






# Semantic Code Search in Software Repositories using Neural Machine Translation

Evangelos Papathomas(✉) , Themistoklis Diamantopoulos , and Andreas Symeonidis 

Electrical and Computer Engineering Dept., Aristotle University of Thessaloniki  
Thessaloniki, Greece

epapathom@ece.auth.gr, thdiaman@issel.ee.auth.gr, symeonid@ece.auth.gr

**Abstract.** Nowadays, software development is accelerated through the reuse of code snippets found online in question-answering platforms and software repositories. In order to be efficient, this process requires forming an appropriate query and identifying the most suitable code snippet, which can sometimes be challenging and particularly time-consuming. Over the last years, several code recommendation systems have been developed to offer a solution to this problem. Nevertheless, most of them recommend API calls or sequences instead of reusable code snippets. Furthermore, they do not employ architectures advanced enough to exploit the semantics of natural language and code in order to form the optimal query from the question posed. To overcome these issues, we propose CodeTransformer, a code recommendation system that provides useful, reusable code snippets extracted from open-source GitHub repositories. By employing a neural network architecture that comprises advanced attention mechanisms, our system effectively understands and models natural language queries and code snippets in a joint vector space. Upon evaluating CodeTransformer quantitatively against a similar system and qualitatively using a dataset from Stack Overflow, we conclude that our approach can recommend useful and reusable snippets to developers.

**Keywords:** code reuse · semantic analysis · neural transformers.

## 1 Introduction

The wide uptake of open-source software in the last few decades has accelerated software development through code reuse. Nowadays, developers search online for ways to solve issues that arise during the development process, such as writing code for complex tasks, integrating APIs, or fixing bugs. The popularity of this paradigm has been further boosted from the introduction of online repositories (e.g. GitHub) and programming communities (e.g. Stack Overflow).

As code reuse has become a vital aspect of today's software development, the challenge of finding appropriate answers to programming-related questions in the vastness of the Internet led to the development of code recommendation systems. While the majority focus on providing API calls and sequences (e.g.

DeepAPI [10]), a selected few have the advantage of recommending reusable code snippets (e.g. DeepCS [9]). Such systems that employ whole snippet extraction mechanisms are greatly valued, as they significantly reduce development time.

However, they are also prone to important limitations. Many accept queries in specialized query languages instead of natural language. In addition, most systems do not employ mechanisms advanced enough to extract the semantics found both in the queries and the source code. And even though some systems engage in semantic analysis (e.g. DeepCS [9], CodeSearchNet [12]), crucial information, such as the control flow of a code snippet, is discarded. Finally, the aforementioned systems typically employ non-annotated datasets and, by extension, lack in terms of training and quantitative evaluation, as ground truth data are essential for the training of a system and the assessment of its performance.

Acknowledging the need for advancing code reuse, GitHub initiated the CodeSearchNet challenge [12], a public competition for code search, specifically aiming to improve on four baseline models using an annotated dataset. These models receive queries in natural language and employ different neural network architectures to return high-quality code snippets. The CodeSearchNet challenge overall provides an interesting testbed due to the variety of programming languages and code snippets in the dataset and the evaluation tools offered.

Given influence by this challenge, in this paper we present CodeTransformer, a system that receives natural language queries and provides reusable code snippets. CodeTransformer uses state-of-the-art neural network and language understanding techniques, while it also employs a custom similarity metric and a custom loss function. Our system does not require some specialized query language; instead, it receives queries in natural language and employs neural machine translation to offer reusable snippets in the form of methods. We train our system on a state-of-the-practice annotated dataset and evaluate its effectiveness against the baseline CodeSearchNet systems [12]. Finally, we assess its applicability in a question-answering context using data from Stack Overflow.

## 2 Related Work

Code search systems can be distinguished into two categories, those producing sequences of API calls and those producing reusable code. The first category includes systems such as SWIM [21] and T2API [19], which translate text queries to API calls and then synthesize their usage code, i.e. code that uses the calls. SWIM extracts API calls related to a query using Bing and forms their usage code, including the control flow. A limitation is that it cannot handle the semantics of queries (e.g. “convert int to string” and “convert string to int”). T2API is trained on Stack Overflow posts and uses the GraLan language model [17] to model dependencies between API calls and synthesize their usage code.

A different approach to API call recommendation is taken by MULAPI [24]. Apart from usage examples, MULAPI also analyzes the source code and API libraries of a project to provide an implementation of the requested feature. The system also maps the repository of the code to recommend files as locations for

the provided API usage code. The architecture of MULAPI comprises a Stanford Word Segmenter for text preprocessing and a Vector Space Model to assess the similarity between texts. FOCUS [18] is a similar system that analyzes a project's repository and other open source repositories using Abstract Syntax Trees and assesses their similarity using Context-Aware Correlation Filter. Next, it mines API calls from the most similar repositories and presents them to the developers.

Other systems treat code recommendation as a machine translation problem. One of them is DeepAPI [10], which utilizes a Neural Network architecture to transform natural language queries to API sequences. It consists of a recurrent neural network (RNN) encoder that processes natural language using attention mechanisms and an RNN decoder using an Inverse Document Frequency (IDF)-based weighting mechanism to output API sequences. BIKER [4] is a similar system that receives natural language queries and assesses their similarity to Stack Overflow question posts and API documentation. Post texts and code snippets are handled as text and are used to train an embedding model that takes into account IDF weights, and recommends relevant API calls.

Word2API [15] also bridges the semantic gap between natural language and code to provide API recommendations. The system creates tuples of method descriptions and API sequences that are used to train a word embedding model for vector generation. A more advanced approach was implemented by DeepAPIRec [6]. Its architecture consists of Tree-LSTMs, a long short-term memory (LSTM) unit variant that organizes information in an inverse tree structure. DeepAPIRec also utilizes a statistical parameter model of data dependency that allows recommending parameter values for the APIs suggested by the Tree-LSTM.

The second category of systems comprises the ones that recommend reusable code snippets instead of API calls. One of them is Seahawk [20], an Eclipse plugin that, given a query, returns a ranked list of relevant Stack Overflow posts. The posts are retrieved using Apache Solr and ranked using tf-idf. The snippets found in the posts can be integrated into the code of a project. Like Seahawk, NLP2Code [5] is an Eclipse plugin that retrieves code snippets from Stack Overflow posts. NLP2Code processes natural language text and snippets using the TaskNav algorithm and measures their grammatical correlation with the Stanford CoreNLP Toolkit. The system receives natural language queries and employs a customized version of Google Search Engine for search. StackSearch [8] also extracts information from Stack Overflow posts and recommends code snippets using a hybrid language model that combines Tf-Idf and fastText [3]. Its results are also accompanied with labels extracted using named entity recognition.

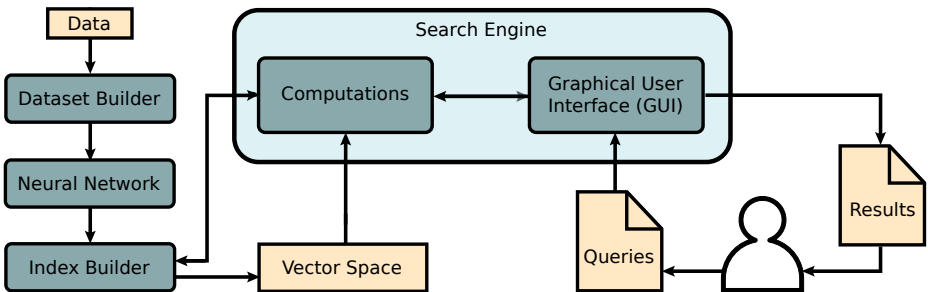
An interesting alternative is DeepCS [9], which recommends reusable code snippets given a natural language query. DeepCS employs two RNN encoders, one that receives natural language descriptions of methods and one that receives a fusion of method names, API sequences and code tokens. Then the system max pools the embeddings generated by the two encoders and assesses their similarity using cosine similarity. DeepCS can understand the semantics of natural language and code to a specific extent, however it relies on the generated vectors to rank its results without considering more code features such as context.

In contrast to systems that utilize raw data dumps from Stack Overflow or code repositories, CodeSearchNet [12] introduced a well curated dataset specifically designed for semantic code search, as it consists of docstring and code tokens which highlight their semantics while also facilitating the preprocessing. Moreover, it introduced four different baselines, each using a different architecture for its encoders (Neural Bag-of-Words, Bidirectional RNNs, 1D Convolutional Neural Networks and Self-Attention). CodeSearchNet outperforms most systems due to the quality of its dataset and its powerful neural architectures. However, it ignores certain semantics, such as the control flow of the code, so it favors keyword-based methods instead of those using semantic information.

Although the aforementioned systems are effective in certain scenarios, they have important limitations. Most of them handle natural language input as keywords, i.e. measuring token frequency instead of analyzing semantics and context. Also, most systems output API calls or API usage code instead of reusable snippets. Deep learning systems often do not employ custom similarity metrics and loss functions. CodeTransformer, is trained on high-quality annotated data from the CodeSearchNet corpus. It analyzes the query and code semantics using word embeddings, generated with state-of-the-art attention mechanisms. We employ a hybrid similarity metric and build a custom loss function that are suited to the challenge at hand. Thus, our system is able to comprehend relations between similar queries (e.g. “how to write to command line” and “how to output to terminal”) and distinguish queries with lexically minor, yet semantically major differences (e.g. “convert int to string” and “convert string to int”).

### 3 Semantic Code Search using Machine Translation

The architecture of our system, shown in Figure 1, comprises four modules: the Dataset Builder, the Neural Network, the Index Builder, and the Search Engine. The Dataset Builder preprocesses the natural language and code data to produce a clean dataset, including the vocabularies of the input and target languages. The Neural Network module generates word embeddings and extracts the most important features per language using attention mechanisms.



**Fig. 1.** The architecture of CodeTransformer

Max pooling is used on the word embeddings to generate a single embedding for each natural language and code sequence. The Index Builder builds a vector space containing the sequence embeddings. Each code vector is assigned to an index to allow nearest neighbor search when a natural language vector is received. The Search Engine receives an input query in the GUI and forwards it to the Computations submodule, where the Neural Network analyzes it and generates a natural language sequence embedding. This vector representation of the query is inserted in the vector space to search for its nearest code vectors. The results are forwarded back to the GUI and presented to the user. These modules are further analyzed in the following subsections.

3.1 Data Preprocessor

**Dataset Overview** The CodeSearchNet corpus comprises over 6.4 million code snippets written in 6 languages, with over 2.3 million of them annotated using docstrings [12]. The snippets were extracted from GitHub repositories, and filtered to remove test functions/constructors, trim long docstrings, and apply de-duplication [16,1]. CodeTransformer was implemented using the Java dataset of the corpus that contains over 1.5 million snippets, of which over 0.54 million come with docstrings. Although we use Java as a proof of concept, it is important to note that our system is mostly language agnostic. Our methodology can be applied to other languages, e.g. Python or JavaScript, with minimal changes.

For each snippet, the dataset contains fields about its origin (repo, path, url, sha) and fields concerning its data (original/full string, method name, extracted code and docstring). The code and the documentation of the snippet (docstring) are also provided as tokens. Table 1 depicts a sample entry of the dataset.

Table 1. An example entry of the dataset

Features	Data
func_name	JsonObjectDeserializer.getRequiredNode
docstring	<pre>/**  * Helper method to return a {@link JsonNode} from the tree.  * @param tree the source tree  * @param fieldName the field name to extract  * @return the {@link JsonNode}  */ protected JsonNode getReqNode(JsonNode tree, String fieldName){     Assert.notNull(tree, "Tree must not be null");     JsonNode node = tree.get(fieldName); code    Assert.state(node != null &amp;&amp; !(node instanceof NullNode), () -&gt;         "Missing JSON field '" + fieldName + "'");     return node; }</pre>

After manual inspection, we concluded that the majority of the dataset entries contain valid natural language docstrings, extracted from each function. However, in certain entries the snippets are not properly annotated and in others the automated natural language text extractor has failed to extract the docstring correctly. For instance, in the docstring of Table 1, the extracted docstring tokens are ['helper', 'method', 'to', 'return', 'a', '{']. To avoid having docstrings that are incorrect or are not properly tokenized, we first preprocess the dataset.

**Data Preprocessing** We create two separate preprocessing pipelines to effectively target the docstrings and the code data. The regular expressions of Table 2 enable modifications in the tokens of the dataset.

**Table 2.** Regular expressions for preprocessing

Regex Name	Regular Expression
remove_non_ascii	<code>[^\x00-\x7f]</code>
remove_special	<code>[^A-Za-z0-9]+</code>
seperate_strings	<code>[A-Z] [a-z] [^A-Z]*</code>
fill_empty	<code>[A-Z] [a-z] [^A-Z]*   [A-Z]* (?![a-z])   [A-Z] [a-z] [^A-Z]*</code>
remove_unnecessary	<code>(\s)   (")   (^//)   (^/\*)   (^/\*\*)</code>
replace_symbols	<code>^[()[\]{}&lt;&gt;+~*/%=&amp;!/?@\.,:;]</code>

For the removal of noisy natural language data, we designed a pipeline of preprocessing steps, as described below:

1. We remove all the tokens of the docstring located after the first dot symbol encounter, thus reducing their size to that of typical natural language queries.
2. The *remove\_non\_ascii* and *remove\_special* expressions are used to replace all non-ASCII characters and all special characters, respectively, in the tokens of the docstring list with empty characters.
3. The *separate\_strings* expression is used to separate all the camelCase tokens of the docstring list and thus augment the data for the neural network.
4. We empty all docstring lists that contain less than 6 or more than 30 tokens<sup>1</sup> as inefficient or lengthy, respectively. The lists are filled with the corresponding camelCase function names and separated using the *fill\_empty* expression.
5. All uppercase characters in the docstring tokens are converted to the corresponding lowercase characters, to achieve structural uniformity between tokens with the same meaning but different writing format.

As an example, the docstring of the snippet shown in Table 1 produces the tokens ['helper', 'method', 'to', 'return', 'json', 'node', 'from', 'the', 'tree'].

<sup>1</sup> The limits were defined after studying the data and concluding that most entries with inefficient docstrings contained less than 6 docstring tokens, while also noting that 30 tokens are adequate for a well-defined description of a function.

Concerning noisy code data, we designed a preprocessing pipeline that slightly differs from those of other systems. Most systems do not sufficiently exploit the control flow information of a code snippet. Instead, they solely focus on function and variable names, as well as control flow words, such as *if*, *else*, *for*, etc. To fully exploit the programming symbols of snippets, we perform the following steps:

- 1. The *remove\_non\_ascii* and *separate\_strings* expressions are used to remove all non-ASCII characters and split the text to tokens.
- 2. We remove all the tokens of the code list that contain space, double quotes, or create a comment using the *remove\_unnecessary* expression.
- 3. We encode programming symbols to unique tokens, as shown in Table 3.

**Table 3.** The encoding of programming symbols to unique tokens

#	Unique Token	#	Unique Token	#	Unique Token
(	openingparen	*	multiplyoperator	<	lessoperator
)	closingparen	/	divideoperator	>=	greaterequaloperator
[	openingbracket	^	poweroperator	<=	lessequaloperator
]	closingbracket	%	modulooperator	++	incrementoperator
{	openingbrace	=	assignoperator	--	decrementoperator
}	closingbrace	==	equaloperator	!	notoperator
+	addoperator	!=	notequaloperator	@	atsign
-	subtractoperator	>	greateroperator	;	semicolon

- 4. The *remove\_special* regular expression is used to remove all the non-alphanumeric characters in the tokens of the code list with empty characters. This step removes symbols that were not replaced in the previous step.
- 5. We limit the length of the code lists to their first 100 tokens, trimming methods of great length and thus enhancing the uniformity of the dataset. Also, all uppercase characters in the code tokens are converted to the corresponding lowercase ones, as in the docstrings, to favor structural uniformity.

As an example, the code of the method snippet shown in Table 1 produces the tokens shown in Figure 2.

'protected', 'json', 'node', 'get', 'required', 'node', 'openingparen', 'json', 'node', 'tree', 'string', 'field', 'name', 'closingparen', 'openingbrace', 'assert', 'not', 'null', 'openingparen', 'tree', 'tree', 'must', 'not', 'be', 'null', 'closingparen', 'semicolon', 'json', 'node', 'node', 'assignoperator', 'tree', 'get', 'openingparen', 'field', 'name', 'closingparen', 'semicolon', 'assert', 'state', 'openingparen', 'notequaloperator', 'null', 'notoperator', 'openingparen', 'node', 'instanceof', 'null', 'node', 'closingparen', 'openingparen', 'closingparen', 'missing', 'json', 'field', 'addoperator', 'field', 'name', 'addoperator', 'closingparen', 'semicolon', 'return', 'node', 'semicolon', 'closingbrace'

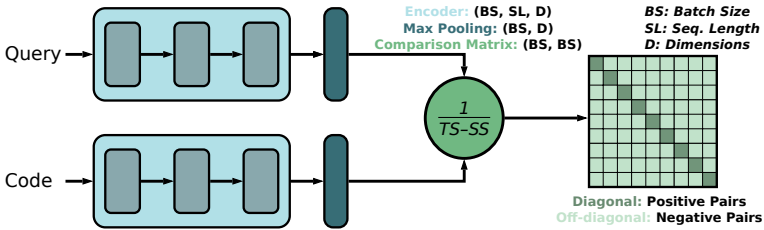
**Fig. 2.** Example tokens extracted from the code of the method snippet of Table 1

Our preprocessing pipeline minimizes the loss of information by performing data augmentation on docstrings and code. In docstrings where the information is insufficient, the pipeline replaces them with separated camelCase function names (e.g. ‘camelCase’ becomes ‘camel case’) that are representative of the code. The pipeline also encodes most code symbols to words instead of removing them and, thus, reinforces code semantics such as control and data flow.

### 3.2 Neural Network

In this subsection we present the main module of our system, a neural network that employs transformers to map natural language queries to source code.

**Network Architecture** The main architecture of CodeTransformer is based on Matching Networks [23], a neural network architecture designed to solve One-Shot Learning problems. Our system, however, follows a slightly different approach, as it uses an improved embedding similarity metric and does not require an external memory to function. As we discuss in the following subsections, our architecture utilizes self-attention encoders and a hybrid geometric similarity metric. In contrast to the original approach, ours does not use a softmax function on its output, as the similarity metric we selected does not natively support it. In Figure 3 we present the architecture of the Neural Network module.

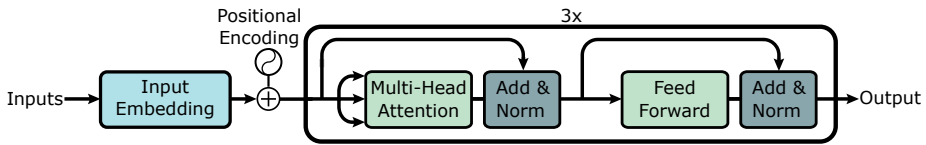


**Fig. 3.** The main architecture of the Neural Network module

**Transformers** To maximize the semantic abilities of our system, we employed the state-of-the-art Transformers architecture on both of its encoders [22]. A Transformer consists of two modules, an encoder and a decoder, with minimal architectural differences. Considering the fact that a Matching Network performs feature extraction and not direct translation of language data, our implementation solely requires encoders for its function. The architecture of the Transformer encoder is presented in Figure 4.

The Transformer encoder comprises an embedding layer, a Positional Encoding layer and encoder layers, i.e. consecutive blocks of Multi-Head Attention and Feed-Forward Network layers. In our implementation, we opted for three stacked encoder layers, as they provide sufficient depth for achieving high efficiency.





**Fig. 4.** The architecture of a Transformer encoder

Before inserting a token sequence to an encoder, we create a vocabulary that includes the most frequently occurring words and then encode them to integers. We build two vocabularies, each consisting of 10,000 unique words. After encoding, we pad each entry with zeros to form tensors of equal dimensions. To enhance the generalization capabilities of our system, we reshuffle the dataset at the start of every training iteration and divide it in batches of 128.

When a token sequence is received as input, the encoder embeds the tokens in a high-dimensional vector space. In other words, the encoder generates word embeddings, i.e. vector representations aiming to extract token information. The encoder generates word embeddings of 128 dimensions using an embedding layer. The natural language encoder and the source code encoder have identical parameter values, but each encoder has its own distinct weights and vocabulary. To generate sequence embeddings we use max pooling, as extracts the most essential features of the embeddings outputted from the stacked encoder layers.

**Similarity Metric** The similarity between natural language and code sequence embeddings is usually quantified using the Euclidean distance or the cosine similarity. However, the computation of the Euclidean distance between two vectors does not contain any information about the angle between the two vectors. On the other hand, cosine similarity does not consider the magnitude of the vectors.

Our system utilizes a hybrid similarity metric, the *Triangle's Area Similarity - Sector's Area Similarity* [11], also known as *TS-SS*, which improves upon the aforementioned metrics by incorporating the Euclidean distance, the magnitude difference and the angle between two vectors to compute their similarity. The Triangle's Area Similarity (TS) comprises the Euclidean distance, the magnitude of each vector and the angle between them, while the Sector's Area Similarity (SS) provides the magnitude difference. The TS of two vectors  $A$  and  $B$  is:

$$TS(A, B) = \frac{|A| \cdot |B| \cdot \sin(\theta')}{2} \quad (1)$$

where, given  $\theta$  is the angle between the two vectors,  $\theta'$  is defined as  $\cos^{-1}(\theta) + 10^\circ$ . We use  $\theta'$  instead of  $\theta$  so that the computation is valid in the case of overlapping vectors (when  $\theta = 0$ ). The SS of two vectors  $A$  and  $B$  is defined as:

$$SS(A, B) = \pi (ED(A, B) + MD(A, B))^2 \cdot \left( \frac{\theta'}{360} \right) \quad (2)$$

where  $\theta'$  is defined as above, while  $ED(A, B)$  and  $MD(A, B)$  correspond to the Euclidean distance and the magnitude difference between the two vectors,

respectively. Given the dimension of the vectors  $N$ , the magnitude difference is:

$$MD(A, B) = \left| \sqrt{\sum_{n=1}^N A_n^2} - \sqrt{\sum_{n=1}^N B_n^2} \right| \quad (3)$$

Merging TS and SS via addition is not possible, as they are in different scale. According to Heidarian and Dinneen [11], their multiplication establishes a new scale that sufficiently represents similarity. Consequently, TS-SS is computed as:

$$TS-SS(A, B) = \frac{|A| \cdot |B| \cdot \sin(\theta') \cdot \theta' \cdot \pi \cdot (ED(A, B) + MD(A, B))^2}{720} \quad (4)$$

TS-SS values range from 0 to infinity, with 0 indicating that two vectors are identical. Accordingly, the TS-SS value of two dissimilar vectors is larger than zero, without any limitations. In our implementation, we decided to calculate the reciprocal TS-SS in favor of the custom loss function we use during our network's training process. The final similarity of the two vectors is computed as:

$$Similarity(A, B) = \frac{1}{TS-SS(A, B)} \quad (5)$$

**Loss Function** The neural network of CodeTransformer outputs a square similarity matrix, where each row represents a natural language embedding and each column represents a source code embedding. The diagonal matrix cells correspond to the positive pairs of natural language and source code and their values ought to be high. The rest of the matrix cells correspond to the negative pairs, and their values ought to be low. At network initialization, all embeddings contain random values and are scattered throughout the vector space. As a result, in order to bring all similar embeddings closer during training, we need to utilize a loss function that is based on the computations of the reciprocal TS-SS.

A loss function typically used by similar systems (such as CodeSearchNet [12] and DeepCS [9]) is a variation of Hinge loss, computed as follows:

$$Loss = \max(0, 1 - positive + negative) \quad (6)$$

Upon testing this variation of Hinge loss, we observed that it did not result in successful integration with the vanilla or reciprocal TS-SS. Even after modifying the function's margin to a value larger than 1, due to TS-SS infinite value range, the result was always the same. The embeddings constantly collapsed to a specific point, not allowing distinct sequence embeddings for each positive pair.

This led us to design a custom loss function, based on the squared variation of Hinge loss. We name this loss function *Squared Margin Loss* and define it as:

$$Loss = (\max(0, margin - positive))^2 + negative^2 \quad (7)$$

Furthermore, the derivatives of our loss function are defined as follows:

$$\frac{\partial}{\partial(positive)} Loss = \begin{cases} 2 \cdot (margin - positive), & \text{if } positive < margin \\ 0, & \text{otherwise} \end{cases} \quad (8)$$

$$\frac{\partial}{\partial(negative)} Loss = 2 \cdot negative \quad (9)$$

The Squared Margin Loss encourages the penalization of larger loss values more, and the penalization of smaller loss values less. Thus, the function ensures the convergence of the network at first epochs and its optimization at later epochs.

By further restricting the function with the function *max*, the positive pairs of similarity value above the margin do not take part in the computation of the loss. In this case, however, the similarity values of the corresponding negative pairs continue to decrease. This allows the similarity values of the diagonal to increase further than the margin. Without the use of the max function, the elements that have crossed the margin would generate useless losses and positive gradients, resulting in the fluctuation of their similarity values around the margin.

**Optimizer** We train our neural network using the *Adaptive Moment Estimation (Adam)* optimizer [14], which computes adaptive learning rates for each parameter. Adam stores the exponentially decaying average of past gradients and the exponentially decaying average of past squared gradients. Using Adam ensures that the network converges fast through momentum estimation. The convergence also depends on the learning rate; a poor choice of its value can slow down the training process, or even derail the network’s weights. To find the ideal learning rate, we examined a range of values generated by the equation:

$$LearningRate = 1.1^{step/100} \cdot 10^{-10} \quad (10)$$

This function generates values starting from  $10^{-10}$  up to a practically infinite value. The learning rate is multiplied by 1.1 once every 100 training steps.

After plotting the accuracy and loss per training step, we noticed a point with a steep increase in accuracy and a steep decrease in loss as well as a point with a steep decrease in accuracy and a steep increase in loss. Next, we isolated the values between these steps and tested those closer to the lower end, where the increase in accuracy and decrease in loss occur. Through trial and error, we selected a learning rate value of  $3.2 \cdot 10^{-4}$ . We set the margin of our network to 5, the number of heads to 8, and the dff to 512, and trained the network for 40 epochs, as these have been shown to be enough for the efficiency of the results.

### 3.3 Index Builder

Due to the complexity of our neural network and the number of its parameters, fast response times cannot be guaranteed. To significantly reduce the processing time of our system, we employed Annoy [2], a tool using a Nearest Neighbor Search algorithm. Using Annoy in the Index Builder module allows us to generate a vector space that contains all the source code embedding vectors of the corpus.

Annoy assigns an index on all code embeddings and then assort them based on their values by building up a forest of trees. The vector dimension of the vector space is set according to the dimension of the output embedding, which is

128. We calculated the Euclidean distance between the vectors and built 10 trees. Regarding the search process in the vector space, we select the first 100 nearest vectors out of the 10.000 nearest forest nodes. Thus, instead of calculating the similarity between a query and the whole corpus using the neural network, Annoy compares the query vector with the nearest 10.000 code vectors. In addition, Annoy’s search time does not seem to be hindered by the embedding dimension.

The search process of a query is executed in three stages. Firstly, the query of the user is preprocessed, so that non-alphanumeric symbols are removed, camel-case tokens are separated and uppercase characters are lowercased. Secondly, every query token is encoded as an integer to be passed as input to the neural network. The neural network, in inference mode, generates the sequence embedding of the query to be inserted to the vector space of the Index Builder. Finally, Annoy extracts the indices of the 10 code vectors nearest to the query, and the corresponding code snippets and GitHub URLs are presented to the user.

## 4 Evaluation

We evaluate our system using two different datasets, the Java corpus of CodeSearchNet [12], and a set of popular Java questions from Stack Overflow<sup>2</sup>.

The performance of our system is assessed using the *Precision at K (P@K)*, the *Mean Reciprocal Rank (MRR)* [7] and the *Normalized Discounted Cumulative Gain (NDCG)* [13]. P@K indicates how many out of the first  $K$  results are relevant to the query. MRR further incorporates the order of the results, computed as the mean of the reciprocal rank of each query (the reciprocal rank of the  $i$ -th query is  $1/\text{rank}_i$ , where  $\text{rank}_i$  is the rank position of the first relevant document). The NDCG is the normalized DCG, computed for  $N$  results as:

$$DCG = \sum_{i=1}^N \frac{2^{rel_i} - 1}{\log_2(i + 1)} \quad (11)$$

where  $rel_i$  is the graded relevance of the result at position  $i$ . Thus, NDCG is computed dividing the result of equation (11) by the ideal DCG, i.e. the one produced if all the results in the list were sorted in the correct order.

### 4.1 Evaluation using CodeSearchNet Queries

CodeTransformer employs the CodeSearchNet corpus [12] for training and inference, allowing its direct comparison with the implementations of CodeSearchNet. CodeSearchNet comprises four different encoder architectures. One of them is the *Self-Attention (SelfAtt)* architecture, which was examined in the previous section. The *Neural Bag of Words (NBoW)* architecture measures word occurrence within a document, therefore it performs well on keyword-based search operations. The *1D Convolutional Neural Network (1D-CNN)* architecture learns to

<sup>2</sup> The code and details used to reproduce our findings can be found at the repository: <https://github.com/AuthEceSoftEng/CodeTransformer>

recognize complex, non-linear patterns. In contrast to NBoW and 1D-CNN, the *Bidirectional RNN (biRNN)* architecture further models the word order.

The four implementations are compared to CodeTransformer on the test set of the CodeSearchNet corpus, which includes 15000 docstring and code snippet pairs, for the computation of MRR. Additionally, the four implementations are compared to CodeTransformer using 99 annotated queries provided by CodeSearchNet for computing NDCG. The results are shown in Table 4. Note that, although our system is not directly compared with DeepCS [9] as the systems use different data, we compare it with the *biRNN* implementation of CodeSearchNet that has a similar neural architecture with DeepCS.

**Table 4.** Evaluation results of CodeTransformer and CodeSearchNet

System	MRR NDCG	
CodeSearchNet-NBoW	0.5140	0.1207
CodeSearchNet-1D-CNN	0.5270	0.1282
CodeSearchNet-biRNN	0.2865	0.0623
CodeSearchNet-SelfAtt	0.5866	0.1003
CodeTransformer	0.6263	0.1028

Concerning MRR, our system outperforms CodeSearchNet measurements, indicating that the different strategies followed for our data pipeline are effective. Another factor that may contribute to this result is our preprocessing methodology, as it may be possible that the replacement of insufficient docstrings with function names led to increased MRR values. As a side note, these results were also clear during the validation phase of the algorithms (e.g. the MRR of CodeTransformer for the validation set was the highest at 0.62604, while the second highest was that of CodeSearchNet-SelfAtt at 0.5513).

Concerning NDCG, our system performs slightly better compared to the corresponding Self-Attention implementation of CodeSearchNet, while the NBoW and 1D-CNN implementations perform better than CodeTransformer, possibly because they use docstrings as natural language. However, we note that only a small amount of data was annotated for the computation of NDCG (i.e. only 823 out of 1.5 million Java code snippets). In addition, as the authors of CodeSearchNet note [12], the annotated data were selected using the top 10 results per query, generated by an ensemble of the CodeSearchNet neural models and ElasticSearch, therefore they are what these systems are more likely to produce. Hence, it is possible that correct results are ignored for computing NDCG.

Figure 5 depicts the distribution and the individual MRR values for 99 queries of the test set of CodeSearchNet [12]. As the annotations were not provided, we annotated the first 10 results returned by our system to compute the MRR. The majority of MRR values are equal to 1, indicating that our system returns a relevant result in the first position for more than half of the queries. By examining the results, we found that our system effectively models the semantic informa-

tion of the text and the code snippets. Indicatively, for Q64, CodeTransformer outputs a function that sorts an array using another array’s order, even though almost none of the exact words of the query are present in the code (except for the word “sort”). Semantically similar terms are also effectively interpreted. E.g., for Q16 that requests exporting data to an excel file, our system returns an `exportXls` method, thus modeling the semantic similarity between terms “excel” and “xls”. Similarly, given Q91 that requests data extraction from a text file, CodeTransformer returns a method using the term “read” instead of “extract”.

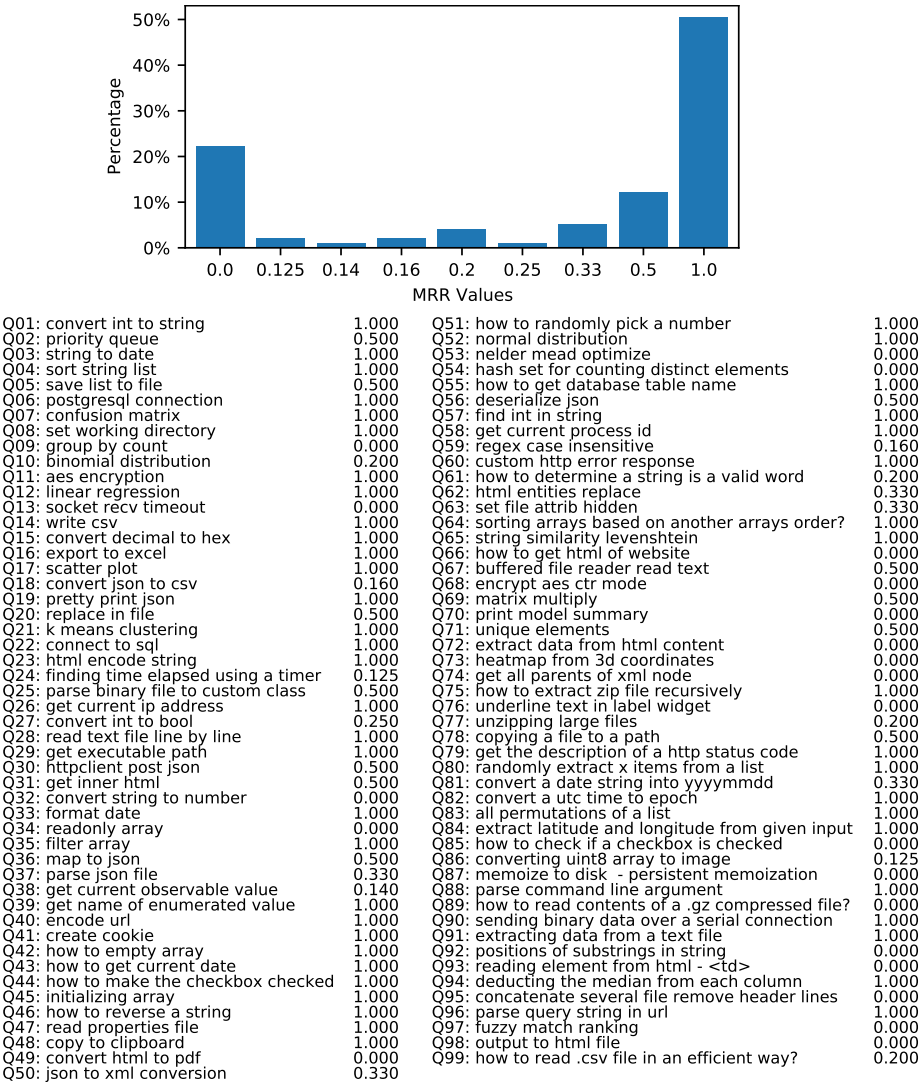


Fig. 5. MRR values of CodeTransformer for the 99 queries of CodeSearchNet dataset

Concerning queries for which our system did not perform as effectively, some of them are relevant to other programming languages and/or are not included in the corpus. Note that these 99 queries are drawn from 6 languages and thus not all of them are relevant to Java. An example unanswered query is Q34, as read-only arrays do not exist in Java and, therefore, a relevant code snippet is not included in the corpus. After manually inspecting the corpus, we concluded that Java code snippets for queries Q53, Q68, Q70, Q73, Q76 and Q97 are focused on other languages. This is also the case for HTML parsing queries, such as queries Q49, Q66, Q72, Q93 and Q98, for which we could find a few Java methods by manual inspection, however they are mainly targeted at other languages. In any case, considering the results of Table 4 and Figure 5, CodeTransformer seems to provide a relevant answer in the first two positions more often than not.

Finally, as a proof of concept, Table 5 depicts the declarations of the methods returned by our system for query Q91, which refers to “extracting data from a text file”. It is clear that the methods respond effectively to the query.

**Table 5.** Declarations of the methods returned by CodeTransformer for query Q91 “extracting data from a text file”

#	Method Declaration
1	<code>public static String readTextFile(Context context, int resId)</code>
2	<code>public static String readTextFile(Context context, String asset)</code>
3	<code>public DataSource&lt;String&gt; readTextFile(String filePath)</code>
4	<code>public static String readTextFile(String fileName)</code>
5	<code>public static String readTextFile(File file)</code>
6	<code>public DataSource&lt;String&gt; readTextFile(String filePath, String charsetName)</code>
7	<code>public static String readTextFile(Context context, int resourceId)</code>
8	<code>public static String readTextFile(File file) throws IOException</code>
9	<code>private ProjectFile readTextFile(InputStream inputStream) throws MPXJException</code>
10	<code>public DataSource&lt;String&gt; readTextFile(String filePath, String charsetName)</code>

## 4.2 Evaluation using Stack Overflow Questions

To further evaluate CodeTransformer, we reviewed its performance on real user queries. Although our model uses docstrings instead of real queries, we consider this experiment adequate for assessing its effectiveness as a proof of concept.

We manually selected the first 40 highest-rated Stack Overflow posts at the time of research, in which the posters search for Java code snippets. After querying our system using their titles, we obtained 10 results for each query, sorted by their similarity to the query. Next, we manually annotated the similarity of each result to the query, making sure that the result is a valid answer. To avoid any threats to validity, the annotations were performed without knowledge of the order of the results. Table 6 depicts the questions as well as the rank of the first relevant result and the precision at the first 10 results for each question.

**Table 6.** Evaluation results of CodeTransformer on the set of the 40 most popular Stack Overflow Java questions

#	Questions	Rank P@10	
S01	How do I read / convert an InputStream into a String in Java?	1	1.0
S02	Create ArrayList from array	2	0.7
S03	How do I generate random integers within a specific range in Java?	2	0.7
S04	Iterate through a HashMap [duplicate]	2	0.2
S05	How do I efficiently iterate over each entry in a Java Map?	2	0.1
S06	How do I convert a String to an int in Java?	1	0.2
S07	Initialization of an ArrayList in one line	—	—
S08	How do I determine whether an array contains a value in Java?	3	0.3
S09	How do I call one constructor from another in Java?	—	—
S10	How do I declare and initialize an array in Java?	—	—
S11	How to get an enum value from a string value in Java?	1	1.0
S12	What's the simplest way to print a Java array?	1	1.0
S13	How to generate a random alpha-numeric string?	1	0.8
S14	How to split a string in Java	1	1.0
S15	Sort a Map<Key, Value> by values	7	0.1
S16	How do I create a Java string from the contents of a file?	7	0.1
S17	How can I convert a stack trace to a string?	1	0.8
S18	Fastest way to determine if an integer's square root is integer	—	—
S19	How do I create a file and write to it in Java?	3	0.3
S20	How can I concatenate two arrays in Java?	1	0.7
S21	How to round a number to n decimal places in Java	1	0.8
S22	Convert ArrayList<String> to String[] array	1	0.7
S23	Sort ArrayList of custom Objects by property	1	0.5
S24	How can I initialise a static Map?	—	—
S25	How to directly initialize a HashMap (in a literal way)?	1	0.9
S26	How to create a generic array in Java?	1	1.0
S27	How to parse JSON in Java	1	0.7
S28	Converting array to list in Java	1	0.4
S29	How to get the current working directory in Java?	1	0.9
S30	Converting 'ArrayList<String>' to 'String[]' in Java	9	0.1
S31	How can I pad an integer with zeros on the left?	1	0.4
S32	How can I get the current stack trace in Java?	1	1.0
S33	Java 8 List<V> into Map<K, V>	—	—
S34	Reading a plain text file in Java	1	1.0
S35	How to check if a String is numeric in Java	3	0.7
S36	Java string to date conversion	1	0.9
S37	A 'for' loop to iterate over an enum in Java	—	—
S38	How do I convert a String to an InputStream in Java?	—	—
S39	Convert InputStream to byte array in Java	2	0.7
S40	How can I read a large text file line by line using Java?	1	0.7

Precision at the first 10 results is relatively high for most queries. Moreover, we may note that CodeTransformer effectively disambiguates among queries with similar context. Consider, e.g., queries S17 and S32 that are both relevant to



stack traces; although these queries are similar, the system was able to comprehend the semantics of each query and return several highly ranked relevant results. Even for queries with low precision in their results, CodeTransformer placed the first relevant result in the first or the second position. Thus, even though for some queries there are not many relevant results, the users typically receive at least one correct answer. An example would be query S06, for which the system returned only two relevant results, but one of them is ranked in the first place. It is also notable, in the same query, that CodeTransformer distinguishes among converting “string to integer” and “integer to string”.

The fact that 8 out of 40 questions were not answered at all occurs mostly because a matching function does not exist in the corpus. For example, queries S07, S09, S10, S24, and S37 do not require a whole method for their implementation and, thus, the corpus does not include relevant code snippets. Other queries may be too complex, such as query S18, for which our system returns some relevant code snippets, however these results do not meet the condition of the fastest way to examine if an integer’s square root is an integer.

In Table 7 we provide three example Stack Overflow queries and the corresponding relevant answers. For the first two queries, CodeTransformer has placed the answers at the first position, while for the third query the answer was placed at the second position. As shown by these examples, CodeTransformer indeed retrieves and recommends useful snippets in a question-answering scenario.

## 5 Conclusion

Although there are several approaches for code snippet retrieval, most of them do not consider semantics of natural language and code, ignoring essential information regarding the data. Furthermore, several of them recommend API calls or sequences instead of reusable code snippets, requiring more effort from the developer. Deep learning systems are usually more effective, however most do not employ advanced neural transformer architectures and are limited by the fact that they are not trained on annotated datasets. Our system, CodeTransformer, overcomes these limitations by employing a state-of-the-art neural network architecture. The advanced attention mechanisms of this architecture, including specialized similarity metric and custom loss function, along with the preprocessing pipeline specifically designed to augment natural language and code semantics, allow the system to generate powerful data representations.

Upon evaluating CodeTransformer against the implementations of CodeSearchNet, we found that our system is more effective, especially when the developer would prefer to receive the method most relevant to the query rather than a list of related methods. We further assessed CodeTransformer on a dataset of actual questions from Stack Overflow, with the results indicating that it is capable of retrieving useful code, even for complex natural language queries.

For future work, we consider implementing our network using real-life natural language data, such as Stack Overflow questions, instead of code documentation. In addition, we could train our network using other (less curated) datasets

**Table 7.** Example Stack Overflow queries and the answers of CodeTransformer

Features	Data
Query S11	How to get an enum value from a string value in Java?
Result	<pre> <b>public static</b> T getEnumFromString(Class c, String string) {     <b>if</b> (c != <b>null</b> &amp;&amp; string != <b>null</b>) {         <b>return</b> Enum.valueOf(c, string.trim().toUpperCase());     }     <b>return null</b>; } </pre>
Query S20	How can I concatenate two arrays in Java?
Result	<pre> <b>public static</b> String[] concat(String[] array1, String[] array2) {     <b>int</b> length1 = array1.length;     <b>int</b> length2 = array2.length;     <b>int</b> length = length1 + length2;     String[] dest = <b>new</b> String[length];     System.arraycopy(array1, 0, dest, 0, length1);     System.arraycopy(array2, 0, dest, length1, length2);     <b>return</b> dest; } </pre>
Query S36	Java string to date conversion
Result	<pre> <b>public static</b> Date serviceStringToDate(String s) {     <b>if</b> (s == <b>null</b>)         <b>return null</b>;     <b>try</b> {         <b>return new</b> SimpleDateFormat(_serviceDateFormat).parse(s);     }     <b>catch</b> (Exception e) {         <b>return null</b>;     } } </pre>

and explore different preprocessing techniques, incorporating the semantics of programming symbols and the information provided by method names to the natural language data. Finally, we could explore whether our system can generate docstrings by providing code snippets as input to the code encoder and comparing their sequence embeddings to docstring sequence embeddings.

## Acknowledgements

This research has been co-financed by the European Regional Development Fund of the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH - CREATE - INNOVATE (project code: T1EDK-02347).

## References

1. Allamanis, M.: 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. p. 143–153. Onward! 2019, Association for Computing Machinery, New York, NY, USA (2019)
2. Bernhardsson, E.: Annoy: Approximate Nearest Neighbors in C++/Python (2018), <https://pypi.org/project/annoy/>, Python package version 1.13.0
3. Bojanowski, P., Grave, E., Joulin, A., Mikolov, T.: Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics* **5**, 135–146 (2017)
4. Cai, L., Wang, H., Huang, Q., Xia, X., Xing, Z., Lo, D.: BIKER: A Tool for Bi-Information Source Based API Method Recommendation. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. p. 1075–1079. ESEC/FSE 2019, ACM, New York, NY, USA (2019)
5. Campbell, B.A., Treude, C.: NLP2Code: Code Snippet Content Assist via Natural Language Tasks. In: Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution. pp. 628–632. ICSME 2017, IEEE Computer Society, Los Alamitos, CA, USA (2017)
6. Chen, C., Peng, X., Sun, J., Xing, Z., Wang, X., Zhao, Y., Zhang, H., Zhao, W.: Generative API Usage Code Recommendation with Parameter Concretization. *Science China Information Sciences* **62**(9), 192103 (2019)
7. Craswell, N.: Mean Reciprocal Rank, p. 1703. In: Liu, Ling and Özsu, M. Tamer (eds), *Encyclopedia of Database Systems*, Springer, Boston, MA (2009)
8. Diamantopoulos, T., Oikonomou, N., Symeonidis, A.: Extracting Semantics from Question-Answering Services for Snippet Reuse. In: Proceedings of the 23rd International Conference on Fundamental Approaches to Software Engineering. pp. 119–139. Dublin, Ireland (2020)
9. Gu, X., Zhang, H., Kim, S.: Deep Code Search. In: Proceedings of the 40th International Conference on Software Engineering. p. 933–944. ICSE '18, Association for Computing Machinery, New York, NY, USA (2018)
10. Gu, X., Zhang, H., Zhang, D., Kim, S.: Deep API Learning. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 631–642. FSE 2016, ACM, New York, NY, USA (2016)
11. Heidarian, A., Dinneen, M.J.: A Hybrid Geometric Approach for Measuring Similarity Level Among Documents and Document Clustering. In: Proceedings of the 2016 IEEE Second International Conference on Big Data Computing Service and Applications. pp. 142–151. BigDataService 2016, IEEE Computer Society, Los Alamitos, CA, USA (2016)
12. Husain, H., Wu, H.H., Gazit, T., Allamanis, M., Brockschmidt, M.: CodeSearchNet Challenge: Evaluating the State of Semantic Code Search (2019)
13. Järvelin, K., Kekäläinen, J.: Cumulated Gain-Based Evaluation of IR Techniques. *ACM Trans. Inf. Syst.* **20**(4), 422–446 (2002)
14. Kingma, D.P., Ba, J.: Adam: A Method for Stochastic Optimization. In: Proceedings of the 3rd International Conference on Learning Representations. pp. 1–15. ICLR 2015, San Diego, CA, USA (2015)
15. Li, X., Jiang, H., Kamei, Y., Chen, X.: Bridging Semantic Gaps between Natural Languages and APIs with Word Embedding. *IEEE Transactions on Software Engineering* pp. 1–17 (2018)

16. Lopes, C.V., Maj, P., Martins, P., Saini, V., Yang, D., Zitny, J., Sajjani, H., Vitek, J.: DéjàVu: A Map of Code Duplicates on GitHub. *Proc. ACM Program. Lang.* **1**(OOPSLA) (2017)
17. Nguyen, A.T., Nguyen, T.N.: Graph-Based Statistical Language Model for Code. In: *Proceedings of the 37th International Conference on Software Engineering - Volume 1*. p. 858–868. ICSE '15, IEEE Press (2015)
18. Nguyen, P.T., Di Rocco, J., Di Ruscio, D., Ochoa, L., Degueule, T., Di Penta, M.: FOCUS: A Recommender System for Mining API Function Calls and Usage Patterns. In: *Proceedings of the 41st International Conference on Software Engineering*. p. 1050–1060. ICSE '19, IEEE Press (2019)
19. Nguyen, T., Rigby, P.C., Nguyen, A.T., Karanfil, M., Nguyen, T.N.: T2API: Synthesizing API Code Usage Templates from English Texts with Statistical Translation. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. pp. 1013–1017. FSE 2016, ACM, New York, NY, USA (2016)
20. Ponzanelli, L., Bacchelli, A., Lanza, M.: Seahawk: Stack Overflow in the IDE. In: *Proceedings of the 2013 International Conference on Software Engineering*. pp. 1295–1298. ICSE '13, IEEE Press, Piscataway, NJ, USA (2013)
21. Raghothaman, M., Wei, Y., Hamadi, Y.: SWIM: Synthesizing What I Mean: Code Search and Idiomatic Snippet Synthesis. In: *Proceedings of the 38th International Conference on Software Engineering*. pp. 357–367. ICSE '16, ACM, New York, NY, USA (2016)
22. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, u., Polosukhin, I.: Attention is All You Need. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. p. 6000–6010. NIPS'17, Curran Associates Inc., Red Hook, NY, USA (2017)
23. Vinyals, O., Blundell, C., Lillicrap, T., Kavukcuoglu, K., Wierstra, D.: Matching Networks for One Shot Learning. In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. p. 3637–3645. NIPS'16, Curran Associates Inc., Red Hook, NY, USA (2016)
24. Xu, C., Sun, X., Li, B., Lu, X., Guo, H.: MULAPI: Improving API method recommendation with API usage location. *Journal of Systems and Software* **142**, 195 – 205 (2018)

**Open Access** This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

