A Multi-Perspective Architecture for Semantic Code Search

Rajarshi Haldar[†], **Lingfei Wu**[‡], **Jinjun Xiong**[‡], **Julia Hockenmaier**[†]
†University of Illinois at Urbana-Champaign, Champaign, IL, USA
‡IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA
{rhaldar2, juliahmr}@illinois.edu
{wuli, jinjun}@us.ibm.com

Abstract

The ability to match pieces of code to their corresponding natural language descriptions and vice versa is fundamental for natural language search interfaces to software repositories. In this paper, we propose a novel multi-perspective cross-lingual neural framework for code–text matching, inspired in part by a previous model for monolingual text-to-text matching, to capture both global and local similarities. Our experiments on the CoNaLa dataset show that our proposed model yields better performance on this cross-lingual text-to-code matching task than previous approaches that map code and text to a single joint embedding space.

1 Introduction

In semantic code search or retrieval, the user provides a natural language query, and the system returns a ranked list of relevant code snippets from a database or repository for that query. This task is usually performed using a matching model that computes the similarity between code snippets and natural language descriptions by mapping code and natural language embeddings into a common space where the distance between a piece of code and its corresponding description is small (Gu et al., 2018; Yao et al., 2019).

But current models do not explicitly model any interactions between the code and the description until the final step when their global similarity is calculated.

In this paper, we propose a novel multi-perspective neural framework for code–text matching that captures both global and local similarities. We show that it yields improved results on semantic code search.

We apply our model to the CoNaLa benchmark dataset (Yin et al., 2018), which consists of Python code snippets and their corresponding annotations in English. We believe that our model could be applied to other programming languages as well. We have made our code publicly available for research purpose [1].

2 Background

Semantic code search is a cross-modal ranking problem where items in one modality (code) need to be ranked according to how well they match queries in another (natural language). One standard way to compute the similarity of items drawn from two different modalities or languages is to map each modality into a common "semantic" vector space such that matching pairs are mapped to vectors that are close to each other.

Gu et al. (2018) propose a code retrieval framework that jointly embeds code snippets and NL descriptions into a high dimensional embedding space such that the vectors representing a code snippet and its corresponding description have high similarity.

A variety of different approaches for learning embeddings for code have been proposed. Because source code is less ambiguous than natural language, there are ways to exploit the underlying structure of code to obtain better representations. Wan et al. (2019); LeClair et al. (2020) show that using features extracted from Abstract Syntax Trees (AST's) and Control Flow Graphs (CFG's) lead to creating better representations of code. Hu et al. (2018); Haque et al. (2020) show that ASTs represented as compact strings can be used to represent code. Following these approaches, we developed a multi-modal framework that generates embeddings for code using both the code tokens and an AST representation.

[1] https://github.com/rajarshihaldar/codetextmatch

Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pages 8563–8568

3 Models

We compare four models: a baseline model (**CT**) that only considers text and source code, a (**CAT**) model that also includes embedding of Abstract Syntax Trees, a multi-perspective model (**MP**) that leverages multi-perspective matching operations as defined in a bilateral multi-perspective model (Wang et al., 2017), and our **MP-CAT** model that combines both **MP** and **CAT** architectures.

3.1 CT: A Baseline Code and Text Model

Our baseline model (**CT**) is based on Gu et al. (2018)'s CODEnn model. It maps both code and natural language descriptions to vectors in the same embedding space and then computes the similarity between these vectors using the L2 distance metric. These vectors are computed by two sets of three layers (one set per modality):

The **Word Embedding Module** consists of two independently pre-trained lookup tables that map code tokens or natural language tokens to embeddings. We use FastText (Bojanowski et al., 2017)) for all embeddings in this paper.

The **Context Representation Module** consists of bi-directional LSTM layers (one for code, one for text) that map the word embedding sequences into another pair of sequences of embeddings that contain contextual information.

The **Maxpool Layer** performs max pool (separately per dimension) over the Context Representation embedding sequences to obtain a single vector.

The **Similarity Module** computes the similarity of the two vectors v_c and v_c produced by the Maxpool Layers as

$$d(v_1, v_2) = \sum_{i=1}^{d} (v_{1i} - v_{2i})^2$$

$$sim(v_c, v_d) = 1 - d(\frac{v_c}{\|v_c\|_2}, \frac{v_d}{\|v_d\|_2})$$

where d returns the L2 distance between d-dimensional vectors v_c and v_d.

3.2 CAT: An AST-Based Model

To capture both syntactic and semantic features, we augment our baseline **CT** model with embeddings based on the Abstract Syntax Tree (AST) representation of the code. Most programming languages, including Python, come with a deterministic parser that outputs the AST representation of a code snippet. Python has a library module called ast that generates AST representations of code. We convert this AST representation to a string using structure-based traversal (SBT) (Hu et al., 2018). The **CAT** model is similar to the **CT** model, except that it extracts features from both the source code tokens and its corresponding AST representation. So the **Word Embedding Module** now contains three lookup tables: for code, AST, and natural language, respectively. Similarly, the **Context Representation Module** has 3 bi-directional LSTM layers which is followed by 3 **Maxpool Layers**. Before the output is passed to the similarity module, the output vectors of the two max pool layers representing code and AST are concatenated to form a single representation of the source code. Because of this, the hidden dimension in the bidirectional LSTM's of the **Context Representation Module** for the natural language sequence is double that of code and AST sequences' LSTM hidden dimensions. This ensures that, after concatenation, the vectors representing the candidate code snippet and the natural language description are of the same dimension. After that, the **Similarity Module** computes the similarity of these vectors via the same L2-distance-based operation as in **CT**.

3.3 MP: A Multi-Perspective Model

The **CT** and **CAT** models learn to map source code and natural language tokens into a joint embedding space such that semantically similar code-natural language pairs are projected to vectors that are close to each other. However, these two representations interact only in the final step when the global similarity of the sequence embeddings is calculated, but not during the first step when each sequence is encoded into its corresponding embedding. Wang et al. (2017) show that, for tasks such as paraphrase identification and natural language inference that require two pieces of texts from the same language to compare, it is beneficial to include a number of different (i.e., multi-perspective) local matching operations between the two input sequences when computing their vector representations. Given contextual sequence encodings P and Q (computed, e.g., by biLSTMs) for the two sequences to be compared, Wang et al. (2017)'s Bilateral Multi-Perspective Matching (BiMPM) model includes a matching mechanism that compares P and Q by matching each position in P with all positions in Q, and by matching each position in Q with all positions in P, under four different match-

ing strategies. We will discuss these strategies in more detail under the Bilateral Multi-Perspective Matching (BiMPM) Module.

We apply the MP model to our cross-modal code-text matching task as follows: The **Word Embedding Layer** takes as input the code sequence, AST sequence, and description sequence. The output of this layer is three independent sequences of token embeddings, one for each input sequence.

The **Context Representation Module** consists of three sets of BiLSTM layers that each computes a contextual representation of each token in the corresponding input sequence. We concatenate the hidden states of the sequences representing the code and AST, respectively, to get one set of sequence embeddings representing the source code input.

The **Bilateral Multi-Perspective Matching (BiMPM) Module** compares the two sequences, say P and Q, by matching each position in P with all positions in Q, and by matching each position in Q with all positions in P, under four different matching strategies m that each produce new embedding sequences P'_m and Q'_m that have the same length as the original P and Q. Each matching strategy is parameterized by a feedforward network (e.g. $P'[i]_m = f_m^{P \to Q}(P[i], Q_m; W_m^{P \to Q})$) that takes in a token embedding $P[i]$ and a strategy-specific single-vector representation of Q_m, and returns a new vector $P'[i]_m$ for $P[i]$. For each token $P[i] \in P$ (and conversely for any $Q[j] \in Q$), Q_m (P_m) is defined as follows:

Full matching sets Q_m (P_m) to be the final hidden state of Q (and vice versa for P).

Maxpool matching obtains Q_m by performing maximum pooling (per dimension) across the elements of Q.

Attentive matching computes Q_m as a weighted average of all $Q[j] \in Q$, where $Q[j]$'s weight is the cosine similarity of $P[i]$ and $Q[j]$.

Max-Attentive matching sets Q_m to be the $Q[j]$ with the highest cosine similarity to $P[i]$.

We concatenate the four $P'[i]_m$ ($Q'[i]_m$) for each token i to get two new sequences P' and Q'.

The **Local Aggregation Module** aggregates these sequence embeddings into two fixed-length multi-perspective hidden representations by passing them through two different bi-LSTM layers (one for each sequence). For each sequence, we concatenate the final hidden states of both the forward and reverse directions to get a vector representation of that sequence.

The **Similarity Module** computes the similarity of the two vectors returned by the Aggregation Module as before.

3.4 MP-CAT: A Combined Model

Our final model combines the MP and the CAT models. It contains the following components:

The **CAT** module reads in the code sequence, the AST sequence, and the natural language sequence and outputs two vectors, one jointly representing the code and the AST and the other representing the natural language description.

The **MP** module also reads in the code sequence, the AST sequence, and the natural language sequence. It returns two vectors, one for code and AST, and the other for the natural language description. The difference between this module and the previous is that **MP** contains local information that is ignored in the global **CAT** embeddings.

The **Global and Local Fusion Module** concatenates the two CAT and MP vectors representing the code to get the final code representation, and does the same for the CAT and MP vectors representing the natural language description, before computing their L2 distance in the same manner as the other similarity modules. Figure 1 shows the pipeline of the MP-CAT framework.

4 Experiments

The CoNaLa Dataset The CoNaLa dataset (Yin et al., 2018) has two parts, a manually curated parallel corpus of 2,379 training and 500 test examples, and a large automatically-mined dataset with 600k examples (which we ignore here). Each example consists of a snippet of Python code and its corresponding English description.

Pre-processing We pre-process the text representing both the source code and the natural language descriptions using sub-word regularization based on unigram language modeling (Kudo, 2018) transforms the original tokens into sequences of shorter (and hence more common) substrings. We use the sentencepiece library (Kudo and Richardson, 2018) and follow the same approach as used by Yin et al. (2018) for the CoNaLa dataset.

Training procedure During training, we use triplets consisting of a code snippet, a correct description, and an incorrect description (obtained by

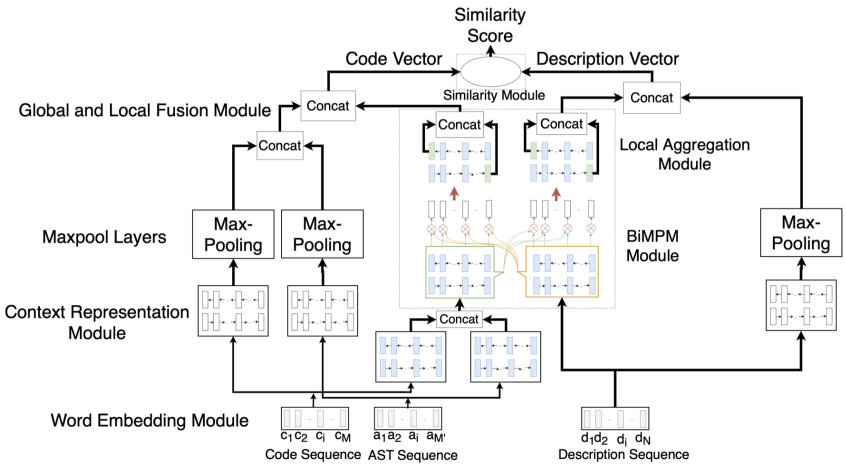

Figure 1: The MP-CAT framework that contains both global-level and local-level features for code–text matching

Framework	Training Time (s)	Evaluation Time (s)
CT	4663.10	6755.62
CAT	6702.69	11050.68
MP	183393.47	17374.14
MP-CAT	240062.38	25306.97

Table 1: Training and Evaluation times for all our models. The models were trained for 100 epochs and the evaluation time was computed on 500 test queries.

Frameworks	MRR	R@1	R@5	R@10
CT	0.172	7.4	24.0	39.6
CAT	0.207	9.0	**32.2**	45.0
MP	0.154	6.4	21.6	33.6
MP-CAT	**0.220**	**11.0**	**32.2**	**47.4**

Table 2: Code Search Results

random sampling from the training set). We sample 5 incorrect descriptions for each code–text pair, giving us five triplets for each training example. During the evaluation phase, for every natural language query $\mathcal{D}$, we calculate the rank of its corresponding code snippet $\mathcal{C}$ among all 500 candidates in the test set.

4.1 Experimental Setup

We train our models on triplets $\langle C, D^+, D^- \rangle$ consisting of a snippet of code C, a natural language description D^+ that correctly describes what the code does (a positive example), and a description D^- that does not describe what the code does (a negative example). We minimize the ranking loss with margin ϵ, following Gu et al. (2018):

$$\mathcal{L}(\theta) = \sum_{\langle C, D^+, D^- \rangle} \max\left(0, \epsilon - \cos(C, D^+) + \cos(C, D^-)\right)$$

In the CAT model, since we first concatenate the vectors for the code and AST before comparing them with the vector for the natural language description, the first two vectors are each half the dimension size of the third one. Our models are implemented in PyTorch (Paszke et al., 2017) and trained using Adam (Kingma and Ba, 2014).

Each model is trained for 100 epochs, and during the evaluation step, we use a set of 500 natural language queries from the test set. The training and evaluation times are shown in Table 2.

4.2 Results

Table 2 shows our test set results for code search. We report Recall@K (K=1,5,10) and mean reciprocal rank (MRR) of the correct answer.

The Impact of Modeling ASTs: In going from the first (CT) row to the second (CAT) row in Table 2, we see that the AST features alone increase MRR from 0.172 to 0.207. There is also an increase in R@k for all values of k. In fact, its R@5 values are competitive with our best model.

Multi-Perspective Results: The results for the multi-perspective models are both surprising and interesting. Row 3 of Table 2 shows that the MP model on its own under-performs and actually has the worst results out of all the models we tested. On the other hand, we see that combining the MP and the CAT models into one framework gives the best performance across the board. This shows that even if we use a multi-perspective framework to model local features, we still need encoders to capture the global features of code and text in addition to the local features; otherwise, we end up missing the forest for the trees.

Query	MP-CAT	CAT
Sort dictionary ‘x‘ by value in ascending order	sorted(list(x.items( )), key = operator.itemgetter(1))	for k in sorted(foo.keys( )): pass
Run a command ‘echo hello world‘ in bash instead of shell	os.system (/bin/bash -c ”echo hello world”)	os.system (’GREPDB= ”echo 123”; /bin/bash -c ”$GREPDB”’)
Select records of dataframe ‘df‘ where the sum of column ’X’ for each value in column ’User’ is 0	df.groupby(’User’)[’X’].filter(lambda x: x.sum() == 0)	print(df.loc[df[’B’].isin([’one’, ’three’])])

Table 3: The top hits returned by the MP-CAT and CAT models for a natural language query.

Query	MP-CAT	MP
Concatenate elements of a list ’x’ of multiple integers to a single integer	sum(d*10**i for i, d in enumerate(x[::-1]))	[float(i) for i in lst]
convert pandas DataFrame ‘df‘ to a dictionary using ‘id‘ field as the key	df.set_index(’id’).to_dict()	data[data[’Value’] == True]
Replace repeated instances of a character ’*’ with a single instance in a string ’text’	re.sub(’**+’, ’*’, text)	re.sub(’^((?:(?!cat).)*cat(?:(?!cat).)*)cat’, ’\\\\1Bull’, s)

Table 4: The top hits returned by the MP-CAT and MP models for a natural language query.

Comparison of MP-CAT, MP and CAT Models In Table 3, we present the retrieval results for select natural language queries from the development set returned by the MP-CAT and CAT models. We do the same thing for MP-CAT and MP models in Table 4. Comparing MP-CAT and CAT, we observe that while CAT correctly identifies the data structures and libraries required to solve the user’s problem, it ends up returning the wrong command. MP, on the other hand, sometimes fails to identify even the correct libraries required. In the second example in Table 4, it fails to understand that there is also a dictionary involved and ends up returning the wrong command. MP-CAT successfully finds the required code snippet when the user queries are longer and have multiple data structures involved.

5 Conclusions

In this paper, we consider the task of semantic code search or retrieval using a code–text similarity model. We propose MP-CAT, a novel multi-perspective deep neural network framework for this task. In contrast to previous approaches, the multi-perspective nature of our model allows it to capture richer similarities between the two sequences.

Acknowledgement

This work is supported by the IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR), a research collaboration as part of the IBM AI Horizons Network.

References

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146.

Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 2018 40th International Conference on Software Engineering (ICSE 2018)*. ACM.

Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. 2020. Improved automatic summarization of subroutines via attention to file context. *ArXiv*, abs/2004.04881.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC ’18, pages 200–210, New York, NY, USA. ACM.

Diederik Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *International Conference on Learning Representations*.

Taku Kudo. 2018. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 66–75, Melbourne, Australia. Association for Computational Linguistics.

Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, Brussels, Belgium. Association for Computational Linguistics.

Alexander LeClair, Sakib Haque, Linfgei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. *ArXiv*, abs/2004.02843.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. In *NIPS-W*.

Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. 2019. Multi-modal attention network learning for semantic source code retrieval. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 13–25.

Zhiguo Wang, Wael Hamza, and Radu Florian. 2017. Bilateral multi-perspective matching for natural language sentences. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 4144–4150.

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 '19, pages 2203–2214, New York, NY, USA. ACM.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories*, MSR, pages 476–486. ACM.