KAPE: *k*NN-Based Performance Testing for Deep Code Search

YUEJUN GUO[*], Luxembourg Institute of Science and Technology, Luxembourg
QIANG HU[+], SnT, University of Luxembourg, Luxembourg
XIAOFEI XIE, Singapore Management University, Singapore
MAXIME CORDY, SnT, University of Luxembourg, Luxembourg
MIKE PAPADAKIS, SnT, University of Luxembourg, Luxembourg
YVES LE TRAON, SnT, University of Luxembourg, Luxembourg

Code search is a common yet important activity of software developers. An efficient code search model can largely facilitate the development process and improve the programming quality. Given the superb performance of learning the contextual representations, deep learning models, especially pre-trained language models, have been widely explored for the code search task. However, studies mainly focus on proposing new architectures for ever-better performance on designed test sets but ignore the performance on unseen test data where only natural language queries are available. The same problem in other domains, e.g., CV and NLP, is usually solved by test input selection that uses a subset of the unseen set to reduce the labeling effort. However, approaches from other domains are not directly applicable and still require labeling effort. In this paper, we propose the **k**NN-b**a**sed **p**erformance t**e**sting (**KAPE**), to efficiently solve the problem without manually matching code snippets to test queries. The main idea is to use semantically similar training data to perform the evaluation. Extensive experiments on six programming language datasets, three state-of-the-art pre-trained models, and seven baseline methods demonstrate that KAPE can effectively assess the model performance (e.g., CodeBERT achieves MRR 0.5795 on JavaScript) with a slight difference (e.g., 0.0261).

CCS Concepts: • **Computing methodologies → Artificial intelligence**; • **Software and its engineering**;

Additional Key Words and Phrases: deep code search, software testing, deep learning testing, test selection

ACM Reference Format:
Yuejun Guo[*], Qiang Hu[+], Xiaofei Xie, Maxime Cordy, Mike Papadakis, and Yves Le Traon. 2018. KAPE: *k*NN-Based Performance Testing for Deep Code Search. *J. ACM* 37, 4, Article 0 (August 2018), 24 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Code search aims to retrieve semantically relevant code snippets from a large code corpus that mostly match a natural language query, which is an essential practice of software developers to avoid "reinventing the wheel". An early case study conducted in Google shows that a developer, on average, makes 5 search sessions with 12 queries every workday [41]. Beyond serving as a critical

[*]This work was partially done while Yuejun Guo worked at SnT, University of Luxembourg.
[+]Corresponding author.

Authors' addresses: Yuejun Guo[*], Luxembourg Institute of Science and Technology, Luxembourg, yuejun.guo@list.lu; Qiang Hu[+], SnT, University of Luxembourg, Luxembourg, qiang.hu@uni.lu; Xiaofei Xie, Singapore Management University, Singapore, xiaofei.xfxie@gmail.com; Maxime Cordy, SnT, University of Luxembourg, Luxembourg, maxime.cordy@uni.lu; Mike Papadakis, SnT, University of Luxembourg, Luxembourg, michail.papadakis@uni.lu; Yves Le Traon, SnT, University of Luxembourg, Luxembourg, yves.letraon@uni.lu.

0004-5411/2018/8-ART0 $15.00
https://doi.org/XXXXXXX.XXXXXXX

development activity, code search can support other software engineering tasks, such as defect localization [50], program repair [2], and code synthesis [39]. Generally, given a functionality, developers seek to reuse previously written code examples by searching over popular platforms, such as Stack Overflow [52], GitHub [11], and Google [31]. For example, developers received over 45.1 billion times coding help from Stack Overflow, and more than 21 million queries were made. Due to constantly growing demand, researchers have leveraged the data from these platforms as a way to power the code search engines. Studies have shown that deep learning (DL) is the most popular modeling technique for code search [28] given its ability to embed the code representation [4, 15, 32].

Although deep code search has attracted popular attention from researchers to devote to the development of ever-better deep neural networks (DNNs) [4, 15, 32], the testing of such models for secure and reliable deployment is lagging behind. For instance, the common scenario of testing the model performance given unknown queries has not been studied. Differing from traditional software systems, the DL-based system has a fundamentally different nature and computing logic. In conventional programming, developers design the computing logic to obtain the executable code for solving a given task. The change of data will not influence the functionality. On the other hand, in DL, developers design the architecture of the DNN that learns the computing logic from the input data and expected results. Specifically, the logic is defined by the weights and bias parameterizing the connections inside a DNN. By contrast, the DNN's behavior may evolve, in response to the change in new data. Namely, given a trained deep code search model, the reported performance on the original test data cannot reflect the actual performance on unseen data, leading to the demand for a dedicated testing before deployment. For example, the mean reciprocal rank (MRR) performance of a pre-trained CodeBERT is 0.8048 on the original Ruby test data but changes to 0.7301 on new test data (see Table 2, more details of the dataset and MRR can be found in Section 4).

Due to the supervised learning nature of DL [12], testing a trained deep code search model requires query-code pairs to calculate the correctness of identifying the best code snippets. In practice, it is easy to collect a large number of natural language queries from public platforms, such as Stack Overflow. However, the corresponding code snippets are usually missing, which makes the testing challenging. Assigning the matched code snippet for each unseen query is straightforward but impractical and almost impossible for four main reasons. ❶ Time cost: collecting queries from online platforms is easy and free, but manually checking the code snippet is time-consuming, especially with the continuous surging queries every day. ❷ Domain knowledge: generally a specific programming language code data is used to test the DL model. Given various languages, e.g., Java, Python, and Go, domain experts are required to give reliable matching. ❸ Correctness: even with domain knowledge, errors are inevitable in human beings' work. ❹ Financial cost: for instance, labeling 1,000 units (50 words per unit) per human labeler of texts for classification costs 129 dollars by the Google Cloud AI platform data labeling and each unit needs at least three labelers to guarantee correctness [14]. Remarkably, the cost increases significantly when it requires excellent query understanding and strong developing experience in specific programming languages. To the best of our knowledge, there is no work in the literature that focuses on this specific testing.

The same issue also happens to DL models in other domains, such as computer vision (CV) [6, 27] and natural language processing (NLP) [20]. Researchers tend to apply the test input selection technique [6] to solve this issue in CV and NLP, especially for classification tasks, e.g., image classification and text classification. In this technique, a subset of data is selected based on a specific selection metric to represent the entire set. Many selection metrics have been proposed in these two domains [6, 18, 27]. However, they are not directly applicable to deep code search models. For instance, in CV and NLP, studies have shown the success of selecting data based on the prediction probability [27]. However, deep code search is not a classification task and it aims to, given a query, find the best-match code snippet from a large codebase. Thus, there is no prediction probability to use but only the contextual representation of natural queries.

This paper focuses on testing a trained deep code search model on unseen data without extra human power, namely, manual query-snippet matching. We propose the kNN-based performance estimation (KAPE) that takes advantage of the similar queries from the training set for each unseen data to undertake the performance estimation. Concretely, KAPE first feeds both the training and test queries into a trained model to obtain the contextual representations for the following similarity calculation. Then, we leverage the widely used cosine similarity [51] to quantify the semantic similarity between representations. Next, to locate the corresponding queries from the training set for each test query, we utilize the simple yet efficient non-parametric k-nearest neighbors algorithm [3]. Finally, concerning the difference in data, we propose to adaptively determine the relevant nearest neighbors and their weights contributing to the performance based on the Z-Score [54]. The evaluation on 6 programming languages and 3 pre-trained models demonstrates that KAPE is capable of estimating the model performance on unseen data. Meanwhile, our investigation on the parameter sensitivity and data distribution influence shows that KAPE is stable and flexible to possible changes. Additionally, our ablation study demonstrates the usefulness of the calculation of the adaptive weights. To summarize, the main contributions of our work are as follows.

(1) To the best of our knowledge, this is the first work that undertakes performance estimation testing for deep code search models.
(2) We propose KAPE that requires no manpower to manually match code snippets to unseen test queries. KAPE is automatic and practical in real-world applications.
(3) We conduct comprehensive experiments on 6 popular programming languages, 3 state-of-the-art pre-trained models, and 7 baseline methods to evaluate the effectiveness of KAPE.

The rest of this paper is organized as follows. Section 2 introduces the background and related work of this paper. Section 3 details the methodology of our proposed KAPE. Section 4 and Section 5 cover the experimental setup and the results analysis. Section 6 discusses the strengths and limitations of KAPE and lists the potential threats that influence our conclusions. In the last Section 7, we conclude our work and point out the future work.

2 Background and related work

2.1 Code search and pre-trained models

Code search is a daily activity of software developers during software development. Given its importance, many code search tools/models have been developed, which can be divided into two categories: traditional and deep learning (DL)-based methods [28]. The traditional manner mainly utilizes the information retrieval technique, such as the Boolean model [24, 33, 56], vector space model [24, 35], and PageRank algorithm [36]. The DL-based models take advantage of DL given its powerful ability in learning language representations [5, 8, 15]. In particular, pre-trained models (PTMs) have gained remarkable attention. Figure 1 illustrates a general process of using deep learning models for code search where the DL model learns the mapping between queries and snippets from a large set.

PTMs were proposed initially for natural language processing (NLP) tasks [38]. A PTM is first fed with a huge text corpus to learn the representations. Then the PTM is transferred to a downstream task with fine-tuning on a specific dataset. The performance of a PTM is usually better than a task-focused model that trains on a specific dataset directly. Typical PTMs are Google's bidirectional encoder representations from transformers (BERT) [7] and Facebook's robustly optimized version named RoBERTa [30]. Due to their considerable functionality, researchers have been seeking to apply PTMs to source code analysis. Built on the top of BERT and RoBERTa, Microsoft proposed

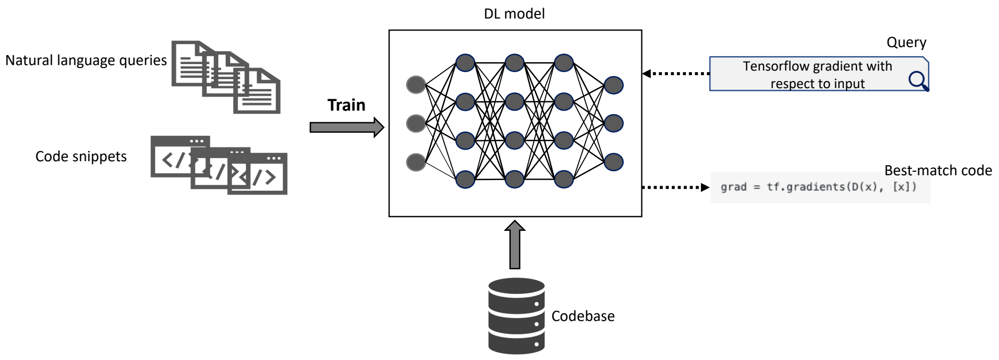


Fig. 1. An illustration of deep code search.

the CodeBERT [10] that learns the representations of both programming and natural languages. Specifically, concerning the inherent structure of code, GraphCodeBERT was developed [16].

2.2 Deep learning testing

Deep learning (DL) is a machine learning technique that learns complex patterns in data by multiple layers of neurons that mathematically transform the data and the connection between neurons forms the data flow. Due to this complex computing logic, DL models are lacking interpretability [60] and require comprehensive testing before being deployed in real-world applications. DL testing refers to evaluating the quality of DL systems for further deployment [21, 58]. The testing-related works in the literature mainly focus on the domains of computer vision (CV) and natural language processing (NLP), while very few consider deep code search systems [47].

As a new type of data-driven software, DL models have advantages in learning features from a large input space. However, the input space is expected to cover all possible cases in the real world to ensure performance, which is infeasible. In practice, only a fixed training set is applied to approximate the input space, and undoubtedly, this approximated input space is much smaller than the real one. Hence, generally, a DL model performs well on designed test data during development but exhibits performance degradation on unseen test data when deployed in the real world [19, 25].

Test input selection is essential for developers to estimate a DL model performance after deployment and has been well studied for classification tasks in the domains of computer vision and natural language processing [6, 27]. This technique decides which test data should be used from available unlabeled tests to cut the cost associated with the labeling effort. Instead of the manual and ad-hoc way, these tests can be selected strategically based on the behavior of DL models. For instance, Chen *et al.* proposed the practical accuracy estimation (PACE) [6] to approximate the accuracy of classifiers and regression models. PACE utilizes the model output at different levels (e.g., first layer, last hidden layer, confidence output) as data features. Based on extracted features, PACE performs clustering to group data into different clusters where representative data are selected proportionally and separately. For the same tasks, Li *et al.* proposed the cross entropy-based sampling (CES) [27]. The main idea is to select data that has the minimum cross entropy with the entire test set. CES also uses the confidence output (prediction probability of belonging to a certain class) as the data feature. However, concerning deep code search, the prediction probability is not available from the model but the contextual representations of text query with the absence of corresponding code snippets. Therefore, these probability-based SOTA approaches are directly inapplicable. Some other test input selection approaches have been proposed to locate error inputs where the model has not learned sufficiently. Note that these approaches are also called test input prioritization in the literature [6]. Pei *et al.* proposed the first white box testing, DeepXplore [37], to find test inputs that trigger the error behavior of DL models with a high neuron coverage. Similarly, [26, 34, 55] also select data based on the neuron coverage. Some other approaches utilize the prediction probability to select data. For instance, Shen *et al.* selects data with a small probability ratio between the first two predicted classes [48]. Feng *et al.* identify test inputs with the largest Gini impurity [9]. However, these types of approaches mainly select test inputs to enhance the model performance, which is different from this paper's focus.

Another relevant technique to test input selection is active learning from the machine learning community [45]. In active learning, a DL model is trained iteratively through multiple steps. At each step, a small set of training data is selected based on a specific acquisition function to update the pre-trained model from the previous step. The goal of both test selection metrics and acquisition functions in active learning is to reduce the labeling cost when given massive unlabeled data. The main difference is that, selection metrics target the test data while acquisition functions select data from the training set. In the literature, studies have proved that these acquisition functions perform effectively as test selection metrics [18–20]. For example, Ozan and Silvio proposed the core set selection [44] for the image classification task. The core set tends to select the representative data from the training set for the current step training. Nevertheless, this approach also requires the prediction probability to select data. Additionally, due to its greedy algorithm in the selection procedure, its execution is prohibitive and impractical.

3 KAPE

We first describe the problem that is targeted in this work. Next, before the detailed explanation of KAPE, we present a motivating example. Via the example, one gets a preliminary insight into the key idea of our proposed KAPE. Finally, we explicitly present KAPE.

3.1 Problem definition

Given a deep code search model f and its training set $\boldsymbol{D}$, where $\boldsymbol{D}$ consists of query-code pairs, this paper focuses on the problem of estimating the model performance $P_{\boldsymbol{T}}$ on a set of unseen data $\boldsymbol{T}$. Specifically, $\boldsymbol{T}$ only includes a number of queries and their corresponding code snippets are missing. However, the code snippets are required to precisely calculate the model performance.

To solve this problem, there is the test input selection technique that is well-studied in the computer vision (CV) and natural language processing (NLP) fields, especially for classification tasks (e.g., image classification, sentiment classification). The core idea is to select a subset of data from the test set and manually assign the corresponding labels (code snippets in code search). This subset is assumed to be representative of the entire set, and, thus, the performance of this subset is considered as the estimated performance of the entire set. Formally,

$$P_{\boldsymbol{T}} \approx P_{\boldsymbol{T}^*} \ , \ \texttt{where}\ \boldsymbol{T^*} \subseteq \boldsymbol{T} \tag{1}$$

However, manually assigning the best-match code snippet to a test query is more challenging than annotating the label (e.g., cat) of an animal in image classification or the review sentiment (e.g., positive, negative) in sentiment classification. Concerning this, we are interested in solving this problem without any manpower.

3.2 Motivating example

Our intuition is that a model outputs the same code snippet if two queries are similar enough. Figure 2 presents a motivating example of our proposed kNN-based performance estimation (KAPE).

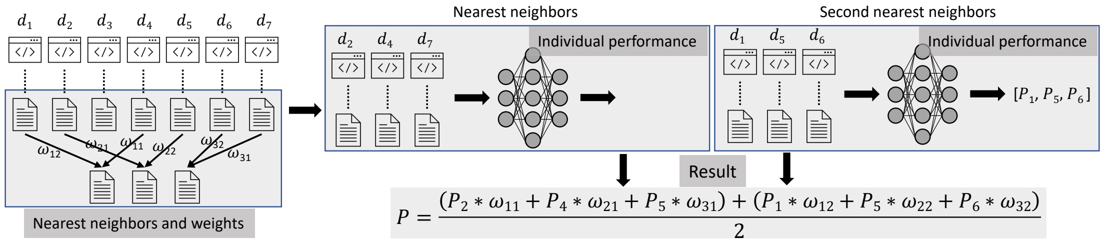

d_1 d_2 d_3 d_4 d_5 d_6 d_7
ω_{12} ω_{21} ω_{11} ω_{22} ω_{32} ω_{31}
Nearest neighbors and weights
Nearest neighbors
d_2 d_4 d_7
Individual performance
Second nearest neighbors
d_1 d_5 d_6
Individual performance
$[P_1, P_5, P_6]$
Result
$P = \frac{(P_2 * \omega_{11} + P_4 * \omega_{21} + P_5 * \omega_{31}) + (P_1 * \omega_{12} + P_5 * \omega_{22} + P_6 * \omega_{32})}{2}$

Fig. 2. An example of performance estimation by KAPE on 3 test queries with 7 training query-code pairs.

In this example, the training set includes 7 query-code pairs $(d_1, d_2, \dots, d_7)$ and the test set has 3 queries. We first find the two nearest neighbors for each test query from the training set. Next, we obtain the individual model performance of each neighbors group. Concretely, we create a neighbors group of query-code pairs where the queries from the training set are the nearest neighbors of test queries. The model can output the performance on each individual pair (P_2, P_4, P_7). Finally, for each test query, given its two nearest neighbors, we assign the weights (e.g., ω_{12} is the weight of the second nearest neighbor for the first test query) and obtain the final result P.

3.3 Methodology

Our proposed KAPE consists of four main steps. Figure 3 presents an overview of KAPE. and Algorithm 1 gives details of KAPE.

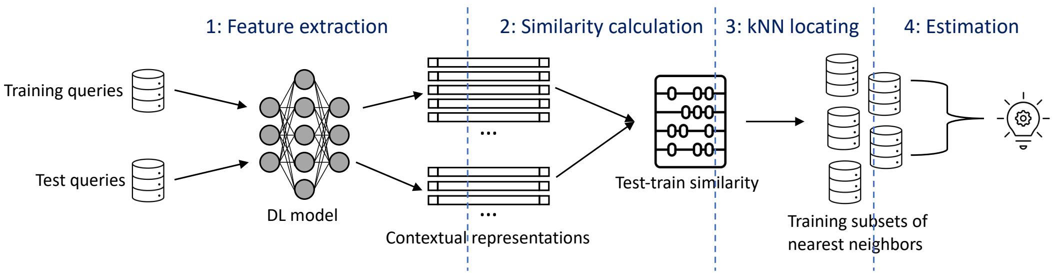


Fig. 3. Overview of KAPE.

Step 1: Feature extraction. Since KAPE relies on the semantic similarity between the unseen test set and training set to conduct performance estimation, the first step is to determine the query feature for the following similarity measure. In this paper, we utilize the contextual representation of natural language queries obtained from the trained model. The reason is that, in general, deep code search models utilize the similarity between the representation vectors of queries and code snippets to find the best match. Namely, if a training query has the same representation vector as a test query, the model will identify the same code snippet. Line 1 extracts the natural language representation vectors of the training and test queries, respectively.

Step 2: Similarity calculation. We utilize the cosine similarity to capture the similarity between two queries, which is defined as:

Algorithm 1: KAPE: KNN-based performance estimation

```
Input  : f: trained model
         D : training set of query-snippet pairs
         T : unseen test set of queries
         k: hyperparameter of kNN
Output : $P_T$: model performance on T
/* Step1: Obtain natural language contextual representations */
1  $E_D, E_T = f(D), f(T)$
/* Step 2: Calculate similarity */
2  $S = \varnothing$
3  for $i = 1 \to m$ do
4      for $j = 1 \to n$ do
5          $S_{i,j} = CosineSimilarity(E_{D\,i}, E_{T\,j})$ ;   // Equation (2)
6      end
7  end
/* Step 3: Locate k nearest neighbors */
8  $NN, NS = \varnothing, \varnothing$ ; // matrix of the first k similar queries and corresponding similarity matrix
9  for $i = 1 \to n$ do
10     $NN_i, NS_i = DecendingSort(S_i, k)$
11 end
/* Step 4: Estimate performance */
12 for $j = 1 \to k$ do
13     $D_j = SelectTrain(D, NN, j)$ ;   // A subset of training data of the ith nearest
14     $P_j = Evaluate(f, D_j)$ ;   // $P_j$ is a vector of model performance on each data
15 end
16 for $i = 1 \to n$ do
17     for $j = 1 \to m$ do
18         $z_{i,j} = \frac{NS_{i,j} - \mu}{\sigma}$ ;   // $\mu$ and $\sigma$ are the mean and standard deviation of $NS_i$, respectively
19     end
20 end
21 $W = WeightCalculate(NS, z)$ ;   // Calculate weights of each nearest neighbor by Equation (3)
22 $P_T = \frac{1}{n}\left(\sum_{i=1}^{n}\sum_{j=1}^{k} P_{i,j} * \omega_{i,j}\right)$
23 return $P_T$
```

$$CosineSimilarity\left(\boldsymbol{E_D}_i, \boldsymbol{E_T}_j\right) = \frac{\boldsymbol{E_D}_i \cdot \boldsymbol{E_T}_j}{\parallel \boldsymbol{E_D}_i \parallel \parallel \boldsymbol{E_T}_j \parallel} = \frac{\sum_{l=1}^{t} {E_D}_i^l {E_T}_j^l}{\sqrt{\sum_{l=1}^{t}\left({E_D}_i^l\right)^2}\sqrt{\sum_{l=1}^{t}\left({E_T}_j^l\right)^2}} \tag{2}$$

where $\boldsymbol{E_D}_i$ and $\boldsymbol{E_T}_j$ are the contextual representations (vectors) of the ith query in the training set an the jth query in the test set, respectively. t is the length of the representation vector. For each test query, we compare its similarity with all the queries in the training set (Lines 2-7).

Step 3: kNN locating. Given the similarity matrix calculated in **Step 2**, the k-nearest neighbors (kNN) algorithm is applied, for each test query, to locate the first k most similar queries from the training set (Lines 8-11). kNN is a simple and easy-to-implement machine learning algorithm that is

Table 1. Examples of the most similar queries from the training set for queries from the test set. The reciprocal rank of the ground truth in the result list is the model performance we consider in this paper. Dataset: Ruby. Model: GraphCodeBERT. For more details of the dataset, model, and performance measure, please refer to Section 4.2.

No.	Query source	Query content	Reciprocal rank	Similarity
1	Test set	Get all comments for a commit	0.1667	0.8223
	Training set	Fetch comments for PRs and add them to comments	0.2500	
2	Test set	Produces a log message	0.1111	0.8528
	Training set	Logs a message.	0.2000	
3	Test set	Create a new builder object for evaluation.	0.1429	0.8427
	Training set	Create a new builder.	0.1429	
4	Test set	Extracts the downloaded archive file into project_dir.	1.0000	0.8334
	Training set	Extract the given tarball to the target directory	1.0000	
5	Test set	Shells out and runs +command+.	0.5000	0.8649
	Training set	Execute shell command	0.1250	
6	Test set	Remove the file at the given path.	1.0000	0.9496
	Training set	Delete the file at the given path	1.0000	
7	Test set	Save the file to disk.	1.0000	0.8842
	Training set	Save the file	1.0000	
8	Test set	Resets all configuration options to the defaults.	0.5000	0.9919
	Training set	Reset all configuration options to defaults.	0.5000	
9	Test set	Start a timer in the included object	1.0000	0.8136
	Training set	Start the timer	1.0000	
10	Test set	Render the barcode to a PNG image	0.3333	0.8659
	Training set	Writes a barcode PNG image.	0.3333	

widely employed in recommendation system [1], classification [59], and regression [46]. We assume that similar queries, respectively, from training and test sets will trigger a similar model performance. Table 1 lists 10 pairs of queries and the corresponding similarities. The examples are from the Ruby dataset and GraphCodeBERT model (more details can be found in Section 4.2). The semantic similarity between two queries is measured by the widely used cosine similarity [51] in NLP. The last two columns (Reciprocal rank, more details in Section 4.2) show the model performance on each test query and its corresponding matched training query. In most cases (7 of 10), the test query shares the same performance as its matched one.

Step 4: Performance estimation. Finally, we estimate the model performance based on the kNN of test data and the corresponding similarity. As shown in Table 1, similar queries do not always have the same performance. For example, the 5th test query has a similarity of 0.8649 to its most similar query from the training set, but the model performance is 0.5 and 0.125, respectively. This is reasonable because if a test query is very similar to several training queries, the output code snippets can vary. As a result, the selected training query-snippet cannot precisely approximate the model performance on this test query. Concerning this, we propose to calculate the weight of each nearest neighbor to approximate the model performance. We undertake this step in two sub-steps (**individual training subset evaluation** and **weight calculation**). In concrete, first, for each $i = 1 \longrightarrow k$ (Line 12), we can extract a subset of training data including query-snippet pairs of the ith nearest to test queries (Line 13). The subset has the same size as the test set and we can obtain the performance on each query-snippet pair by simply evaluating the model (Line 14).

In **weight calculation**, we utilize the similarity between queries to obtain the weight. Given the similarity matrix NS, for each test query, we first determine the neighbors/similarities to use. This is to avoid the impact of low similarities. For example, the similarities between a test query and its 5 nearest neighbors from the training set are 1, 0.8743, 0.8718, 0.8472, and 0.8443, respectively. The actual performance on this test query is 0.2 and 0.2, 0.2, 1, 1, and 1, respectively, on its 5 neighbors. Using the first nearest neighbor can precisely estimate the performance. However, if we take the average of the 5 results or use similarity as the weight to calculate the final result, there will be an inevitable difference. To solve this problem, we use the Z-Score [54] to adaptively identify to-be-used neighbors (Lines 16-20). In statistics, the Z-Score tells how far a data is from the mean, which can be used to identify outliers in a set of data [40]. A Z-Score of 1.0 indicates that the value is one standard deviation from the mean. A value with a high Z-Score is usually considered as an outlier in the group. In this paper, we experimentally take the neighbors that have an absolute Z-Score of less than 1 into consideration. Thus, given the Z-Scores, $\boldsymbol{z_i}$, of the ith test query and its similarity matrix, $\boldsymbol{NS_i}$, the weight of each neighbor is:

$$\omega_{i,j} = \begin{cases} \frac{NS_{i,j}}{\sum\left\{NS_{i,l} \middle| 1 \leq l \leq k, z_{i,l} \leq 1\right\}} & , z_{i,j} \leq 1 \\ 0 & , otherwise \end{cases} \tag{3}$$

Finally, the model performance is calculated given the weight (Line 21) and performance on each individual training subset (Lines 22-23).

4 Experimental Setup

Our experiments aim to address four research questions:

RQ1 **Effectiveness.** How effective is KAPE in estimating the model performance given an unseen test set?
RQ2 k **sensitivity.** How sensitive is KAPE to the setting of k in the kNN algorithm?
RQ3 **Impact of data distribution.** Does the data distribution w.r.t. the similarity affect KAPE's effectiveness?
RQ4 **Impact of nearest neighbors' weights.** What is the impact of weight calculation on KAPE?

RQ1 gives an insight into KAPE's effectiveness in assessing the model performance using different datasets and models. By **RQ2**, we analyze if KAPE performs consistently given different settings of its only parameter k, which will demonstrate how flexible KAPE is. Since KAPE takes advantage of the similarity between the test queries and training set to approximate the model performance, we conduct **RQ3** to explore if KAPE is stable with different data distributions w.r.t. the similarity. Finally, as an important component of KAPE, the weight calculation is adaptive in the number of k for each test data as well as the weight from each nearest neighbor, which makes KAPE practical for real-world applications. We undertake an ablation study for **RQ4** to verify the importance of this component.

4.1 Implementation details

All experiments were conducted on a high-performance computer cluster and each cluster node runs a 2.6 GHz Intel Xeon Gold 6132 CPU with an NVIDIA Tesla V100 16G SXM2 GPU. We implement KAPE and baseline methods using the PyTorch 1.6.0 framework. We repeat each experiment three times to reduce the influence of randomness. Additionally, for reproducing the results, we use fixed random seeds of 0, 1, and 2 in the experiments. Due to the space limitation, we only report the results of the largest dataset PHP for RQ3, and the remaining results corroborating our findings are available on our companion project website [17].

4.2 Datasets, models, and performance measure

Datasets and models. We use the six benchmark datasets provided by the CodeSearchNet challenge [22] including different programming languages, namely, JavaScript, Java, Python, Ruby, PHP, and Go. For all the datasets, we utilize three state-of-the-art pre-trained models for deep code search, RoBERTa [30], CodeBERT [10], and GraphCodeBERT [16], that are superb in learning the contextual representations of both natural and programming language data. The pre-trained models are obtained by the implementation provided by the CodeXGLUE project [32] (epoch number is 5 and default for the other parameters). Each model is fine-tuned using the training set and tested on the validation set. The test set is regarded as unseen data and untouched in the fine-tuning procedure. Table 2 lists more details of each dataset.

Table 2. Summary of datasets and the MRR on the validation (left) and test sets (right).

Dataset	#Training	#Validation	#Test	MRR					
				RoBERTa		CodeBERT		GraphCodeBERT	
JavaScript	123,889	8,253	6,483	0.4867	0.5125	0.5493	0.5795	0.5682	0.5992
Java	454,451	15,328	26,909	0.5005	0.4828	0.5329	0.5227	0.5385	0.5291
Python	412,178	23,107	22,176	0.5812	0.6149	0.6401	0.6735	0.6546	0.6905
Ruby	48,791	2,209	2,279	0.6214	0.5717	0.7080	0.6360	0.7244	0.6602
PHP	523,712	26,015	28,391	0.4750	0.4475	0.5142	0.4856	0.5242	0.4947
Go	317,832	14,242	14,291	0.7872	0.7018	0.8048	0.7301	0.8054	0.7296

MRR. We adopt the widely used mean reciprocal rank (MRR) [15, 32] in our experiments to measure the model performance. MRR is calculated by:

$$MRR = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{rank_i} \tag{4}$$

where $rank_i$ is the position of the matched code snippet in the returned results of the ith test query. The higher the MRR, the better the searching performance. Note that, $P_{i,j}$ in Algorithm 1 is equal to $\frac{1}{rank_i}$ instead of the MRR. The reciprocal rank of each query in Table 1 is also $\frac{1}{rank_i}$.

4.3 Baseline methods

Concerning that this is the first performance estimation work for deep code search, we take three test selection metrics (random sampling, PACE, and DeepGini) and three acquisition functions (LC, Margin sampling, and MaxEntropy) from active learning that are widely studied in the CV and NLP domains [6, 20] as the baseline methods. Note that, studies [18–20] have demonstrated that the acquisition functions can act as test selection metrics. In addition, as KAPE selects data from the training set, we propose the baseline method of randomly selecting data from the training set. Without loss of generality, we use **random sampling (test)** to refer to the sampling from the test set and **random sampling (train)** for sampling from the training set.

- **Random sampling (test)** A fixed number of test data is randomly selected and the corresponding code snippets are manually matched.
- **PACE [6]** The **P**ractical **AC**curacy **E**stimation (PACE) method first divides the test queries into different clusters using the HDBSCAN (hierarchical density-based spatial clustering of applications with noise) clustering algorithm. Then PACE utilizes the MMD-critic algorithm [23] to select the most representative data from each cluster proportionally concerning the cluster size. PACE was initially proposed for image classification and regression tasks.

- **DeepGini [9]** For a classification task (e.g., image classification), it selects the most informative data that have the highest Gini impurity. The Gini impurity measures how likely a sample is wrongly classified based on the prediction possibilities of all classes.
- **LC [18]** The least confidence (LC) metric selects data where the model has the least confidence (probability) in the most likely class label.
- **Margin sampling [42]** Similar to LC, the margin sampling considers the prediction confidence. Instead of using the most likely class, it selects data that have the smallest difference between the first and second most probable class labels.
- **MaxEntropy [18]** This metric selects the most uncertain data where the Shannon entropy of the prediction probability is the highest. The only difference with DeepGini is the measure (Gini impurity and Shannon entropy) used to calculate the uncertainty.
- **Random sampling (train)** A set of training data is randomly selected from the training set.

Note that PACE, DeepGini, LC, Margin sampling, and MaxEntropy require the prediction probability to perform the clustering procedure or the uncertainty calculation. Since in the deep code search task, the prediction probability is unavailable, these metrics are not directly applicable. To solve this issue, we calculate the similarity between each test query and its predicted 10 best-match code snippets from the training set and apply softmax function [13] to obtain the probability for each best-match. The probabilities of 10 best-match are considered as the prediction probability of 10 classes.

Table 3 presents the four main differences between KAPE and baseline methods. First, compared to test selection metrics where a subset of test data approximates the model performance on the entire test set, KAPE selects data from the training set to achieve the goal. Second, in test selection metrics, the selected data size depends on the given budget of manpower and is usually much smaller than the given test set size. By contrast, KAPE selects the same size of training data. Third, since KAPE relies on the training set to undertake the performance estimation, no manpower is required, which is more practical. Finally, due to the sampling randomness, the random manner (from test or training data) has low stability of performance estimation. Namely, the estimated performance by random sampling varies among several repetitions.

Table 3. Differences between KAPE and baseline methods.

Method	Selection object	Selection number	Manpower free	Stability
Random sampling (test)	test data	limit to manpower budget	×	×
PACE				√
LC, Margin				√
MaxEntropy, DeepGini				√
Random sampling (train)	training data	the same as test data	√	×
KAPE				√

For the six test selection metrics, we use different labeling percentages, i.e., 1%, 3%, 5%, . . . , and 50%. 1% means that 1% test data is selected. The sampling size of random sampling (train) is the same as the unseen test set.

5 Results and Discussion

In this section, we first compare the performance estimation effectiveness of KAPE and baseline methods. Next, by assigning different ks we explore the KAPE's sensitivity to k. Third, we investigate if the type of data influences KAPE's effectiveness. The type of data means test queries are very similar or different to the training set. Finally, we discuss the necessity of adaptive weight calculation in Section 3.3 via an ablation study.

5.1 RQ1: Effectiveness

Figure 4 shows the comparison between KAPE and seven baseline methods based on the CodeBERT model. Compared to random sampling (train), KAPE always estimates the model performance more accurately for all datasets and models. We can first conclude that selecting data from the training set based on the semantic similarity between training and test data is more reasonable than simple test-independent sampling. On the other hand, the effectiveness of random sampling (test) and Margin sampling improves along with increasing the labeling percentage. The other test selection metrics perform inconsistently across different datasets. For instance, LC, MaxEntropy, and DeepGini perform well on JavaScript and Ruby but act extremely badly on Java, PHP, and Go. In particular, when the labeling percentage is less than 10%, these three metrics estimate the model performance as around 0, which is far from the ground truth. In addition, PACE improves the effectiveness along with the increment of labeling budget on JavaScript, Python, and Ruby but degrades on the other datasets. By contrast, in most cases, KAPE outperforms baselines in the six datasets regardless of the labeling percentage. For instance, in PHP, random sampling (test), LC, MaxEntropy, Margin, and DeepGini can only reach a competitive performance when manually matching more than 50% (14,196) code snippets from the unseen test queries. In addition, due to the sampling randomness, there is a performance deviation in random sampling (train) and random sampling (test) over three repetitions, which is avoided by KAPE.

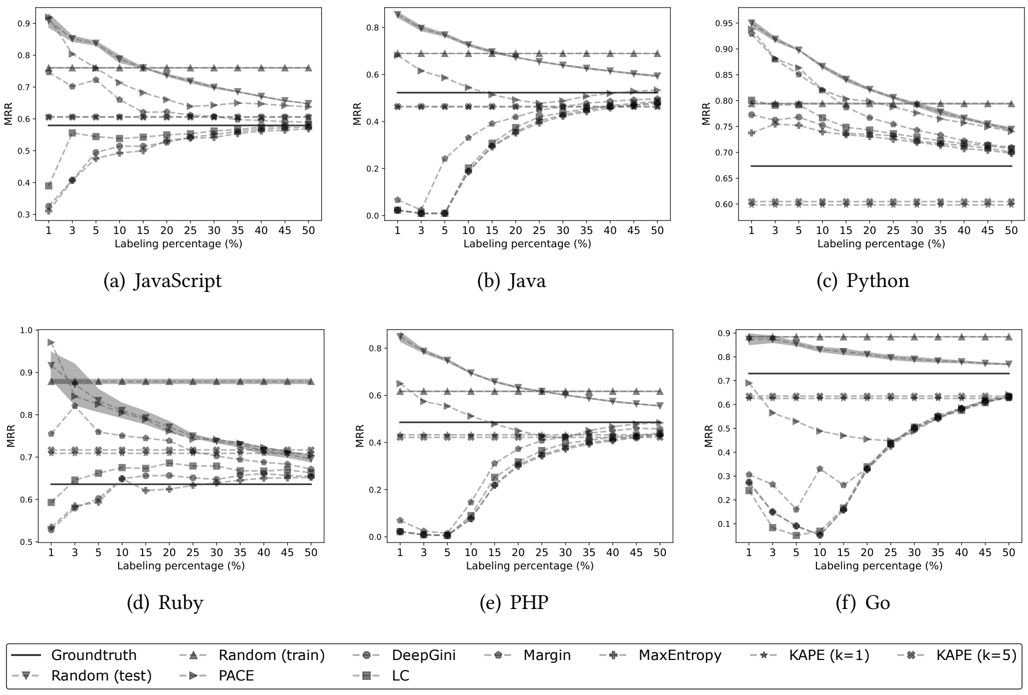


Fig. 4. Effectiveness comparison between KAPE and baseline methods given the CodeBERT model. **Groundtruth:** the actual MRR of the model on the test set. Shaded area illustrates the standard deviation of three experiment repetitions.

Table 4 lists the results of KAPE on the PHP dataset based on the RoBERTa and GraphCodeBERT models. KAPE always outperforms random sampling (test), random sampling (train), DeepGini, Margin sampling, and MaxEntropy regardless of the labeling percentage and model. With more manpower added, the effectiveness of LC improves. However, it still requires at least 50% (14196) and 45% (12776) manually matched query-code pairs given the RoBERTa and GraphCodeBERT models, respectively, to achieve similar performance to KAPE. Similar to the case in CodeBERT (Figure 4(e)), PACE occasionally degrades the performance when increasing the labeling percentage. For instance, in GraphCodeBERT, PACE outperforms KAPE when the labeling percentage is 5%, 10%, and 15% but fails when increasing the percentage to 20%, 25%, and 30%.

Table 4. Effectiveness comparison between KAPE and baseline methods on PHP given the RoBERTa and GraphCodeBERT models. Values highlighted in grey indicate that KAPE outperforms the baseline methods. **Groundtruth:** the actual MRR of the model on the test set.

Dataset	**Labeling percentage (%)**											
	1	**3**	**5**	**10**	**15**	**20**	**25**	**30**	**35**	**40**	**45**	**50**
						RoBERTa						
Groundtruth	0.4475	0.4475	0.4475	0.4475	0.4475	0.4475	0.4475	0.4475	0.4475	0.4475	0.4475	0.4475
Random (test)	0.8295	0.7647	0.7222	0.6612	0.6249	0.5986	0.5814	0.5648	0.5500	0.5360	0.5255	0.5162
Random (train)	0.5686	0.5686	0.5686	0.5686	0.5686	0.5686	0.5686	0.5686	0.5686	0.5686	0.5686	0.5686
PACE	0.6320	0.5676	0.5240	0.4696	0.4342	0.4113	0.3932	0.4006	0.4232	0.4375	0.4449	0.4513
DeepGini	0.0219	0.0086	0.0055	0.0808	0.1919	0.2619	0.3054	0.3247	0.3464	0.3636	0.3772	0.3857
LC	0.0219	0.0086	0.0055	0.0862	0.2286	0.2817	0.3234	0.3468	0.3565	0.3735	0.3857	0.3918
Margin	0.5030	0.2665	0.1547	0.1243	0.2836	0.3501	0.3718	0.3936	0.4066	0.4126	0.4149	0.4151
MaxEntropy	0.0219	0.0086	0.0055	0.0788	0.1928	0.2616	0.2901	0.3211	0.3429	0.3571	0.3719	0.3801
KAPE (k = 1)	0.3880	0.3880	0.3880	0.3880	0.3880	0.3880	0.3880	0.3880	0.3880	0.3880	0.3880	0.3880
KAPE (k = 5)	0.3766	0.3766	0.3766	0.3766	0.3766	0.3766	0.3766	0.3766	0.3766	0.3766	0.3766	0.3766
						GraphCodeBERT						
Groundtruth	0.4947	0.4947	0.4947	0.4947	0.4947	0.4947	0.4947	0.4947	0.4947	0.4947	0.4947	0.4947
Random (test)	0.8512	0.7967	0.7559	0.7061	0.6699	0.6466	0.6308	0.6140	0.5988	0.5853	0.5741	0.5657
Random (train)	0.6328	0.6328	0.6328	0.6328	0.6328	0.6328	0.6328	0.6328	0.6328	0.6328	0.6328	0.6328
PACE	0.6445	0.5553	0.5298	0.4776	0.4467	0.4267	0.4109	0.4349	0.4647	0.4820	0.4943	0.4991
DeepGini	0.0219	0.0086	0.0055	0.0762	0.2106	0.2842	0.3391	0.3742	0.3946	0.4113	0.4253	0.4367
LC	0.0219	0.0086	0.0055	0.0809	0.2347	0.3001	0.3552	0.3813	0.4055	0.4187	0.4309	0.4438
Margin	0.5572	0.5939	0.3959	0.1901	0.2887	0.3643	0.3986	0.4136	0.4300	0.4401	0.4492	0.4565
MaxEntropy	0.0219	0.0086	0.0055	0.0713	0.2071	0.2862	0.3334	0.3682	0.3896	0.4074	0.4197	0.4323
KAPE (k = 1)	0.4442	0.4442	0.4442	0.4442	0.4442	0.4442	0.4442	0.4442	0.4442	0.4442	0.4442	0.4442
KAPE (k = 5)	0.4380	0.4380	0.4380	0.4380	0.4380	0.4380	0.4380	0.4380	0.4380	0.4380	0.4380	0.4380

Additionally, as shown in Figure 4 and Table 4, KAPE performs consistently with different settings of k, i.e., $k = 1$ and $k = 5$ and the difference is very slight. To analyze the sensitivity of KAPE to this parameter, we conduct the next experiment in Section 5.2.

Answer to RQ1: KAPE is efficient in performance estimation and outperforms the test selection metrics in most cases (e.g., even with 50% manual matching in PHP, GraphCodeBERT, MaxEntropy still performs worse than KAPE by 0.0119). In addition, KAPE outperforming random sampling (train) indicates that KAPE's efficiency comes from the similarity-based selection strategy.

5.2 RQ2: k sensitivity analysis

KAPE only includes one parameter, k, that determines the number of nearest neighbors of test queries to consider for performance estimation. To investigate whether KAPE is sensitive to this parameter or not, we conduct the sensitivity analysis experiment by using 10 different ks ($k = 1, 2, \ldots, 10$).

Table 5 shows KAPE's effectiveness given different ks with the statistical average and standard deviation. In general, the deviation is slight and ranges from 0.0011 to 0.0090, which draws the conclusion that KAPE is insensitive to k. On the other hand, we observe that, in many cases, KAPE performs equally with $k = 1$ and $k = 2$. For example, for JavaScript, KAPE obtains the same result regardless of the model. Additionally, given the GraphCodeBERT, Table 5 shows that KAPE has a greater deviation on Ruby (0.009) and Python (0.0053) than other datasets. We conjecture the reason is that the unseen test sets in these two datasets include many data that have very low and different similarities to their nearest neighbors. Thus, more neighbors and their corresponding similarities are considered in the weight calculation. To verify this, we conduct the next experiment to explore the impact of data similarity on KAPE.

Table 5. Comparison of KAPE using 10 different k settings. "Avg" and "Std" are short for average and standard deviation, respectively.

Dataset	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$	**Avg**	**Std**
						RoBERTa						
JavaScript	0.5136	0.5136	0.5141	0.5118	0.5119	0.5111	0.5120	0.5139	0.5137	0.5140	0.5130	0.0011
Java	0.3993	0.4020	0.4021	0.4040	0.4047	0.4051	0.4065	0.4069	0.4073	0.4077	0.4045	0.0026
Python	0.5347	0.5391	0.5389	0.5394	0.5406	0.5416	0.5423	0.5435	0.5445	0.5447	0.5409	0.0029
Ruby	0.6307	0.6405	0.6427	0.6474	0.6486	0.6523	0.6548	0.6566	0.6589	0.6608	0.6493	0.0089
PHP	0.3880	0.3841	0.3828	0.3792	0.3766	0.3738	0.3723	0.3708	0.3697	0.3685	0.3766	0.0064
Go	0.6036	0.6066	0.6078	0.6106	0.6134	0.6139	0.6148	0.6158	0.6164	0.6174	0.6120	0.0044
						CodeBERT						
JavaScript	0.6065	0.6065	0.6066	0.6042	0.6056	0.6075	0.6073	0.6079	0.6088	0.6091	0.6070	0.0014
Java	0.4627	0.4627	0.4626	0.4629	0.4637	0.4654	0.4663	0.4677	0.4688	0.4701	0.4653	0.0027
Python	0.5983	0.6015	0.6008	0.6032	0.6049	0.6065	0.6072	0.6080	0.6092	0.6104	0.6050	0.0038
Ruby	0.7089	0.7111	0.7111	0.7127	0.7171	0.7216	0.7247	0.7266	0.7281	0.7292	0.7191	0.0074
PHP	0.4325	0.4275	0.4254	0.4229	0.4217	0.4198	0.4190	0.4181	0.4172	0.4163	0.4220	0.0049
Go	0.6258	0.6309	0.6313	0.6341	0.6356	0.6357	0.6363	0.6374	0.6381	0.6388	0.6344	0.0038
						GraphCodeBERT						
JavaScript	0.6270	0.6270	0.6287	0.6311	0.6317	0.6330	0.6345	0.6360	0.6360	0.6367	0.6322	0.0035
Java	0.4770	0.4770	0.4766	0.4785	0.4810	0.4824	0.4837	0.4841	0.4853	0.4864	0.4812	0.0035
Python	0.6200	0.6251	0.6264	0.6284	0.6307	0.6321	0.6337	0.6348	0.6366	0.6377	0.6306	0.0053
Ruby	0.7398	0.7433	0.7433	0.7483	0.7519	0.7564	0.7587	0.7627	0.7646	0.7655	0.7534	0.0090
PHP	0.4442	0.4428	0.4408	0.4390	0.4380	0.4365	0.4355	0.4342	0.4334	0.4328	0.4377	0.0037
Go	0.6204	0.6230	0.6240	0.6280	0.6293	0.6312	0.6320	0.6324	0.6330	0.6336	0.6287	0.0044

Answer to RQ2: KAPE's effectiveness is stable with different k settings with a slight deviation less than 0.01.

5.3 RQ3: Impact of data distribution

Concerning that KAPE is based on the semantic similarity between test queries and the training set, we explore, in this research question, the impact of data distribution. Namely, if KAPE is flexible to perform well when given a test set that has a big or small similarity with the training set. Here, the data distribution refers to the distribution based on semantic similarity. We evenly split each test set into two subsets, i.e., a similar set and a different set. In concrete, the similarity between each test query and its nearest neighbor is first calculated. Then we group each test data into a similar set if its similarity is greater than half of the test, and vice versa. Figure 5 shows the density distribution of similarities of the two sets for each dataset. In PHP and Go, the difference between similar and different test sets is smaller than in other datasets since most data have a similarity between 0.8 and 1.0. For example, in Ruby, the lower bound of data similarity reaches 0.5.

Figure 6 shows the results on the similar and different sets on the PHP dataset. In general, KAPE still outperforms baseline methods in most cases and performs better on similar test sets.

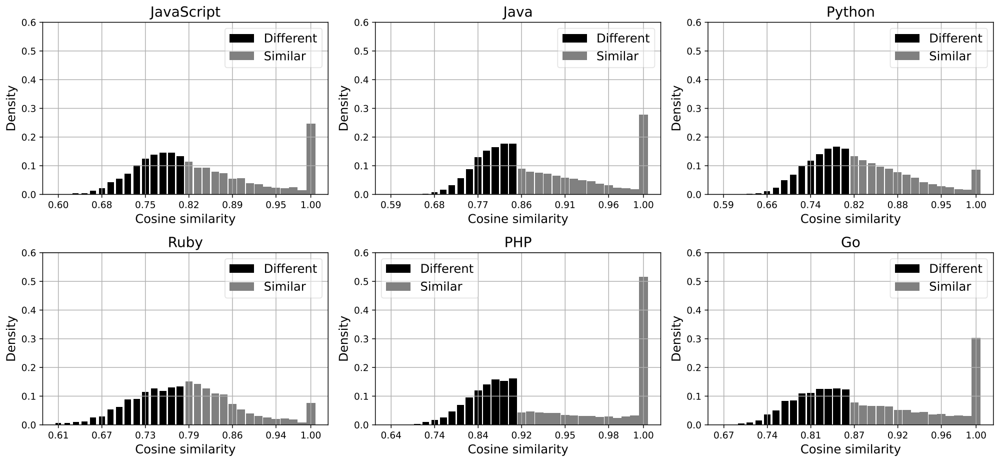


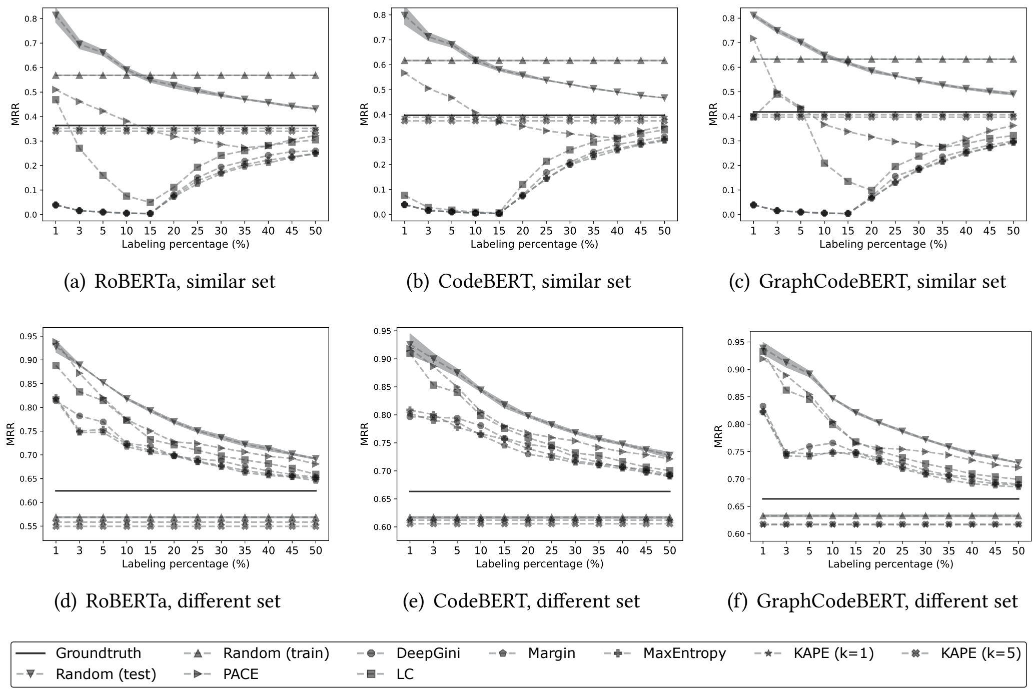
Fig. 5. Similarity density distribution of test sets where data is "similar" to or "different" from the training set. Model: CodeBERT.

Concerning the similar set, KAPE performs the best regardless of the model by achieving only 0.0118, 0.0079, and 0.0112 differences with the ground truth given RoBERTa, CodeBERT, and GraphCodeBERT, respectively. Random sampling (train) still performs worse than KAPE as in Section 5.1. Random sampling (test) improves the performance with a greater labeling percentage but consistently performs worse than KAPE even with the labeling percentage at 50%. LC, MaxEntropy, and DeepGini estimate the model performance around 0 when the labeling percentage is less than 20% (5678 test data). The effectiveness of Margin sampling and PACE varies much when increasing the labeling percentage. For instance, Margin sampling performs the worst when 15% of test queries are manually matched with code snippets and improves the performance by increasing and decreasing the percentage. Concerning the different sets, in general, all test selection metrics improve the effectiveness when increasing the labeling percentage. In all models, when the labeling percentage is less than 30%, KAPE still outperforms all these metrics. By comparison, the estimation error increases given the different sets and is up to 0.0660 (RoBERTa). The reason is that when the nearest neighbors from the training set are similar to the test data, the performance of the training data is more reliable to be transferred to the test set.

Answer to RQ3: Although KAPE is based on the similarity between test and training sets, its effectiveness is flexible to various (similar or different) data distributions. In addition, KAPE benefits more from the similar data distribution where most test data are close to the training set.

5.4 RQ4: Impact of nearest neighbors' weights

Recall that in the last step of KAPE, given k, the number ($\leq k$) of used nearest neighbors of each test query and the weight of each nearest neighbor are assigned adaptively based on the Z-Score and similarity (Section 3.3). To verify the importance of this adaptive calculation, we conduct an ablation study. We compare KAPE to the other three manners of obtaining the estimated performance (Equation (3), Line 21 in Algorithm 1):



(a) RoBERTa, similar set (b) CodeBERT, similar set (c) GraphCodeBERT, similar set

(d) RoBERTa, different set (e) CodeBERT, different set (f) GraphCodeBERT, different set

Fig. 6. Effectiveness of KAPE and baseline methods given the similar (first row) and different (second row) PHP test set. **Groundtruth:** the actual MRR of the model on the test set. Shaded area illustrates the standard deviation of three experiment repetitions.

- Fixed k, equal weight: for each test query, we assign equal weights to its k nearest neighbors. The variant of Equation (3) is defined as:

$$\omega_{i,j} = \frac{1}{k} \tag{5}$$

- Fixed k, adaptive weight: we utilize the similarity to determine the weight:

$$\omega_{i,j} = \frac{NS_{i,j}}{\sum_{l=1}^{k} NS_{i,l}} \tag{6}$$

- Adaptive k, equal weight: we utilize a flexible number of k for each test query based on the Z-Score and re-define Equation (3) as:

$$\omega_{i,j} = \begin{cases} \frac{1}{|\{NS_{i,l} | 1 \le l \le k, z_{i,l} \le 1\}|} & , z_{i,j} \le 1 \\ 0 & , otherwise \end{cases} \tag{7}$$

Table 6 presents the statistical result of adaptive k calculated by the Z-Score method. It shows that with the adaptive manner, test queries have different numbers of nearest neighbors contributing to the weight calculation. For example, in Go, GraphCodeBERT, given the pre-set maximum $k = 10$, the 14,291 test queries have an average use of nearest neighbors of 8.97, and the minimum number drops to 5. In concrete, only 43.96% (6282) test data fully take all the 10 neighbors and 1 data uses 5 neighbors. 20.26% (2896), 25.21% (3604), 9.94% (1406), and 0.71% (102) test data use 9, 8, 7, and 6 neighbors for the weight calculation, respectively. This is reasonable given that, for each test query, the similarity between it and its nearest neighbors can be significantly different from the others.

Table 6. Statistics (Min: minimum and Avg: average) of involved k nearest neighbors by KAPE given a fixed k ($k = 3, 4, \ldots, 10$).

Model	$k = 3$		$k = 4$		$k = 5$		$k = 6$		$k = 7$		$k = 8$		$k = 9$		$k = 10$	
	Min	**Avg**	**Min**	**Avg**	**Min**	**Avg**	**Min**	**Avg**	**Min**	**Avg**	**Min**	**Avg**	**Min**	**Avg**	**Min**	**Avg**
JavaScript																
RoBERTa	2	2.49	3	3.45	3	4.33	4	5.28	4	6.16	5	7.09	5	8.01	6	8.92
CodeBERT	2	2.47	3	3.45	3	4.33	3	5.27	4	6.18	5	7.09	5	8.00	6	8.92
GraphCodeBERT	2	2.48	2	3.44	3	4.32	3	5.27	4	6.17	5	7.09	5	7.99	5	8.90
Java																
RoBERTa	2	2.50	3	3.48	3	4.36	3	5.29	4	6.19	5	7.10	5	8.00	5	8.91
CodeBERT	2	2.50	3	3.48	3	4.37	3	5.29	4	6.18	5	7.09	5	7.99	5	8.91
GraphCodeBERT	2	2.50	3	3.47	3	4.35	3	5.27	4	6.17	5	7.09	5	8.00	5	8.91
Python																
RoBERTa	2	2.43	3	3.41	3	4.29	3	5.20	4	6.10	5	7.01	5	7.92	6	8.84
CodeBERT	2	2.44	3	3.41	3	4.29	3	5.21	4	6.11	5	7.01	5	7.93	6	8.85
GraphCodeBERT	2	2.44	3	3.41	3	4.30	3	5.21	4	6.11	5	7.03	5	7.93	5	8.84
Ruby																
RoBERTa	2	2.44	3	3.40	3	4.29	4	5.20	4	6.10	5	6.98	5	7.89	6	8.82
CodeBERT	2	2.44	3	3.38	3	4.26	4	5.18	4	6.10	5	6.99	5	7.91	6	8.83
GraphCodeBERT	2	2.41	2	3.38	3	4.27	4	5.19	4	6.09	5	6.98	5	7.90	6	8.82
PHP																
RoBERTa	2	2.56	3	3.54	3	4.41	3	5.35	4	6.25	5	7.17	5	8.08	5	9.00
CodeBERT	2	2.56	3	3.54	3	4.40	3	5.35	4	6.25	5	7.16	5	8.07	5	8.99
GraphCodeBERT	2	2.56	3	3.54	3	4.42	3	5.35	4	6.25	5	7.17	5	8.07	5	8.99
Go																
RoBERTa	2	2.49	3	3.48	3	4.37	4	5.30	4	6.23	5	7.16	5	8.08	5	9.00
CodeBERT	2	2.50	3	3.48	3	4.38	4	5.30	4	6.21	5	7.12	5	8.05	6	8.95
GraphCodeBERT	2	2.49	3	3.47	3	4.37	4	5.30	4	6.22	5	7.14	5	8.06	5	8.97

Table 7 presents the results of the ablation study on the JavaScript dataset. Regardless of the model, the adaptive setting of k and weight achieves the best result in most cases. With a very small k ($k = 1$ and $k = 2$), the 4 ways of weight calculation obtain the same results, but the estimation precision can be lower than using greater ks. For example, in RoBERTa, using 7 nearest neighbors can reach a difference at 0.0005 but 0.0011 by only using the first neighbor. When increasing the k, the advantage of using the adaptive setting stands out. Additionally, given fixed k, using adaptive weight is always better than the equal manner in most cases, which also happens when given adaptive k. On the other hand, given adaptive weight, using adaptive k is always better.

Answer to RQ4: Our proposed adaptive strategy to determine the number of nearest neighbors and the corresponding weight contributes positively to KAPE's effectiveness.

5.5 Human evaluation

We conducted a human study to evaluate the effectiveness of KAPE. Five experienced software developers, familiar with Java, were invited to participate in the study. First, we obtained the answers by the RoBERTa and CodeBERT models (see more details in Section 4.2) to 23 Java-related queries collected from StackOverflow. Subsequently, the developers were asked to manually inspect and verify the correctness of these answers. Specifically, each developer was given the answers generated by two models along with corresponding queries, all while being unaware of the specific details (e.g., name, architecture, or any other distinguishing characteristics.) about the underlying model. The results in Figure 7 revealed that, on average, RoBERTa provided 16.4 out of 23 correct answers, whereas CodeBERT yielded 17.8 out of 23 correct answers. This suggests that

Table 7. Ablation study on the k nearest neighbors' weights of KAPE. Values highlighted in grey indicate the best performance. **Groundtruth:** the actual MRR of the model on the test set. Dataset: JavaScript.

Dataset	Base	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$	$k = 6$	$k = 7$	$k = 8$	$k = 9$	$k = 10$
RoBERTa											
Fixed k, equal weight	0.5125	0.5136	0.5136	0.5135	0.5120	0.5120	0.5127	0.5136	0.5145	0.5151	0.5149
Fixed k, adaptive weight	0.5125	0.5136	0.5136	0.5136	0.5120	0.5121	0.5128	0.5136	0.5145	0.5151	0.5149
Adaptive k, equal weight	0.5125	0.5136	0.5136	0.5141	0.5117	0.5119	0.5110	0.5119	0.5139	0.5137	0.5140
Adaptive k, adaptive weight (KAPE)	0.5125	0.5136	0.5136	0.5141	0.5118	0.5119	0.5111	0.5120	0.5139	0.5137	0.5140
CodeBERT											
Fixed k, equal weight	0.5795	0.6065	0.6065	0.6047	0.6057	0.6075	0.6077	0.6083	0.6083	0.6092	0.6101
Fixed k, adaptive weight	0.5795	0.6065	0.6065	0.6048	0.6057	0.6074	0.6076	0.6083	0.6082	0.6091	0.6100
Adaptive k, equal weight	0.5795	0.6065	0.6065	0.6066	0.6042	0.6056	0.6076	0.6073	0.6079	0.6089	0.6092
Adaptive k, adaptive weight (KAPE)	0.5795	0.6065	0.6065	0.6066	0.6042	0.6056	0.6075	0.6073	0.6079	0.6088	0.6091
GraphCodeBERT											
Fixed k, equal weight	0.5992	0.6270	0.6270	0.6306	0.6317	0.6334	0.6357	0.6361	0.6371	0.6380	0.6387
Fixed k, adaptive weight	0.5992	0.6270	0.6270	0.6305	0.6316	0.6333	0.6355	0.6359	0.6368	0.6377	0.6384
Adaptive k, equal weight	0.5992	0.6270	0.6270	0.6287	0.6312	0.6318	0.6331	0.6346	0.6362	0.6363	0.6370
Adaptive k, adaptive weight (KAPE)	0.5992	0.6270	0.6270	0.6287	0.6311	0.6317	0.6330	0.6345	0.6360	0.6360	0.6367

the developers considered CodeBERT to be a superior search model for these particular queries. On the other hand, we employed KAPE to estimate the MRR for both RoBERT and CodeBERT. Notably, CodeBERT was estimated to have a higher MRR (0.9783) than RoBERTa (0.9565), which aligns with the conclusion from the human inspection.

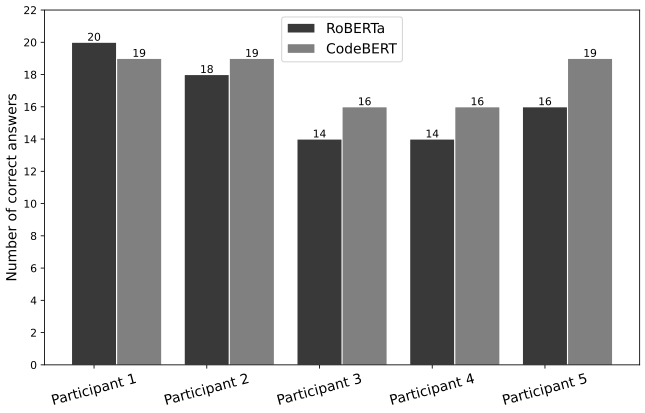


Fig. 7. Results of the user study to evaluate the effectiveness of KAPE. The x-axis represents the 5 independent participants and the y-axis corresponds to the number of correct answers identified by participants.

6 Discussion

6.1 Strengths and limitations

Strengths First, unlike the test selection metrics where a subset of test data is selected and manually labeled, KAPE does not require manpower for the test data. Second, although based on the similarity between training and test data, KAPE is flexible to different data distributions, e.g., the unseen test data is very similar or very different to the training set.

Limitations Since KAPE performs the performance estimation based on the semantic similarity between training and test queries, the training set is required to be accessible. In addition, as demonstrated by RQ3 in Section 5.3, KAPE benefits more from the similarity when the test data is more similar to the training set than different. A new method is in demand for accurate performance estimation.

6.2 Threats to validity

The internal threat mainly comes from the implementations of baseline methods, KAPE, model preparation, and testing. For random sampling (test) and random sampling (train), we apply the random module in Python. We use the original implementation of PACE [6] and use the same parameters, e.g., the maximum number of clusters and MMD-critic-related settings. The definitions of DeepGini [9], LC [18], Margin sampling, and MaxEntropy [18] are simple and these metrics are easy to implement. For model preparation and evaluation, we use the original implementation on GitHub [32] provided by Lu *et al.* For KAPE, the cosine similarity calculation is implemented using the public library SciPy [43].

The external threat is due to selected datasets, models, baseline methods, and evaluation measures. Regarding the datasets, we test on all the 6 benchmark datasets provided by the CodeSearchNet challenge [22]. For the models, we employ 3 popular and state-of-the-art pre-trained models for deep code search. For comparison, concerning that our work is the first and the test input selection metrics in other fields are inapplicable, we consider the most widely used baseline method, random sampling (test), and apply different labeling percentages. In addition, since KAPE utilizes training data to estimate the model performance, we also implement the baseline method, random sampling (train), that selects training data for comparison. Regarding the performance measure, we consider the widely used MRR. There are other measures, such as Answerd@k [5, 53], which can be considered in a further study.

The construct threat mainly lies in the sampling randomness in the baseline methods. To reduce the impact of randomness, we repeat each experiment 3 times and report the results of both average and standard deviation. Additionally, to allow for reproducibility, we use fixed random seeds (0, 1, and 2) for all the environment settings (e.g., random module in Python, Numpy, Torch, and CUDA's manual seed setting).

7 Conclusion

In this paper, we introduce KAPE, a manpower-free testing approach that takes advantage of the training set to efficiently estimate the performance on unseen test data of deep code search models. Via the kNN algorithm, we map the unseen test data to the training space and assign adaptive weights to the neighbors based on the semantic similarity between training and test data. Experimental results on six programming languages and three pre-trained models demonstrate that KAPE is effective in estimating the model performance and outperforms test selection metrics. In addition, we show that KAPE is randomness-free, stable to parameters, and flexible to data distributions.

For future work, more advanced deep code search models and evaluation measures will be investigated.

Acknowledgments

This work is supported by the Luxembourg National Research Funds (FNR) through CORE project C18/IS/12669767/STELLAR/LeTraon.

References

[1] David Adedayo Adeniyi, Zhaoqiang Wei, and Yang Yongquan. 2016. Automated web usage data mining and recommendation system using K-Nearest Neighbor (KNN) classification method. *Applied Computing and Informatics* 12, 1 (2016), 90–108.

[2] Afsoon Afzal, Manish Motwani, Kathryn T. Stolee, Yuriy Brun, and Claire Le Goues. 2021. SOSRepair: expressive semantic search for real-world program repair. *IEEE Transactions on Software Engineering* 47, 10 (2021), 2162–2181. https://doi.org/10.1109/TSE.2019.2944914

[3] N. S. Altman. 1992. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician* 46, 3 (1992), 175–185. https://doi.org/10.1080/00031305.1992.10475879

[4] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 964–974. https://doi.org/10.1145/3338906.3340458

[5] Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 964–974. https://doi.org/10.1145/3338906.3340458

[6] Junjie Chen, Zhuo Wu, Zan Wang, Hanmo You, Lingming Zhang, and Ming Yan. 2020. Practical accuracy estimation for efficient deep neural network testing. *ACM Transactions on Software Engineering and Methodology* 29, 4, Article 30 (oct 2020), 35 pages. https://doi.org/10.1145/3394112

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: pre-training of deep bidirectional transformers for language understanding. https://doi.org/10.48550/ARXIV.1810.04805

[8] Sen Fang, You-Shuai Tan, Tao Zhang, and Yepang Liu. 2021. Self-attention networks for code search. *Information and Software Technology* 134 (2021), 106542. https://doi.org/10.1016/j.infsof.2021.106542

[9] Yang Feng, Qingkai Shi, Xinyu Gao, Jun Wan, Chunrong Fang, and Zhenyu Chen. 2020. DeepGini: prioritizing massive tests to enhance the robustness of deep neural networks. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 177–188. https://doi.org/10.1145/3395363.3397357

[10] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: a pre-trained model for programming and natural languages. https://doi.org/10.48550/ARXIV.2002.08155

[11] GitHub. 2008. GitHub: a platform and cloud-based service for software development and version control. https://github.com/. Online; accessed 23th August 2023.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Softmax units for multinoulli output distributions. Deep Learning*. MIT Press.

[14] Google. 2007. AI platform data labeling service pricing. https://cloud.google.com/ai-platform/data-labeling/pricing. Online; accessed 23th August 2023.

[15] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 933–944. https://doi.org/10.1145/3180155.3180167

[16] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2020. GraphCodeBERT: pre-training code representations with data flow. https://doi.org/10.48550/ARXIV.2009.08366

[17] Yuejun Guo. 2022. Project site of KAPE. https://sites.google.com/view/kape4dcs/. Online; accessed 23th August 2023.

[18] Yuejun Guo, Qiang Hu, Maxime Cordy, Mike Papadakis, and Yves Le Traon. 2021. Robust active learning: sample-efficient training of robust deep learning models. *CoRR* abs/2112.02542 (2021). arXiv:2112.02542 https://arxiv.org/abs/2112.02542

[19] Qiang Hu, Yuejun Guo, Maxime Cordy, Xiaofei Xie, Lei Ma, Mike Papadakis, and Yves Le Traon. 2022. An empirical study on data distribution-aware test selection for deep learning enhancement. *ACM Transactions on Software Engineering and Methodology* (January 2022). https://doi.org/10.1145/3511598

[20] Qiang Hu, Yuejun Guo, Maxime Cordy, Xiaofei Xie, Wei Ma, Mike Papadakis, and Yves Le Traon. 2021. Towards exploring the limitations of active learning: an empirical study. In *36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 917–929. https://doi.org/10.1109/ASE51524.2021.9678672

[21] Qiang Hu, Yuejun Guo, Xiaofei Xie, Maxime Cordy, Mike Papadakis, Lei Ma, and Yves Le Traon. 2023. Aries: Efficient Testing of Deep Neural Networks via Labeling-Free Accuracy Estimation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1776–1787.

[22] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[23] Been Kim, Rajiv Khanna, and Oluwasanmi O Koyejo. 2016. Examples are not enough, learn to criticize! criticism for interpretability. *Advances in neural information processing systems* 29 (2016).

[24] Kisub Kim, Dongsun Kim, Tegawendé F. Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. FaCoY – a code-to-code search engine. In *IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 946–957. https://doi.org/10.1145/3180155.3180187

[25] Pang Wei Koh, Shiori Sagawa, Henrik Marklund, Sang Michael Xie, Marvin Zhang, Akshay Balsubramani, Weihua Hu, Michihiro Yasunaga, Richard Lanas Phillips, Irena Gao, Tony Lee, Etienne David, Ian Stavness, Wei Guo, Berton Earnshaw, Imran Haque, Sara M Beery, Jure Leskovec, Anshul Kundaje, Emma Pierson, Sergey Levine, Chelsea Finn, and Percy Liang. 2021. WILDS: a benchmark of in-the-wild distribution shifts. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 5637–5664. https://proceedings.mlr.press/v139/koh21a.html

[26] Seokhyun Lee, Sooyoung Cha, Dain Lee, and Hakjoo Oh. 2020. Effective white-box testing of deep neural networks with adaptive neuron-selection strategy. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 165–176. https://doi.org/10.1145/3395363.3397346

[27] Zenan Li, Xiaoxing Ma, Chang Xu, Chun Cao, Jingwei Xu, and Jian Lü. 2019. Boosting operational dnn testing efficiency through conditioning. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 499–509. https://doi.org/10.1145/3338906.3338930

[28] Chao Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and John Grundy. 2021. Opportunities and challenges in code search tools. *ACM Comput. Surv.* 54, 9, Article 196 (oct 2021), 40 pages. https://doi.org/10.1145/3480027

[29] Shangqing Liu, Xiaofei Xie, Lei Ma, Jing Kai Siow, and Yang Liu. 2021. GraphSearchNet: enhancing GNNs via capturing global dependency for semantic code search. *CoRR* abs/2111.02671 (2021). arXiv:2111.02671 https://arxiv.org/abs/2111.02671

[30] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: a robustly optimized BERT pretraining approach. https://doi.org/10.48550/ARXIV.1907.11692

[31] Google LLC. 1998. Google. https://www.google.com/. Online; accessed 23th August 2023.

[32] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: a machine learning benchmark dataset for code understanding and generation. *CoRR* abs/2102.04664 (2021).

[33] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. CodeHow: effective code search based on API understanding and extended boolean model. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 260–270. https://doi.org/10.1109/ASE.2015.42

[34] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. *DeepGauge: multi-granularity testing criteria for deep learning systems.* Association for Computing Machinery, New York, NY, USA, 120–131. https://doi-org.proxy.bnl.lu/10.1145/3238147.3238202

[35] Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Chen Fu, and Qing Xie. 2012. Exemplar: a source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering* 38, 5 (2012), 1069–1087. https://doi.org/10.1109/TSE.2011.84

[36] Collin Mcmillan, Denys Poshyvanyk, Mark Grechanik, Qing Xie, and Chen Fu. 2013. Portfolio: searching for relevant functions and their usages in millions of lines of code. *ACM Transactions on Software Engineering and Methodology* 22, 4, Article 37 (October 2013). https://doi.org/10.1145/2522920.2522930

[37] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 1–18. https://doi.org/10.1145/3132747.3132785

[38] XiPeng Qiu, TianXiang Sun, YiGe Xu, YunFan Shao, Ning Dai, and XuanJing Huang. 2020. Pre-trained models for natural language processing: A survey. *Science China Technological Sciences* 63, 10 (September 2020), 1872–1897. https://doi.org/10.1007/s11431-020-1647-3

[39] Mukund Raghothaman, Yi Wei, and Youssef Hamadi. 2016. SWIM: synthesizing what I mean - code search and idiomatic snippet synthesis. In *IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 357–367. https://doi.org/10.1145/2884781.2884808

[40] Peter J Rousseeuw and Mia Hubert. 2011. Robust statistics for outlier detection. *Wiley interdisciplinary reviews: Data mining and knowledge discovery* 1, 1 (2011), 73–79.

[41] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. 2015. How developers search for code: a case study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 191–201. https://doi.org/10.1145/2786805.2786855

[42] Tobias Scheffer, Christian Decomain, and Stefan Wrobel. 2001. Active hidden Markov models for information extraction. In *Advances in Intelligent Data Analysis*. Springer Berlin Heidelberg, Berlin, Heidelberg, 309–318.
[43] SciPy. 2023. SciPy: open-source Python library. https://scipy.org/. Online; accessed 23 August 2023.
[44] Ozan Sener and Silvio Savarese. 2018. Active learning for convolutional neural networks: a core-set approach. In *International Conference on Learning Representations* (Vancouver, Canada).
[45] Burr Settles. 2010. *Active learning literature survey.* Technical Report 1648. University of Wisconsin, Madison.
[46] Kanish Shah, Henil Patel, Devanshi Sanghvi, and Manan Shah. 2020. A comparative analysis of logistic regression, random forest and KNN models for the text classification. *Augmented Human Research* 5, 1 (2020), 1–16.
[47] Tushar Sharma, Maria Kechagia, Stefanos Georgiou, Rohit Tiwari, and Federica Sarro. 2021. A survey on machine learning techniques for source code analysis. https://doi.org/10.48550/ARXIV.2110.09610
[48] Weijun Shen, Yanhui Li, Lin Chen, Yuanlei Han, Yuming Zhou, and Baowen Xu. 2020. Multiple-boundary clustering and prioritization to promote neural network retraining. In *IEEE/ACM International Conference on Automated Software Engineering* (Melbourne, VIC, Australia). Association for Computing Machinery, New York, United States, 410–422.
[49] Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. 2020. Improving code search with co-attentive representation learning. In *Proceedings of the 28th International Conference on Program Comprehension* (Seoul, Republic of Korea) *(ICPC '20)*. Association for Computing Machinery, New York, NY, USA, 196–207. https://doi.org/10.1145/3387904.3389269
[50] Bunyamin Sisman and Avinash C. Kak. 2013. Assisting code search with automatic query reformulation for bug localization. In *10th Working Conference on Mining Software Repositories (MSR)*. 309–318. https://doi.org/10.1109/MSR.2013.6624044
[51] Pinky Sitikhu, Kritish Pahi, Pujan Thapa, and Subarna Shakya. 2019. A comparison of semantic similarity methods for maximum human interpretability. In *Artificial Intelligence for Transforming Business and Society (AITB)*. IEEE. https://doi.org/10.1109/aitb48515.2019.8947433
[52] StackOverflow. 2008. StackOverflow. https://stackoverflow.com/. Online; accessed 23th August 2023.
[53] Zhensu Sun, Li Li, Yan Liu, Xiaoning Du, and Li Li. 2022. On the importance of building high-quality training datasets for neural code search. https://doi.org/10.48550/ARXIV.2202.06649
[54] GJG Upton. 1987. An introduction to mathematical statistics and its applications , by RJ Larsen and ML Marx. Pp 630.£ 17· 95. 1987. ISBN 13-487166-9 (Prentice-Hall). *The Mathematical Gazette* 71, 458 (1987), 330–330.
[55] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. *DeepHunter: a coverage-guided fuzz testing framework for deep neural networks.* Association for Computing Machinery, New York, NY, USA, 146–157. https://doi-org.proxy.bnl.lu/10.1145/3293882.3330579
[56] R Baeza Yates and B Ribeiro Neto. 2011. Modern information retrieval: the concepts and technology behind search. *Addison-Wesley Professional* (2011).
[57] Chen Zeng, Yue Yu, Shanshan Li, Xin Xia, Zhiming Wang, Mingyang Geng, Linxiao Bai, Wei Dong, and Xiangke Liao. 2022. DeGraphCS: embedding variable-based flow graph for neural code search. *ACM Transactions on Software Engineering and Methodology* (July 2022). https://doi.org/10.1145/3546066
[58] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. 2019. Machine learning testing: survey, landscapes and horizons. *CoRR* abs/1906.10742 (2019). arXiv:1906.10742 http://arxiv.org/abs/1906.10742
[59] Shichao Zhang, Xuelong Li, Ming Zong, Xiaofeng Zhu, and Debo Cheng. 2017. Learning k for knn classification. *ACM Transactions on Intelligent Systems and Technology (TIST)* 8, 3 (2017), 1–19.
[60] Yu Zhang, Peter Tiňo, Aleš Leonardis, and Ke Tang. 2021. A survey on neural network interpretability. *IEEE Transactions on Emerging Topics in Computational Intelligence* 5, 5 (2021), 726–742. https://doi.org/10.1109/TETCI.2021.3100641

8 Appendex: Beyond pre-trained models

In addition to pre-trained models, we evaluate KAPE on other deep learning models that do not benefit from a large code corpus in advance and are trained from scratch using the training data.

DeepCS [15]. Gu *et al.* proposed the CODEnn (Code-Description Embedding Neural Network) model and integrated it into the DeepCS tool. CODEnn embeds code snippets into code vectors via a code embedding network (CoNN) and embeds queries into description vectors via a description embedding network (DeNN). Specifically, CoNN takes the method name, API invocation sequence, and tokens contained in the source code as features to embed code.

UNIF [5]. Cambronero *et al.* built the UNIF (Embedding Unification) model that simply uses a bag-of-words-based network. In particular, given a bag of code embedding vectors, UNIF designs an attention-based weighing scheme to calculate the weight of each vector.

CARLCS-CNN[49]. Unlike DeepCS and UNIF which learn individual embeddings for code snippets and queries, respectively, CARLCS-CNN (co-attentive representation learning code search-CNN) learns interdependent representations with a co-attention mechanism. Similar to DeepCS, CARLCS-CNN also considers the method name, API invocation sequence, and tokens as features of code.

Tok-Att[57]. Introduced by Chen *et al.*, Tok-Att only exploits the token feature of code to generate code embeddings.

GraphSearchNet[29]. Proposed by Liu *et al.*, GraphSearchNet consists of a program encoder and a summary that learn the vector representations of code and queries, respectively. Concretely, GraphSearchNet constructs graphs for code snippets and queries to capture the structural information. In the corresponding encoder (e.g., program encoder), the graphs (e.g., graphs of code) are fed into a bidirectional gated graph neural network (BiGGNN) with a multi-head attention layer that learns the node and word embeddings.

Dataset and implementation For DeepCS, UNIF, and CARLCS-CNN, since they share similar data features, we use two datasets (Example and GitHub with Java code files) provided by DeepCS to perform the evaluation under the Tensorflow 2.0.0 framework. For GraphSearchNet, we use its provided dataset named Python. Each model has trained 500 epochs and the best model with respect to the validation set is saved. Table 8 lists the details of the provided datasets[1] and model performance.

Table 8. Summary of three datasets provided by DeepCS and the MRR on the validation and test sets by different models.

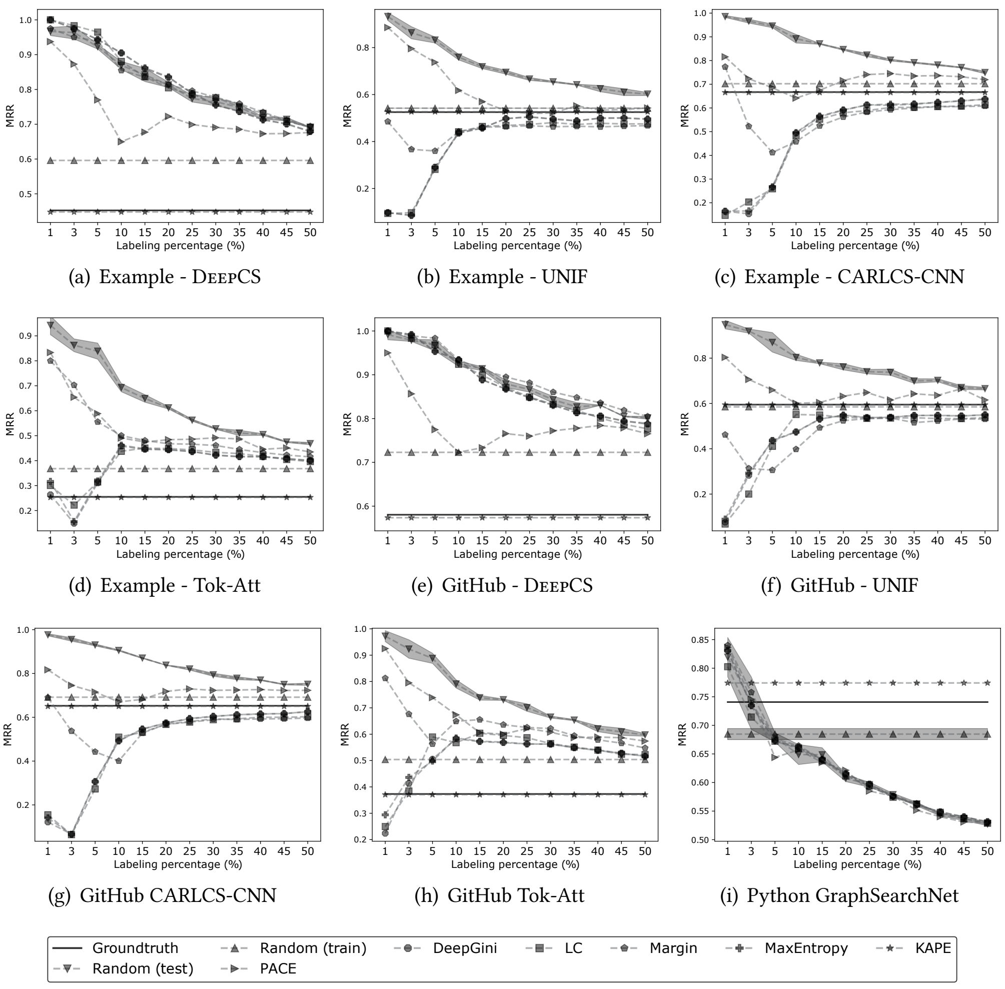
Dataset	#Training/Validation/Test	MRR of DeepCS		MRR of UNIF		MRR of CARLCS-CNN		MRR of Tok-Att	
		Validation	Test	Validation	Test	Validation	Test	Validation	Test
Example	10,000/5,000/5,000	0.4649	0.4524	0.5472	0.5251	0.6788	0.6670	0.2564	0.2549
GitHub	10,000/5,000/5,000	0.5939	0.5804	0.5830	0.5949	0.6650	0.6526	0.3725	0.3731

Dataset	#Training/Validation/Test	MRR of GraphSearchNet	
		Validation	Test
Python	283,318/15,486/15,290	0.7060	0.7408

Figure 8 shows the comparison results. In general, KAPE outperforms baseline methods regardless of datasets and the labeling budget.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009

[1]All datasets by DeepCS are provided with the pre-processed token, method name, API sequence, and description files. No pre-processing implementation or source code is available, thus reproduction on other datasets is inapplicable.



(a) Example - DeepCS (b) Example - UNIF (c) Example - CARLCS-CNN

(d) Example - Tok-Att (e) GitHub - DeepCS (f) GitHub - UNIF

(g) GitHub CARLCS-CNN (h) GitHub Tok-Att (i) Python GraphSearchNet

Fig. 8. Effectiveness comparison between KAPE and baseline methods given different models. (a)-(d): result on the Example dataset. (e)-(h): result on the GitHub dataset. (i): result on the Python dataset. **Groundtruth:** the actual MRR of the model on the test set. Shaded area illustrates the standard deviation of three experiment repetitions.