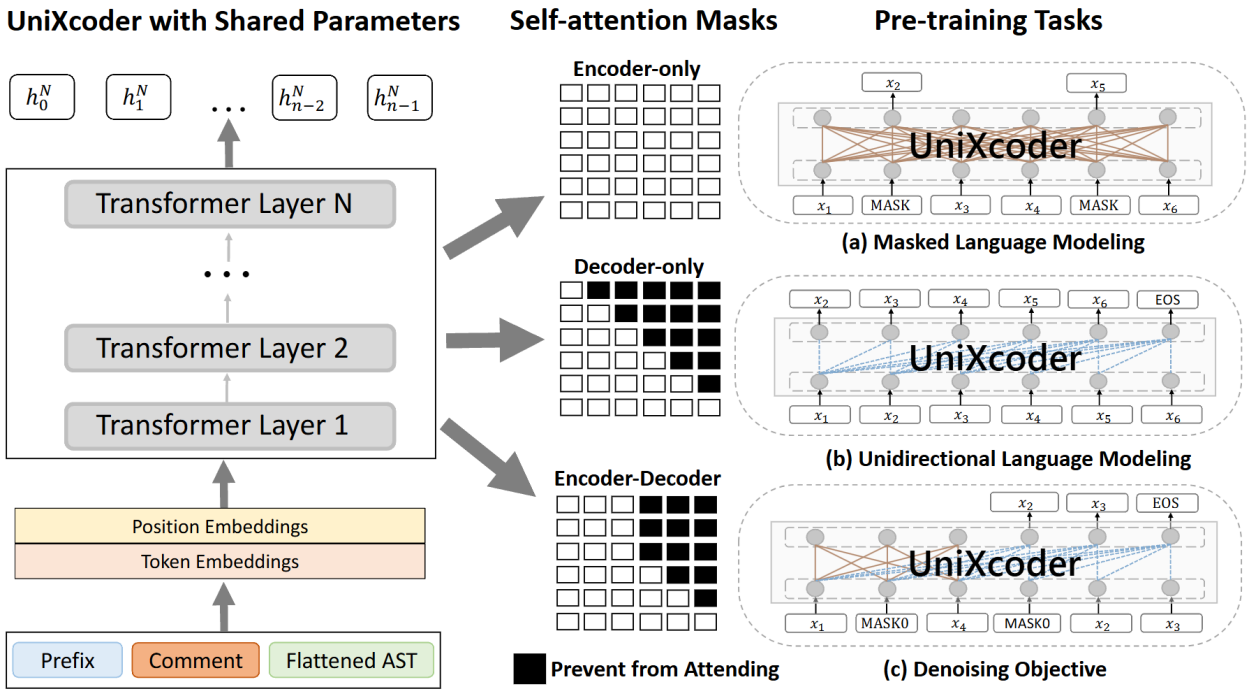


Figure 3: Architecture of UniXcoder [13]. The model supports encoder-only, decoder-only, and encoder-decoder modes using prefix tokens and mode-specific self-attention masks. Inputs consist of code comments and flattened AST sequences.

Algorithm 1: AST Mapping Function F

Input: The root node `root` of the AST
Output: A flattened token sequence
function `F(`*root*`)`**:**

```
seq ← empty list;
name ← name of root;
if root is a leaf then
    seq.append(name);
else
    seq.append(name::left);
    foreach child in children of root do
        seq.extend(F(child));
    seq.append(name::right);
```

The input to UniXcoder concatenates a mode-specific prefix $p \in [\text{Enc}], [\text{Dec}], [\text{E2D}]$, a code comment $W = w_0, \ldots, w_{m-1}$, and the flattened AST sequence $F(T(C)) = c_0, \ldots, c_{k-1}$. The self-attention mask $M \in \mathbb{R}^{n \times n}$ is mode-dependent: fully unmasked for encoder-only, causal for decoder-only, and hybrid for encoder–decoder. As shown in Figure 3, N Transformer layers process this input. Each layer applies multi-headed self-attention [22] and a feed-forward network.

Pre-training UniXcoder is a unified pre-trained model that supports encoder-only, decoder-only, and encoder-decoder modes, allowing it to handle a wide range of code-related tasks through three distinct training objectives:

- **Encoder-only mode**: Uses masked language modeling (MLM) [5]. Here, 15% of the tokens are selected for prediction, with 80% replaced by the special `[MASK]` token, 10% replaced by a random token, and 10% left unchanged. This bidirectional training captures both semantic and syntactic information from code comments and the abstract syntax tree (AST).
- **Decoder-only mode**: Applies unidirectional language modeling (ULM) [19]. Each token is predicted based only on the preceding tokens using a causal attention mask. This enables autoregressive tasks such as code completion.
- **Encoder-decoder mode**: Employs a denoising objective (DNS) [20, 16]. The input sequence is divided into chunks, random spans are masked, and the model learns to reconstruct them. This setup supports tasks like code summarization and code translation.

Beyond these pre-training objectives, UniXcoder integrates **multi-modal code representation learning**. Multi-modal contrastive learning (MCL) [12] aligns embeddings from different modalities, while cross-modal generation (CMG) generates code comments, effectively fusing semantic information into the learned code embeddings.

2.2.4 CodeT5

CodeT5 [24] is a unified encoder-decoder Transformer model based on the T5 architecture [20], designed for both code understanding and generation tasks. Unlike encoder-only models like CodeBERT, CodeT5's sequence-to-sequence framework supports bidirectional conversion between natural language (NL) and programming languages (PL).

Input CodeT5 adopts a standard T5 encoder-decoder structure (12 layers, 768 hidden dimensions, 12 attention heads). Input sequences are formatted with special tokens:

- *Bimodal inputs* (NL-PL): $x =$ `[CLS]`$, w_1, \ldots, w_n,$ `[SEP]`$, c_1, \ldots, c_m,$ `[SEP]`
- *Unimodal inputs* (PL-only): Omit NL segment

where w_i = NL tokens, c_j = PL tokens. To capture code-specific semantics, the PL segment is parsed into an **abstract syntax tree (AST)** to generate a binary sequence $y \in \{0,1\}^m$ where $y_i = 1$ indicates c_i is a developer-assigned identifier (e.g., variables/functions). This identifier-type information is embedded and combined with token embeddings.

Training CodeT5 employs two complementary pre-training strategies to learn robust representations from both unimodal (PL-only) and bimodal (NL–PL) data, following CodeBERT in using CodeSearchNet as the pre-training dataset.

- **Identifier-aware Denoising**: Extends standard denoising with code-specific enhancements:

- Masked Span Prediction (**MSP**): Randomly masks token spans (15% corruption rate, average 3-token length) and reconstructs original text. Uses whole-word masking to avoid partial sub-token corruption.
- Identifier Tagging (**IT**): Classifies each token as identifier/non-identifier through sequence labeling. Teaches the model to recognize developer-assigned elements like variables and functions.
- Masked Identifier Prediction (**MIP**): Obfuscates all identifier occurrences with unique sentinels and reconstructs original names. Requires understanding semantic relationships between identifiers.

These three tasks are optimized alternately with equal probability.

- **Bimodal Dual Generation**: Bridges the gap between pre-training and downstream tasks through bidirectional conversion. The model is trained on NL→PL and PL→NL generation as dual tasks, using the same NL-PL bimodal data in both directions. Each instance is tagged with language IDs (e.g., `<java>`, `<en>`) and treated as a special case of T5's span masking, where the entire NL or PL segment is masked to enhance NL–PL alignment.

After pre-training, CodeT5 is adapted to downstream tasks via **task-specific transfer** or **multi-task learning**. For generation tasks, it uses its Seq2Seq framework; for understanding tasks, it either generates the label as a single token or classifies from the last decoder state. Multi-task learning trains a shared model with task control codes (e.g., "Translate Java to CSharp:") and balanced sampling ($\alpha = 0.7$) to reduce bias, improving generalization and efficiency across diverse code tasks.

2.2.5 InCoder

InCoder [10] is a decoder-only Transformer model based on the dense 6.7B architecture described in Artetxe et al. [1], featuring 32 layers and a hidden dimension of 4096.

Training InCoder is pre-trained on two main sources: (1) public code with permissive, non-copyleft open-source licenses from GitHub and GitLab, and (2) StackOverflow questions, answers, and comments. Although the primary focus is on Python, the corpus includes code from 28 programming languages and StackOverflow content in all available languages. To prevent data leakage, any overlap with evaluation datasets is removed. The nine most represented languages are Python, JavaScript, HTML, C, C++, Java, Jupyter Notebook, TypeScript, and Go, with Python and JavaScript being the largest.

The model employs a causal masking objective that combines the benefits of autoregressive and masked language modeling. During training, contiguous spans of tokens are randomly selected and replaced with a special mask token `<Mask:k>`. These removed spans are appended to the end of the document with an end-of-mask token `<EOM>` to mark their boundaries. InCoder is trained to maximize the likelihood of the modified sequence in a left-to-right fashion using cross-entropy loss on all tokens except the mask tokens. This setup allows the model to both infill masked regions and autoregressively generate code.

3 Experimental Setup

3.1 Dataset Description

Our dataset is derived from the code corpus used in the work by Bui et al. on cross-language program classification [2, 18]. It contains function-level implementations of ten classic algorithms—*breadth-first search* (`bfs`), *bubble sort* (`bs`), *depth-first search* (`dfs`), *heap* (`heap`), *knapsack* (`kns`), *linked list* (`ll`), *merge sort* (`ms`), *queue* (`queue`), *quick sort* (`qs`), and *stack* (`stack`)—written in two programming languages: Java and C++/c.

- **Java code samples**: 5,874 function implementations
- **C++/c code samples**: 6,531 function implementations
- **Total**: 12,405 samples across ten algorithmic categories

Table 1 summarizes the distribution of samples per algorithm and programming language.

Language	bfs	bs	dfs	kns	ll	ms	heap	queue	qs	stack	Total
Java	630	631	644	630	630	631	317	459	630	622	5,874
C++	630	631	602	630	630	631	817	861	630	959	6,531

Table 1: Number of code samples per programming language and algorithm.

The dataset is well-balanced acros, with approximately 500–630 samples per combination, facilitating robust evaluation of cross-language clustering and classification performance. It serves as the foundation for assessing how well code embeddings capture algorithmic structure across different languages.

Process For each programming language, we organized the code samples by algorithmic category and generated vector representations using four pre-trained models: CodeBERT, GraphCodeBERT, UniXcoder, and CodeT5, as well as InCoder for causal language modeling. The code files were tokenized and truncated to a maximum length of 512 tokens. For CodeBERT, GraphCodeBERT, and CodeT5, embeddings were obtained from the final hidden states, with either the [CLS] token or mean-pooled token representations used as the fixed-size embedding for each function. UniXcoder embeddings were extracted using its encoder-only mode, while InCoder embeddings were obtained by averaging the last-layer hidden states, masking padding tokens to ensure correct vector representations. All embeddings were stored in a DataFrame alongside their corresponding algorithm labels and saved to disk for subsequent clustering and clustering experiments.

3.2 Clustering Framework

We implement an active learning framework based on semi-supervised learning using either *Label Propagation* or *Label Spreading* with an RBF kernel. The process begins by randomly selecting a small number of labeled samples per class (e.g., 5), while the remaining data are treated as unlabeled. At each iteration, the model is trained on this partially labeled dataset

to infer label distributions for all instances. Unlabeled samples are then selected for labeling according to one of the following strategies:

- **Random**: uniformly samples from the pool of unlabeled data.
- **Entropy-based**: queries instances with the highest predictive entropy, i.e., those for which the model is most uncertain.
- **Class-uniform**: ensures balanced selection across classes by querying the highest-entropy samples within each class.
- **Hybrid**: initially applies random sampling, then transitions to entropy-based selection once a predefined labeling budget (e.g., 100 or 250 samples) is reached, thereby combining class balance with uncertainty sampling.

We hypothesize that the more informed strategies (e.g., entropy-based, class-uniform, and hybrid) will yield superior performance compared to purely random sampling.stones (50, 100, 250, and 500 labeled samples). To balance efficiency and stability in the active learning process, we employ a dynamic batch size schedule. Initially, queries are made in small batches of 10 samples to minimize noise and ensure that early model updates are guided by high-quality labels. As the number of labeled samples grows, the batch size is progressively increased (from 10 to 30 and finally to 50) once predefined labeling milestones (50, 100, and 250 labeled samples) are reached. This adaptive strategy allows the model to benefit from fine-grained exploration in the early stages while improving computational efficiency and scalability in later stages.

The metrics we measured include *Accuracy*, *Adjusted Rand Index (ARI)*, *Normalized Mutual Information (NMI)*, *F1-score*, and *Recall*, with *Accuracy* being the most prominent. All experiments are repeated multiple times, and reported results correspond to the average across runs to ensure robustness.

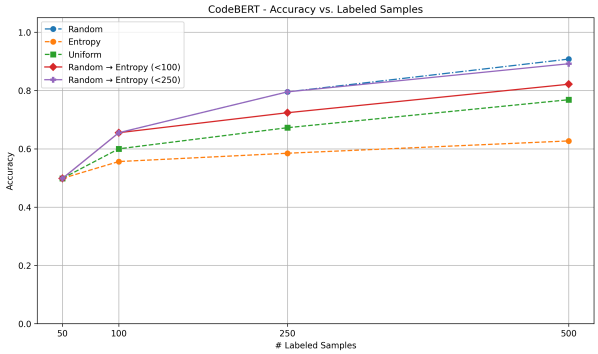
4 Results

This section presents the performance evaluation of various code representation models within our active learning framework across both Java and C/C++ datasets. Each model was evaluated using both Label Propagation and Label Spreading techniques under four distinct sampling strategies.

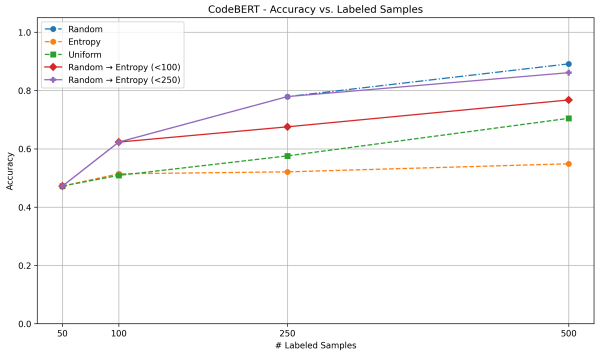
Given the comprehensive nature of our results, this section provides a overview of accuracy trends across different strategies, embedding models, and programming languages. The complete dataset of results, including detailed metrics for ARI, NMI, F1-score, and Recall, is available in the appendix. A deeper analysis of these results and their implications is presented in the subsequent Analysis section.

4.1 Java Results

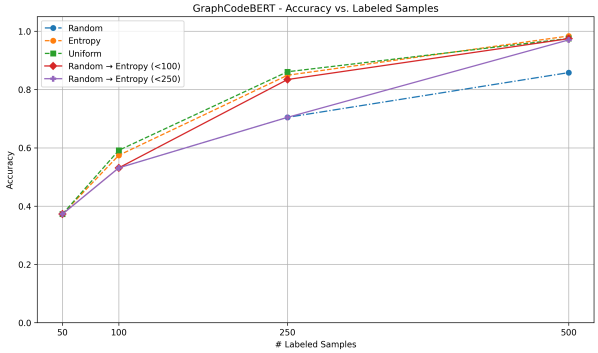
Figures 4 and 5 summarize the Java dataset results.



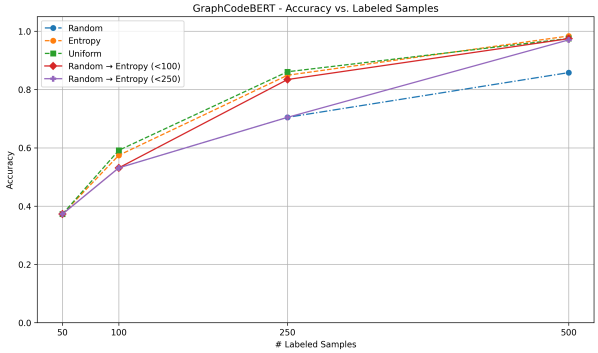
(a) Label Propagation



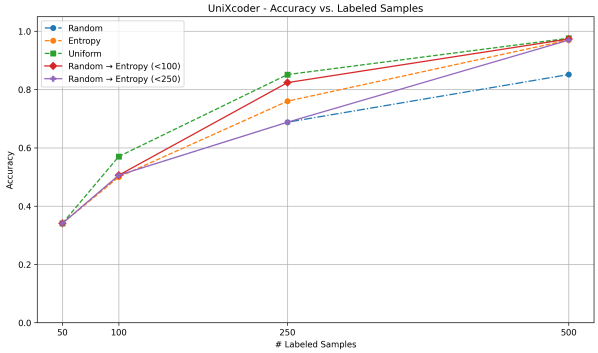
(b) Label Spreading



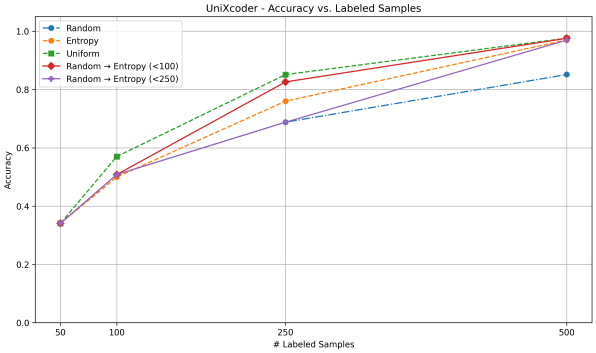
(c) Label Propagation



(d) Label Spreading

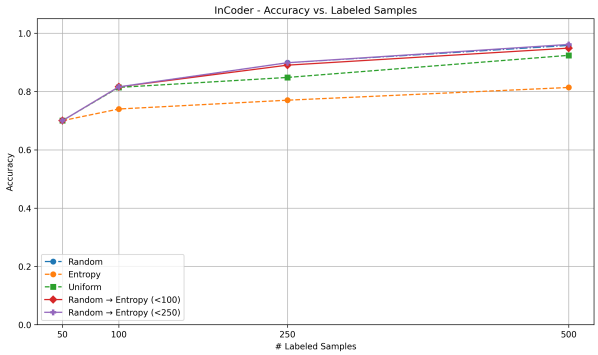


(e) Label Propagation

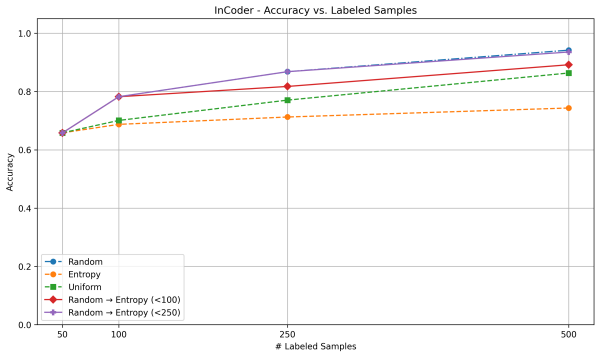


(f) Label Spreading

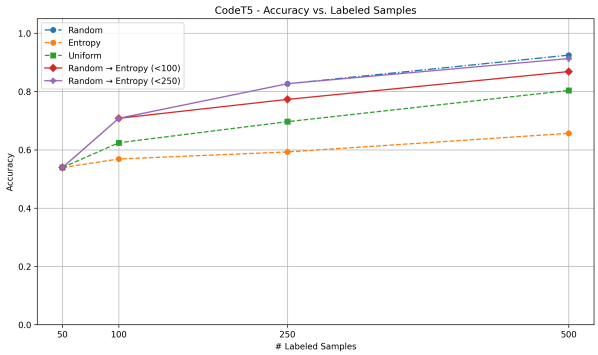
Figure 4: Java dataset performance (Part 1): (a-b) CodeBERT, (c-d) GraphCodeBERT, (e-f) UniXcoder



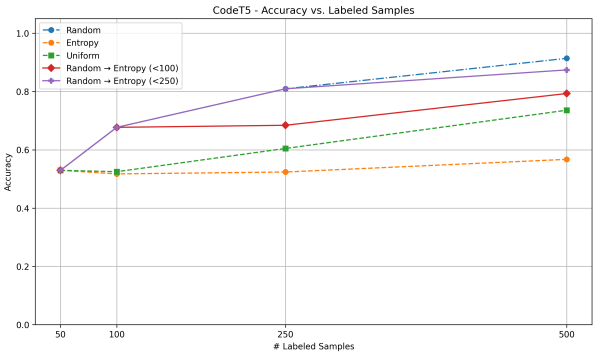
(a) Label Propagation



(b) Label Spreading



(c) Label Propagation

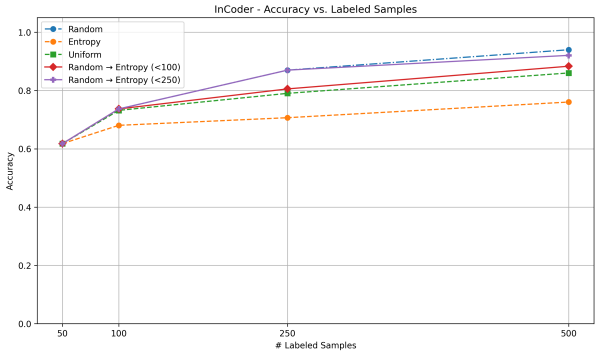


(d) Label Spreading

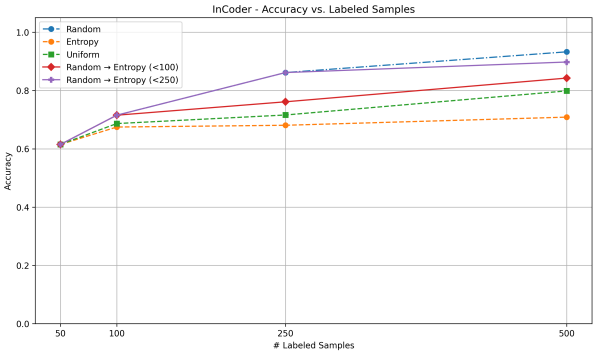
Figure 5: Java dataset performance (Part 2): (a-b) InCoder, (c-d) CodeT5

4.2 C/C++ Results

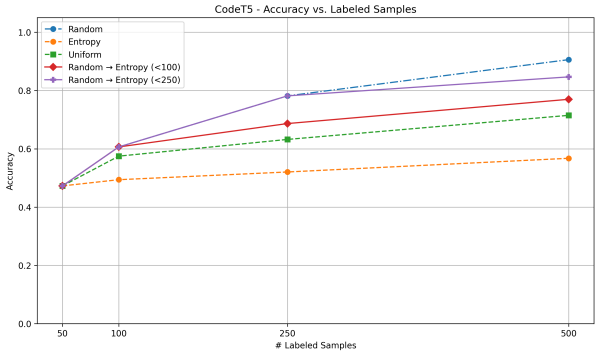
Figures 6 and 7 summarize the C/C++ dataset results.



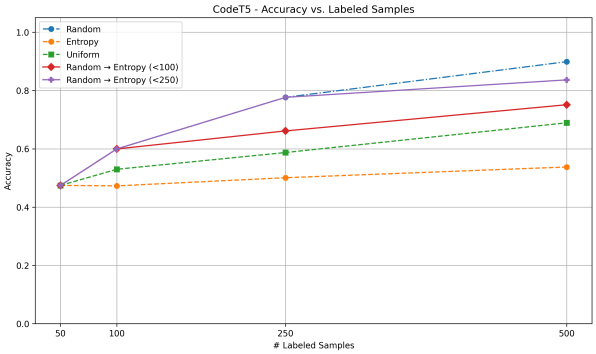
(a) Label Propagation



(b) Label Spreading

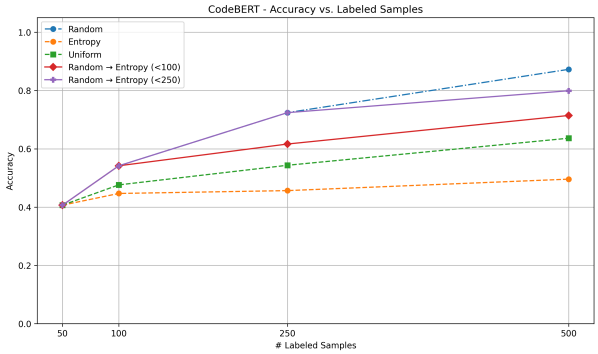


(c) Label Propagation

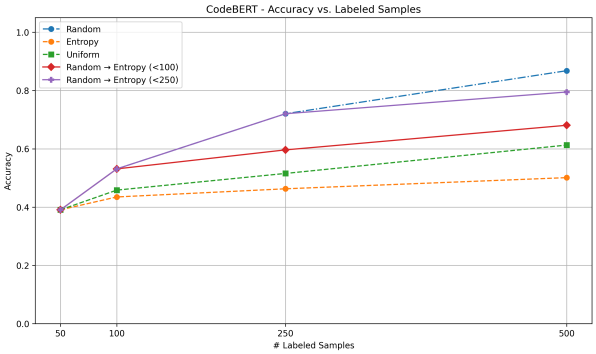


(d) Label Spreading

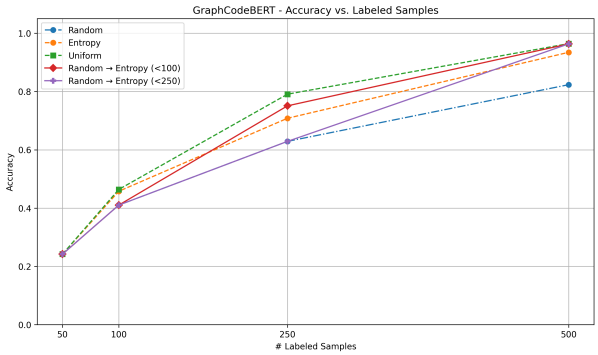
Figure 6: C/C++ dataset performance (Part 1): (a-b) InCoder, (c-d) CodeT5



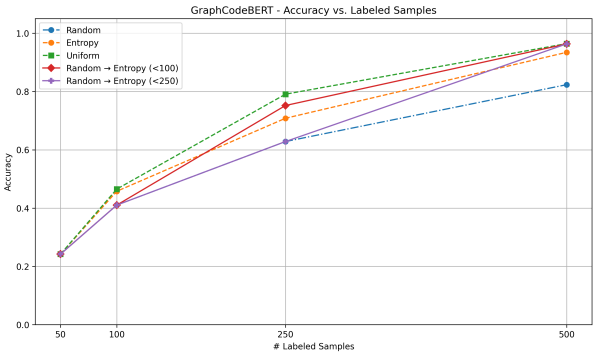
(a) Label Propagation



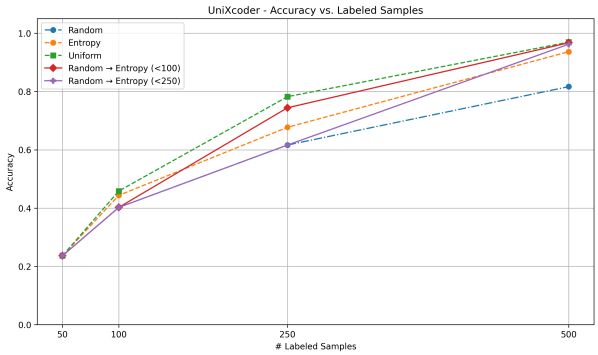
(b) Label Spreading



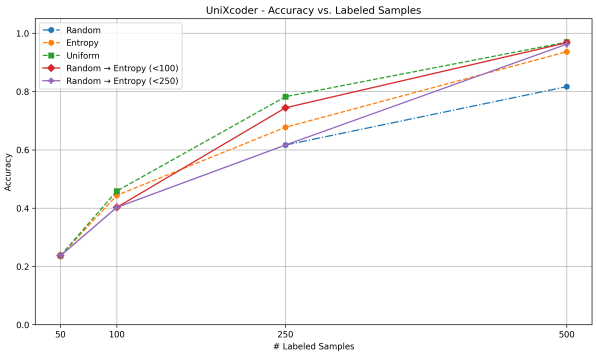
(c) Label Propagation



(d) Label Spreading



(e) Label Propagation



(f) Label Spreading

Figure 7: C/C++ dataset performance (Part 2): (a-b) CodeBERT, (c-d) GraphCodeBERT, (e-f) UniXcoder

5 Analysis

With as few as 50 labeled samples (approximately 0.85% of the dataset), evaluated methods achieve 40%-50% accuracy, demonstrating surprisingly strong initial performance given the limited supervision. This performance scales effectively with additional labels, reaching over 80% accuracy with only 500 labeled samples (approximately 8.5% of the dataset). These results establish a robust baseline for active learning and suggest that even minimal labeled data can capture meaningful patterns in code representation.

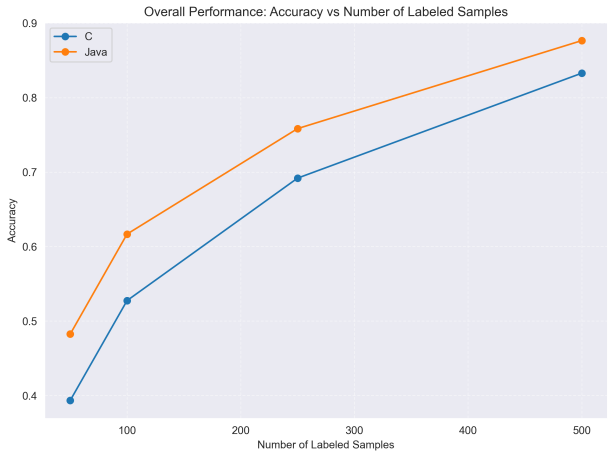


Figure 8: Classification accuracy versus number of labeled samples across all methods and languages.

5.1 Cross-Language Cluster Analysis: Java vs. C

To assess the transferability of code embeddings across programming languages, we compare clustering performance between Java and C under identical experimental conditions. Table 2 reports the mean performance gaps and the proportion of cases in which Java outperforms C across multiple metrics.

Metric	Mean Diff.	Java Better (%)	C Better (%)	Equal (%)	Total Cases
Accuracy	0.072	99.5	0.5	0.0	200
ARI	0.091	99.5	0.5	0.0	200
NMI	0.089	100.0	0.0	0.0	200
F1	0.066	96.0	4.0	0.0	200
Recall	0.067	96.5	3.5	0.0	200

Table 2: Performance gap between Java and C across evaluation metrics. "Mean Diff." refers to the average difference (Java–C). Percentages denote the fraction of cases in which Java or C achieved higher scores.

Figure 9 further illustrates the distribution of performance differences through boxplots. The median values for all metrics are above zero, reinforcing the observation that Java generally achieves higher clustering quality than C.

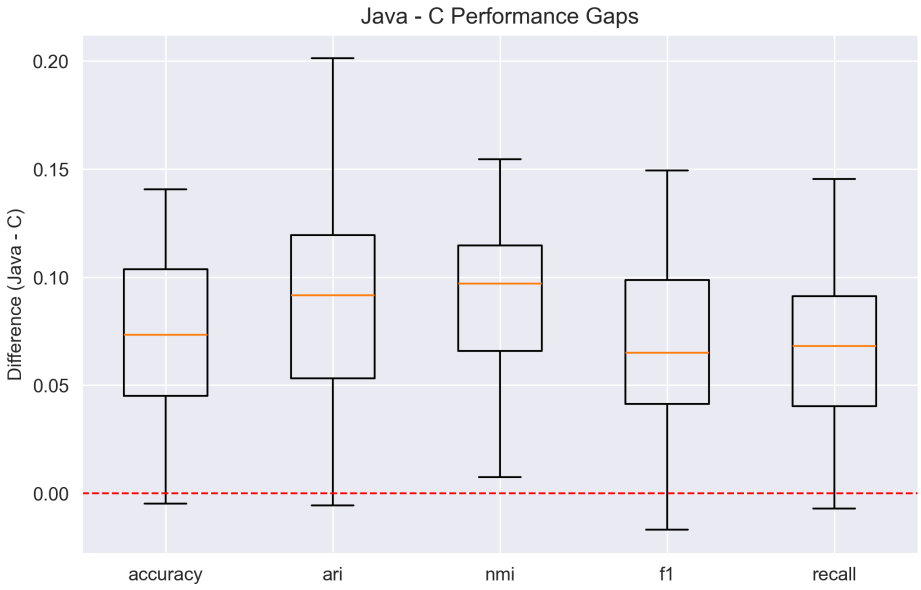


Figure 9: Distribution of performance gaps (Java–C) across metrics. The dashed red line at 0 indicates parity between the two languages.

The consistent advantage of Java over C can be attributed to several factors. First, Java's more regular syntax and richer context (e.g., class structures, method declarations, and explicit typing) may provide embeddings with clearer semantic signals, leading to better cluster separability, as demonstrated by research showing that structural focus improves embedding quality [4]. In contrast, C code often exhibits lower-level constructs and more flexible syntactic patterns, which may reduce embedding consistency across similar tasks. Furthermore, Java corpora used in pretraining are often larger and more diverse than C, potentially biasing representation quality towards Java.

5.2 Approach & Embedding Differences

Impact of Class Balance Enforcing class balance through the class-uniform query strategy consistently yields superior active learning performance compared to vanilla entropy sampling. As visually confirmed in Figure 10, the class-uniform strategy maintains a significant performance advantage across all label budgets (50, 100, 250, and 500). This performance advantage highlights the critical importance of balanced query selection, particularly for imbalanced code classification tasks where the naive entropy strategy tends to oversample majority classes. These findings align with and extend prior work in natural language processing that demonstrates incorporating class balancing heuristics into active learning frameworks significantly improves efficiency and final model performance on imbalanced datasets [7, 8].

Onset of Active Learning The optimal onset of active learning is a critical determinant of final model performance. Our results demonstrate that deferring the transition from random to informed sampling yields significant benefits. As shown in Figure 11, initializing

Figure 10: Comparison of class-uniform versus entropy sampling strategies across increasing label budgets.

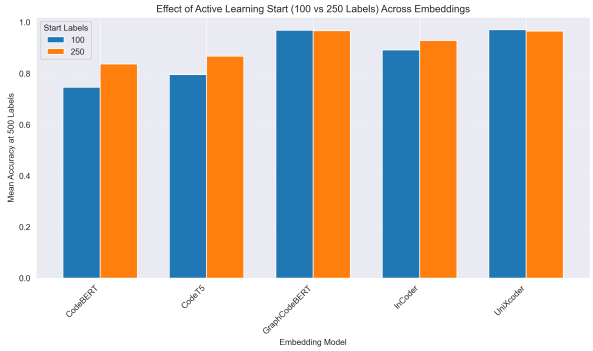


Figure 11: Macro F1-score at 500 labels for AL onset at 100 vs. 250 samples across embeddings.

active learning at 250 labeled samples consistently matches or surpasses the performance of earlier onset (100 samples) when evaluated at the 500-label benchmark. This pattern is particularly pronounced for CodeBERT, CodeT5, and InCoder embeddings, where the later onset provides a clear advantage. For GraphCodeBERT and UniXCoder, performance remains similar across onset strategies, suggesting these embeddings generate sufficiently stable initial representations to support effective active learning even with fewer labels.

This finding indicates that stronger initial supervision—acquired through a longer phase of random sampling—provides a more robust foundational representation. This foundation reduces the risk of sampling bias in subsequent active learning cycles, preventing the reinforcement of early model errors and enabling more informative query selection once the representation space has stabilized.

Label Propagation vs. Label Spreading Our analysis reveals a statistically significant performance advantage of Label Propagation over Label Spreading across all evaluation metrics and experimental conditions. As shown in Figure 12, Label Propagation consistently achieves superior mean accuracy (left panel) and maintains stronger final performance across diverse embedding methods (right panel). This performance gap arises from fundamental algorithmic properties that align closely with the structural characteristics of code.

Label Propagation employs hard clamping, preserving original high-confidence labels from method signatures and AST features throughout propagation, thereby preventing semantic drift in code representations. In contrast, Label Spreading utilizes soft clamping, which allows neighborhood influence to overwrite initial labels and introduces noise in structured code graphs. The difference is further amplified by normalization schemes: Label Propagation's random-walk normalized Laplacian naturally aligns with probabilistic traversal patterns in code (e.g., control-flow paths), while Label Spreading's symmetric normalization emphasizes mutual influence at the expense of directional dependencies critical for program analysis. As [23] demonstrate, graph quality strongly impacts propagation effectiveness—a factor that

explains why Label Propagation achieves consistent gains in code embedding spaces where AST-derived graphs exhibit exceptionally high structural consistency.

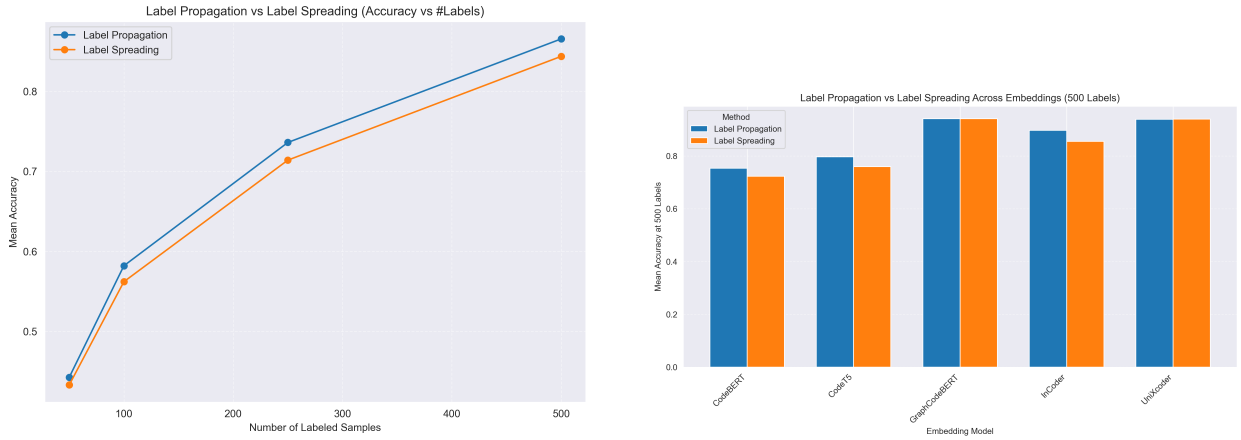


Figure 12: Comparison of Label Propagation (blue) versus Label Spreading (orange) performance. Left: Learning curves showing nearly identical performance across label budgets. Right: Performance comparison across different embedding methods, showing minimal and inconsistent differences between propagation algorithms.

Random sampling vs. Active learning Unexpectedly, random sampling outperforms entropy-based active learning strategies for CodeBERT, CodeT5, and InCoder embeddings, challenging the conventional assumption that uncertainty-driven queries inherently provide more informative samples. As shown in Table 3, the performance gap (Random - Entropy) becomes increasingly negative for these embeddings as the label budget grows (e.g., -0.342 for CodeBERT at 500 labels), indicating clear random sampling superiority. Conversely, GraphCodeBERT and UniXcoder exhibit the expected active learning advantage, with entropy sampling outperforming random by margins up to 0.119. This divergence suggests that embedding quality and structural awareness critically mediate active learning effectiveness.

Performance reversals arise from embedding sensitivity to initial bias and query drift. Models such as CodeBERT, CodeT5, and InCoder amplify early sampling errors under entropy-based selection, prioritizing outliers or ambiguous samples that poorly represent the underlying data distribution. Moreover, active learning is most effective when dataset characteristics align with the query strategy; when this alignment is absent, entropy-based sampling may underperform even naive random selection.

GraphCodeBERT and UniXcoder mitigate these issues by incorporating structural information, such as data flow graphs and Abstract Syntax Trees (ASTs). This structural grounding produces more stable embeddings and improves uncertainty estimation by capturing functional relationships between code elements[11]. Consequently, entropy-based sampling more reliably identifies informative samples, yielding robust active learning performance in structurally sensitive tasks. By contrast, CodeBERT, CodeT5, and InCoder rely primarily on sequential token-level representations without structural grounding, resulting in semantically irregular embeddings. Their uncertainty estimates are less reliable, leading entropy

strategies to select noisy or uninformative points. In such cases, random sampling provides more diverse coverage of the data space and can outperform uncertainty-driven approaches.

Table 3: Performance of Entropy vs. Random Strategy Across All Embeddings

Labeled	**CodeT5**			**InCoder**		
	Entropy	Random	Diff	Entropy	Random	Diff
100	0.513	0.648	-0.135	0.696	0.762	-0.067
250	0.534	0.798	-0.264	0.717	0.874	-0.157
500	0.582	0.911	-0.329	0.756	0.943	-0.187

CodeBERT			**GraphCodeBERT**			**UniXcoder**		
Entropy	Random	Diff	Entropy	Random	Diff	Entropy	Random	Diff
0.488	0.588	-0.100	0.516	0.471	0.045	0.472	0.455	0.017
0.506	0.755	-0.248	0.778	0.666	0.112	0.719	0.652	0.067
0.543	0.885	-0.342	0.959	0.841	0.118	0.953	0.834	0.119

6 Conclusion

This study provides a preliminary analysis of semi-supervised learning techniques for clustering code by algorithm, evaluating the complex interplay between code embeddings, label propagation algorithms, and active learning strategies. Our findings demonstrate that effective code clustering in low-label regimes is not determined by a single factor, but rather by the careful alignment of several critical components.

First, the choice of code embedding model proves to be the most significant determinant of overall performance. Embeddings that explicitly incorporate structural information through Abstract Syntax Trees (ASTs) and Data Flow Graphs (DFGs)—specifically GraphCodeBERT and UniXcoder—consistently outperform sequential token-based models across all experimental conditions. Their structurally-aware representations create a more semantically meaningful latent space where both clustering algorithms and uncertainty estimates for active learning can operate more effectively.

Second, we identify that proper sampling strategy design is crucial for successful active learning. The class-uniform query strategy, which explicitly maintains class balance during sample selection, universally outperforms naive entropy-based sampling across all embedding types. Furthermore, we find that deferring the onset of active learning until a sufficient initial representation is learned (approximately 250 labeled samples in our setup) provides significant performance gains, particularly for less structurally-aware embeddings.

Third, our cross-language analysis reveals a consistent performance advantage for Java over C, attributable to Java's more structured syntax and richer contextual information that provides clearer semantic signals for embedding models. This finding highlights the importance of considering language-specific characteristics when building code understanding systems.

Finally, our comparison of label propagation algorithms shows that Label Propagation's hard clamping mechanism outperforms Label Spreading's soft constraints for code cluster-

ing tasks. This result suggests that preserving high-confidence labels from method signatures and structural features is more beneficial than allowing gradual label updates through neighborhood influence in code representation spaces.

7 Future Work

While this study has provided several insights into semi-supervised clustering with active learning for code embeddings, there remain important directions for further exploration:

- **Computational Efficiency Benchmarking:** We observed a substantial runtime disparity between Label Propagation and Label Spreading algorithms, with the former often requiring hours to converge while the latter completed in minutes. Future work should systematically profile the computational requirements of these methods across varying dataset sizes and graph complexities to provide practical guidance for method selection.
- **Continuous Learning Curves:** Our evaluation used discrete labeling milestones (50, 100, 250, 500 samples). A more nuanced analysis employing continuous learning curves would better capture knowledge acquisition dynamics and identify critical inflection points in the learning process.
- **Dataset Scaling and Balancing:** The current study utilized a relatively small dataset with imbalanced Java and C test sets. Expanding to larger, more balanced cross-language corpora would strengthen the generalizability of findings and enable more robust cross-linguistic comparisons.
- **Advanced Outlier Handling:** Preliminary experiments with simple entropy-based outlier removal proved ineffective. Future research should develop more sophisticated approaches, potentially incorporating clustering-based filtering or density-aware sampling strategies to better distinguish between informative uncertain samples and true outliers.

References

[1] Mikel Artetxe, Shruti Bhosale, Naman Goyal, Todor Mihaylov, Myle Ott, Sam Shleifer, Xi Victoria Lin, Jingfei Du, Srinivasan Iyer, Ramakanth Pasunuru, Giri Anantharaman, Xian Li, Shuohui Chen, Halil Akin, Mandeep Baines, Louis Martin, Xing Zhou, Punit Singh Koura, Brian O'Horo, Jeff Wang, Luke Zettlemoyer, Mona Diab, Zornitsa Kozareva, and Ves Stoyanov. Efficient large scale language modeling with mixtures of experts, 2022.

[2] Nghi D. Q. Bui, Lingxiao Jiang, and Yijun Yu. Cross-language learning for program classification using bilateral tree-based convolutional neural networks. In *The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018.*, pages 758–761, 2018.

[3] David A. Cohn, Les E. Atlas, and Richard E. Ladner. Improving generalization with active learning. *Machine Learning*, 15:201–221, 1994.

[4] Rhys Compton, Eibe Frank, Panos Patros, and Abigail M. Y. Koay. Embedding java classes with code2vec: Improvements from variable obfuscation. *CoRR*, abs/2004.02942, 2020.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[6] Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. Unified language model pre-training for natural language understanding and generation, 2019.

[7] Yaron Fairstein, Oren Kalinsky, Zohar Karnin, Guy Kushilevitz, Alexander Libov, and Sofia Tolmach. Class balancing for efficient active learning in imbalanced datasets. In Sophie Henning and Manfred Stede, editors, *Proceedings of the 18th Linguistic Annotation Workshop (LAW-XVIII)*, pages 77–86, St. Julians, Malta, March 2024. Association for Computational Linguistics.

[8] Yaron Fairstein, Zohar Karnin, Guy Kushilevitz, Oren Kalinsky, Alex Libov, and Sofia Tolmach. Class balancing for efficient active learning in imbalanced datasets. 2024.

[9] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1536–1547, 2020.

[10] Daniel Fried, Jacob Austin, Daniel Andrejczuk, J. Hu, Zico Kolter, Luke Metz, Miltiadis Allamanis, and Marc Brockschmidt. Incoder: A generative model for code infilling and synthesis. In *Proceedings of the 11th International Conference on Learning Representations (ICLR)*, 2023.

[11] Dominik Fuchsgruber, Tom Wollschläger, Bertrand Charpentier, Antonio Oroz, and Stephan Günnemann. Uncertainty for active learning on graphs, 2025.

[12] Tianyu Gao, Xingcheng Yao, and Danqi Chen. Simcse: Simple contrastive learning of sentence embeddings, 2022.

[13] Daya Guo, Shuai Lu, Shuo Ren, Zhangyin Feng, Duyu Tang, Nan Duan, Long Zhou, Alexey Svyatkovskiy, Shuo Fu, Michele Tufano, et al. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 7212–7225, 2022.

[14] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Nan Duan, Long Zhou, Linjun Shou, Ming Gong, Daxin Jiang, et al. Graphcodebert: Pre-training code representations with data flow. In *Proceedings of the 9th International Conference on Learning Representations (ICLR)*, 2021.

[15] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search, 2020.

[16] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension, 2019.

[17] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

[18] B. Nghi D. Q., Y. Yu, and L. Jiang. Bilateral dependency neural networks for cross-language algorithm classification. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 422–433, Feb 2019.

[19] Alec Radford and Karthik Narasimhan. Improving language understanding by generative pre-training. 2018.

[20] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[21] Burr Settles. Active learning literature survey. Technical report, University of Wisconsin–Madison, Computer Sciences Technical Report 1648, 2009.

[22] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[23] Fei Wang, Lei Zhu, Liang Xie, Zheng Zhang, and Mingyang Zhong. Label propagation with structured graph learning for semi-supervised dimension reduction. *Knowledge-Based Systems*, 225:107130, 2021.

[24] Yue Wang, Weishi Wang, Shafiq Joty, Steven C.H. Lin, and See-Kiong Ng. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and

generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 8696–8708, 2021.

[25] Dengyong Zhou, Olivier Bousquet, Thomas Navin Lal, Jason Weston, and Bernhard Schölkopf. Learning with local and global consistency. *Advances in neural information processing systems*, 16:321–328, 2004.

[26] Xiaojin Zhu, Zoubin Ghahramani, and John Lafferty. Semi-supervised learning using gaussian fields and harmonic functions. In *Proceedings of the 20th International Conference on Machine Learning (ICML)*, pages 912–919, 2003. Originally Technical Report CMU-CS-02-107, Carnegie Mellon University, 2002.

A Code Repository

The code supporting the experiments and analyses presented in this report is available on GitHub: `https://github.com/t3acup/MIE8888-KuangRen`

B Complete Experimental Results

Table 4: Complete Active Learning Performance Across Models, Propagation Methods and Languages

Model	Propagation	Language	Strategy	Labels	Acc	ARI	NMI	F1	Recall
CodeBERT	Propagation	Java	Random Sampling	50	0.499	0.258	0.386	0.501	0.497
				100	0.655	0.416	0.527	0.656	0.659
				250	0.795	0.614	0.679	0.792	0.792
				500	0.908	0.817	0.833	0.901	0.902
			Entropy-based	50	0.499	0.258	0.386	0.501	0.497
				100	0.556	0.312	0.437	0.557	0.555
				250	0.585	0.340	0.455	0.579	0.582
				500	0.627	0.387	0.498	0.621	0.624
			Class-balance	50	0.499	0.258	0.386	0.501	0.497
				100	0.600	0.350	0.466	0.595	0.597
				250	0.672	0.437	0.554	0.668	0.671
				500	0.768	0.577	0.673	0.761	0.767
			Random → Entropy (100)	50	0.499	0.258	0.386	0.501	0.497
				100	0.655	0.416	0.527	0.656	0.659
				250	0.724	0.508	0.600	0.719	0.728
				500	0.821	0.662	0.735	0.817	0.824
			Random → Entropy (250)	50	0.499	0.258	0.386	0.501	0.497
				100	0.655	0.416	0.527	0.656	0.659
				250	0.795	0.614	0.679	0.792	0.792
				500	0.892	0.793	0.823	0.884	0.889
		C/C++	Random Sampling	50	0.407	0.154	0.272	0.402	0.414
				100	0.542	0.271	0.398	0.539	0.550
				250	0.724	0.499	0.582	0.721	0.724
				500	0.873	0.744	0.777	0.871	0.872
			Entropy-based	50	0.407	0.154	0.272	0.402	0.414
				100	0.447	0.186	0.302	0.445	0.452
				250	0.457	0.193	0.317	0.454	0.461
				500	0.496	0.227	0.355	0.493	0.502
			Class-balance	50	0.407	0.154	0.272	0.402	0.414
				100	0.476	0.208	0.333	0.475	0.486
				250	0.543	0.275	0.407	0.542	0.555
				500	0.636	0.383	0.518	0.638	0.652
			Random → Entropy (100)	50	0.407	0.154	0.272	0.402	0.414
				100	0.542	0.271	0.398	0.539	0.550
				250	0.616	0.357	0.476	0.614	0.625
				500	0.714	0.487	0.588	0.713	0.725
			Random → Entropy (250)	50	0.407	0.154	0.272	0.402	0.414
				100	0.542	0.271	0.398	0.539	0.550
				250	0.724	0.499	0.582	0.721	0.724
				500	0.799	0.614	0.679	0.797	0.803
	Spreading	Java	Random Sampling	50	0.472	0.235	0.373	0.480	0.475
				100	0.623	0.376	0.488	0.621	0.630
				250	0.779	0.590	0.655	0.773	0.778

Continued on next page

Table 4 continued from previous page

Model	Propagation	Language	Strategy	Labels	Acc	ARI	NMI	F1	Recall
				500	0.891	0.789	0.814	0.884	0.886
			Entropy-based	50	0.472	0.235	0.373	0.480	0.475
				100	0.514	0.252	0.409	0.520	0.520
				250	0.521	0.242	0.425	0.529	0.529
				500	0.548	0.266	0.442	0.553	0.554
			Class-balance	50	0.472	0.235	0.373	0.480	0.475
				100	0.509	0.213	0.402	0.521	0.521
				250	0.576	0.285	0.477	0.589	0.583
				500	0.704	0.448	0.614	0.712	0.710
			Random → Entropy (100)	50	0.472	0.235	0.373	0.480	0.475
				100	0.623	0.376	0.488	0.621	0.630
				250	0.675	0.420	0.562	0.677	0.686
				500	0.768	0.550	0.679	0.769	0.776
			Random → Entropy (250)	50	0.472	0.235	0.373	0.480	0.475
				100	0.623	0.376	0.488	0.621	0.630
				250	0.779	0.590	0.655	0.773	0.778
				500	0.861	0.736	0.791	0.854	0.861
		C/C++	Random Sampling	50	0.391	0.139	0.261	0.388	0.400
				100	0.531	0.258	0.389	0.530	0.539
				250	0.720	0.491	0.575	0.717	0.720
				500	0.868	0.731	0.771	0.868	0.867
			Entropy-based	50	0.391	0.139	0.261	0.388	0.400
				100	0.435	0.176	0.300	0.435	0.440
				250	0.463	0.190	0.336	0.469	0.468
				500	0.501	0.214	0.367	0.508	0.505
			Class-balance	50	0.391	0.139	0.261	0.388	0.400
				100	0.458	0.184	0.345	0.468	0.469
				250	0.515	0.236	0.396	0.526	0.529
				500	0.613	0.340	0.504	0.626	0.629
			Random → Entropy (100)	50	0.391	0.139	0.261	0.388	0.400
				100	0.531	0.258	0.389	0.530	0.539
				250	0.596	0.316	0.469	0.603	0.604
				500	0.681	0.421	0.566	0.689	0.695
			Random → Entropy (250)	50	0.391	0.139	0.261	0.388	0.400
				100	0.531	0.258	0.389	0.530	0.539
				250	0.720	0.491	0.575	0.717	0.720
				500	0.795	0.598	0.676	0.798	0.799
GraphCodeBERT	Propagation	Java	Random Sampling	50	0.373	0.064	0.382	0.406	0.371
				100	0.531	0.150	0.527	0.591	0.539
				250	0.704	0.347	0.669	0.749	0.703
				500	0.858	0.668	0.807	0.870	0.851
			Entropy-based	50	0.373	0.064	0.382	0.406	0.371
				100	0.574	0.255	0.613	0.591	0.566
				250	0.849	0.714	0.849	0.842	0.834
				500	0.984	0.971	0.976	0.979	0.979
			Class-balance	50	0.373	0.064	0.382	0.406	0.371
				100	0.591	0.207	0.583	0.651	0.599
				250	0.861	0.678	0.815	0.874	0.860
				500	0.975	0.953	0.962	0.970	0.970
			Random → Entropy (100)	50	0.373	0.064	0.382	0.406	0.371
				100	0.531	0.150	0.527	0.591	0.539
				250	0.834	0.613	0.788	0.854	0.837
				500	0.974	0.954	0.961	0.969	0.972
			Random → Entropy (250)	50	0.373	0.064	0.382	0.406	0.371
				100	0.531	0.150	0.527	0.591	0.539
				250	0.704	0.347	0.669	0.749	0.703

Continued on next page

Table 4 continued from previous page

Model	Propagation	Language	Strategy	Labels	Acc	ARI	NMI	F1	Recall
				500	0.970	0.945	0.952	0.964	0.964
			Random Sampling	50	0.242	0.011	0.233	0.284	0.255
				100	0.410	0.059	0.404	0.486	0.418
				250	0.629	0.235	0.594	0.693	0.631
				500	0.823	0.600	0.767	0.847	0.824
		C/C++	Entropy-based	50	0.242	0.011	0.233	0.284	0.255
				100	0.458	0.161	0.493	0.461	0.430
				250	0.708	0.512	0.752	0.692	0.689
				500	0.934	0.880	0.919	0.931	0.930
			Class-balance	50	0.242	0.011	0.233	0.284	0.255
				100	0.464	0.083	0.456	0.551	0.475
				250	0.791	0.508	0.756	0.833	0.803
				500	0.964	0.923	0.947	0.965	0.966
			Random → Entropy (100)	50	0.242	0.011	0.233	0.284	0.255
				100	0.410	0.059	0.404	0.486	0.418
				250	0.751	0.438	0.709	0.797	0.759
				500	0.963	0.921	0.944	0.964	0.966
			Random → Entropy (250)	50	0.242	0.011	0.233	0.284	0.255
				100	0.410	0.059	0.404	0.486	0.418
				250	0.629	0.235	0.594	0.693	0.631
				500	0.963	0.919	0.944	0.965	0.966
	Spreading	Java	Random Sampling	50	0.373	0.064	0.382	0.406	0.371
				100	0.531	0.150	0.527	0.591	0.539
				250	0.704	0.347	0.669	0.749	0.703
				500	0.858	0.668	0.807	0.870	0.851
			Entropy-based	50	0.373	0.064	0.382	0.406	0.371
				100	0.574	0.255	0.613	0.591	0.566
				250	0.849	0.714	0.849	0.842	0.834
				500	0.984	0.971	0.976	0.979	0.979
			Class-balance	50	0.373	0.064	0.382	0.406	0.371
				100	0.591	0.207	0.583	0.651	0.599
				250	0.861	0.678	0.815	0.874	0.860
				500	0.975	0.953	0.962	0.970	0.970
			Random → Entropy (100)	50	0.373	0.064	0.382	0.406	0.371
				100	0.531	0.150	0.527	0.591	0.539
				250	0.834	0.613	0.788	0.854	0.837
				500	0.974	0.954	0.961	0.969	0.972
			Random → Entropy (250)	50	0.373	0.064	0.382	0.406	0.371
				100	0.531	0.150	0.527	0.591	0.539
				250	0.704	0.347	0.669	0.749	0.703
				500	0.970	0.945	0.952	0.964	0.964
		C/C++	Random Sampling	50	0.242	0.011	0.233	0.284	0.255
				100	0.410	0.059	0.404	0.486	0.418
				250	0.628	0.237	0.591	0.692	0.630
				500	0.823	0.600	0.767	0.847	0.824
			Entropy-based	50	0.242	0.011	0.233	0.284	0.255
				100	0.458	0.161	0.493	0.461	0.430
				250	0.708	0.512	0.752	0.692	0.689
				500	0.934	0.880	0.919	0.931	0.930
			Class-balance	50	0.242	0.011	0.233	0.284	0.255
				100	0.465	0.085	0.454	0.551	0.476
				250	0.791	0.508	0.756	0.833	0.803
				500	0.964	0.923	0.947	0.965	0.966
			Random → Entropy (100)	50	0.242	0.011	0.233	0.284	0.255
				100	0.410	0.059	0.404	0.486	0.418
				250	0.752	0.440	0.710	0.798	0.760

Continued on next page

Table 4 continued from previous page

Model	Propagation	Language	Strategy	Labels	Acc	ARI	NMI	F1	Recall
				500	0.963	0.921	0.944	0.964	0.966
			Random → Entropy (250)	50	0.242	0.011	0.233	0.284	0.255
				100	0.410	0.059	0.404	0.486	0.418
				250	0.628	0.237	0.591	0.692	0.630
				500	0.963	0.919	0.944	0.965	0.966
UniXcoder	Propagation	Java	Random Sampling	50	0.341	0.045	0.339	0.375	0.340
				100	0.506	0.122	0.497	0.569	0.513
				250	0.688	0.315	0.650	0.736	0.685
				500	0.851	0.650	0.799	0.865	0.845
			Entropy-based	50	0.341	0.045	0.339	0.375	0.340
				100	0.500	0.170	0.536	0.525	0.499
				250	0.760	0.526	0.772	0.773	0.759
				500	0.970	0.947	0.964	0.965	0.968
			Class-balance	50	0.341	0.045	0.339	0.375	0.340
				100	0.570	0.181	0.561	0.633	0.578
				250	0.851	0.651	0.803	0.868	0.851
				500	0.976	0.956	0.963	0.971	0.972
			Random → Entropy (100)	50	0.341	0.045	0.339	0.375	0.340
				100	0.506	0.122	0.497	0.569	0.513
				250	0.824	0.587	0.777	0.846	0.826
				500	0.973	0.951	0.959	0.968	0.971
			Random → Entropy (250)	50	0.341	0.045	0.339	0.375	0.340
				100	0.506	0.122	0.497	0.569	0.513
				250	0.688	0.315	0.650	0.736	0.685
				500	0.969	0.943	0.951	0.963	0.963
		C/C++	Random Sampling	50	0.237	0.010	0.225	0.275	0.249
				100	0.402	0.054	0.396	0.476	0.411
				250	0.616	0.218	0.584	0.683	0.619
				500	0.817	0.581	0.761	0.843	0.819
			Entropy-based	50	0.237	0.010	0.225	0.275	0.249
				100	0.444	0.146	0.478	0.444	0.415
				250	0.677	0.459	0.729	0.660	0.656
				500	0.936	0.884	0.926	0.932	0.932
			Class-balance	50	0.237	0.010	0.225	0.275	0.249
				100	0.458	0.079	0.452	0.544	0.471
				250	0.782	0.493	0.742	0.825	0.795
				500	0.969	0.931	0.953	0.971	0.972
			Random → Entropy (100)	50	0.237	0.010	0.225	0.275	0.249
				100	0.402	0.054	0.396	0.476	0.411
				250	0.744	0.420	0.704	0.793	0.755
				500	0.967	0.927	0.950	0.969	0.970
			Random → Entropy (250)	50	0.237	0.010	0.225	0.275	0.249
				100	0.402	0.054	0.396	0.476	0.411
				250	0.616	0.218	0.584	0.683	0.619
				500	0.962	0.915	0.943	0.965	0.966
	Spreading	Java	Random Sampling	50	0.341	0.045	0.339	0.375	0.340
				100	0.508	0.123	0.501	0.573	0.515
				250	0.688	0.315	0.650	0.736	0.685
				500	0.851	0.650	0.799	0.865	0.845
			Entropy-based	50	0.341	0.045	0.339	0.375	0.340
				100	0.500	0.170	0.536	0.525	0.499
				250	0.760	0.526	0.772	0.773	0.759
				500	0.970	0.947	0.964	0.965	0.968
			Class-balance	50	0.341	0.045	0.339	0.375	0.340
				100	0.570	0.181	0.561	0.633	0.578
				250	0.851	0.651	0.803	0.868	0.851

Continued on next page

Table 4 continued from previous page

Model	Propagation	Language	Strategy	Labels	Acc	ARI	NMI	F1	Recall
				500	0.976	0.956	0.963	0.971	0.972
			Random → Entropy (100)	50	0.341	0.045	0.339	0.375	0.340
				100	0.508	0.123	0.501	0.573	0.515
				250	0.826	0.591	0.780	0.849	0.829
				500	0.976	0.956	0.962	0.970	0.973
			Random → Entropy (250)	50	0.341	0.045	0.339	0.375	0.340
				100	0.508	0.123	0.501	0.573	0.515
				250	0.688	0.315	0.650	0.736	0.685
				500	0.969	0.943	0.951	0.963	0.963
			Random Sampling	50	0.237	0.010	0.225	0.275	0.249
				100	0.402	0.054	0.396	0.477	0.411
				250	0.616	0.218	0.584	0.683	0.619
				500	0.817	0.581	0.761	0.843	0.819
		C/C++	Entropy-based	50	0.237	0.010	0.225	0.275	0.249
				100	0.444	0.146	0.478	0.444	0.415
				250	0.677	0.459	0.729	0.660	0.656
				500	0.936	0.884	0.926	0.932	0.932
			Class-balance	50	0.237	0.010	0.225	0.275	0.249
				100	0.458	0.079	0.452	0.544	0.471
				250	0.782	0.493	0.742	0.825	0.795
				500	0.969	0.931	0.953	0.971	0.972
			Random → Entropy (100)	50	0.237	0.010	0.225	0.275	0.249
				100	0.402	0.054	0.396	0.477	0.411
				250	0.744	0.420	0.704	0.793	0.755
				500	0.967	0.927	0.950	0.969	0.970
			Random → Entropy (250)	50	0.237	0.010	0.225	0.275	0.249
				100	0.402	0.054	0.396	0.477	0.411
				250	0.616	0.218	0.584	0.683	0.619
				500	0.962	0.915	0.943	0.965	0.966
CodeT5	Propagation	Java	Random Sampling	50	0.539	0.297	0.418	0.535	0.538
				100	0.708	0.495	0.584	0.700	0.710
				250	0.827	0.675	0.724	0.818	0.821
				500	0.925	0.853	0.864	0.917	0.918
			Entropy-based	50	0.539	0.297	0.418	0.535	0.538
				100	0.568	0.333	0.461	0.562	0.569
				250	0.593	0.362	0.493	0.582	0.593
				500	0.656	0.438	0.552	0.646	0.654
			Class-balance	50	0.539	0.297	0.418	0.535	0.538
				100	0.624	0.396	0.513	0.614	0.623
				250	0.696	0.485	0.601	0.688	0.695
				500	0.804	0.643	0.734	0.796	0.803
			Random → Entropy (100)	50	0.539	0.297	0.418	0.535	0.538
				100	0.708	0.495	0.584	0.700	0.710
				250	0.773	0.589	0.668	0.764	0.772
				500	0.868	0.749	0.797	0.861	0.868
			Random → Entropy (250)	50	0.539	0.297	0.418	0.535	0.538
				100	0.708	0.495	0.584	0.700	0.710
				250	0.827	0.675	0.724	0.818	0.821
				500	0.913	0.838	0.859	0.903	0.905
		C/C++	Random Sampling	50	0.473	0.214	0.348	0.470	0.482
				100	0.606	0.344	0.464	0.602	0.614
				250	0.782	0.586	0.654	0.780	0.784
				500	0.906	0.804	0.827	0.906	0.906
			Entropy-based	50	0.473	0.214	0.348	0.470	0.482
				100	0.494	0.235	0.364	0.490	0.504
				250	0.520	0.256	0.389	0.518	0.533

Continued on next page

Table 4 continued from previous page

Model	Propagation	Language	Strategy	Labels	Acc	ARI	NMI	F1	Recall
				500	0.567	0.303	0.430	0.566	0.577
			Class-balance	50	0.473	0.214	0.348	0.470	0.482
				100	0.575	0.312	0.441	0.577	0.589
				250	0.632	0.375	0.503	0.633	0.647
				500	0.715	0.482	0.597	0.717	0.731
			Random → Entropy (100)	50	0.473	0.214	0.348	0.470	0.482
				100	0.606	0.344	0.464	0.602	0.614
				250	0.687	0.439	0.555	0.689	0.696
				500	0.770	0.562	0.654	0.771	0.782
			Random → Entropy (250)	50	0.473	0.214	0.348	0.470	0.482
				100	0.606	0.344	0.464	0.602	0.614
				250	0.782	0.586	0.654	0.780	0.784
				500	0.847	0.694	0.746	0.846	0.853
		Java	Random Sampling	50	0.530	0.287	0.423	0.529	0.532
				100	0.677	0.440	0.556	0.673	0.681
				250	0.809	0.641	0.704	0.803	0.806
				500	0.914	0.834	0.852	0.905	0.907
			Entropy-based	50	0.530	0.287	0.423	0.529	0.532
				100	0.517	0.246	0.418	0.518	0.522
				250	0.524	0.237	0.416	0.523	0.529
				500	0.567	0.286	0.462	0.566	0.570
	Spreading		Class-balance	50	0.530	0.287	0.423	0.529	0.532
				100	0.525	0.243	0.434	0.531	0.532
				250	0.605	0.318	0.522	0.615	0.609
				500	0.736	0.496	0.658	0.741	0.740
			Random → Entropy (100)	50	0.530	0.287	0.423	0.529	0.532
				100	0.677	0.440	0.556	0.673	0.681
				250	0.684	0.415	0.580	0.691	0.692
				500	0.793	0.591	0.715	0.795	0.801
			Random → Entropy (250)	50	0.530	0.287	0.423	0.529	0.532
				100	0.677	0.440	0.556	0.673	0.681
				250	0.809	0.641	0.704	0.803	0.806
				500	0.874	0.758	0.812	0.867	0.872
		C/C++	Random Sampling	50	0.474	0.219	0.353	0.471	0.484
				100	0.599	0.334	0.463	0.598	0.608
				250	0.776	0.575	0.650	0.776	0.778
				500	0.899	0.787	0.819	0.900	0.900
			Entropy-based	50	0.474	0.219	0.353	0.471	0.484
				100	0.473	0.197	0.360	0.479	0.478
				250	0.501	0.210	0.387	0.521	0.510
				500	0.538	0.241	0.420	0.559	0.547
			Class-balance	50	0.474	0.219	0.353	0.471	0.484
				100	0.530	0.249	0.414	0.548	0.539
				250	0.587	0.311	0.467	0.601	0.598
				500	0.689	0.435	0.570	0.701	0.700
			Random → Entropy (100)	50	0.474	0.219	0.353	0.471	0.484
				100	0.599	0.334	0.463	0.598	0.608
				250	0.661	0.389	0.528	0.674	0.667
				500	0.751	0.518	0.635	0.762	0.759
			Random → Entropy (250)	50	0.474	0.219	0.353	0.471	0.484
				100	0.599	0.334	0.463	0.598	0.608
				250	0.776	0.575	0.650	0.776	0.778
				500	0.836	0.668	0.735	0.841	0.842
		Java	Random Sampling	50	0.700	0.521	0.630	0.687	0.692
				100	0.816	0.671	0.741	0.806	0.815
				250	0.899	0.808	0.837	0.891	0.894

Continued on next page

Propagation

Table 4 continued from previous page

Model	Propagation	Language	Strategy	Labels	Acc	ARI	NMI	F1	Recall
				500	0.958	0.917	0.919	0.953	0.955
			Entropy-based	50	0.700	0.521	0.630	0.687	0.692
				100	0.740	0.560	0.664	0.729	0.734
				250	0.770	0.602	0.698	0.761	0.767
				500	0.814	0.666	0.745	0.805	0.814
			Class-balance	50	0.700	0.521	0.630	0.687	0.692
				100	0.814	0.669	0.740	0.803	0.810
				250	0.848	0.720	0.794	0.840	0.846
				500	0.924	0.854	0.887	0.918	0.923
			Random → Entropy (100)	50	0.700	0.521	0.630	0.687	0.692
				100	0.816	0.671	0.741	0.806	0.815
				250	0.890	0.790	0.835	0.885	0.892
				500	0.949	0.900	0.921	0.944	0.951
			Random → Entropy (250)	50	0.700	0.521	0.630	0.687	0.692
				100	0.816	0.671	0.741	0.806	0.815
				250	0.899	0.808	0.837	0.891	0.894
				500	0.961	0.927	0.937	0.957	0.960
		C/C++	Random Sampling	50	0.618	0.377	0.525	0.616	0.624
				100	0.737	0.527	0.627	0.734	0.738
				250	0.870	0.748	0.783	0.866	0.867
				500	0.940	0.877	0.887	0.938	0.939
			Entropy-based	50	0.618	0.377	0.525	0.616	0.624
				100	0.680	0.461	0.579	0.675	0.678
				250	0.707	0.489	0.596	0.701	0.704
				500	0.760	0.562	0.654	0.757	0.756
			Class-balance	50	0.618	0.377	0.525	0.616	0.624
				100	0.732	0.518	0.629	0.730	0.735
				250	0.790	0.606	0.696	0.788	0.790
				500	0.860	0.725	0.788	0.859	0.861
			Random → Entropy (100)	50	0.618	0.377	0.525	0.616	0.624
				100	0.737	0.527	0.627	0.734	0.738
				250	0.805	0.635	0.708	0.802	0.804
				500	0.883	0.764	0.813	0.883	0.884
			Random → Entropy (250)	50	0.618	0.377	0.525	0.616	0.624
				100	0.737	0.527	0.627	0.734	0.738
				250	0.870	0.748	0.783	0.866	0.867
				500	0.921	0.839	0.866	0.919	0.921
	Spreading	Java	Random Sampling	50	0.658	0.447	0.600	0.658	0.651
				100	0.782	0.616	0.714	0.776	0.784
				250	0.868	0.749	0.801	0.861	0.867
				500	0.942	0.886	0.898	0.936	0.940
			Entropy-based	50	0.658	0.447	0.600	0.658	0.651
				100	0.687	0.475	0.650	0.689	0.688
				250	0.712	0.527	0.674	0.705	0.716
				500	0.743	0.574	0.701	0.733	0.747
			Class-balance	50	0.658	0.447	0.600	0.658	0.651
				100	0.701	0.500	0.665	0.697	0.706
				250	0.770	0.587	0.717	0.766	0.777
				500	0.863	0.741	0.827	0.859	0.869
			Random → Entropy (100)	50	0.658	0.447	0.600	0.658	0.651
				100	0.782	0.616	0.714	0.776	0.784
				250	0.817	0.658	0.760	0.812	0.825
				500	0.892	0.790	0.855	0.888	0.897
			Random → Entropy (250)	50	0.658	0.447	0.600	0.658	0.651
				100	0.782	0.616	0.714	0.776	0.784
				250	0.868	0.749	0.801	0.861	0.867

Continued on next page

Table 4 continued from previous page

Model	Propagation	Language	Strategy	Labels	Acc	ARI	NMI	F1	Recall
				500	0.935	0.873	0.902	0.931	0.936
		C/C++	Random Sampling	50	0.615	0.369	0.518	0.616	0.622
				100	0.715	0.486	0.609	0.718	0.720
				250	0.861	0.727	0.773	0.860	0.860
				500	0.933	0.856	0.880	0.934	0.934
			Entropy-based	50	0.615	0.369	0.518	0.616	0.622
				100	0.675	0.438	0.576	0.675	0.675
				250	0.681	0.418	0.571	0.690	0.678
				500	0.709	0.455	0.601	0.719	0.708
			Class-balance	50	0.615	0.369	0.518	0.616	0.622
				100	0.687	0.442	0.590	0.691	0.692
				250	0.716	0.474	0.617	0.726	0.722
				500	0.799	0.601	0.715	0.806	0.804
			Random → Entropy (100)	50	0.615	0.369	0.518	0.616	0.622
				100	0.715	0.486	0.609	0.718	0.720
				250	0.761	0.550	0.671	0.767	0.770
				500	0.843	0.684	0.769	0.847	0.850
			Random → Entropy (250)	50	0.615	0.369	0.518	0.616	0.622
				100	0.715	0.486	0.609	0.718	0.720
				250	0.861	0.727	0.773	0.860	0.860
				500	0.898	0.782	0.834	0.900	0.900