SiamBERT: Siamese Bert-based Code Search

Francisco J. Peña[†], Angel Luis Gonzalez[†], Sepideh Pashami[‡], Ahmad Al-Shishtawy[‡], Amir H. Payberah[†‡]
[†]KTH Royal Institute of Technology, Stockholm, Sweden
[‡] RISE Research Institutes of Sweden, Stockholm, Sweden
[†]{frape,algon,payberah}@kth.se [‡]{sepideh.pashami,ahmad.al-shishtawy}@ri.se

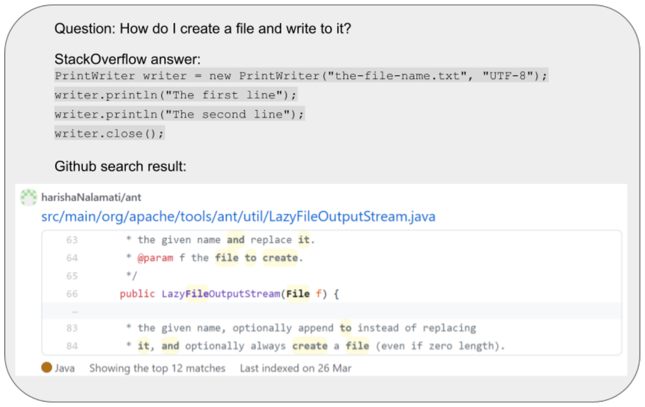
***Abstract*— Code Search is a practical tool that helps developers navigate growing source code repositories by connecting natural language queries with code snippets. Platforms such as StackOverflow resolve coding questions and answers; however, they cannot perform a semantic search through the code. Moreover, poorly documented code adds more complexity to search for code snippets in repositories. To tackle this challenge, this paper presents SiamBERT, a BERT-based model that gets the question in natural language and returns relevant code snippets. The SiamBERT architecture consists of two stages, where the first stage, inspired by Siamese Neural Network, returns the top K relevant code snippets to the input questions, and the second stage ranks the given snippets by the first stage. The experiments show that SiamBERT outperforms non-BERT-based models having improvements that range from 12% to 39% on the Recall@1 metric and improves the inference time performance, making it 15x faster than standard BERT models.**

I. Introduction

In recent years, there has been an explosion in the amount of source code available on platforms like GitHub or GitLab. Code availability benefits software developers as they can read and execute source code from other projects. Moreover, software developers can benefit from open source projects by reusing parts of the code and including them in their projects. However, finding which part of the project contains proper code is a complex problem.

Consider a scenario where Alice, a software developer, wants to know how to create a file and write to it. She goes to Q&A websites like StackOverflow and writes *"How do I create a file and write to it?"*. If she is lucky, she will find an answer based on similar questions that other users have asked in the past. An answer to her query is possible if other users have posted their queries using the same wording. Figure 1 (on its top part) illustrates this, where we can see the top result for our example query. However, if she writes the same question on a code hosting website like GitHub, she would struggle to obtain an answer to her query because natural language does not directly match the source code. This problem is aggravated if the source code lacks documentation. On the bottom part of Figure 1, we can see that even though the top result matches the words of our query, it does not help the developer to implement the code.

Models like Bidirectional Encoder Representations from Transformers (BERT) [10] can be successfully applied to solve the code search problem. Several BERT-base models [28, 12, 7] attempt to solve this problem by producing code embeddings. These models usually create pairs of



Question: How do I create a file and write to it?

StackOverflow answer:

```
PrintWriter writer = new PrintWriter("the-file-name.txt", "UTF-8");
writer.println("The first line");
writer.println("The second line");
writer.close();
```

Github search result:

harishaNalamati/ant
src/main/org/apache/tools/ant/util/LazyFileOutputStream.java

```
     * the given name and replace it.
     * @param f the file to create.
     */
    public LazyFileOutputStream(File f) {
...
     * the given name, optionally append to instead of replacing
     * it, and optionally always create a file (even if zero length).
```

Java   Showing the top 12 matches   Last indexed on 26 Mar

Fig. 1: Example of a question on StackOverflow. The code of its accepted answer and the result of searching for that sentence on GitHub. The latter references code unrelated to the answer but contains some of the words in the query.

queries and code snippets in the dataset to calculate their similarity. Nevertheless, this process is slow and can become performance bottlenecks for large datasets. To overcome this problem, this paper presents SiamBERT, which uses BERT to embed developers' queries and code snippets. Developers, then, can use SiamBERT to search for code snippets by expressing the question in natural language. To this end, SiamBERT follows two stages: (i) first, it uses a code embedding model, inspired by Siamese Neural Network, trained on a triple-loss function [29] to select the best K candidates, and (ii) then it uses the code similarity model to re-rank the K candidates according to the query.

We evaluate the SiamBERT's prediction using Top-N metrics. Our results show that SiamBERT outperforms other state-of-the-art code embedding models in ranking prediction. We also evaluate our model in terms of time execution, showing that it outperforms code similarity models by being 15x faster without compromising the prediction power. Moreover, we demonstrate the robustness of our model by rephrasing queries and showing that it still holds as the top performer.

II. Background

A. Embedding representation

Word embedding is a vectorization representation approach that maps words into vectors such that similar words get close vectors in the new space. For example, the word `connect` can be represented by the vector $\langle 0.2, 3, -1.5\rangle$, the word `join` can be represented by the vector $\langle 0.3, 3, -1.0\rangle$, and the word `method` can be represented by the vector $\langle 6, -1.2, 8\rangle$. In this illustrative example, we can see that the vectors for the words `connect` and `join` are close to each other in the space (i.e., $\sqrt{(0.2-0.3)^2+(3-3)^2+(-1.5+1.0)^2} \sim 0.5$) because their words hold similar meanings, while vectors of words with different meanings, such as `method` will be further apart (i.e., $\sqrt{(0.2-6)^2+(3+1.2)^2+(-1.5-8)^2} \sim 11.89$).

Similar to words, sentences can also be encoded into vectorial representations. For instance, the code presented in the top part of Figure 1 can be encoded into the vector $\langle 2.7, 0.1, -0.8\rangle$. Traditionally, sentences are encoded into vectors by aggregating the word vectors together [24]; however, this has the drawback that the order of the words in the sentence is lost, resulting in inaccurate representations. Recurrent Neural Networks (RNNs), such as Long Short-Term Memory (LSTM) [15] are one of the first code search models that use sentence embedding, such as [12]. They can generate vector representations for sentences in which the information about the order of the words is preserved. However, RNNs present two major drawbacks: they cannot capture relations between words in long sentences, and training them is slow since words have to be fed into them one at a time, which prevents them from benefiting from parallel computing [30].

B. BERT and Sentence-BERT

In recent years, transformers have shown superior performance in Natural Language Processing (NLP) tasks [32]. They can capture relations between words in long sentences, word embeddings hold information about each word's context, and they can be trained in parallel, as they do not require each word to be input as a sequence. BERT is a language model based on the transformer architecture that has achieved state-of-the-art performance in multiple language modeling tasks [10]. BERT is pre-trained on a large corpus of text on two tasks: Masked Language Modeling (MML), where 15% of the words in a sentence are masked and BERT has to predict them from the context, and Next Sentence Prediction (NSP), where two sentences are given and BERT has two predict if the second sentence follows the first one. After pre-training, BERT can be fined-tuned to other tasks on smaller datasets.

The success of BERT and its good results applied in various fields of NLP is sometimes overshadowed when used in text extraction tasks, mainly when applied to large corpora. The main problem is that BERT's base architecture requires pairing an input query with each document in the corpus, generating performance problems [21]. However, many variants of BERT have emerged to improve its performance issue. For example, Sentence-BERT (SBERT) [26] is a model that addresses the latency problems by generating a single embedding vector for an entire sentence using max or mean pooling to embed all the words in the sentence. Unlike BERT, which needs to receive two sentences for tasks such as sentence similarity, SBERT uses a Siamese Neural Network (SNN) architecture [6] to receive each sentence individually for the parallel BERT module. This architectural difference results in a significant improvement in performance at the inference stage.

Suppose we have a group of sentences, and for each new sentence, we want to obtain the most similar one in our collection. There are two ways to do this with BERT. The first way is to input each sentence separately to BERT and obtain a sentence embedding from the `[CLS]` token or by mean pooling. Reimers and Gurevych [26] have shown that the embeddings obtained this way do not help calculate the semantic similarity of sentences. The second way is to input the two sentences simultaneously (details on how to do this are explained in Section IV-A.1.a). This implies calculating similarity values between the new sentence and each of the sentences in the dataset every time a new sentence is provided. With SBERT, we can precompute the embeddings of all the sentences in the dataset, and upon getting a new sentence, we only need to calculate the embedding of the new sentence and calculate its similarity with the rest of the embeddings in the dataset. Although the improvement in inference time is remarkable when using code embedding models for code similarity models, there is a significant loss in prediction quality [21].

C. UNIF

Embedding Unification (UNIF) [7] is a supervised extension of the Neural Code Search (NCS) [28] that uses two embedded matrices and an attention mechanism. Both code and description are embedded with their matrices, and then the cosine similarity of both embeddings is calculated. The embedding and attention parameters are tuned during the training process to maximize the similarity between a code snippet and its description. The advantages of this proposed model are that it outperforms previous state-of-the-art approaches by using a simpler model than other deep learning models, such as Deep Code Search (DeepCS) [12].

III. Methodology

The general idea of SiamBERT is as follows: after getting the input query, first, SiamBERT transforms it into an embedding vector and compares it with the pre-calculated embedding vectors of existing code. After selecting K similar existing codes, it applies a code similarity algorithm to sort them based on similarity to the query.

Figure 2 presents the multi-stage architecture of our SiamBERT. In the pre-selection stage, a code embedding model obtains K representative candidates, taking advantage of the efficiency of this architecture to search for candidates in large code repositories. The embedded coded snippets are stored in a database. Thus, it is only necessary to embed the new query and calculate the cosine similarity between the query vector and the stored vectors at the inference time.

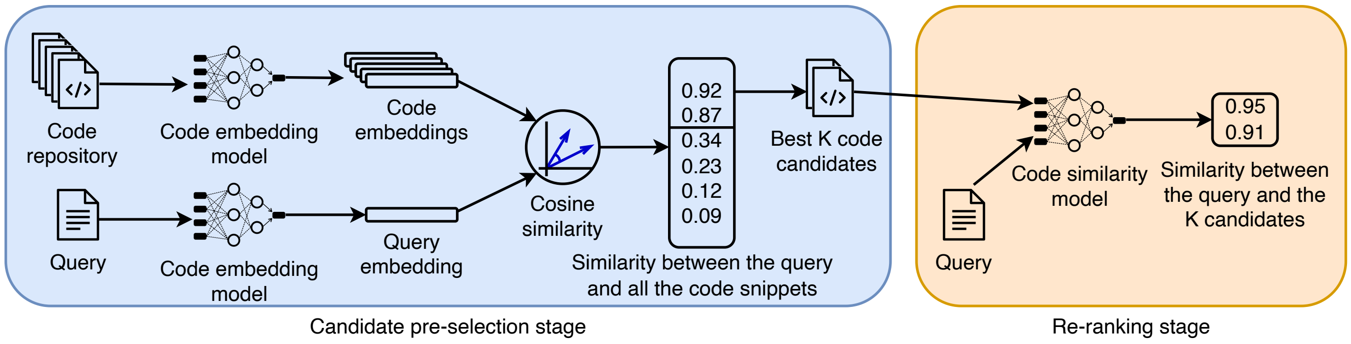


Fig. 2: Multi-stage architecture.

The code similarity model sorts the K candidates to obtain a final ranking in the re-ranking stage. Note that the code similarity model does not produce embeddings but directly calculates the similarity between a query and a code snippet.

We can use various models in the pre-selection stage. One of the models we implement is UNIF-SNN, which is inspired by UNIF [7]. This model embeds both description and code tokens to the same vector space. The motivation behind this idea is that using the same vector space could allow the model to learn the semantics of words used in code snippets and descriptions or queries. To this end, we feed both the code and description to an SNN consisting of an embedding layer and an attention layer. Experiments in Section IV show that using an SNN improves the prediction performance of UNIF.

We also extend UNIF-SNN by using a triplet loss function, as presented in [26], to train the model. Inspired by [26], we fine-tune UNIF-SNN using the triplet loss function to learn source code methods embedding and description embeddings. Our goal is to make the embedding such that the distance between a description embedding and its corresponding code embedding is small, while the distance between a description and a non-matching code is large. To form a triplet, we choose a query as an *anchor* and two code snippets as *positive* and *negative*. The positive code snippet contains the functionality described in the query, while the negative one has different functionality. Given the anchor, positive, and negative embeddings (denoted by a, p, and n, respectively), we define the triplet loss $\mathcal{L}(a,p,n)$ as:

$$\mathcal{L}(a,p,n) = max(d(a,p) - d(a,n) + \alpha, 0), \qquad (1)$$

where $d(x,y)$ measures the distance between x and y, and α is the margin that ensures the positive code snippet is closer to the anchor than the negative code snippet. We can consider different distance metrics for d, such as the Euclidean and Cosine distances.

To construct a triplet, we first choose a description randomly from the dataset as an anchor and then select its corresponding code snippet as positive and a random code snippet as negative for the anchor accordingly. As Figure 3 shows, the triplet loss function consists of three instances of the same code embedding model, sharing the weights. However, in practice, we use only one single instance of the code embedding model with three different input channels to adhere to the triplet structure. First, to train the model, we tokenize each method in the triplet (anchor, positive, and negative) and give them as input to the code embedding model to compute the features. We then use a pooling layer to aggregate the set of features that the code embedding model computes for each token to obtain a single fixed-length vector representing the given method. Finally, we use the embedding of the anchor, positive, and negative methods (i.e., a, p, and n, respectively) to compute the triplet loss $\mathcal{L}(a,p,n)$.

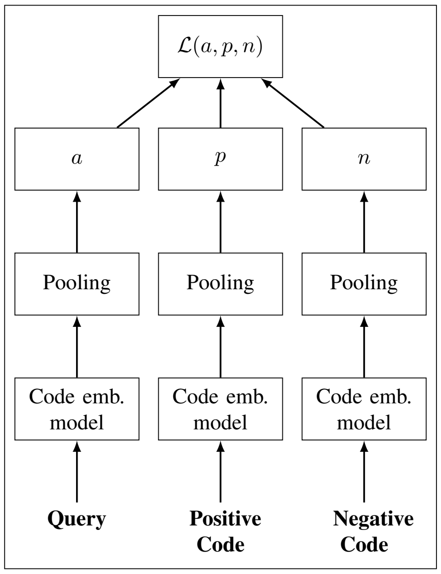


Fig. 3: Triplet loss function.

IV. Evaluation

This section will evaluate our model in different tasks and compare its performance against state-of-the-art baselines.

A. Experimental setup

We conduct all the experiments on a single machine running Ubuntu 18.04. The machine has four Nvidia RTX 2070 GPUs, a 12-core AMD Ryzen Threadripper 2920X CPU, 128 GBs of RAM, and 500 GBs of SSD storage. We implement all models using TensorFlow [1] and Keras [8], which are packaged and deployed using Docker [23]. For the loading and configuration of BERT-based models, we used the HuggingFace transformers framework [33].

To evaluate our approach, we use a subset of the dataset proposed in [12] containing 600k + 500 (training + test sets) Java methods organized in tuples and containing four pieces of information ⟨method name, API sequence, tokens, description ⟩. We use the dataset from Java projects hosted on GitHub, which have documented methods and were created between August 2008 and June 2016. The method body has also been tokenized, as detailed in [12]. We discard the method names and API sequences as we are only interested in matching code snippets with descriptions.

1) Baselines: We fine-tune all the baselines and SIAMBERT to predict if a given description matches a code snippet.

a) BERT: For BERT we give the input of the description and the code in the following way:

`[CLS] <CODE> [SEP] <DESC.> [SEP] [PAD]`

where `[CLS]` is a token that BERT expects to receive at the beginning of each pair of sentences, `[SEP]` is a token used to separate sentences, and it is placed at the end of each sentence, and `[PAD]` is a padding token (see [10]).

b) RoBERTa: Robustly Optimized BERT Pretraining Approach (RoBERTa) is a language model that has the same architecture as BERT, but it is pre-trained with more data, dynamic masking, and without the NSP task [22]. We feed RoBERTa the data using the same input format as with BERT.

c) SBERT and SRoBERTa:: SBERT and Sentence-RoBERTa (SRoBERTa) are modifications of BERT and RoBERTa that use SNN to learn embeddings to capture the semantic similarity of sentences [26]. Equally to [26], we perform experiments using the SNN along with BERT and RoBERTa.

d) UNIF: The code embedding model is described in Section II-C.

2) Metrics: To measure the performance of the code search models, we use the Recall@N metric [9], which is widely used in information retrieval and recommender systems [14, 31]. Recall@N calculates the number of hits or true positives among the entire test set as follows:

$$Recall@N = \frac{\#hits}{|T|} \tag{2}$$

where $|T|$ is the number of elements in the test set.

3) Evaluation procedure: To evaluate the models, we iterate over the test set, generate an embedding for each description, calculate the similarity with $|T|$ code snippets embeddings and extract the ranking of the correct snippet. Based on this ranking, we calculate the Recall@N. In all our experiments, we used $|T| = 500$. We train all the models with learning rates between $4 - e5$ and $1e - 6$. The experiments show that larger learning rates like $1e - 4$ negatively impact BERT models. We also evaluated different triplet margin values in the range $[0.2 - 0.6]$ and saved the best results.

B. Results

Here, we present the results.

1) Ranking prediction: In the first set of experiments, we compare the ranking prediction of SIAMBERT and the rest of the baselines. Table I shows the results of our experiments. Our goal is to reach the same prediction performance as BERT in a fraction of the time, so in the Rec@N columns, we show the absolute value for the Recall@N accompanied by the percentage of BERT's performance that other models were able to reach. We can see that our model UNIF-SNN is the best code embedding model for code search. It can consistently outperform SBERT, SRoBERTa, and UNIF.

Our experiments confirm that the code similarity models outperform code embedding models in code search tasks in other text retrieval tasks [21]. However, the high inference time in these models makes them unfeasible to use in production environments. For this reason, we combine code embedding models with code similarity models to reduce latency and minimize the impact on the output quality. We achieve the best performance when combining code similarity with our proposed code embedding model UNIF-SNN.

Table I shows all three SIAMBERT consistently outperform the rest of the baselines, with SIAMBERT-UNIF-SNN having the best performance overall. The four SIAMBERT models are approximately 15x faster than BERT. The improvement produced by using code embedding models and then using code similarity models (in this case, BERT) to re-rank the candidates shows that the re-ranking done by BERT consistently improves the order of the candidates. In other words, code similarity models are good at selecting the best candidates. The fact that we can achieve a prediction performance almost as good as BERT (98% in the worst case) shows that code embedding models do an excellent job at discarding bad candidates. We also have improvements between 12% and 39% on Recall@1 compared to non-BERT models. Our multi-stage architecture exploits this fact to make predictions virtually as good as the BERT predictions 15-times faster.

2) Ranking prediction with rephrasing: To validate the robustness of our models, we modified 41 descriptions from the test set and ran the experiments to compare the performance of these sentences against their original versions. Table III shows that BERT outperforms all the code embedding models, and all the multi-stage models outperform the code embedding models. The results obtained by BERT are maintained or even improved by the multi-stage models.

In Table II, we present some rephrasing examples that we give to the SIAMBERT-SRoBERTa model, which has the best rephrasing performance in this experiment. The rank column shows the number of code snippets with a better rank than the ground truth. For example, if the rank is 0, the model returns the correct snippet at the first position for that query. If the rank is 4, then four code snippets got a better rank than the correct code, which was fifth.

For instance, examples 83 and 329 got a better new rank (moves from second to the first position) with the rephrase

Model	Rec@1	Rec@3	Rec@5	Time	Speed
BERT	0.63	0.83	0.86	17m 30s	– –
SBERT	0.49 (78%)	0.71 (86%)	0.77 (90%)	2m 2s	9x
SRoBERTa	0.46 (73%)	0.66 (80%)	0.74 (86%)	42s	25x
UNIF	0.47 (75%)	0.66 (80%)	0.74 (86%)	**36s**	**29x**
UNIF-SNN	0.57 (90%)	0.73 (88%)	0.80 (93%)	1m 07s	16x
SIAMBERT-SBERT	**0.64 (102%)**	**0.81 (98%)**	0.83 (97%)	1m 14s	14x
SIAMBERT-SRoBERTa	0.62 (98%)	0.79 (95%)	0.82 (95%)	1m 13s	14x
SIAMBERT-UNIF	**0.64 (102%)**	0.79 (95%)	0.83 (97%)	1m 9s	15x
SIAMBERT-UNIF-SNN	**0.64 (102%)**	**0.81 (98%)**	**0.84 (98%)**	1m 10s	15x

TABLE I: Recall@N performance of all the models executed with 500 code-description pairs from the test set.

query, which contains the original sentence's semantic meaning but fewer words. Example 5 got a worse ranking even when both original and rephrased sentences are very similar. In example 153, the ranking has worsened significantly with the new sentence, an example of a poorly detailed query. The word "java" probably made this search more difficult, as it is pretty generic and may not be a common word in the training set method description.

V. Related work

Code clustering is a helpful unsupervised task that gives an overview of source code repositories in a fast and inexpensive way, given that no labeled data is necessary. Hägglund et al. [13] present a work similar to ours because they also use the triplet loss function based on an anchor, positive, and negative samples to learn the source code embeddings. Other clustering methods include [27] and [19].

Code summarization is the task of creating descriptive summaries from source code snippets to help developers understand the functionality of the source code. Alon et al. [3] present the code2seq model, which encodes each Abstract Syntax Tree (AST) path with its values as a vector, and uses the average of all paths as the decoder's start state. The decoder generates an output sequence while attending over the encoded paths. Attention is used to select the relevant paths while decoding. code2seq directly uses paths in the AST for end-to-end generation of sequences. Liang and Zhu [20] introduce Code-RNN, which encodes different source code segments into vectors. An RNN then uses these vectors to generate code summaries. A known problem with RNNs is that long-term dependencies are often not captured. Feng et al. [11] attempt to overcome the long-dependency problem with their model called Fret. Fret is composed of two encoders and a decoder. The first encoder, which acts as a reinforcer, captures the code's functionality based on method calls. The second encoder is a pre-trained BERT that takes the code tokens as an input and the functionality embeddings from the first encoder. The outputs of both embeddings are given to a decoder that generates the code summaries.

Code clone detection is the task of finding duplicate code among a corpora of code. Detecting code clones has multiple benefits, such as reducing the code maintenance or avoiding having the cloned copies in the train and test set of a machine learning dataset. Allamanis [2], and Kang, Bissyandé, and Lo [18] generate code embeddings to detect code clones beyond using an exact text match.

Method name prediction involves predicting the name of a method or function to give a developer essential information about the method's functionality. Alon et al. [4] introduce code2vec, a method to predict method names that decomposes the AST into a collection of paths and learns the representation for each path. Alon et al. claim these representations can be used for multiple tasks, including code retrieval, captioning, classification, and tagging or clone detection. Other models to predict method names include [25].

Code search, as we described in Section I is the task of matching natural language descriptions with their corresponding code snippets. We describe three code search models published recently. Gu, Zhang, and Kim [12] present the DeepCS model, which learns code embeddings by fusing the resulting vectors of three neural networks: two RNNs and a multi-layer perceptron that process method names, API sequences, and code tokens, respectively. Description embeddings are learned using an RNN, and a triplet loss function based on cosine similarity is used to bring the code and description vectors closer in the latent space. Sachdev et al. [28] introduce the NCS model, which transforms code and description tokens into embeddings using the fastText library [5]. An average of the token embeddings is used to create the description embedding. The code embedding is created by using a TF-IDF weighted average in which the weights are learned to maximize the cosine similarity between the code and description embeddings. The UNIF model presented by Cambronero et al. in [7] is similar to the NCS model, but it differs in that the token embeddings are not taken from fastText; instead, they are learned by the model. For the description, the token embeddings are averaged while an attention mechanism is used for the code. Note that none of these models use transformer-based architecture and therefore suffer from the long-dependency problem and their produced embeddings lack contextual information.

Pre-trained models. Recently, variants of the BERT model have been designed to create context-rich code embeddings [11, 17, 16]. They mainly use the same encoding architecture as BERT but differ in the pre-training tasks and the data used to train them.

Id	**Original rank**	**New rank**	**Original**	**Rephrased**
83	0	0	return the canonical path of this file	how to get canonical path of the given file
329	1	0	generate a random int as a token	how to get a random integer
73	1	0	turn array of bytes into string representing each byte as unsigned hex number	how to transform list bytes into string hexadecimal representation
5	0	17	pause processing at the socket	how to stop socket processing
153	2	223	get an xml representation of this object	from java object to xml
144	0	1	set the maximum allowed number of threads	how to define max threads
15	0	16	do a form post method call	how to perform a http post request

TABLE II: Example of the ranking of descriptions and its paraphrased version with the SIAMBERT -SRoBERTa (SIAMBERT-SRoBERTa) model.

Model	**Original**			**Rephrased**		
	Rec@1	**Rec@3**	**Rec@5**	**Rec@1**	**Rec@3**	**Rec@5**
BERT	0.70	0.97	0.97	0.51	0.68	0.78
SRoBERTa	0.58	0.75	0.85	0.39	0.56	0.68
UNIF	0.53	0.85	0.90	0.29	0.53	0.63
UNIF-SNN	0.70	0.82	0.90	0.43	0.63	0.70
SIAMBERT-SRoBERTa	0.75	**0.95**	**0.95**	**0.53**	**0.73**	**0.75**
SIAMBERT-UNIF	**0.78**	0.92	**0.95**	**0.53**	0.70	0.73
SIAMBERT-UNIF-SNN	**0.78**	**0.95**	**0.95**	**0.53**	**0.73**	**0.75**

TABLE III: Recall@N performance with 41 selected queries and their rephrased versions, with 500 code snippets

VI. CONCLUSIONS

In this paper, we have exploited the speed and ability to discard bad candidates of code embedding models and combined them with the ability to select the best candidates of the code similarity models to produce a model 15-times faster and without sacrificing the performance of the code similarity models.

The results presented in Section IV suggest that code embedding models are good at discarding bad candidates but underperform in selecting the best candidates. On the other hand, code similarity models like BERT and RoBERTa have an excellent ability to discard bad candidates and select the best candidates at the expense of large response times. The experiments show that SIAMBERT outperforms non-BERT-based models having improvements that range from 12% to 39% on the Recall@1 metric. Our model SIAMBERT combines the advantages of both models to reduce the inference time without compromising prediction quality.

We introduced a new model called UNIF-SNN, which combines the simplicity of the UNIF with a Siamese neural network architecture. Our experiments show that it is the best code embedding model for the task of code search.

We also studied the robustness of the models, testing them with sentences from outside their test set. Although all models show a drop in performance with the new sentences, which could be mitigated by training them with more data, code similarity models maintain their performance with the multi-stage architecture.

REFERENCES

[1] Martín Abadi et al. “{TensorFlow}: A System for {Large-Scale} Machine Learning”. In: *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 2016, pp. 265–283.

[2] Miltiadis Allamanis. “The adverse effects of code duplication in machine learning models of code”. In: *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 2019, pp. 143–153.

[3] Uri Alon et al. “code2seq: Generating sequences from structured representations of code”. In: *arXiv preprint arXiv:1808.01400* (2018).

[4] Uri Alon et al. “code2vec: Learning distributed representations of code”. In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), pp. 1–29.

[5] Piotr Bojanowski et al. “Enriching word vectors with subword information”. In: *Transactions of the association for computational linguistics* 5 (2017), pp. 135–146.

[6] Jane Bromley et al. “Signature verification using a "siamese" time delay neural network”. In: *Advances in neural information processing systems* 6 (1993).

[7] Jose Cambronero et al. “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*. 2019, pp. 964–974.

[8] François Chollet et al. *Keras*. `https://keras.io`. 2015.

[9] Paolo Cremonesi, Yehuda Koren, and Roberto Turrin. "Performance of recommender algorithms on top-n recommendation tasks". In: *Proceedings of the fourth ACM conference on Recommender systems*. 2010, pp. 39–46.

[10] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[11] Zhangyin Feng et al. "Codebert: A pre-trained model for programming and natural languages". In: *arXiv preprint arXiv:2002.08155* (2020).

[12] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. "Deep code search". In: *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE. 2018, pp. 933–944.

[13] Marcus Hägglund et al. "COCLUBERT: Clustering Machine Learning Source Code". In: *2021 20th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE. 2021, pp. 151–158.

[14] Jonathan L Herlocker et al. "Evaluating collaborative filtering recommender systems". In: *ACM Transactions on Information Systems (TOIS)* 22.1 (2004), pp. 5–53.

[15] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: `10.1162/neco.1997.9.8.1735`. eprint: `https://direct.mit.edu/neco/article-pdf/9/8/1735/813796/neco.1997.9.8.1735.pdf`. URL: `https://doi.org/10.1162/neco.1997.9.8.1735`.

[16] Aditya Kanade et al. "Learning and evaluating contextual embedding of source code". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 5110–5121.

[17] Aditya Kanade et al. "Pre-trained contextual embedding of source code". In: (2019).

[18] Hong Jin Kang, Tegawendé F Bissyandé, and David Lo. "Assessing the generalizability of code2vec token embeddings". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2019, pp. 1–12.

[19] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. "Semantic clustering: Identifying topics in source code". In: *Information and software technology* 49.3 (2007), pp. 230–243.

[20] Yuding Liang and Kenny Zhu. "Automatic generation of text descriptive comments for code blocks". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 32. 1. 2018.

[21] Jimmy Lin, Rodrigo Nogueira, and Andrew Yates. "Pretrained transformers for text ranking: Bert and beyond". In: *Synthesis Lectures on Human Language Technologies* 14.4 (2021), pp. 1–325.

[22] Yinhan Liu et al. "Roberta: A robustly optimized bert pretraining approach". In: *arXiv preprint arXiv:1907.11692* (2019).

[23] Dirk Merkel et al. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux journal* 2014.239 (2014), p. 2.

[24] Tomas Mikolov et al. "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (2013).

[25] Veselin Raychev, Martin Vechev, and Andreas Krause. "Predicting program properties from" big code"". In: *ACM SIGPLAN Notices* 50.1 (2015), pp. 111–124.

[26] Nils Reimers and Iryna Gurevych. "Sentence-bert: Sentence embeddings using siamese bert-networks". In: *arXiv preprint arXiv:1908.10084* (2019).

[27] Dimitris Rousidis and Christos Tjortjis. "Clustering data retrieved from Java source code to support software maintenance: A case study". In: *Ninth European Conference on Software Maintenance and Reengineering*. IEEE. 2005, pp. 276–279.

[28] Saksham Sachdev et al. "Retrieval on source code: a neural code search". In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. 2018, pp. 31–41.

[29] Florian Schroff, Dmitry Kalenichenko, and James Philbin. "Facenet: A unified embedding for face recognition and clustering". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 815–823.

[30] Alex Sherstinsky. "Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network". In: *CoRR* abs/1808.03314 (2018). arXiv: `1808.03314`. URL: `http://arxiv.org/abs/1808.03314`.

[31] Daniel Valcarce et al. "On the robustness and discriminative power of information retrieval metrics for top-N recommendation". In: *Proceedings of the 12th ACM conference on recommender systems*. 2018, pp. 260–268.

[32] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[33] Thomas Wolf et al. "Transformers: State-of-the-art natural language processing". In: *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*. 2020, pp. 38–45.