Active Code Learning: Benchmarking Sample-Efficient Training of Code Models

Qiang Hu, Yuejun Guo, Xiaofei Xie, Maxime Cordy, Lei Ma, *Member, IEEE*, Mike Papadakis, and Yves Le Traon, *Fellow, IEEE*

***Abstract*—The costly human effort required to prepare the training data of machine learning (ML) models hinders their practical development and usage in software engineering (ML4Code), especially for those with limited budgets. Therefore, efficiently training models of code with less human effort has become an emergent problem. Active learning is such a technique to address this issue that allows developers to train a model with reduced data while producing models with desired performance, which has been well studied in computer vision and natural language processing domains. Unfortunately, there is no such work that explores the effectiveness of active learning for code models. In this paper, we bridge this gap by building the first benchmark to study this critical problem - active code learning. Specifically, we collect 11 acquisition functions (which are used for data selection in active learning) from existing works and adapt them for code-related tasks. Then, we conduct an empirical study to check whether these acquisition functions maintain performance for code data. The results demonstrate that feature selection highly affects active learning and using output vectors to select data is the best choice. For the code summarization task, active code learning is ineffective which produces models with over a 29.64% gap compared to the expected performance. Furthermore, we explore future directions of active code learning with an exploratory study. We propose to replace distance calculation methods with evaluation metrics and find a correlation between these evaluation-based distance methods and the performance of code models.**

***Index Terms*—Active learning, machine learning for code, benchmark, empirical analysis.**

Manuscript received 8 June 2023; revised 27 February 2024; accepted 29 February 2024. Date of publication 13 March 2024; date of current version 16 May 2024. The work of Yuejun Guo was supported by the European Union's Horizon Research and Innovation Programme, as part of the Project LAZARUS under Grant 101070303. This work was supported by Luxembourg National Research Funds (FNR) through CORE Project C18/IS/12669767/STELLAR/LeTraon. Recommended for acceptance by D. Hao. *(Corresponding author: Lei Ma.)*

Qiang Hu, Maxime Cordy, Mike Papadakis, and Yves Le Traon are with the University of Luxembourg, L-1359 Belval, Luxembourg (e-mail: qianghu0515@gmail.com; maxime.cordy@uni.lu; mike.papadakis@uni.lu; yves.letraon@uni.lu).

Yuejun Guo is with Luxembourg Institute of Science and Technology, L-1359 Belval, Luxembourg (e-mail: yuejun.guo@list.lu).

Xiaofei Xie is with Singapore Management University, Singapore 188065 (e-mail: xiaofei.xfxie@gmail.com.

Lei Ma is with the University of Tokyo, Tokyo 113-0033, Japan, and also with the University of Alberta, Edmonton, AB T6G 2R3, Canada (e-mail: ma.lei@acm.org).

This article has supplementary downloadable material available at https://doi.org/10.1109/TSE.2024.3376964, provided by the authors.

Digital Object Identifier 10.1109/TSE.2024.3376964

I. Introduction

Leveraging machine learning (ML) to help developers solve software problems (ML4Code) [1] has been a hot direction in both software engineering (SE) and ML communities in recent years. Deep learning (DL), one of the advanced ML techniques, has achieved great success in various software tasks, such as code summarization [2], code clone detection [3], and vulnerability detection [4]. Typically, developing a code model involves two main steps: first, building a pre-trained model that learns general code information; second, fine-tuning this model using datasets that target a specific downstream task. Both components contribute to the success of ML4Code and are still under exploration for further improving the performance of code models.

Generally, pre-trained code models can be easily accessed from open resources, e.g., Hugging Face [5], or built by using self-supervised learning without data labeling effort. This means that for developers planning to use code models, the first step of pre-trained model preparation is not challenging and can be fully automated. However, collecting datasets to fine-tune pre-trained models is not easy. The main reason is that the general fine-tuning process follows the procedure of fully-supervised learning, which requires carefully labeled training data. Unfortunately, the data labeling process is time-consuming and labor-intensive [4].

To alleviate the aforementioned heavy effort of training data labeling, active learning [6] is used to enable sample-efficient model training in other famous fields, e.g., computer vision [7] and natural language processing [8]. The key idea of active learning is to iteratively select a subset of training data to label and use them to train the model. The existing studies [9] have shown that labeling only a few (less than 10%) training data can train a model with similar performance as the model trained by using the entire training data. In this way, the labeling effort can be significantly reduced and made flexible to a fixed budget. However, despite being well-studied in many application domains, the usefulness of active learning in ML4Code is still unknown. Researchers mainly focus on designing new model architectures, proposing novel code representation methods, and studying more software tasks. The study of how to lighten the model training cost is missed. There is a need to provide a benchmark to support the exploration of this important problem.

In this paper, we aim to bridge this gap and build a benchmark to study how active learning can help us efficiently build code

© 2024 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License. For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

models – active code learning. We collect acquisition functions (i.e., active learning methods) that can be used for code data from existing studies [9], [10]. In total, we implement 11 acquisition functions (including random selection). These functions can be categorized into five output-uncertainty-based functions, four clustering-based functions, and one function that uses both the input and output information for the selection. Then, based on these collected acquisition functions, we conduct experiments on six datasets (including classification tasks and non-classification tasks) and four famous pre-trained code models to answer the following research questions:

RQ1: What features should be used for clustering-based acquisition functions? Feature selection [11] is an important problem for clustering methods. However, it is unclear which features should be used for clustering-based methods in active code learning. Our first study is to explore how different features affect the effectiveness of active code learning. Here, we consider three types of features, code tokens, code embedding vectors, and model output vectors. *Findings:* The results show that clustering-based methods are sensitive to the used features. Interestingly, in more than half of cases (60%), output vector-based acquisition functions achieve significantly better results than code token and code embedding-based functions.

RQ2: How do acquisition functions perform on code models? After determining features to use, we compare all the acquisition functions across different tasks. *Findings:* Firstly, contrary to the previous study [10], which suggests that simple techniques perform better for active learning, we found that clustering-based methods consistently outperform simple uncertainty methods in our considered binary classification code tasks (Clone detection and vulnerability detection). Secondly, unlike prior research [9], [10], which shows that only a small set of training data (less than 10%) is sufficient to produce good models, our results on non-classification tasks (code summarization and code translation) indicate that existing methods are not yet capable of achieving this goal. There is at least a 30% performance gap between the models trained by active learning (with 10% data) and the models trained using the entire training data. Finally, our case study reveals that smaller token distribution differences between the selected code data and the whole training data and higher code token (and label) diversity contribute to the acquisition functions to produce code models with better performance. In total, in the first two studies, we trained 9900 models considering each labeling budget. The training process takes more than 5000 GPU hours.

Exploratory study. Additionally, using our benchmark, we conduct a study to further explore potential directions for proposing new acquisition functions. We focus on clustering-based methods since existing methods can not perform well on non-classification tasks. Concretely, due to clustering-based methods tending to select diverse data (data have a bigger distance to each other), we first check if there is a correlation between the distance of selected data and the accuracy of the trained model. Then, we propose a novel view to consider the distance of code for active learning – *using evaluation metrics (e.g., CodeBERTScore) as distance calculation methods*. Based on the evaluation, we found that, for non-classification code tasks, 1) there is no correlation between the distance (calculated by Cosine similarity and Euclidean distance) of selected data to each other and the accuracy of models. 2) There is a weak correlation between the evaluation metrics-based distance of selected data to each other and the accuracy of models, indicating that future proposed methods can be based on evaluation metrics-based distance.

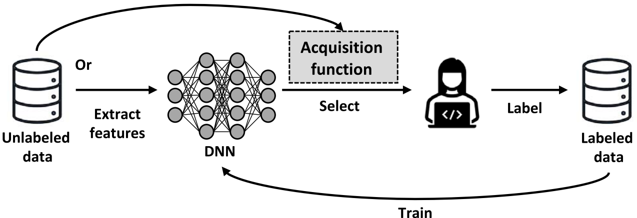


Fig. 1. Overview of active learning.

To summarize, the main contributions of this paper are:

- This is the first work that builds a benchmark for sample-efficient training across both classification and non-classification code tasks. *The source code as well as the datasets can be found on our project site [12].*
- Our analysis, grounded in the empirical study from this work, reveals a significant divergence from established findings [9], [10] in image and text data when applied to code data.
- We design a novel strategy that uses evaluation metrics to quantify the distance between code pairs to support future research when proposing new acquisition functions.

II. Background

A. Active Learning

Active learning is a well-known technique that enables sample-efficient model training. Fig. 1 depicts the overview workflow of active learning. Given an unlabeled dataset and a model under training, the first step of active learning is to choose the features used for conducting data sampling. Generally, two types of features can be used, 1) the data features itself (e.g., image pixels, and code tokens), and 2) features extracted from the model (e.g., output probabilities, and code embeddings). After obtaining the features, acquisition functions are used to select the most valuable data for labeling. Here, important means the model can learn more information from this kind of data and achieve better performance on test data. Finally, developers label these selected data and use them to train the model. In this way, developers can train a model under a fixed labeling budget. When a new budget is allowed, developers repeat this process and further enhance the trained model.

The acquisition function is the most important part of active learning which decides the quality of labeled training data. Existing acquisition functions can be roughly divided into two groups, output-uncertainty-based functions, and clustering-based functions. Output-uncertainty-based functions obtain the model predictions of the unlabeled data first and then use some uncertainty metrics (Entropy of output probabilities) to rank these data and select the most uncertain ones for training.

TABLE I
OUR CONSIDERED ACQUISITION FUNCTIONS. BOTH: CLASSIFICATION AND NON-CLASSIFICATION TASKS

Method Name	Support Tasks	Description
Random	Both	Randomly select a budget of data.
Least Confidence (LC) [13]	Classification	Select the data with the least top-1 probability.
DeepGini [14]	Classification	Select data with minimum Gini impurity $\sum_{i=1}^{N}(p_i(x))^2$.
Bayesian Active Learning by Disagreement (BALD) [15]	Classification	Select data minimum disagreement $\frac{count(mode(y_x^1,\ldots,y_x^T))}{T}$ over T dropouts.
Entropy [13]	Classification	Select data with the minimum Shannon entropy on output probabilities.
Margin [13]	Classification	Select data with a minimum difference between the top-1 probability and top-2 probability.
Contrastive Active Learning (CAL) [16]	Classification	Select a subset that 1) is far away from the labeled data, 2) included data samples have big output divergence.
K-Means (KM) [7]	Both	Use the K-means clustering method to divide data into K groups and select the center of each group.
K-Center (KC) [7]	Both	Iteratively select data that are far from the labeled data.
BADGE [17]	Both	Select data by using K-means on the gradient embedding level.
Coreset [7]	Both	An advanced version of K-Center with a distance upper bound.

The intuition behind this type of function is that uncertain data are usually close to the decision boundaries of the model, thus, training the model using these data can force the model to build clear boundaries and improve its correctness. Clustering-based functions assume that the models have better performance if they learn more diverse data information. Thus, people use different distance methods to measure the distance of data to each other and select the central data for model training, e.g., using K-means to do data clustering and then select the center data.

In this work, we follow the previous works [9], [10] and collect and implement 11 acquisition functions for active code learning. Let X be the training data and $x \in X$ be an input sample. N is the number of classes. y and y_x are the ground-true label and predicted labels. $p_i(x)$ represents the predicted probability of x belonging to the ith class. Table I presents these acquisition functions with their brief descriptions. For detailed information on each function, please refer to the original paper.

B. Machine Learning for Code (ML4Code)

ML4Code [1] is a new but hot direction in the current software engineering research field. The basic idea of ML4Code is to train a machine learning model to solve software problems, for example, program repair, vulnerability detection, and code summarization. Machine learning models and training data play key roles in ML4Code. For machine learning models, due to the *naturalness* of software, researchers believe model architectures that are good at handling text data are also promising for solving code data. Thus, the famous code models mainly come from modifying natural language models. For instance, CodeBERT [18] is a bi-modal extension of the Bidirectional Encoder Representations from Transformers (BERT) [19] which was proposed for natural language processing. The training data are usually processed by some code representation techniques to transfer the raw data to a machine-readable format, e.g., transfer the strings of code to a sequence of tokens with integer format. How to design this transformation (i.e., code representation) is another research topic that highly affects the performance of trained models.

In this work, we focus on the most practical paradigm of ML4Code – building models of code for specific downstream code tasks by fine-tuning pre-trained code models. Existing studies [18], [20] already demonstrated that fine-tuning pre-trained models can achieve better results than training the models from scratch. Here, the pre-trained code model has been trained on multi-language datasets, therefore, they already learned general information of code, e.g., the pre-trained CodeBERT model learns knowledge from six datasets with different programming languages. As a result, developers only need to prepare their dataset for a specific task and use it to fine-tune the pre-trained model.

III. BENCHMARK

After collecting massive acquisition functions whose effectiveness on image data and text data has already been demonstrated in previous studies, we plan to study their unknown use case – if they are still useful for efficient training models of code. As mentioned in Section I, we have two main questions want to answer, 1) since features used for clustering-based acquisition functions are important and highly related to the final performance of active learning, we explore for each acquisition function, which kind of feature is suitable for conducting code selection. 2) As the previous study [10] claimed that – simple methods perform well on active learning, we want to check if this conclusion still holds on active code learning. Concretely, we compare output-uncertainty-based functions (i.e., simple methods mentioned in [10]) with clustering-based functions with carefully chosen features.

A. Study Design

To address the question of suitable feature selection (**RQ1**), we first prepare different versions of clustering-based acquisition functions based on the features they use. Note that, output-based acquisition functions directly use the output-probability to measure the uncertainty of the corresponding inputs. Since the output probability has a fixed format, one-hot vectors, there are no feature selection problems in such acquisition functions. Then, we conduct active learning on different code tasks using

TABLE II
DETAILS OF DATASETS AND MODELS. ACCURACY (%) FOR PROBLEM CLASSIFICATION AND VULNERABILITY DETECTION, F1-SCORE (%) FOR CLONE DETECTION, AND PPL/BLEU FOR CODE SUMMARIZATION AND CODE TRANSLATION

			Classification			
Task	**Dataset**	**Language**	**Train/Dev/Test**	**CodeBERT**	**GraphCodeBERT**	**CodeT5**
Problem Classification	Java250	Java	62500/-/12500	98.10%	98.49%	98.39%
Clone Detection	BigCloneBench	Java	90102/4000/4000	97.15%	97.05%	97.87%
Vulnerability Detection	Devign	C	21854/2732/2732	63.76%	62.26%	62.85%
			Non-Classification			
Task	**Dataset**	**Language**	**Train/Dev/Test**	**CodeBERT**	**GraphCodeBERT**	**RoBERTa**
Code Summarization	CodeSearchNet	JavaScript	58025/3885/3291	3.85/14.34	3.79/14.89	5.15/13.34
Code Summarization	CodeSearchNet	Ruby	24927/1400/1261	4.04/12.80	3.99/13.54	5.15/11.25
Code Translation	CodeTrans	Java-C#	10300/500/1000	1.44/78.57	1.43/79.15	1.44/76.99

these functions and compare the performance of trained models. we can draw conclusions about which features are most suitable for each acquisition function. Here, we focus on three types of features from different perspectives, 1) code embeddings, 2) sequence of code tokens, and 3)model output vectors.

- **Code Embeddings** It is common to consider code embeddings as the input of clustering methods since code embeddings produced by pre-trained code models can present the general information of code. Code embedding is similar to the image data after image pre-processing. In our study, code embeddings extracted from the fine-tuned code models which can better represent the downstream task are used as the features.
- **Sequence of code tokens** Regardless of the code representation techniques, the raw program will be converted into a sequence of integer values. Thus, it is also possible to use these sequences as the inputs of the clustering methods. It is similar to using the pixel numbers of images as inputs. The only difference is that code tokens often with bigger data space, i.e., the image pixels are from 0 to 255, while the range of code tokens depends on the vocabulary size, which is generally much bigger than 255, e.g., the vocabulary size of our used CodeBERT model is 50265.
- **Model outputs** Different from the above two features that have been widely explored in other fields and can be easily considered for active code learning, in our study, we propose to use the model outputs as the input features of clustering methods. The intuition behind this idea is that the output can be seen as the *understanding* of the model on this input data which should be useful for data selection. Specifically, for classification tasks, we use the one-hot output probabilities as the features, and for the non-classification tasks, the output vectors produced by decoders are used as the features.

For the comparison of each acquisition function (**RQ2**), the main goal is to find the recommended function that has consistently better performance than others in active code learning. At the same time, we also compare the output-uncertainty-based functions and clustering-based functions to determine if the previous findings [10] hold true in active code learning. Here, based on the first study, we use suitable features as the input for clustering-based acquisition functions and perform code selection.

B. Dataset and Model

Table II presents the datasets and models we used in the study. We follow the previous works [21], [22], [23] and consider five code tasks and one dataset per task including a multi-class classification task (problem classification), two binary classification tasks (clone detection and vulnerability detection), and two non-classification tasks (code summarization and code translation).

- **Problem classification** is a multi-class classification task. Given a program, the model predicts the target problem that the program solves. We use the dataset JAVA250 [24] provided by IBM for this task. JAVA250 contains 250 code problems, e.g., problem: *write a program which prints the heights of the top three mountains in descending order.*
- **Clone detection** is a well-studied task in the software engineering field to lighten the effort of software maintenance. The main purpose of this task is to check if two programs are semantically equivalent. Thus, code clone detection is often seen as a binary classification problem. In our study, we use the dataset provided by BigCloneBench [25] which contains a large number of Java code clone pairs. Besides, for the computation friendly, we follow the previous work [22] and only use a subset of data from BigCloneBench.
- **Vulnerability detection** is a security-critical task that aims at localizing the vulnerable functions in source code to protect the software. Given a program, the code model identifies whether the program is vulnerable or not, which is also a binary classification problem. We use Devign [26] for this task which is constructed by four C libraries.
- **Code summarization** is a common code task for helping developers understand code snippets. Given a program, the model generates natural language comments to describe the functionality of this program. We use two datasets provided by Microsoft [27] in our study.
- **Code translation** transfers programs from one language to another which is a typical task for software migration. In this work, we study this task using the dataset provided by [27] which focuses on translating Java programs to C# programs.

For the code models, we focus on pre-trained code models since they achieved much better results than models trained from scratch [18], [20] and are widely studied now. We follow the previous work [22] and use two well-known pre-trained

code models in our study, CodeBERT [18] and GraphCodeBERT [20]. Besides, we added one advanced pre-train model CodeT5 [23] in our work. Note that, we found CodeT5 is already well pre-trained for our studied two non-classification tasks, using active learning to fine-tune it can not significantly improve its performance, e.g., the pre-trained CodeT5 model already has around 16 BLEU score on code summarization task (JavaScript), and after the fine-tuning, the BLEU score is also less than 17. Therefore, we only train CodeT5 for classification tasks and build another model RoBERTa (Robustly Optimized BERT) [28] for non-classification tasks. Other models can be easily added and evaluated to our provided project. Considering the downstream tasks, the pre-trained model is used as an encoder to generate code embeddings, and a decoder is followed to produce the final results for a specific code problem. For example, a dense layer is used as the decoder to produce the output probabilities of classification tasks. The same as the original works of our considered code models, during the fine-tuning, we also fine-tune the parameters of pre-trained encoders to ensure a better performance on downstream tasks. Then, we briefly introduce the two pre-trained models here, and the detailed model information can be found on our project site [12].

- **CodeBERT** is a bimodal model that shares a similar architecture with the well-known BERT model in the field of natural language processing. The CodeBERT model is initialized through pre-training with six code datasets featuring various programming languages such as Java and Python. During the training process, programs are transformed into sequences of tokens, which is similar to text data. As a result, CodeBERT is able to learn the semantic information of code data.
- **GraphCodeBERT** is a newer pre-trained code model that incorporates both sequences of tokens and data-flow information to learn code knowledge. This allows pre-trained code models to understand the structural information of programs and generate more precise code representations. In our study, the base model is learned by this combination of code information and fine-tuned for our considered downstream tasks only using the code token information. The reason is that we found in some downstream tasks, adding the data-flow information to fine-tune the model harms the performance of models, e.g., for the problem classification task, adding data-flow information and without data-flow information, the accuracy of fine-tuned models is 82.30% and 98.39%, respectively.
- **RoBERTa** is a variant of the BERT [19] model. The difference lies in the pre-training strategy. Compared to BERT, the training of RoBERTa lasts longer with bigger batches and additional new training data. Rather than performing masking once during data pre-processing, as done in BERT, RoBERTa employs the dynamic masking strategy where the masking is performed every time when feeding a sequence to the model during pre-training. Additionally, both the masked language modeling (MLM) objective and the next sentence prediction (NSP) loss are applied in the BERT pre-training procedure, while RoBERTa removes the NSP loss to improve the downstream task performance. Furthermore, RoBERTa is trained with a larger byte-level Byte-Pair Encoding (BPE) vocabulary while BERT uses the character-level BPE vocabulary. RoBERTa is widely used as the basic baseline for code learning [18], [20]. In this work, we employ RoBERTa for non-classification tasks.
- **CodeT5** is a variant of the Text-To-Text Transfer Transformer (T5) [29] model specifically tailored for code understanding tasks. It adopts the same architecture as T5 but incorporates code-specific knowledge. CodeT5 is pre-trained on a large corpus of code with accompanying natural language comments. It employs the Masked Span Prediction (MSP) objective during pre-training. Additionally, two auxiliary tasks, Identifier Tagging (IT) and Masked Identified Prediction (MIP), are applied to fuse more code-specific structural information into the model. It achieved superior results in multiple downstream tasks such as code translation and defect detection. In this work, we employ CodeT5 for classification tasks.

C. Evaluation Metrics

Our study contains three types of code tasks. For each task, we use the most practical metrics to evaluate the trained models.

Accuracy on the test data is the basic way to evaluate the performance of multi-class classification models. It calculates the percentage (%) of correctly classified data over the entire input data.

F1-score is a commonly used metric for binary classification problems. It calculates the harmonic mean of the precision and recall scores. Given that, true positive (TP) represents the number of samples correctly predicted as positive, false positive (FP) represents the number of samples wrongly predicted as positive, and false negative (FN) represents the number of samples wrongly predicted as negative, F1-score is calculated as:

$$F1-score = \frac{TP}{TP + \frac{1}{2}(FP + FN)}$$

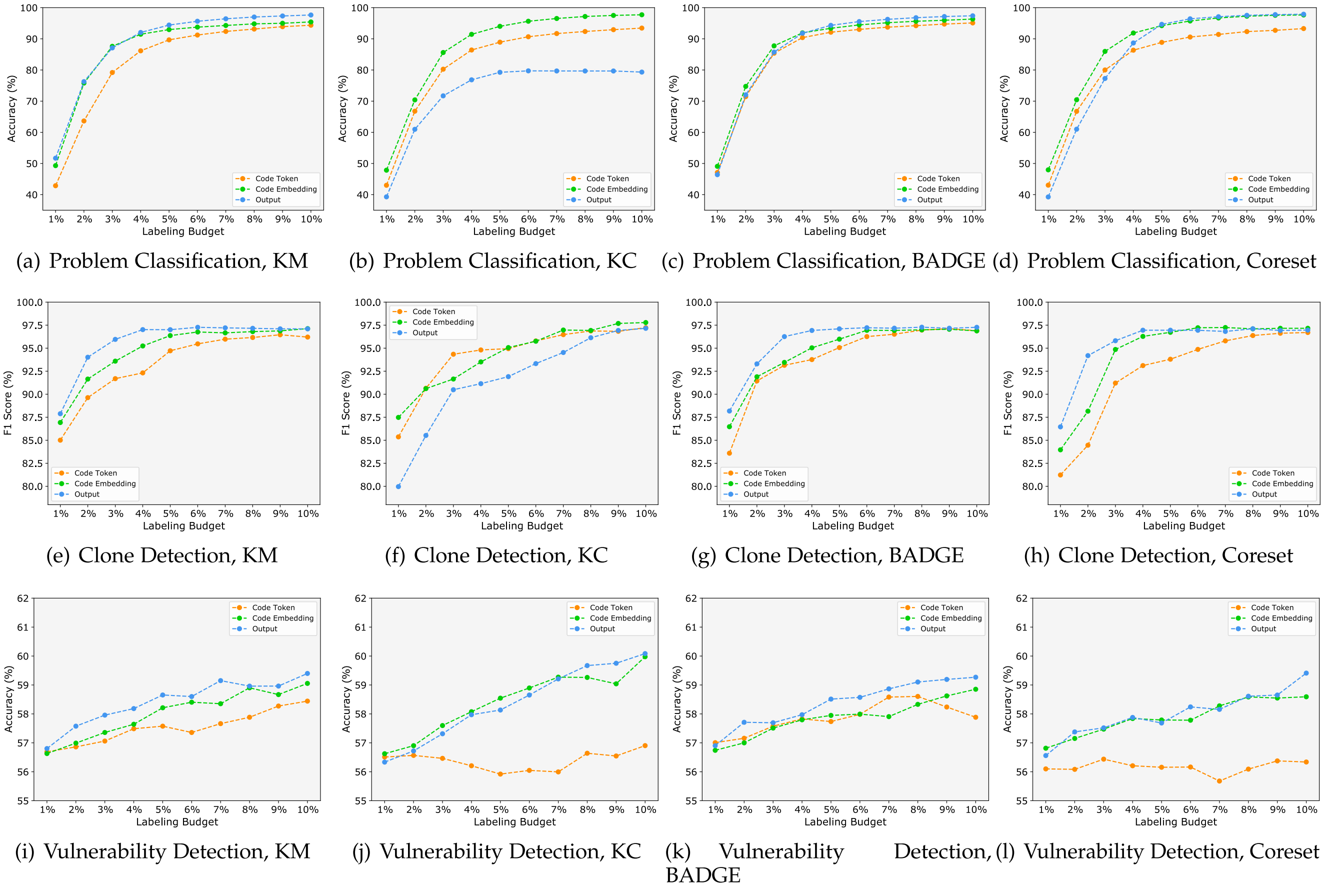
Perplexity (PPL) is a widely used metric to evaluate language models. PPL can be seen as the loss of language models that can record the logs of the training process. A lower PPL score means a better performance of the model. Recently, researchers applied PPL to record to evaluate the code summarization models [30]. Specifically, PPL is calculated by:

$$PPL(X) = \exp\left\{-\frac{1}{t}\sum_{i=0}^{t}\log p_\theta(x_i|x_{<i})\right\}$$

where $X = (x_0,\ x_1,\ldots,x_t)$ is the set of code tokens, and $logp_\theta(x_i|x_{<i})$ is the log-likelihood of the ith token conditioned on the preceding tokens $x_{<i}$ by the model.

BLEU (Bilingual Evaluation Understudy) is a metric to evaluate the quality of the generated text to another (the reference). Simply, BLEU score is calculated by:

$$BLEU = BP \times \exp\sum_{n=1}^{N} w_n \log p_n$$



(a) Problem Classification, KM (b) Problem Classification, KC (c) Problem Classification, BADGE (d) Problem Classification, Coreset

(e) Clone Detection, KM (f) Clone Detection, KC (g) Clone Detection, BADGE (h) Clone Detection, Coreset

(i) Vulnerability Detection, KM (j) Vulnerability Detection, KC (k) Vulnerability Detection, BADGE (l) Vulnerability Detection, Coreset

Fig. 2. Results of different feature-based active code learning on classification tasks.

The value of N depends on the used N-gram precision, and the weights w_n = N / 4. In our study, 4 is used. p_n is the ratio of length n sub-sequences in both the candidate sequence and the reference. BP is the brevity penalty calculated by:

$$BP = \begin{cases} 1, & if\ c > r \\ e^{1-r/c}, & if\ c \leq r \end{cases}$$

where c is the length of the generated sequence and r is the length of the reference sequence.

Besides, to perform a significance test for the comparison between different acquisition functions, we conduct statistical analysis by using Student's t-test [31] in our study.

D. Configurations

Training configuration. Considering the hyperparameters used for model training, we follow the previous work [22] and use the same batch size and learning rate for each model. All the detailed settings can be found on our project site. For model fine-tuning, we set the training epoch as 10 which is enough to ensure the convergence of models.

Active learning configuration. We initialize the models (which is a common setting of active learning that we start from a model with a little knowledge of the datasets) by training them on randomly selected 500 samples. We set the labeling budget as 1% of the entire training set and do 10 times iterations of active learning.

Acquisition function configuration. For all clustering-based methods, we follow the previous work [9] and set the number of centers as the labeling budget. For BALD, we set the times of dropout prediction as 20. For all clustering-based methods, to get more precise code information, we obtain the embedding features from the fine-tuned encoders.

E. Implementation and Environment

The project is based on Python-3.6 and PyTorch-1.10.2 framework. The key implementation of code models is modified from the open source project [27]. We adopt acquisition functions implemented by [9], [10] that are used for image data and text data to code data. All the source code can be found on our project site. We conduct all the experiments on a 2.6 GHz Intel Xeon Gold 6132 CPU with an NVIDIA Tesla V100 16G SXM2 GPU. We repeat all the experiments five times to reduce the influence of randomness and report the average results in the following sections.

F. RQ1: Feature Selection

First, we explore the features of clustering-based acquisition functions. Fig. 2 and Fig. 3 depict the performance of models trained by different labeling budgets. Here, we only list the results of CodeBERT models and the whole results can be found on our project site.

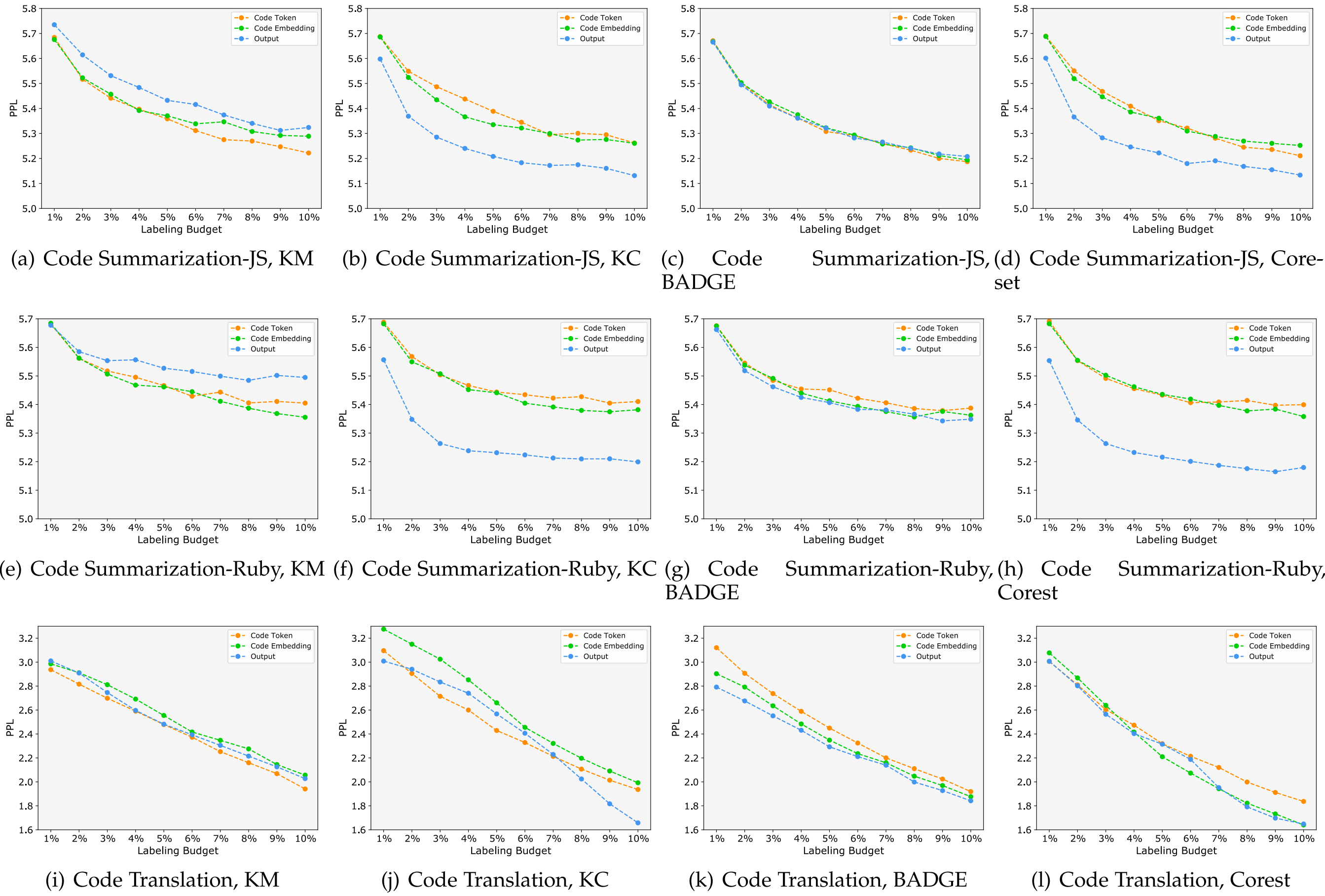


Fig. 3. Results of different feature-based active code learning on non-classification tasks. JS is short for JavaScript.

From the results, we can see that the same acquisition function could train models with significantly different performances depending on the features used. In particular, for the *K-Means*, *K-Center*, and *Coreset* functions, the performance difference is substantial and should not be overlooked. In contrast, the *BADGE* function is relatively stable compared to the others. Then, we analyze the most suitable features that should be used for each function for solving each code task. Table III displays the percentage of each function that produces models with the best performance across all labeling budgets. Surprisingly, overall, the results demonstrate that the output vectors extracted from the models are superior features compared to code tokens and code embeddings. This suggests that active code learning should focus on output vectors rather than input vectors when conducting data selection, unlike active learning for image and text data. We also use t-test metric to test the signification of the comparison. Table IV summarizes the results. This result also indicates that the output feature is the best one among our considered features. For example, for clone detection tasks, the output feature beats token and embedding features in KM, BADGE, and Coreset three acquisition functions.

Besides, we observe that the feature selection of some acquisition functions also depends on the types of downstream tasks. In both classification tasks, output vectors and code embeddings are the best choices for *K-Means* and *K-center*, respectively. However, In the non-classification code summarization task, the choices become code tokens and output vectors. On the other hand, output vectors are the most useful features for *BADGE* and *Coreset* regardless of the type of tasks.

TABLE III
RATIO OF TRAINED MODELS ACHIEVING BEST RESULTS USING DIFFERENT FEATURES

	KM	**KC**	**BADGE**	**Coreset**	**Average**
			Problem Classification		
Token	0%	0%	0%	0%	0%
Embedding	10%	100%	40%	40%	47.5%
Ouput	90%	0%	60%	60%	52.5%
			Clone Detection		
Token	0%	40%	0%	0%	10%
Embedding	10%	60%	0%	40%	27.5%
Output	90%	0%	100%	60%	62.5%
			Vulnerability Detection		
Token	0%	0%	10%	0%	2.5%
Embedding	0%	70%	0%	70%	35%
Output	100%	30%	90%	30%	62.5%
			Code Summarization		
Token	50%	0%	20%	0%	17.5%
Embedding	45%	0%	15%	0%	15%
Output	5%	100%	65%	100%	67.5%
			Code Translation		
Token	100%	60%	0%	10%	42.5%
Embedding	0%	0%	0%	40%	10%
Output	0%	40%	100%	50%	47.5%

TABLE IV
COMPARISON BETWEEN DIFFERENT FEATURES (t-TEST)

	KM	KC	Badge	Coreset
Problem Classification				
Token vs. Embedding	-0.9431	-1.7146	-0.6646	-1.7504
Token vs. Output	-1.6223	-1.6208	-0.6277	-1.6824
Embedding vs. Output	-0.6610	5.2458	-0.2266	0.3993
Clone Detection				
Token vs. Embedding	-1.9726	-0.5992	-1.1039	-2.7805
Token vs. Output	-2.1633	1.8298	-2.3139	-3.1667
Embedding vs. Output	-1.6111	3.7102	-2.6944	-3.0582
Vulnerability Detection				
Token vs. Embedding	-3.0216	-10.8523	-0.0595	-11.7910
Token vs. Output	-5.4160	-9.6567	-3.3622	-11.9319
Embedding vs. Output	-2.1204	0.1349	-2.7474	-0.6913
Code Summarization				
Token vs. Embedding	-0.2400	2.0836	2.4792	2.8578
Token vs. Output	-7.3257	6.2842	0.5840	6.5609
Embedding vs. Output	-7.0504	4.6275	-1.9706	4.0003
Code Translation				
Token vs. Embedding	-1.3914	-2.0507	1.2889	1.0177
Token vs. Output	-0.7749	0.1396	2.1774	1.1127
Embedding vs. Output	0.6059	1.9714	0.9068	0.0563

Based on the experimental results, we provide recommendations for the feature selection in clustering-based acquisition functions for active code learning.

- **K-Means-C (KM-C)**: use model output vectors (code tokens) for classification (non-classification) tasks.
- **K-Center-C (kC-C)**: use code embeddings (model output vectors) for classification (non-classification) tasks.
- **BADGE-C**: use model output vectors for all code tasks.
- **Coreset-C**: use model output vectors for all code tasks.

Answer to RQ1: Feature selection highly influences the effectiveness of clustering-based acquisition functions to perform active learning. Surprisingly, in most (60%) cases, leveraging output vectors extracted from models in active learning outperforms using code tokens and code embeddings, resulting in improved performance of code models.

G. RQ2: Acquisition Function Comparison

After studying the feature selection, we compare all acquisition functions on different code tasks. Here, we use our recommended features to build the clustering-based function. Table V presents the results of classification tasks.

Acquisition function comparison in classification tasks. For the multi-class classification task (problem classification), we can see that output-uncertainty-based methods often achieve better results than the clustering-based methods. In which, *Margin* which only uses the top-1 and top-2 probabilities of the outputs performs the best in 5 out of 6 cases. This phenomenon draws a similar conclusion to the previous work [10], that simple methods perform well on active learning. However, regarding the binary classification task (clone detection and vulnerability detection), interestingly, the results show that the previous conclusion cannot stand. For the clone detection task, in all cases, clustering-based methods outperform *simple methods* (output-uncertainty-based methods). Table VI shows the results of the comparison between BADGE-C (the best clustering-based method) and all output-uncertainty-based methods on the code clone detection task. The results also demonstrate that except for labeling budget 10%, BADGE-C outperforms *simple methods* in this task. For the vulnerability task, clustering-based methods also perform the best in 6 out of 9 cases. In summary, the first conclusion we can draw is – no acquisition functions consistently perform better than others, and findings [10] from previous works can not be directly applied to active code learning.

Acquisition function comparison in non-classification tasks. Then, move to the non-classification task (two code summarization tasks and the code translation task), Table VII, Table VIII, and Table IX show the results. The first finding reveals disparate conclusions drawn from the utilization of different evaluation metrics. For instance, the PPL scores demonstrate that *KC-C* is the best acquisition function for code summarization tasks while the BLEU scores suggest *Coreset-C*. However, regardless of the evaluation metrics we used, the gap between the performance of active learning-trained models and the performance of models trained by using the entire data is big. For example, for JavaScript-CodeBERT, under 10% labeling budget, the best PPL score and BLEU score we get are 5.1313 and 10.09, which are 33.28% and 29.64% lower than the 3.85 and 14.34 computed from the model trained by entire training data. These results are totally different from the ones drawn by the classification tasks that using 10% data can train a model with similar and even better performance, e.g., for the clone detection task, models trained with 10% (97.79%) data have better performance than the models trained by entire training data (97.15%). Therefore, we can say that active code learning in non-classification code tasks is still in a very early stage, the conclusions from classification tasks can not be migrated to non-classification tasks.

Labeling budgets study. To study how many labeling budgets we need for active code learning to train a good model (with similar performance to the model trained by using the whole training data), we further conduct studies by increasing the labeling budgets for vulnerability detection and non-classification tasks that 10% labeling budgets are not enough. The results can be found in the Appendix (available online).

Comparison between acquisition functions to random selection. Additionally, to investigate the importance of using suitable acquisition functions in active code learning, we compare the best acquisition functions in each task to the random selection function. Specifically, we compare Margin, Badge, KM, and Coreset to Random for problem classification, clone detection, vulnerability detection, and three non-classification tasks, respectively using T-test. Table X presents the coefficient and P-value for this comparison. From the results, we can see that, in 16 out of 18 cases, the statistical results demonstrate that the selected acquisition functions achieve better results than the random selection. Besides, in half of these 16 cases, the models trained by the selected acquisition functions have significantly (with a P-value less than 0.03) better performance than models trained using random selection. We can conclude that, when using active code learning to prepare the code models,

TABLE V
MODEL PERFORMANCE OF ACTIVE LEARNING TRAINED MODELS FOR TWO CLASSIFICATION TASKS. VALUES HIGHLIGHTED IN RED AND BLUE INDICATE THE BEST AND SECOND BEST. OUTPUT: OUTPUT-UNCERTAINTY-BASED METHODS. CLUSTERING: CLUSTERING-BASED METHODS. UPPER: UPPER BOUND ACHIEVED BY ALL ACQUISITION FUNCTIONS

		CodeBERT			**GraphCodeBERT**			**CodeT5**		
		1%	5%	10%	1%	5%	10%	1%	5%	10%
					Problem Classification					
Output	**LC**	29.61	52.90	59.14	33.99	56.93	61.06	35.84	47.52	50.64
Output	**Gini**	43.96	90.95	97.75	44.31	93.02	98.04	52.61	94.24	98.18
-	**Random**	48.01	92.67	95.09	55.05	93.75	95.76	58.08	93.96	95.82
Output	**BALD**	40.63	94.51	97.86	50.83	94.32	98.02	55.29	95.62	98.12
Output	**Entropy**	42.67	90.05	97.67	44.50	92.21	97.99	52.31	93.75	98.15
Output	**Margin**	51.22	95.41	97.91	57.76	96.33	98.16	59.62	96.99	98.22
-	**CAL**	29.99	54.51	64.99	34.04	57.68	65.33	34.78	56.58	67.76
Clustering	**KM**	51.69	94.42	97.65	57.38	95.41	97.99	48.36	87.32	95.01
Clustering	**KC**	47.84	94.01	97.73	56.18	95.57	97.33	44.44	87.46	93.93
Clustering	**Badge**	46.41	94.33	97.39	53.48	95.31	97.91	52.04	95.57	97.91
Clustering	**Coreset**	47.96	94.23	97.66	56.35	95.48	97.24	47.10	96.52	98.21
	Upper	51.69	95.41	97.91	57.76	96.33	98.16	59.62	96.99	98.22
					Clone Detection					
Output	**LC**	80.08	83.51	90.85	79.90	81.73	90.12	93.48	94.75	95.66
Output	**Gini**	85.45	96.58	96.74	84.45	96.41	97.18	95.30	97.02	96.50
-	**Random**	84.22	95.98	96.86	84.83	95.52	96.81	94.18	97.12	96.99
Output	**BALD**	87.61	96.81	97.16	88.73	96.47	97.12	94.11	94.97	96.75
Output	**Entropy**	86.16	96.32	96.89	85.28	96.89	97.17	95.40	97.01	96.72
Output	**Margin**	86.38	96.82	96.92	84.92	97.02	97.17	95.46	96.98	96.67
-	**CAL**	79.57	91.10	95.14	77.60	88.45	92.06	92.64	92.87	96.55
Clustering	**KM**	87.88	97.01	97.10	87.80	96.92	97.23	95.58	97.30	96.78
Clustering	**KC**	87.48	95.07	97.79	85.77	96.85	97.62	96.25	96.90	97.50
Clustering	**Badge**	88.17	97.10	97.27	90.60	97.25	97.09	96.70	97.70	97.14
Clustering	**Coreset**	86.45	96.95	96.96	92.10	97.19	97.06	96.52	97.22	97.10
	Upper	88.17	97.10	97.79	92.10	97.25	97.62	96.70	97.70	97.50
					Vulnerability Detection					
	LC	56.34	56.73	58.09	56.27	57.12	57.84	55.99	55.12	56.46
Output	**Gini**	57.12	58.73	59.47	57.01	59.64	60.28	55.81	57.23	57.59
-	**Random**	56.51	57.74	59.00	57.14	58.53	59.29	55.71	56.19	57.02
Output	**BALD**	57.50	58.41	59.63	57.17	59.35	59.69	56.48	56.94	58.60
Output	**Entropy**	57.00	58.48	59.25	57.19	59.31	59.79	56.30	56.79	57.91
Output	**Margin**	56.85	58.42	59.44	57.27	59.15	59.77	56.13	56.49	57.93
-	**CAL**	56.35	57.45	58.13	57.07	58.67	59.19	56.47	57.20	58.00
Clustering	**KM**	57.30	59.15	59.90	57.63	59.49	60.52	56.59	57.27	58.57
Clustering	**KC**	56.62	58.54	59.98	57.06	57.66	59.10	56.38	56.48	57.88
Clustering	**Badge**	56.90	58.51	59.27	57.06	58.83	59.33	56.13	56.32	57.32
Clustering	**Coreset**	56.56	57.69	59.01	56.99	58.37	59.75	56.27	56.93	57.48
	Upper	57.50	59.15	59.98	57.63	59.64	60.52	56.59	57.27	58.60

TABLE VI
COMPARISON BETWEEN BADGE AND OUTPUT-UNCERTAINTY-BASED METHODS (t-TEST). TASK: CLONE DETECTION

	1%	5%	10%
		CodeBERT	
BADGE-C vs. LC	8.4889	5.9074	6.2645
BADGE-C vs. Gini	2.5009	4.0470	3.1028
BADGE-C vs. Blad	0.4731	1.7192	0.5137
BADGE-C vs. Entropy	2.1334	2.2341	1.7796
BADGE-C vs. Margin	1.9832	1.6134	2.1428
		GraphCodeBERT	
BADGE-C vs. LC	23.2590	10.0688	12.3815
BADGE-C vs. Gini	11.4256	3.4707	-0.5947
BADGE-C vs. Blad	1.8989	2.8951	-0.2334
BADGE-C vs. Entropy	10.2222	1.5331	-0.7654
BADGE-C vs. Margin	12.1774	1.9214	-0.4843
		CodeT5	
BADGE-C vs. LC	18.7178	6.5411	3.8040
BADGE-C vs. Gini	2.3301	1.2324	0.2511
BADGE-C vs. Blad	7.4407	1.1518	0.1810
BADGE-C vs. Entropy	3.5711	1.4365	2.0307
BADGE-C vs. Margin	12.3686	1.6058	3.4134

the acquisition function is important and needs to be carefully considered.

> **Answer to RQ2**: In contrast to previous work on classification tasks [10], our findings reveal that simple methods are ineffective for the binary code classification task – clone detection and vulnerability detection. Clustering-based acquisition functions consistently outperform output-uncertainty-based functions in this task. In addition, active learning is ineffective for non-classification tasks such as code summarization, as the performance of models trained via active learning lags behind those trained using the entire dataset by at least 29.64%.

H. Case Study

To check how each acquisition function works, we follow the work [32] to study the characteristics of its selected code data.

TABLE VII
PPL of Active Learning Trained Code Summarization Models With Different Training Budgets. Values Highlighted in Red Indicate the Best. Upper: Upper Bound Achieved by All Acquisition Functions

	JavaScript			Ruby		
	1%	5%	10%	1%	5%	10%
Roborta						
Random	5.9526	5.7142	5.6593	5.8720	5.6888	5.6546
KM-C	5.9747	5.6990	5.6503	5.8696	5.7227	5.6443
KC-C	5.7308	5.5647	5.6282	5.6726	5.5735	5.5833
Badge-C	5.8927	5.6350	5.5887	5.8509	5.7256	5.6471
Coreset-C	5.7286	5.5235	5.5146	5.6729	5.5642	5.5909
Upper	5.7286	5.5235	5.5146	5.6726	5.5642	5.5833
CodeBERT						
Random	5.6771	5.3407	5.2038	5.6671	5.4198	5.3662
KM-C	5.6840	5.3586	5.2219	5.6819	5.4662	5.4050
KC-C	5.5978	5.2082	5.1313	5.5565	5.2313	5.1992
Badge-C	5.6654	5.3214	5.2079	5.6623	5.4067	5.3487
Coreset-C	5.6013	5.2222	5.1334	5.5536	5.2158	5.1796
Upper	5.5978	5.2082	5.1313	5.5536	5.2158	5.1796
GraphCodeBERT						
Random	5.3806	5.1560	5.0938	5.3806	5.1560	5.0938
KM-C	5.3852	5.1589	5.0821	5.4038	5.1843	5.1100
KC-C	5.2883	5.0748	5.0178	5.4366	5.2179	5.1881
Badge-C	5.3630	5.1379	5.0639	5.5942	5.3490	5.3015
Coreset-C	5.2912	5.0987	5.0346	5.4368	5.2120	5.1412
Upper	5.2883	5.0748	5.0178	5.3806	5.1560	5.0938

TABLE VIII
PPL of Active Learning Trained Code Translation Models With Different Training Budgets. Values Highlighted in Red Indicate the Best. Upper: Upper Bound Achieved by All Acquisition Functions

	1%	5%	10%
RoBERTa			
Random	3.5825	3.2714	2.8544
KM-C	3.5628	3.3849	2.9817
KC-C	3.1155	2.8576	2.7099
Badge-C	3.3967	2.9880	2.6429
Coreset-C	3.0766	2.8869	2.7516
Upper	3.0766	2.8576	2.6429
CodeBERT			
Random	2.9491	2.4044	1.9459
KM-C	2.9369	2.4787	1.9409
KC-C	3.0087	2.5677	1.6580
Badge-C	2.7917	2.2928	1.8422
Coreset-C	3.0072	2.3149	1.6485
Upper	2.7917	2.2928	1.6485
GraphCodeBERT			
Random	2.6850	2.1740	1.7254
KM-C	2.8086	2.3040	1.6766
KC-C	2.8934	2.3440	1.6113
Badge-C	2.7820	2.2095	1.6900
Coreset-C	2.7724	2.2982	1.5398
Upper	2.6850	2.1740	1.5398

TABLE IX
BLEU Score of Active Learning Trained Code Summarization Models With Labeling Budget 10%. Values Highlighted in Red Indicate the Best

	CS-Ruby			CS-JS			CT		
	RB	CB	GCB	RB	CB	GCB	RB	CB	GCB
Random	7.60	9.62	10.10	8.24	10.36	11.60	20.93	37.93	39.58
KM	7.45	9.94	10.37	8.22	10.15	11.44	20.18	33.46	36.97
KC	7.65	9.96	10.32	7.80	10.46	11.55	18.46	39.13	41.71
Badge	7.47	9.41	9.84	8.01	10.71	11.70	19.81	37.55	40.20
Coreset	7.72	10.09	10.42	7.78	10.45	11.71	19.34	39.71	42.21

TABLE X
Comparison Between the Best Acquisition Function and Random Selection. Comparison Method: T-Test

		CB	GCB	T5
		Problem Classification		
Margin vs. Random	Coefficient	23.8281	24.1301	19.8977
	P-value	0.0003	0.0007	0.0007
		Clone Detection		
Badge vs. Random	Coefficient	4.5579	5.7687	1.8635
	P-value	0.0531	0.0284	0.3912
		Vulnerability Detection		
KM vs. Random	Coefficient	1.6347	3.0595	2.3576
	P-value	0.2658	0.0914	0.1126
		RB	CB	GCB
		Code Summarization-JavaScript		
Coreset vs. Random	Coefficient	-3.5122	-1.7361	-1.6903
	P-value	0.0025	0.0996	0.1082
		Code Summarization-Ruby		
Coreset vs. Random	Coefficient	-5.0270	-4.1700	1.0314
	P-value	0.0001	0.0006	0.3160
		Code Translation		
Coreset vs. Random	Coefficient	-3.7578	-0.8499	0.1577
	P-value	0.0014	0.4065	0.8765

In addition to the 1) label diversity considered by [32], we add two code distribution features in our study, 2) token difference between the selected data and the whole training data, and 3) the token diversity of selected data. Tokens are the direct input of code models and represent fundamental units of the structural and semantic aspects of code data for code models. Their analysis provides further insights into code distribution and diversity beyond the label diversity alone. By incorporating token-related characteristics such as token difference and token diversity, we aim to explore the impact of these inherent aspects on the model performance, thereby enhancing the depth of our analysis.

Label diversity measures the label balance of selected data which is quantified using the entropy score of label histograms.

Token difference measures the token difference between the selected code and the whole training code. Specifically, we build the histograms of tokens in the selected code data and the whole training code data. Then, we normalize the values of histograms to 0 to 1. Finally, we calculate the absolute difference between the two histograms as the difference value.

Token diversity is similar to the label diversity. We use the entropy score of the token histograms to quantify the diversity of selected tokens.

We adopt a balanced approach by selecting one classification task and one non-classification task for this case study. Regarding classification, problem classification is chosen for its consistent performance superiority among available tasks, supported by three distinct models. Meanwhile, regarding non-classification, we focus on code summarization (JavaScript), leveraging its larger dataset size for robust statistical analysis. To facilitate fair comparisons, we employ the same model architecture for both tasks. CodeBERT is selected over GraphBERT due to its lower model complexity and wider community adoption. Table XI presents the characteristic results of the selected data by different acquisition functions and Table XII

TABLE XI
CHARACTERISTIC OF SELECTED DATA

	Token Difference	Token Diversity	Label Diversity
Problem Classification			
LC	881.18	7.78	6.14
Gini	849.95	7.80	7.58
Random	856.22	7.77	7.77
BALD	846.83	7.80	7.62
Entropy	857.24	7.80	7.57
Margin	859.58	7.80	7.74
CAL	994.89	7.64	5.74
KM	839.22	7.76	7.07
KC	783.44	7.94	7.72
Badge	847.61	7.83	7.76
Coreset	854.69	7.81	7.57
Code Summarization			
Random	1495.39	8.55	-
KM	1569.46	8.51	-
KC	807.31	8.79	-
Badge	857.30	8.68	-
Coreset	807.31	8.79	-

TABLE XII
CORRELATION BETWEEN DATA FEATURES AND THE PERFORMANCE OF TRAINED MODELS

	Coefficient	P-Value
Problem Classification		
Token difference	-0.6951	0.0176
Token diversity	0.5003	0.1171
Label diversity	0.8252	0.0018
Code Summarization		
Token difference	0.8572	0.0634
Token diversity	-0.9662	0.0074

summarizes the correlation between the characteristic and the performance of trained models. From the results, we can see that, there is a clear correlation (with an absolute coefficient value from 0.50 to 0.97) between the characteristics of selected data and the performance of the trained models using these selected data. Concretely, 1) when the selected code data have a smaller code token distribution difference to the token distribution of the whole training data, the trained code models have better performance, and 2) when the selected code data have higher token/label diversity, the trained code models have better performance.

Finding: Acquisition functions that select data with smaller token distribution differences to the whole training data and higher code token (label) diversity can produce code models with better performance.

IV. EXPLORATORY STUDY

As discussed in Section III, active code learning for non-classification code summarization tasks is still in an early stage. In this section, we tend to explore the potential directions to propose new effective acquisition functions and mainly focus on non-classification code tasks as existing acquisition functions are effective enough to produce good code models for classification tasks. Since clustering-based acquisition functions are the main techniques used for non-classification tasks, the main goal of our exploratory study is to explore ways to improve this type of acquisition function.

The key idea of clustering-based acquisition functions is to select a diverse set of data, each data sample in this set has big distances from the others. As a result, the calculation of the distance between each data is important in clustering-based functions. Thus, the straightforward way to improve the existing acquisition functions is to provide a precise way to measure the distance between code pairs (or their features). Generally, in the existing acquisition functions, the distance is computed by the Euclidean distance between two vectors that represent two programs. The main concern of this distance calculation is that vectors, e.g., code tokens, code embeddings, and output vectors, highly rely on code representation techniques or code models. It is difficult to say such computed distance can represent the real distance between two programs.

Fortunately, recent research has proposed some evaluation metrics [33], [34] for code generation. Roughly speaking, different from the existing distance methods, these metrics are specifically designed for code to measure the similarity between the machine-generated code snippets and the reference. The studies conducted by the original works show that there is a strong connection between the evaluation metrics and human preference. Inspired by these works, we propose to *use the evaluation metrics as the distance methods* for the clustering-based acquisition functions. Ideally, this new distance should be more precise and the active code learning should be more effective.

A. Study Design

To validate our conjectures, we empirically explore the following two problems:

- *Is there a connection between the distance of selected data to each other and the performance of trained models based on these data?* Through this study, we explore the probability of improving active code learning from the perspective of providing new distance methods for clustering-based acquisition functions.
- *Is there a connection between the existing distance methods and the code evaluation metrics?* Through this study, we explore whether our proposed distance methods are different from the old ones or not. The positive answer demonstrates that evaluation-metric-based methods cannot be replaced by the existing distance methods.

Addressing the first problem follows the following steps:

Step 1 We prepare two groups of models, the first group contains the initialized models (the same as the initialized models used in Section III – models trained using 500 initial training data) that represent models in the early stage of active learning, and the second group includes models that have already been trained using 5% of training data that represent models at a late stage of active learning. In this way, we can see if the correlation we want to study holds in models with different performances. Note that we have prior knowledge of the data used to train the models.

Step 2 We conduct active code learning by using *Random* acquisition function for each group of models 100 times and record the 100 groups of selected data as well as the trained models for further analysis.

Step 3 We measure the accuracy of the 100 trained models on the test data and calculate the average distance between the selected data samples in each group. Finally, we can get 100 accuracy values and corresponding 100 distance values. Here, the distance can be calculated by different methods.

Step 4 We use Spearman's rank correlation coefficient to compute the correlation between the accuracy and the distance provided by **step 3**.

For the second problem, we use different distance calculation methods in **Step 3** to obtain the distance scores and then use Spearman's rank correlation coefficient to compute the correlation between these distance methods.

B. Setup

We select one classification task (problem Classification) and one non-classification task (Code summarization for JavaScript programs) to conduct this exploratory study. For both tasks, we choose CodeBERT as our base model. For the distance calculation methods, we consider four in this study, cosine similarity as distance, Euclidean distance, BLEU score as distance, and CodeBERTScore [34] as distance. Here, CodeBERTScore is a state-of-the-art evaluation metric for code generation. It uses pre-trained contextual embeddings to vectorize each token in the reference program and the generated program first. Then, it computes the pairwise cosine similarity between every embedded token in the reference and every encoded token in the generated code. Finally, the maximum similarity score in each row of the pairwise matrix is used to compute the final similarity of these two programs. We compute the cosine distance and Euclidean distance based on both input embedding (code embedding) and output vectors. For BLEU-based distance, we compute it using both the sequence of input and output tokens. Since CodeBERTScore can be only computed on the code data, we only use the input data to calculate this distance.

C. Results Analysis

Table XIII presents the results of the correlation between data distance and model accuracy. Surprisingly, the results indicate that regardless of the code tasks, when the model has a poor performance, i.e., at the early stage of active code learning, the trained model performance is not related to the diversity (the distance of data to each other) of used training data. However, for a model that was already trained on a few data and with a good performance, i.e., at the late stage of active code learning, the conclusion changed. Considering the classification task, there is a weak correlation between the cosine and Euclidean distance with the accuracy of the models. That means clustering-based acquisition functions that use these two distance calculation methods are promising to train a model with high accuracy. As already shown in Table V, all these acquisition functions based on Euclidean distance achieve good performance in classification tasks. Considering the non-classification task, the results show this correlation based on the input embeddings becomes weaker, e.g., for cosine similarity, the correlation results of $Coreset_i$ change from -0.2555 to 0.1328 in $Model_l$. This is also the reason that the existing acquisition functions do not work well on code summarization tasks based on the token embeddings and have no advantage over random selection as shown in Table VII and Table IX. However, there is a clear correlation between the Euclidean ($Euclidean_o$) and Cosine ($Cosine_o$) distance computed based on the output vectors of selected data and the performance of trained models, which is constant with the conclusion drawn in RQ1. On the other hand, we can see there is a weak correlation between the evaluation scores-based distance and the accuracy of models which does not happen in our considered two classification tasks. The correlation result of CodeBERTScore on $Model_l$ is significant (with a p-value less than 0.05). These results lead to a promising direction of proposing new acquisition functions that use evaluation metrics as the distance calculation method for the code summarization task.

TABLE XIII
CORRELATION BETWEEN THE SELECTED DATA DISTANCE TO EACH OTHER AND THE PERFORMANCE OF TRAINED MODELS BASED ON THESE DATA. EACH VALUE REPRESENTS THE CORRELATION COEFFICIENT. $Model_e$: MODEL AT THE EARLY STAGE OF ACTIVE LEARNING. $Model_l$: AT THE LATE STAGE OF ACTIVE LEARNING. $i(o)$: INPUT(OUTPUT)-BASED FEATURE. VALUE WITH * INDICATES THE P-VALUE IS LESS THAN 0.05

	Classification (Accuracy)		**Non-Classification (PPL)**	
	$Model_e$	$Model_l$	$Model_e$	$Model_l$
$Cosine_i$	0.0589	-0.2555*	0.0705	0.1328
$Euclidean_i$	-0.0773	-0.2262*	0.0868	0.0947
$BLEU_i$	0.0309	-0.0589	0.0504	-0.1699
$Cosine_o$	0.1018	-0.3118*	0.1108	0.2241*
$Euclidean_o$	0.1134	-0.2764*	0.1364	0.2108*
$BLEU_o$	0.0506	0.0309	0.053	-0.079
$CodeScore$	0.0463	-0.0203	0.0128	-0.1965*

Takeaway: In non-classification code summarization tasks, our analysis shows that in the selected dataset, greater distances between data samples as calculated by evaluation-metrics-based distance methods lead to better model performance.

Table XIV presents the results of the correlation between different distance calculation methods. For models with low performance ($Model_e$), there is always a correlation between cosine similarity or Euclidean distance with CodeBERTScore, which means CodeBERTScore is able to produce similar distance ranking of data to the existing distance methods in these models. Combining the conclusion from the last study, we can see that for $Model_e$, all methods have a similar distance ranking of data, but this ranking is not connected to the performance of models. However, for $Model_l$, we can see there is no correlation between cosine and Euclidean to the BLEU and CodeBERTScore. That is the reason why cosine and

TABLE XIV
CORRELATION BETWEEN DIFFERENT DISTANCE CALCULATION METHODS. EACH VALUE REPRESENTS THE CORRELATION COEFFICIENT. $Model_e$: MODEL AT THE EARLY STAGE OF ACTIVE LEARNING. $Model_l$: MODEL AT THE LATE STAGE OF ACTIVE LEARNING. $i(o)$: INPUT(OUTPUT)-BASED FEATURE. VALUE WITH * INDICATES THE P-VALUE IS LESS THAN 0.05

	Classification		**Non-Classification**	
	$Model_e$	$Model_l$	$Model_e$	$Model_l$
$Cosine_i$-$Euclidean_i$	0.9438*	0.9155*	0.9717*	0.9769*
$Cosine_i$-$BLEU_i$	-0.0943	-0.0356	-0.1012	-0.0111
$Cosine_i$-$CodeBERTScore$	-0.2522	-0.0838	-0.4600*	-0.0030
$Euclidean_i$-$BLEU_i$	-0.0963	-0.0230	-0.0856	-0.0856
$Euclidean_i$-$CodeBERTScore$	-0.2457	-0.0615	-0.4176*	-0.0038
$BLEU_i$-$CodeBERTScore$	0.6464*	0.6464*	0.2422	0.2422
$Cosine_o$-$Euclidean_o$	0.9531*	0.9244*	0.9769*	0.9802*
$Cosine_o$-$BLEU_o$	0.0513	-0.0378	-0.0745	0.0621
$Euclidean_o$-$BLEU_o$	-0.0374	-0.0558	-0.0671	-0.0216

euclidean (BLEU and CodeBERTScore) correlate with the model performance while BLEU and CodeBERTScore (cosine and euclidean) do not have this correlation in our considered classification (non-classification) tasks.

Takeaway: Distance methods that are correlated with model performance produce significantly different rankings of data compared to methods that do not exhibit such a correlation.

D. Case Study

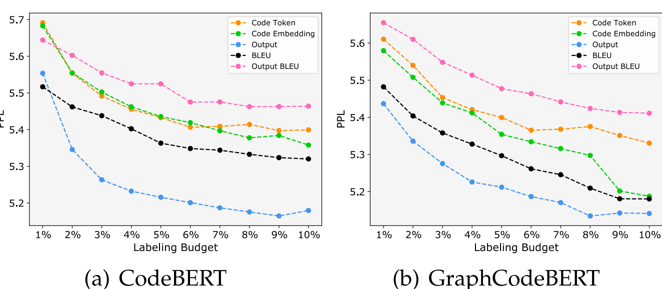
According to our exploratory study, we know that evaluation metrics are promising to be used as distance methods in clustering-based acquisition functions. In this part, we conduct a case study to show if it can really help improve such acquisition functions. Specifically, we modify the distance calculation method in the *Coreset* function from Euclidean distance to BLEU and run experiments on code summarization tasks on Ruby. The reason we choose Ruby is that running evaluation metrics (especially CodeBERTScore) is time-consuming, e.g., it takes more than one month to run an active learning experiment once on code summarization tasks of JavaScript since it contains 2 times more training data than Ruby. The reason for choosing *Coreset* is that it performs the best in code summarization of Ruby as shown in Table VII and Table IX. However, since CodeBERTScore does not support Ruby language now, we can only replace the original distance method in *Coreset* with the BLEU metric. Note that we still use the Euclidean distance between code embeddings to compute $Pairwise\ Distances$ which is the initial step of *Coreset*. It is almost impossible to use the BLEU score here (the time cost is monthly). The algorithm of our modified Coreset is shown as Algorithm 1.

Fig. 4 depicts the results. We can see that *Coreset* with a BLEU score at the input level performs significantly better than *Coreset* with Euclidean distance calculated from code tokens and code embeddings. These results demonstrate the potential of using evaluation metrics as distance methods in active learning. However, using output vectors as clustering features which is also proposed by us is still the best choice which achieves the best results among all the cases. There is a big room to be improved in terms of the performance of

Algorithm 1: Coreset with BLEU

Input : X_l: labeled data
X_u: unlabeled data
M: model under training
$budget$: labeling budget
Output : $X_seleted$: selected data

1 $X_seleted = []$
2 $Dis = Pairwise_Distances(X_u,\ X_l)$
3 $init_data = X_u(Min(Dis))$
4 $X_seleted.append(init_data)$
5 $X_u = X_u \setminus init_data$
6 $count_num = 1$
7 **while** $count_num < budget$ **do**
8 $\quad BLEU_list = BLEU(init_data, X_u)$
9 $\quad init_data = X_u(Min(BLEU_list))$
10 $\quad X_seleted.append(init_data)$
11 $\quad X_u = X_u \setminus init_data$
12 $\quad count_num += 1$
13 **end**
14 **return** $X_seleted$;



(a) CodeBERT (b) GraphCodeBERT

Fig. 4. Active learning with coreset acquisition functions. Code task: code summarization for ruby. BLEU: replacing original Euclidean distance in coreset to BLEU metric at the input level. Output BLEU: replacing original Euclidean distance in coreset to BLEU metric at the output level.

trained models and how to propose a new acquisition function based on the evaluation metrics is still an open problem and our future research.

V. DISCUSSION

A. The Importance of Active Code Learning

Recently, large deep-learning models, especially foundation models like GPT-4 [35] have gained huge attention and achieved many state-of-the-art results in various application domains. This hot trend almost changes the research focus of ML4Code from designing new code model architectures or code representation techniques to how to reuse these foundation models for our specific code tasks. Generally, model reuse involves a fine-tuning step that further optimizes the model parameters and improves performance. As a result, active code learning becomes more and more important since it allows us to fine-tune the pre-trained models with a controllable human effort, i.e., budgets allocated for labeling the datasets used in fine-tuning. This technique provides opportunities for researchers and developers with limited resources to leverage and improve existing big models.

B. Threat to Validity

The **external threat** lies in our considered acquisition functions used for active learning, code tasks and datasets, and code models. For the acquisition functions, we collect 10 functions that are specifically proposed for active learning and already studied the most in recent works [9], [10]. Other functions such as neural coverage methods are not considered since they are proposed for a different purpose. For code tasks and datasets, we consider both classification tasks that study important problems, problem classification, clone detection, and code summarization. For code models, we prepare two well-known code pre-trained models. Based on our open-source projects, other tasks, and models can be easily added to our benchmark. The **internal threat** can be the implementation of acquisition functions and code models. All implementations of acquisition functions are based on the existing active learning works [9], [36], [37] and after carefully checking. The implementation of code models is also modified from the famous open source project [27]. The **construct threat** can be the configuration of active learning. Since this is the first work that studies active code learning, we follow our best practice to initialize the code models and set the labeling budgets. Besides, since we compare code models under the same labeling budgets, the comparison results are not affected by the configuration of active learning.

VI. Related Work

We review related works in two aspects: empirical study on active learning and empirical study on code learning.

A. Empirical Study on Active Learning

Since active learning plays an important role in efficient model training, multiple works [8], [38], [39], [40] conducted empirical studies on this topic. Ramirez-Loaiza et al. [41] compared different active learning methods using different measurements and concluded that improvements obtained by active learning for one performance measure often came at the expense of another measure. Heilbron et al. [42] studied active learning for a specific task, action localization. They found that using acquisition functions the select previously labeled data and combine them with the newly selected data is a more useful strategy of active learning for action localization. More recently, Hu et al. [9] explored the limitations of active learning. They studied adversarial robustness and the ability to handle model compression of models trained by using active learning. The results showed that models trained with active learning can achieve competitive test accuracy but suffer from robustness and compression ability loss. Michael et al. [10] conducted a replicability study and showed that the simple active learning methods, e.g., *DeepGini*, perform better in active learning than neuron coverage-based methods. The most recent work is [36] which did a very large empirical study of 19 active learning methods including both fully-supervised active learning methods and semi-supervised active learning methods. They concluded that semi-supervised learning benefits active learning and should be considered in proposing new methods.

Different from the existing works which mainly study image data or conventional text data, our work is the first one to focus on program data which is the key type of data in the software engineering field.

B. Empirical Study on ML4Code

ML4Code gained huge attention recently, researchers also conducted multiple empirical studies to explore the problems in the ML4Code field. In the very early work [43], Chirkova et al. empirically study the backbone model architecture of the later code models–Transformer on different code tasks, code completion, function naming, and bug fixing. They found that Transformers can utilize syntactic information in source code to solve code tasks. Niu et al. [44] conducted a large-scale empirical study to compare pre-trained models of source code. In total, we studied 19 pre-trained models and 13 software tasks and gave fine-grained suggestions for using and evaluating these models. Steenhoek et al. [45] studied the deep learning models for a specific code task, vulnerability detection. Their experimental results showed that models trained by specific types of vulnerability perform better than models trained by all vulnerabilities. Increasing the size of the training dataset has limited benefits to the performance of trained models. Jiang et al. [46] conducted the first study of pre-trained model reuse. Concretely, they interviewed practitioners from Hugging Face and identified the challenges of pre-trained model reuse, e.g., missing attributes, discrepancies, and model risks. Besides considering only the clean performance of models of code, Mastropaolo et al. [47] studied the robustness of GitHub Copilot which is a famous code generation model. They generated semantically equivalent natural language descriptions based on the seed description and checked if Copilot can generate the same code functions as the ones generated by the seed description. They found that almost half of the semantically equivalent but different method descriptions result in different code recommendations which means the code generation models are not robust. Hu et al. [48] provided shifted datasets and studied the generalization ability of code models under data distribution shift. They found that code models are not robust and can not handle distribution shifts properly. Finally, Nie et al. [49] studied the influence of used evaluation methods on code summarization and found that different evaluation methods lead to conflicting results which should gain attention for users during testing the code models.

More recently, researchers employed large language models (LLM) to solve code-related tasks and reported impressive results [50], [51], [52]. Xia et al. [53] used LLM to solve automated program repair problems and achieved SOTA results with acceptable budgets. Deng et al. [54] utilized LLMs to synthesize unusual programs to test deep learning libraries and found 49 unknown bugs in famous deep learning frameworks. Ma et al. [55] empirically studied the ability of ChatGPT (the most famous LLM) on program syntax, static, and dynamic understanding. The results demonstrate that ChatGPT is good at analyzing syntax and static information of programs, but tends to generate non-existent semantic information.

The above works focused on the clean performance, robustness, challenges of reuse, and evaluation methods of code models. However, none of them studied the important problem of how to reduce the labeling effort of training data and efficiently train the code models, which is our main purpose.

C. Active Learning for Code

Active learning has been applied to solve software tasks due to its cost-saving ability. Moskovitch et al. [56] proposed a very early framework to use active learning methods to detect malicious code. In the work, the Margin selection strategy has been considered to select new code from the code pool. Inspired by this work, other works also tried to employ active learning in other software domains, e.g., Nir et al. used active learning to identify new unknown malware in Android systems [57] and Windows OS [58]. Recently, Berezov et al. [59] combined active learning and code generation techniques to help build code benchmarks. In their work, the Greedy Sampling [60] acquisition function was used to select the most valuable code for the active learning component. The latest work [61] employed active learning to predict software performance (execution time). Four acquisition functions such as Random Sampling and Coreset Sampling have been studied in this work.

Compared to the previous works, each of which only considers a single type of code task (e.g., malware detection), limited acquisition functions (no more than four), and simple models (e.g., Graph2Vec), our work provides the first benchmark that covers various types of code tasks, over 10 types of acquisition functions, and advanced pre-trained code models.

VII. Conclusion

This paper introduced the first benchmark and an empirical study for the important yet unexplored problem – active code learning. Our experimental results demonstrated that active code learning is effective in training code models with expected high performance for classification tasks such as problem classification and clone detection. However, it is still in the early stage for non-classification tasks like code summarization. Besides, we conducted an exploratory study to show using evaluation metrics as distance calculation methods is a promising way to propose new clustering-based acquisition methods. We believe that our benchmark as well as empirical studies will provide developers and researchers insights into efficiently reusing (i.e., with little human effort) existing large pre-trained models for their specific code tasks.

In the future, we plan to

- extend our benchmark to support semi-supervised learning and combine it with active learning to future improve the effectiveness of active code learning.
- explore clustering ensemble methods to combine the clustering results produced by different features (e.g., output vectors and evaluation-based metrics) to enhance active code learning.

Acknowledgment

The content of this article does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the authors.

References

[1] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, vol. 51, no. 4, pp. 1–37, 2018.

[2] X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing source code with transferred API knowledge," in *Proc. 27th Int. Joint Conf. Artif. Intell. (IJCAI)*, 2018, pp. 2269–2275, doi:10.24963/ijcai.2018/314.

[3] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "CCLearner: A deep learning-based clone detection approach," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, 2017, pp. 249–260.

[4] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019.

[5] T. Wolf et al., "Transformers: State-of-the-art natural language processing," in *Proc. Conf. Empirical Methods Natural Lang. Process., Syst. Demonstrations*, Association for Computational Linguistics, Oct. 2020, pp. 38–45. Accessed: Oct. 2020. [Online]. Available: https://aclanthology.org/2020.emnlp-demos.6

[6] B. Settles, "Active learning literature survey," Univ. Wisconsin-Madison, Madison, WI, USA, CS Tech. Rep. TR1648, 2009.

[7] O. Sener and S. Savarese, "Active learning for convolutional neural networks: A core-set approach," in *Proc. Int. Conf. Learn. Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=H1aIuk-RW

[8] B. Settles and M. Craven, "An analysis of active learning strategies for sequence labeling tasks," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2008, pp. 1070–1079.

[9] Q. Hu et al., "Towards exploring the limitations of active learning: An empirical study," in *Proc. 36th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2021, pp. 917–929.

[10] M. Weiss and P. Tonella, "Simple techniques work surprisingly well for neural network test prioritization and active learning (replicability study)," in *Proc. 31st ACM SIGSOFT Int. Symp. Softw. Testing Anal. (ISSTA)*, 2022, pp. 139–150.

[11] H. Liu and L. Yu, "Toward integrating feature selection algorithms for classification and clustering," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 4, pp. 491–502, Apr. 2005.

[12] "Active code learning." Google Sites. Accessed: 2023. [Online]. Available: https://sites.google.com/view/activecodelearning

[13] D. Wang and Y. Shang, "A new active labeling method for deep learning," in *Proc. Int. Joint Conf. Neural Netw. (IJCNN)*, Piscataway, NJ, USA: IEEE Press, 2014, pp. 112–119.

[14] Y. Feng, Q. Shi, X. Gao, J. Wan, C. Fang, and Z. Chen, "DeepGini: Prioritizing massive tests to enhance the robustness of deep neural networks," in *Proc. 29th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2020, pp. 177–188.

[15] Y. Gal, R. Islam, and Z. Ghahramani, "Deep Bayesian active learning with image data," in *Proc. Int. Conf. Mach. Learn.*, PMLR, 2017, pp. 1183–1192.

[16] K. Margatina, G. Vernikos, L. Barrault, and N. Aletras, "Active learning by acquiring contrastive examples," in *Proc. Conf. Empirical Methods Natural Lang. Process.,* Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 650–663. Accessed: Nov. 2021. [Online]. Available: https://aclanthology.org/2021.emnlp-main.51

[17] J. T. Ash, C. Zhang, A. Krishnamurthy, J. Langford, and A. Agarwal, "Deep batch active learning by diverse, uncertain gradient lower bounds," in *Proc. Int. Conf. Learn. Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=ryghZJBKPS

[18] Z. Feng et al., "CodeBERT: A pre-trained model for programming and natural languages," in *Proc. Findings Assoc. Comput. Linguistics (EMNLP)*, Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. Accessed: 2020. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[19] J. D. M.-W. C. Kenton and L. K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics, Human Lang.*

Technol., Association for Computational Linguistics, 2019, vol. 1 (Long and Short Papers), pp. 4171–4186.
[20] D. Guo et al., "GraphCodeBERT: Pre-training code representations with data flow," 2020, *arXiv:2009.08366*.
[21] Z. Tian, J. Chen, and Z. Jin, "Code difference guided adversarial example generation for deep code models," in *Proc. 38th IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, Los Alamitos, CA, USA: IEEE Comput. Soc. Press, Sep. 2023, pp. 850–862, doi:10.1109/ASE56229.2023.00149.
[22] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," in *Proc. 44th Int. Conf. Softw. Eng.*, 2022, pp. 1482–1493.
[23] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, M.-F. Moens, X. Huang, L. Specia, and S. W.-t. Yih, Eds., Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 8696–8708. Accessed: 2021. [Online]. Available: https://aclanthology.org/2021.emnlp-main.685
[24] R. Puri et al, "CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks," in *Proc. Neural Inf. Process. Syst. (NeurIPS) Track Datasets Benchmarks*, 2021. Accessed: 2021. [Online]. Available: https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/a5bfc9e07964f8dddeb95fc584cd965d-Abstract-round2.html
[25] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, Piscataway, NJ, USA: IEEE Press, 2014, pp. 476–480.
[26] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Vancouver, Canada: Curran Associates, Inc., 2019. Accessed: Dec. 8, 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/49265d2447bc3bbfe9e76306ce40a31f-Paper.pdf
[27] S. Lu et al., "CodeXGLUE: A machine learning benchmark dataset for code understanding and generation," 2021, *arXiv:2102.04664*.
[28] Y. Liu et al., "RoBERTa: A robustly optimized BERT pretraining approach," 2019, *arXiv:1907.11692*.
[29] C. Raffel et al., "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 1, pp. 5485–5551, Jan. 2020.
[30] Y. Wan et al., "Improving automatic source code summarization via deep reinforcement learning," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 397–407.
[31] D. B. Owen, "The power of student's t-test," *J. Amer. Statist. Assoc.*, vol. 60, no. 309, pp. 320–333, 1965.
[32] Q. Hu et al., "An empirical study on data distribution-aware test selection for deep learning enhancement," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 4, pp. 1–30, 2022.
[33] A. Eghbali and M. Pradel, "CrystalBLEU: Precisely and efficiently measuring the similarity of code," in *Proc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2022, pp. 1–12.
[34] S. Zhou, U. Alon, S. Agarwal, and G. Neubig, "CodeBERTScore: Evaluating code generation with pretrained models of code," 2023, *arXiv:2302.05527*.
[35] OpenAI, "GPT-4 technical report," 2023, *arXiv:2303.08774*.
[36] Y. Li, M. Chen, Y. Liu, D. He, and Q. Xu, "An empirical study on the efficacy of deep active learning for image classification," 2022, *arXiv:2212.03088*.
[37] Y. Guo, Q. Hu, M. Cordy, M. Papadakis, and Y. Le Traon, "DRE: Density-based data selection with entropy for adversarial-robust deep learning models," *Neural Comput. Appl.*, vol. 45, pp. 4009–4026, Feb. 2023.
[38] D. Pereira-Santos, R. B. C. Prudencio, and A. C. de Carvalho, "Empirical investigation of active learning strategies," *Neurocomputing*, vol. 326, pp. 15–27, Jan. 2019.
[39] Z. Yu, N. A. Kraft, and T. Menzies, "Finding better active learners for faster literature reviews," *Empirical Softw. Eng.*, vol. 23, pp. 3161–3186, Dec. 2018.
[40] A. Siddhant and Z. C. Lipton, "Deep Bayesian active learning for natural language processing: Results of a large-scale empirical study," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, Brussels, Belgium: Association for Computational Linguistics, Oct./Nov. 2018, pp. 2904–2909. Accessed: 2018. [Online]. Available: https://aclanthology.org/D18-1318
[41] M. E. Ramirez-Loaiza, M. Sharma, G. Kumar, and M. Bilgic, "Active learning: An empirical study of common baselines," *Data Mining Knowl. Discovery*, vol. 31, pp. 287–313, Mar. 2017.
[42] F. C. Heilbron, J.-Y. Lee, H. Jin, and B. Ghanem, "What do I annotate next? An empirical study of active learning for action localization," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2018, pp. 199–216.
[43] N. Chirkova and S. Troshin, "Empirical study of transformers for source code," in *Proc. 29th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2021, pp. 703–715.
[44] C. Niu, C. Li, V. Ng, D. Chen, J. Ge, and B. Luo, "An empirical comparison of pre-trained models of source code," in *Proc. 45th Int. Conf. Softw. Eng. (ICSE)*, May 2023, pp. 2136–2148.
[45] B. Steenhoek, M. M. Rahman, R. Jiles, and W. Le, "An empirical study of deep learning models for vulnerability detection," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, 2023, pp. 2237–2248.
[46] W. Jiang et al., "An empirical study of pre-trained model reuse in the hugging face deep learning model registry," in *Proc. 45th Int. Conf. Softw. Eng. (ICSE)*, May 2023, pp. 2463–2475.
[47] A. Mastropaolo et al., "On the robustness of code generation techniques: An empirical study on GitHub Copilot," in *Proc. IEEE/ACM 45th Int. Conf. Softw. Eng. (ICSE)*, 2023, pp. 2149–2160.
[48] Q. Hu et al., "CodeS: Towards code model generalization under distribution shift," in *Proc. Int. Conf. Softw. Eng. (ICSE), New Ideas Emerg. Results (NIER)*, 2023, pp. 1–6.
[49] P. Nie, J. Zhang, J. J. Li, R. Mooney, and M. Gligoric, "Impact of evaluation methodologies on code summarization," in *Proc. 60th Annu. Meeting Assoc. Comput. Linguistics,* vol. 1 (Long Papers). Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 4936–4960. Accessed: 2022. [Online]. Available: https://aclanthology.org/2022.acl-long.339
[50] H. Tian et al., "Is ChatGPT the ultimate programming assistant—How far is it?" 2023, *arXiv:2304.11938*.
[51] W. Sun et al., "Automatic code summarization via ChatGPT: How far are we?" 2023, *arXiv:2305.12865*.
[52] Y. Charalambous, N. Tihanyi, R. Jain, Y. Sun, M. A. Ferrag, and L. C. Cordeiro, "A new era in software security: Towards self-healing software via large language models and formal verification," 2023, *arXiv:2305.14752*.
[53] C. S. Xia and L. Zhang, "Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT," 2023, *arXiv:2304.00385*.
[54] Y. Deng, C. S. Xia, C. Yang, S. D. Zhang, S. Yang, and L. Zhang, "Large language models are edge-case fuzzers: Testing deep learning libraries via FuzzGPT," 2023, *arXiv:2304.02014*.
[55] W. Ma et al., "The scope of ChatGPT in software engineering: A thorough investigation," 2023, *arXiv:2305.12138*.
[56] R. Moskovitch, N. Nissim, and Y. Elovici, "Malicious code detection using active learning," in *Proc. Int. Workshop Privacy, Secur., Trust KDD*, Berlin, Germany: Springer-Verlag, 2008, pp. 74–91.
[57] N. Nissim, R. Moskovitch, O. BarAd, L. Rokach, and Y. Elovici, "ALDROID: Efficient update of android anti-virus software using designated active learning methods," *Knowl. Inf. Syst.*, vol. 49, pp. 795–833, Dec. 2016.
[58] N. Nissim, R. Moskovitch, L. Rokach, and Y. Elovici, "Novel active learning methods for enhanced PC malware detection in windows OS," *Expert Syst. Appl.*, vol. 41, no. 13, pp. 5843–5857, 2014.
[59] P. Samoaa, L. Aronsson, A. Longa, P. Leitner, and M. H. Chehreghani, "A unified active learning framework for annotating graph data with application to software source code performance prediction," 2023, *arXiv:2304.13032*.
[60] D. Wu, C.-T. Lin, and J. Huang, "Active learning for regression using greedy sampling," *Inf. Sci.*, vol. 474, pp. 90–105, Feb. 2019.
[61] M. Berezov, C. Ancourt, J. Zawalska, and M. Savchenko, "COLA-Gen: Active learning techniques for automatic code generation of benchmarks," in *Proc. 13th Workshop Parallel Program. Run-Time Manage. Techn. Many-Core Archit./11th Workshop Des. Tools Archit. Multicore Embedded Comput. Platforms (PARMA-DITAM)*, Wadern, Germany: Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022, pp. 3:1–3:14.