Accelerating Code Search with Deep Hashing and Code Classification

Wenchao Gu[1]*, **Yanlin Wang**[2,†], **Lun Du**[2], **Hongyu Zhang**[3],
Shi Han[2], **Dongmei Zhang**[2], **and Michael R. Lyu**[1]
[1] Department of Computer Science and Engineering,
The Chinese University of Hong Kong, China.
[2] Microsoft Research Asia, Beijing, China
[3] The University of Newcastle, Australia

Abstract

Code search is to search reusable code snippets from source code corpus based on natural languages queries. Deep learning-based methods on code search have shown promising results. However, previous methods focus on retrieval accuracy, but lacked attention to the efficiency of the retrieval process. We propose a novel method CoSHC to accelerate code search with deep hashing and code classification, aiming to perform efficient code search without sacrificing too much accuracy. To evaluate the effectiveness of CoSHC, we apply our method on five code search models. Extensive experimental results indicate that compared with previous code search baselines, CoSHC can save more than 90% of retrieval time meanwhile preserving at least 99% of retrieval accuracy.

1 Introduction

Code reuse is a common practice during software development process. It improves programming productivity as developers' time and energy can be saved by reusing existing code. According to previous studies (Brandt et al., 2009; Lv et al., 2015), many developers tend to use natural language to describe the functionality of desired code snippets and search the Internet/code corpus for code reuse.

Many code search approaches (Brandt et al., 2009; McMillan et al., 2011; Lv et al., 2015; Du et al., 2021) have been proposed over the years. With the rapid growth of open source code bases and the development of deep learning technology, recently deep learning based approaches have become popular for tackling the code search problem (Gu et al., 2018; Husain et al., 2019; Gu et al., 2021). Some of these approaches adopt neural network models to encode source code and query descriptions into representation vectors in the same embedding space. The distance between the representation vectors whose original code or description are semantically similar should be small. Other approaches (Feng et al., 2020; Guo et al., 2021; Du et al., 2021) regard the code search task as a binary classification task, and calculate the probability of code matching the query.

In the past, deep learning-based methods focused on retrieval accuracy, but lacked attention to the efficiency of retrieval on large-scale code corpus. However, both types of these deep learning-based approaches directly rank all the source code snippets in the corpus during searching, which will incur a large amount of computational cost. For the approaches that separately encode code and description representation vectors, the similarity of the target query vector with all code representation vectors in the corpus needs to be calculated for every single retrieval. In order to pursue high retrieval accuracy, a high dimension is often set for the representation vectors. For example, in CodeBERT, the dimension of the final representation vector is 768. The similarity calculation between a pair of code and query vectors will take 768 multiplications and 768 additions between two variables with double data type. The total calculation of single linear scan for the whole code corpus containing around 1 million code snippets is extremely large - around 1 billion times of multiplications and additions. As for the approaches adopting binary classification, there is no representation vectors stored in advance and the inference of the target token sequence with all the description token sequences needs to be done in real time for every single retrieval. Due to the large number of parameters in the current deep learning models, the computation cost will be significant.

Hashing is a promising approach to improve the retrieval efficiency and widely adopted in other retrieval tasks such as image-text search and image-image search. Hashing techniques can convert high

* Work done while this author was an intern at Microsoft Research. Wenchao Gu (wcgu@cse.cuhk.edu.hk).
† Yanlin Wang is the corresponding author (yanlwang@microsoft.com).

Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics
Volume 1: Long Papers, pages 2534 - 2544

dimensional vectors into low dimensional binary hash code, which greatly reduce the cost of storage and calculation (Luo et al., 2020). Hamming distance between two binary hash code can also be calculated in a very efficient way by running XOR instruction on the modern computer architectures (Wang et al., 2016). However, the performance degradation is still not avoidable during the conversion from representation vectors to binary hash codes even the state-of-the-art hashing models are adopted. The tolerance of performance degradation from most users is quite low and many of them are willing to sweep the performance with efficiency. In order to preserve the performance of the original code search models that adopt bi-encoders for the code-query encoding as much as possible, we integrate deep hashing techniques with code classification, which could mitigate the performance degradation of hashing model in the recall stage by filtering out the irrelevant data.

Specifically, in this paper, we propose a novel approach **CoSHC** (Accelerating Semantic Code Search with Deep Hashing and Code Classification) for accelerating the retrieval efficiency of deep learning-based code search approaches. CoSHC firstly clusters the representation vectors into different categories. It then generates binary hash codes for both source code and queries according to the representation vectors from the original models. Finally, CoSHC gives the normalized prediction probability of each category for the given query, and then CoSHC will decide the number of code candidates for the given query in each category according to the probability. Comprehensive experiments have been conducted to validate the performance of the proposed approach. The evaluation results show that CoSHC can preserve more than 99% performance of most baseline models. We summarize the main contributions of this paper as follows:

- We propose a novel approach, CoSHC, to improve the retrieval efficiency of previous deep learning based approaches. CoSHC is the first approach that adopts the recall and re-rank mechanism with the integration of code clustering and deep hashing to improve the retrieval efficiency of deep learning based code search models.
- We conduct comprehensive experimental evaluation on public benchmarks. The results demonstrate that CoSHC can greatly improve the retrieval efficiency meanwhile preserve almost the same performance as the baseline models.

2 Background

2.1 Code Search

In this subsection, we briefly review some deep learning based code search approaches. Sachdev et al. (2018) firstly propose the neural network based model NCS to retrieve the source code from a large source code corpus according to the given natural language descriptions. Cambronero et al. (2019) propose a neural network model UNIF based on bag-of-words, which embeds code snippets and natural language descriptions into a shared embedding space. Gu et al. (2018) propose to encode source code representation with API sequences, method name tokens and code tokens. Yao et al. (2019) treat code annotation and code search as dual tasks and utilize the generated code annotations to improve code search performance. Husain et al. (2019) explore different neural architectures for source code representation and discover that the self-attention model achieves the best performance. Gu et al. (2021) extract the program dependency graph from the source code and adopt long short term memory (LSTM) networks to model this relationship. Feng et al. (2020) propose a pre-trained model for source code representation and demonstrate its effectiveness on the code search task.

2.2 Deep Hashing

In this subsection, we briefly introduce some representative unsupervised cross-modal hashing methods. In order to learn a unified hash code, Ding et al. (2014) propose to adopt collective matrix factorization with latent factor model from different modalities to merge multiple view information sources. Zhou et al. (2014) firstly utilize sparse coding and matrix factorization to extract the latent features for images and texts, respectively. Then the learned latent semantic features are mapped to a shared space and quantized to the binary hash codes. Wang et al. (2014) suggest using stacked auto-encoders to capture the intra- and inter-modal semantic relationships of data from heterogeneous sources. He et al. (2017) and Zhang et al. (2018) adopt adversarial learning for cross-modal hash codes generation. Wu et al. (2018) propose an approach named UDCMH that integrates deep learning and matrix factorization with binary latent factor models to generate binary hash codes for multi-

modal data retrieval. By incorporating Laplacian constraints into the objective function, UDCMH preserve not only the nearest neighbors but also the farthest neighbors of data. Unlike using Laplacian constraints in the loss function, Su et al. (2019) construct a joint-semantic affinity matrix that integrates the original neighborhood information from different modalities to guide the learning of unified binary hash codes.

3 Method

We propose a general framework to accelerate existing Deep Code Search (DCS) models by decoupling the search procedure into a recall stage and a re-rank stage. Our main technical contribution lies in the recall stage. Figure 1 illustrates the overall framework of the proposed approach. CoSHC consists of two components, i.e., Offline and Online. In Offline, we take the code and description embeddings learned in the given DCS model as input, and learn the corresponding hash codes by preserving the relations between the code and description embeddings. In Online, we recall a candidate set of code snippets according to the Hamming distance between the query and code, and then we use the original DCS model to re-rank the candidates.

3.1 Offline Stage

Multiple Code Hashing Design with Code Classification Module Since the capacity of binary hashing space is very limited compared to Euclidean space, the Hamming distance between similar code snippets will be too small to be distinguishable if we adopt a single Hashing model. To be specific, we cluster the codebase using K-Means algorithm with the code embeddings learned from the given DCS model. The source code whose representation vectors are close to each other will be classified into the same category after the clustering.

Deep Hashing Module The deep hashing module aims at generating the corresponding binary hash codes for the embeddings of code and description from the original DCS model. Figure 2 illustrates the framework of the deep hashing module. To be specific, three fully-connected (FC) layers with $\tanh(\cdot)$ activation function are adopted to replace the output layer in the original DCS model to convert the original representation vectors into a soft binary hash code.

The objective of the deep hashing module is to force the Hamming distance between hashing representations of code pairs and description pairs approaching the Euclidean distance between the corresponding embeddings. Thus, we need to calculate the ground truth similarity matrix between code pairs and description pairs firstly. For performance consideration, we calculate the similarity matrix within a mini-batch.

To construct such a matrix, we first define the code representation vectors and the description representation vectors in the original code search model as $V_C = \{v_c^{(1)}, ..., v_c^{(n)}\}$ and $V_D = \{v_d^{(1)}, ..., v_d^{(n)}\}$, respectively. V_C and V_D represent the representation vectors matrix for the entire batch, while $v_c^{(i)}$ and $v_d^{(i)}$ represent the representation vector for the single code snippet or query. After normalizing V_C, V_D to $\hat{V}_C$, $\hat{V}_D$ with l_2-norm, we can calculate the code similarity matrices $S_C = \hat{V}_C \hat{V}_C^T$ and summary similarity matrices $S_D = \hat{V}_D \hat{V}_D^T$ to describe the similarity among code representation vectors and summary representation vectors, respectively. In order to integrate the similarity information in both S_C and S_D, we combine them with a weighted sum:

$$\tilde{S} = \beta S_C + (1 - \beta) S_D, \beta \in [0, 1] \quad (1)$$

where β is the weight parameter. Since the pairwise similarity among the code representation vectors and description representation vectors still cannot comprehensively present the distribution condition of them in the whole embedding space, we involve a matrix $\tilde{S}\tilde{S}^T$ to describe a high order neighborhood similarity that two vectors with high similarity should also have the close similarity to other vectors. Finally, we utilize a weighted equation to combine both of these two matrices as follows:

$$S = (1 - \eta)\tilde{S} + \eta \frac{\tilde{S}\tilde{S}^T}{m}, \quad (2)$$

where η is a hyper-parameter and m is the batch size which is utilized to normalize the second term in the equation. Since we hope the binary hash codes of the source code and its corresponding description to be the same, we replace the diagonal elements in the similarity matrix with one. The final high order similarity matrix is:

$$S_{F_{ij}} = \begin{cases} 1, & i = j \\ S_{ij}, & \text{otherwise} \end{cases} \quad (3)$$

Binary Hash Code Training We propose to replace the output layer of the original code search

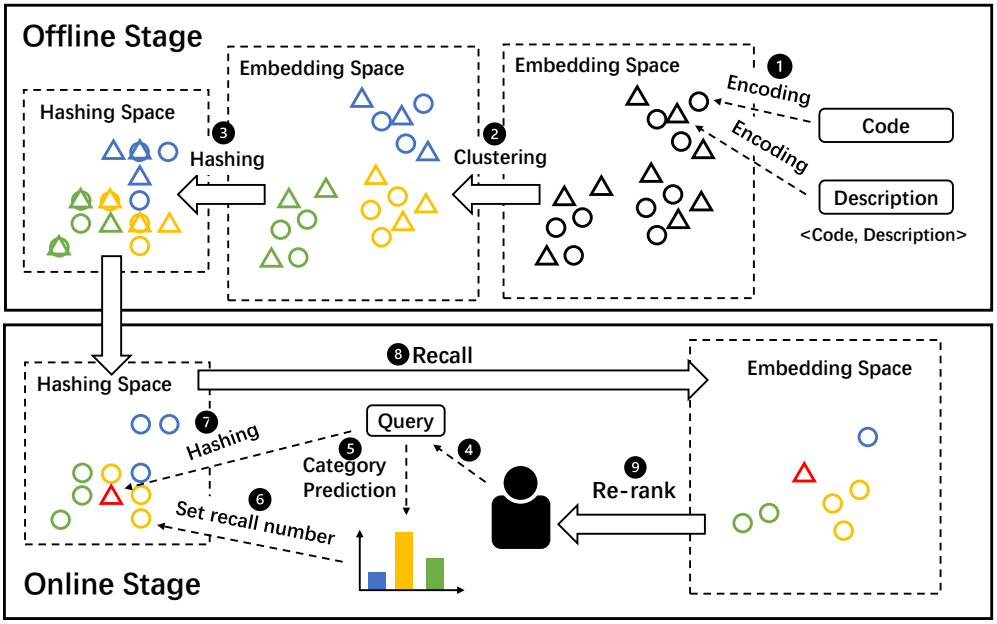

Figure 1: Overview of the proposed CoSHC. ❶ Encoding the code token sequence and description token sequence via original code retrieval models. ❷ Clustering the code representation vectors into several categories. ❸ Converting the original code representation vectors into binary hash codes.❺❻ Predicting the category of the query given by users and set the number of code candidates for different categories. ❼ Converting the input query into binary hash code. ❽ Recall the code candidates according to the hamming distance and the number of code candidates for each category. ❾ Re-ranking all the code candidates according to the cosine similarity between the input query description vectors and code candidates' representation vectors and return the results to the user.

model with three FC layers with $\tanh(\cdot)$ activate function. We define the trained binary hash code for code and description as $B_C = \{b_c^{(1)}, ..., b_c^{(n)}\}$ and $B_D = \{b_d^{(1)}, ..., b_d^{(n)}\}$, respectively. To ensure that the relative distribution of binary hash codes is similar to the distribution of representation vectors in the original embedding space, the following equation is utilized as the loss function of the deep hashing module:

$$\begin{aligned}\mathscr{L}(\theta) = \min_{B_C, B_D} & \| \min(\mu S_F, 1) - \frac{B_C B_D^T}{d} \|_F^2 \\ & + \lambda_1 \| \min(\mu S_F, 1) - \frac{B_C B_C^T}{d} \|_F^2 \\ & + \lambda_2 \| \min(\mu S_F, 1) - \frac{B_D B_D^T}{d} \|_F^2, \\ s.t.\ & B_C, B_D \in \{-1, +1\}^{m \times d},\end{aligned} \tag{4}$$

where θ are model parameters, μ is the weighted parameters to adjust the similarity score between different pairs of code and description, λ_1, λ_2 are the trade-off parameters to weight different terms in the loss function, and d is the dimension of the binary hash code generated by this deep hashing module. These three terms in the loss function are adopted to restrict the similarity among binary hash codes of the source codes, the similarity among binary hash codes of the descriptions, and the similarity between the binary hash codes of source code and description, respectively.

Note that we adopt $B_C B_D^T/d$ to replace $\cos(B_C, B_D)$ because $\cos(B_C, B_D)$ only measures the angle between two vectors but neglects the length of the vectors, which makes $\cos(B_C, B_D)$ can still be a very large value even the value of every hash bits is close to zero. Unlike $\cos(B_C, B_D)$, $B_C B_D^T/d$ can only achieve a high value when every bit of the binary hash code is 1 or -1 since the value of $B_C B_D^T/d$ will be close to zero if the value of every hash bits is close to zero.

Since it is impractical to impose on the output of neural network to be discrete values like 1 and -1, we adopt the following equation to convert the output of deep hashing module to be strict binary hash code:

$$B = \mathrm{sgn}(H) \in \{-1, +1\}^{m \times d}, \tag{5}$$

where H is the output of the last hidden layer without the activation function in the deep hashing module and $\mathrm{sgn}(\cdot)$ is the sign function and the output of this function is 1 if the input is positive and the output is -1 otherwise.

However, the gradient of the sign function will be zero in backward propagation which will induce

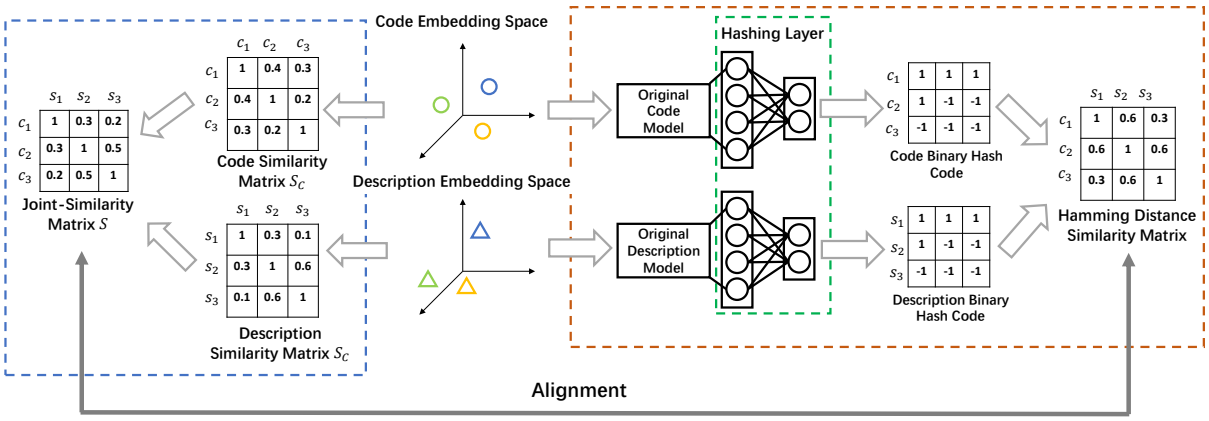

Figure 2: Architecture of the hashing module. The original representation vectors will be utilized for the joint-similarity matrix construction at first. Then the joint-similarity matrix will be utilized as the labels for training binary hash codes generation. The training objective is to make the Hamming distance similarity matrix to be identical as the joint-similarity matrix.

the vanishing gradients problem and affect model convergence. To address this problem, we follow the previous research (Cao et al., 2017; Hu et al., 2019) and adopt a scaling function:

$$B = \tanh(\alpha H) \in \{-1, +1\}^{m \times d}, \quad (6)$$

where α is the parameter which is increased during the training. The function of $\tanh(\alpha H)$ is an approximate equation of $\text{sgn}(H)$ when α is large enough. Therefore, the output of Eq. 6 will finally be converged to 1 or -1 with the increasing of α during the training and the above problem is addressed.

3.2 Online Stage

Recall and Re-rank Mechanism The incoming query from users will be fed into the description category prediction module to calculate the normalized probability distribution of categories at first. Then the number of code candidates R_i for each category i will be determined according to this probability distribution. The Hamming distance between the hash code of the given query and all the code inside the database will be calculated. Then code candidates will be sorted by Hamming distance in ascending order and the top R_i code candidates in each category i will be recalled. In the re-rank step, the original representation vectors of these recalled code candidates will be retrieved and utilized for the cosine similarity calculation. Finally, code snippets will be returned to users in descending order of cosine similarity.

Description Category Prediction Module The description category prediction module aims to predict the category of source code that meets user's requirement according to the given natural language description. The model adopted for category prediction is the same as the original code search model, except that the output layer is replaced with a one-hot category prediction layer and the cross-entropy function is adopted as the loss function of the model.

Since the accuracy of the description category prediction module is not perfect, we use the probability distribution of each category instead of the category with the highest predicted probability as the recall strategy for code search. We define the total recall number of source code as N, the normalized predicted probability for each code category as $P = \{p_1, ..., p_k\}$, where k is the number of categories. The recall number of source code in each category is:

$$R_i = \min(\lfloor p_i \cdot (N - k) \rfloor, 1), \ i \in 1, ..., k, \quad (7)$$

where R_i is the recall number of source code in category i. To ensure that the proposed approach can recall at least one source code from each category, we set the minimum recall number for a single category to 1.

4 Experiments

4.1 Dataset

We use two datasets (Python and Java) provided by CodeBERT (Feng et al., 2020) to evaluate the performance of CoSHC. CodeBERT selects the data from the CodeSearchNet (Husain et al., 2019)

dataset and creates both positive and negative examples of <description, code> pairs. Since all the baselines in our experiments are bi-encoder models, we do not need to predict the relevance score for the mismatched pairs so we remove all the negative examples from the dataset. Finally we get 412,178 <description, code> pairs as the training set, 23,107 <description, code> pairs as the validation set, and 22,176 <description, code> pairs as the test set in the Python dataset. We get 454,451 <description, code> pairs as the training set, 15,328 <description, code> pairs as the validation set, and 26,909 <description, code> pairs as the test set in the Java dataset.

4.2 Experimental Setup

In the code classification module, we set the number of cluster to 10. In the deep hashing module, we add three fully connected (FC) layer in all the baselines, the hidden size of each FC layer is the same as the dimension of the original representation vectors. Specifically, the hidden size of FC layer for CodeBERTa, CodeBERT, GraphCodeBERT is 768. The hidden size of FC layer for UNIF is 512 and for RNN is 2048. The size of the output binary hash code for all the baselines is 128. The hyper parameters $\beta, \eta, \mu, \lambda_1, \lambda_2$ are 0.6, 0.4, 1.5, 0.1, 0.1, respectively. The parameter α is the epoch number and will be linear increased during the training. In the query category prediction module, a cross-entropy function is adopted as the loss function and the total recall number is 100.

The learning rate for CodeBERTa, CodeBERT and GraphCodeBERT is 1e-5 and the learning rate for UNIF, RNN is 1.34e-4. All the models are trained via the AdamW algorithm (Kingma and Ba, 2015).

We train our models on a server with four 4x Tesla V100 w/NVLink and 32GB memory. Each module based on CodeBERT, GraphCodeBERT and CodeBERTa are trained with 10 epochs and Each module based on RNN and UNIF are trained with 50 epochs. The early stopping strategy is adopted to avoid overfitting for all the baselines. The time efficiency experiment is conducted on the server with Intel Xeon E5-2698v4 2.2Ghz 20-core. The programming for evaluation is written in C++ and the program is allowed to use single thread of CPU.

4.3 Baselines

We apply CoSHC on several state-of-the-art and representative baseline models. UNIF (Cambronero et al., 2019) regards the code as the sequence of tokens and embeds the sequence of code tokens and description tokens into representation vectors via full connected layer with attention mechanism, respectively. RNN baseline adopts a two-layer bi-directional LSTM (Cho et al., 2014) to encode the input sequences. CodeBERTa [1] is a 6-layer, Transformer-based model trained on the CodeSearchNet dataset. CodeBERT (Feng et al., 2020) is a pre-trained model based on Transformer with 12 layers. Similar to CodeBERT, GraphCodeBERT (Guo et al., 2021) is a pre-trained Transformer-based model pre-trained with not only tokens information but also dataflow of the code snippets. As we introduced, the inference efficiency of cross-encoder based models like CodeBERT is quite low and the purpose of our approach is to improve the calculation efficiency between the representation vectors of code and queries. Here we slightly change the model structure of CodeBERTa, CodeBERT, and GraphCodeBERT. Rather than concatenating code and query together and inputting them into a single encoder to predict the relevance score of the pair, we adopt the bi-encoder architecture for the baselines, which utilize the independent encoder to encoding the code and queries into representation vectors, respectively. Also, cosine similarity between the given representation vector pairs is adopted as the training loss function to replace the cross entropy function of the output relevance score.

4.4 Evaluation Metric

$SuccessRate@k$ is widely used by many previous studies (Haldar et al., 2020; Shuai et al., 2020; Fang et al., 2021; Heyman and Cutsem, 2020). The metric is calculated as follows:

$$SuccessRate@k = \frac{1}{|Q|}\sum_{q=1}^{Q}\delta(FRank_q \leq k), \quad (8)$$

where Q denotes the query set and $FRank_q$ is the rank of the correct answer for query q. If the correct result is within the top k returning results, $\delta(FRank_q \leq k)$ returns 1, otherwise it returns 0. A higher $R@k$ indicates better performance.

[1] https://huggingface.co/huggingface/CodeBERTa-small-v1

	Python	Java
	Total Time	
CodeBERT	572.97s	247.78s
CoSHC	33.87s (↓94.09%)	15.78s (↓93.51%)
	(1) Vector Similarity Calculation	
CodeBERT	531.95s	234.08s
CoSHC	14.43s (↓97.29%)	7.25s (↓96.90%)
	(2) Array Sorting	
CodeBERT	41.02s	13.70s
CoSHC	19.44s (↓53.61%)	8.53s (↓37.74%)

Table 1: Time Efficiency of CoSHC.

4.5 Experimental Results

In this section, we present the experimental results and evaluate the performance of CoSHC from the aspects of retrieval efficiency, overall retrieval performance, and the effectiveness of the internal classification module.

4.5.1 RQ1: How much faster is CoSHC than the original code search models?

Table 1 illustrates the results of efficiency comparison between the original code search models and CoSHC. Once the representation vectors of code and description are stored in the memory, the retrieval efficiency mainly depends on the dimension of representation vectors rather than the complexity of the original retrieval model. Therefore, we select CodeBERT as the baseline model to illustrate efficiency comparison. Since code search process in both approaches contains vector similarity calculation and array sorting, we split the retrieval process into these two steps to calculate the time cost.

In the vector similarity calculation step, CoSHC reduces 97.29% and 96.90% of time cost in the dataset of Python and Java respectively, which demonstrates that the utilization of binary hash code can effectively reduce vector similarity calculation cost in the code retrieval process.

In the array sorting step, CoSHC reduces 53.61% and 37.74% of time cost in the dataset of Python and Java, respectively. The classification module makes the main contribution on the improvement of sorting efficiency. The sorting algorithm applied in both original code search model and CoSHC is quick sort, whose time complexity is $O(n\log n)$. Classification module divides a large code dataset into several small code datasets, reducing the average time complexity of sorting to $O(n\log\frac{n}{m})$. The reason why the improvement of sorting in the Java dataset is not so significant as in the Python dataset is that the size of Java dataset is much smaller than the size of Python dataset. However, the combination of the algorithm of divide and conquer and max-heap, rather than quick sort, is widely applied in the big data sorting, which can greatly shrink the retrieval efficiency gap between these two approaches. Therefore, the improvement of efficiency in the sorting process will not be as large as what shown in Table 1.

In the overall code retrieval process, the cost time is reduced by 94.09% and 93.51% in the dataset of Python and Java, respectively. Since the vector similarity calculation takes most of cost time in the code retrieval process, CoSHC still can reduce at least 90% of cost time, which demonstrates the effectiveness on the efficiency improvement in the code search task.

4.5.2 RQ2: How does CoSHC affect the accuracy of the original models?

Table 2 illustrates the retrieval performance comparison between the original code search models and CoSHC. We have noticed that the performance of the conventional approaches like BM25 (Robertson and Zaragoza, 2009) is not good enough. For example, we set the token length for both code and queries as 50, which is the same as the setting in CodeBERT, and apply BM25 to recall top 100 code candidates for the re-rank step on the Python dataset. BM25 can only retain 99.3%, 95.6% and 92.4% retrieval accuracy of CodeBERT in terms of $R@1$, $R@5$ and $R@10$ on the Python dataset. Here we only compare the performance of our approach with the original code search models since the purpose of our approach is to preserve the performance of the original code search models. As can be observed, CoSHC can retain at least 99.5%, 99.0% and 98.4% retrieval accuracy of most original code search models in terms of $R@1$, $R@5$ and $R@10$ on the Python dataset. CoSHC can also retain 99.2%, 98.2% and 97.7% of the retrieval accuracy as all original code search baselines in terms of $R@1$, $R@5$ and $R@10$ on the Java dataset, respectively. We can find that CoSHC can retain more than 97.7% of performance in all metrics. $R@1$ is the most important and useful metric among these metrics since most users hope that the first returned answer is the correct answer during the search. CoSHC can retain at least 99.2% of performance on $R@1$ in both datasets, which demonstrates that CoSHC can retain almost the same performance as the original code search model.

Model	Python			Java		
	R@1	R@5	R@10	R@1	R@5	R@10
UNIF	0.071	0.173	0.236	0.084	0.193	0.254
$\text{CoSHC}_{\text{UNIF}}$	**0.072 (↑1.4%)**	**0.177 (↑2.3%)**	**0.241 (↑2.1%)**	**0.086 (↑2.4%)**	**0.198 (↑2.6%)**	**0.264 (↑3.9%)**
−w/o classification	0.071 (0.0%)	0.174 (↑0.6%)	0.236 (0.0%)	0.085 (↑1.2%)	0.193 (0.0%)	0.254 (0.0%)
−one classification	0.069 (↓2.8%)	0.163 (↓5.8%)	0.216 (↓8.5%)	0.083 (↓1.2%)	0.183 (↓5.2%)	0.236 (↓7.1%)
−ideal classification	0.077 (↑6.9%)	0.202 (↑16.8%)	0.277 (↑17.4%)	0.093 (↑10.7%)	0.222 (↑15.0%)	0.296 (↑16.5%)
RNN	0.111	0.253	0.333	0.073	0.184	0.250
$\text{CoSHC}_{\text{RNN}}$	**0.112 (↑0.9%)**	**0.259 (↑2.4%)**	**0.343 (↑5.0%)**	**0.076 (↑4.1%)**	**0.194 (↑5.4%)**	**0.265 (↑6.0%)**
−w/o classification	**0.112 (↑0.9%)**	0.254 (↑0.4%)	0.335 (↑0.6%)	0.073 (0.0%)	0.186 (↑1.1%)	0.253 (↑1.2%)
−one classification	**0.112 (↑0.9%)**	0.243 (↓4.0%)	0.311 (↓6.6%)	0.075 (↑2.7%)	0.182 (↓1.1%)	0.240 (↓4.0%)
−ideal classification	0.123 (↑10.8%)	0.289 (↑14.2%)	0.385 (↑15.6%)	0.084 (↑15.1%)	0.221 (↑20.1%)	0.302 (↑20.8%)
CodeBERTa	0.124	0.250	0.314	0.089	0.203	0.264
$\text{CoSHC}_{\text{CodeBERTa}}$	**0.123 (↓0.8%)**	**0.247 (↓1.2%)**	**0.309 (↓1.6%)**	**0.090 (↑1.1%)**	**0.210 (↑3.4%)**	**0.272 ((↑3.0%)**
−w/o classification	0.122 (↓1.6%)	0.242 (↓3.2%)	0.302 (↓3.8%)	0.089 (0.0%)	0.201 (↓1.0%)	0.258 (↓2.3%)
−one classification	0.116 (↓6.5%)	0.221 (↓11.6%)	0.271 (↓13.7%)	0.085 (↓4.5%)	0.189 (↓6.9%)	0.238 (↓9.8%)
−ideal classification	0.135 (↑8.9%)	0.276 (↑10.4%)	0.346 (↑10.2%)	0.100 (↑12.4%)	0.235 (↑15.8%)	0.305 (↑15.5%)
CodeBERT	0.451	0.683	0.759	0.319	0.537	0.608
$\text{CoSHC}_{\text{CodeBERT}}$	**0.451 (0.0%)**	**0.679 (↓0.6%)**	**0.750 (↓1.2%)**	**0.318 (↓0.3%)**	**0.533 (↓0.7%)**	**0.602 (↓1.0%)**
−w/o classification	0.449 (↓0.4%)	0.673 (↓1.5%)	0.742 (↓2.2%)	0.316 (↓0.9%)	0.527 (↓1.9%)	0.593 (↓2.5%)
−one classification	0.425 (↓5.8%)	0.613 (↓10.2%)	0.665 (↓12.4%)	0.304 (↓4.7%)	0.483 (↓10.1%)	0.532 (↓12.5%)
−ideal classification	0.460 (↑2.0%)	0.703 (↑2.9%)	0.775 (↑2.1%)	0.329 (↑3.1%)	0.555 (↑3.4%)	0.627 (↑3.1%)
GraphCodeBERT	0.485	0.726	0.792	0.353	0.571	0.640
$\text{CoSHC}_{\text{GraphCodeBERT}}$	**0.483 (↓0.4%)**	**0.719 (↓1.0%)**	**0.782 (↓1.3%)**	**0.350 (↓0.8%)**	**0.561 (↓1.8%)**	**0.625 (↓2.3%)**
−w/o classification	0.481 (↓0.8%)	0.713 (↓1.8%)	0.774 (↓2.3%)	0.347 (↓1.7%)	0.553 (↓3.2%)	0.616 (↓3.7%)
−one classification	0.459 (↓5.4%)	0.653 (↓10.1%)	0.698 (↓11.9%)	0.329 (↓7.8%)	0.505 (↓11.6%)	0.551 (↓13.9%)
−ideal classification	0.494 (↑1.9%)	0.741 (↑2.1%)	0.803 (↑1.4%)	0.361 (↑2.3%)	0.585 (↑2.5%)	0.649 (↑1.4%)

Table 2: Results of code search performance comparison. The best results among the three CoSHC variants are highlighted in **bold** font.

It is interesting that CoSHC presents a relatively better performance when the performance of the original code retrieval models is worse. $\text{CoSHC}_{\text{CodeBERTa}}$ even outperforms the original baseline model in Java dataset. $\text{CoSHC}_{\text{RNN}}$ and $\text{CoSHC}_{\text{UNIF}}$ outperform the original model in both Python and Java datasets. The integration of deep learning and code classification with recall make the contribution on this result. The worse performance indicates more misalignment between the code representation vectors and description representation vectors. Since the code classification and deep hashing will filter out most of irrelevant codes in the recall stage, some irrelevant code representation vectors but has high cosine similarity with the target description representation vectors are filtered, which leads the improvement on the final retrieval performance.

4.5.3 RQ3: Can the classification module help improve performance?

Table 2 illustrates the performance comparison between the CoSHC variants which adopt different recall strategies with query category prediction results. $\text{CoSHC}_{\text{w/o classification}}$ represents CoSHC without code classification and description prediction module. $\text{CoSHC}_{\text{one classification}}$ represents the CoSHC variant that recalls $N - k + 1$ candidates in the code category with highest prediction probability and one in each of the rest categories. $\text{CoSHC}_{\text{ideal classification}}$ is an ideal classification situation we set. Assuming the correct description category is known, $N - k + 1$ candidates are recalled in the correct category and one candidate is recalled in each of the rest categories. Note that the display of $\text{CoSHC}_{\text{ideal classification}}$ is only to explore the upper threshold of performance improvement of the category prediction module and will not be counted as a variant of CoSHC we compare.

Model	Python Acc.	Java Acc.
$\text{CoSHC}_{\text{UNIF}}$	0.558	0.545
$\text{CoSHC}_{\text{RNN}}$	0.610	0.535
$\text{CoSHC}_{\text{CodeBERTa}}$	0.591	0.571
$\text{CoSHC}_{\text{CodeBERT}}$	0.694	0.657
$\text{CoSHC}_{\text{GraphCodeBERT}}$	0.713	0.653

Table 3: Classification accuracy of the code classification module in each model.

By comparing the performance be-

tween $\text{CoSHC}_{\text{ideal classification}}$ and $\text{CoSHC}_{\text{w/o classification}}$, we can find that correct classification can significantly improve the retrieval performance. With the ideal category labels, CoSHC can even outperform all baseline models. As mentioned in Sec. 4.5.2, code classification can mitigate the problem of vector pairs misalignment via filtering out wrong candidates whose representation vectors has high cosine similarity with the target representation vectors in the recall stage. The more serious the misalignment problem, the more effective the code classification. That is the reason why the improvement of CoSHC with ground-truth labels on UNIF, RNN, and CodeBERTa is more significant than the improvement of it on CodeBERT and GraphCodeBERT since the retrieval accuracy of former models is much lower than the latter ones. Similar conclusions can also be drawn at the aspect of binary hash code distribution via the comparison between CoSHC and $\text{CoSHC}_{\text{ideal classification}}$ since CoSHC utilizes the distribution of the original representation vectors as the guidance for model training. Therefore, the distribution of binary hash codes will be similar to the distribution of original representation vectors.

Since we have explored the theoretical upper limit of the effectiveness of code classification for code retrieval, the effectiveness of code classification for code retrieval in the real application will be validated. By comparing the experimental results between $\text{CoSHC}_{\text{w/o classification}}$ and $\text{CoSHC}_{\text{one classification}}$, we can find that the performance of CoSHC with predicted labels is even worse than the performance of CoSHC without code classification module. The reason is that the accuracy of description category prediction is far from the satisfactory. Table 3 illustrates the accuracy of description category prediction module in all baseline models. We regard the category with the highest probability as the predicted category from the description category prediction module and check whether the module could give a correct prediction. It can be seen that the classification accuracy is not very high (less than 75%). By observing the experimental results of CoSHC in GraphCodeBERT on the Java dataset, we can also find that low accuracy greatly affect the performance of $\text{CoSHC}_{\text{oneclassification}}$, which makes 7.8%, 11.6%, and 13.9% performance drop in terms of $R@1$, $R@5$, and $R@10$, respectively.

Fortunately, although the description category prediction module cannot accurately tell the exact category which this description belongs to, the module still can give a relative high predicted probability on the correct category. By comparing the experimental results among all the variants of CoSHC, we can find the performance is increased significantly once the recall strategy is replaced to that the number of code candidates for each category is determined by the normalized predication probability. CoSHC with new recall strategy almost achieve the best performance in all metrics on all baseline models. Even on RNN in the Python dataset, CoSHC still achieve the same performance as CoSHC without classification under $R@1$ and achieve similar performance in other metrics. Above experimental results have demonstrated the effectiveness of the adoption of code classification in code search.

5 Conclusion

To accelerate code search, we present CoSHC, a general method that incorporates deep hashing techniques and code classification. We leverage the two-staged recall and re-rank paradigm in information retrieval field and apply deep hashing techniques for fast recall. Furthermore, we propose to utilize a code classification module to retrieve better quality code snippets. Experiments on five code search models show that compared with the original code search models, CoSHC can greatly improve the retrieval efficiency meanwhile preserve almost the same performance.

6 Acknowledgement

Wenchao Gu's and Michael R. Lyu's work described in this paper was in part supported by the Research Grants Council of the Hong Kong Special Administrative Region, China (CUHK 14210920 of the General Research Fund).

References

Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems, CHI 2009, Boston, MA, USA, April 4-9, 2009*, pages 1589–1598. ACM.

José Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search. In *Proceedings of the ACM*

Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019, pages 964–974. ACM.

Zhangjie Cao, Mingsheng Long, Jianmin Wang, and Philip S. Yu. 2017. Hashnet: Deep learning to hash by continuation. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, pages 5609–5618. IEEE Computer Society.

Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the properties of neural machine translation: Encoder-decoder approaches. In *Proceedings of SSST@EMNLP 2014, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation, Doha, Qatar, 25 October 2014*, pages 103–111. Association for Computational Linguistics.

Guiguang Ding, Yuchen Guo, and Jile Zhou. 2014. Collective matrix factorization hashing for multimodal data. In *2014 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2014, Columbus, OH, USA, June 23-28, 2014*, pages 2083–2090. IEEE Computer Society.

Lun Du, Xiaozhou Shi, Yanlin Wang, Ensheng Shi, Shi Han, and Dongmei Zhang. 2021. Is a single model enough? mucos: A multi-model ensemble learning approach for semantic code search. In *CIKM '21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021*, pages 2994–2998. ACM.

Sen Fang, Youshuai Tan, Tao Zhang, and Yepang Liu. 2021. Self-attention networks for code search. *Inf. Softw. Technol.*, 134:106542.

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. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics.

Wenchao Gu, Zongjie Li, Cuiyun Gao, Chaozheng Wang, Hongyu Zhang, Zenglin Xu, and Michael R. Lyu. 2021. Cradle: Deep code retrieval based on semantic dependency learning. *Neural Networks*, 141:385–394.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 933–944. ACM.

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 B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

Rajarshi Haldar, Lingfei Wu, Jinjun Xiong, and Julia Hockenmaier. 2020. A multi-perspective architecture for semantic code search. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 8563–8568. Association for Computational Linguistics.

Li He, Xing Xu, Huimin Lu, Yang Yang, Fumin Shen, and Heng Tao Shen. 2017. Unsupervised cross-modal retrieval through adversarial learning. In *2017 IEEE International Conference on Multimedia and Expo, ICME 2017, Hong Kong, China, July 10-14, 2017*, pages 1153–1158. IEEE Computer Society.

Geert Heyman and Tom Van Cutsem. 2020. Neural code search revisited: Enhancing code snippet retrieval through natural language intent. *CoRR*, abs/2008.12193.

Di Hu, Feiping Nie, and Xuelong Li. 2019. Deep binary reconstruction for cross-modal hashing. *IEEE Trans. Multim.*, 21(4):973–985.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *CoRR*, abs/1909.09436.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Xiao Luo, Chong Chen, Huasong Zhong, Hao Zhang, Minghua Deng, Jianqiang Huang, and Xiansheng Hua. 2020. A survey on deep hashing methods. *CoRR*, abs/2003.03369.

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 (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 260–270. IEEE Computer Society.

Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of*

the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011, pages 111–120. ACM.

Stephen E. Robertson and Hugo Zaragoza. 2009. The probabilistic relevance framework: BM25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389.

Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. 2018. Retrieval on source code: a neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pages 31–41. ACM.

Jianhang Shuai, Ling Xu, Chao Liu, Meng Yan, Xin Xia, and Yan Lei. 2020. Improving code search with co-attentive representation learning. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*, pages 196–207. ACM.

Shupeng Su, Zhisheng Zhong, and Chao Zhang. 2019. Deep joint-semantics reconstructing hashing for large-scale unsupervised cross-modal retrieval. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*, pages 3027–3035. IEEE.

Jun Wang, Wei Liu, Sanjiv Kumar, and Shih-Fu Chang. 2016. Learning to hash for indexing big data - A survey. *Proc. IEEE*, 104(1):34–57.

Wei Wang, Beng Chin Ooi, Xiaoyan Yang, Dongxiang Zhang, and Yueting Zhuang. 2014. Effective multi-modal retrieval based on stacked auto-encoders. *Proc. VLDB Endow.*, 7(8):649–660.

Gengshen Wu, Zijia Lin, Jungong Han, Li Liu, Guiguang Ding, Baochang Zhang, and Jialie Shen. 2018. Unsupervised deep hashing via binary latent factor models for large-scale cross-modal retrieval. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 2854–2860. ijcai.org.

Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. 2019. Coacor: Code annotation for code retrieval with reinforcement learning. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, pages 2203–2214. ACM.

Jian Zhang, Yuxin Peng, and Mingkuan Yuan. 2018. Unsupervised generative adversarial cross-modal hashing. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 539–546. AAAI Press.

Jile Zhou, Guiguang Ding, and Yuchen Guo. 2014. Latent semantic sparse hashing for cross-modal similarity search. In *The 37th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '14, Gold Coast , QLD, Australia - July 06 - 11, 2014*, pages 415–424. ACM.