# Deep Code Search

Xiaodong Gu[1], Hongyu Zhang[2], and Sunghun Kim[1,3]
[1]The Hong Kong University of Science and Technology, Hong Kong
guxiaodong1987@126.com,hunkim@cse.ust.hk
[2]The University of Newcastle, Callaghan, Australia
hongyu.zhang@newcastle.edu.au
[3]Clova AI Research, NAVER

## ABSTRACT

To implement a program functionality, developers can reuse previously written code snippets by searching through a large-scale codebase. Over the years, many code search tools have been proposed to help developers. The existing approaches often treat source code as textual documents and utilize information retrieval models to retrieve relevant code snippets that match a given query. These approaches mainly rely on the textual similarity between source code and natural language query. They lack a deep understanding of the semantics of queries and source code.

In this paper, we propose a novel deep neural network named CODEnn (Code-Description Embedding Neural Network). Instead of matching text similarity, CODEnn jointly embeds code snippets and natural language descriptions into a high-dimensional vector space, in such a way that code snippet and its corresponding description have similar vectors. Using the unified vector representation, code snippets related to a natural language query can be retrieved according to their vectors. Semantically related words can also be recognized and irrelevant/noisy keywords in queries can be handled.

As a proof-of-concept application, we implement a code search tool named DeepCS using the proposed CODEnn model. We empirically evaluate DeepCS on a large scale codebase collected from GitHub. The experimental results show that our approach can effectively retrieve relevant code snippets and outperforms previous techniques.

## CCS CONCEPTS

• **Software and its engineering → Reusability**;

## KEYWORDS

code search, deep learning, joint embedding

**ACM Reference Format:**
Xiaodong Gu[1], Hongyu Zhang[2], and Sunghun Kim[1,3]. 2018. Deep Code Search. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180167

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180167

## 1 INTRODUCTION

Code search is a very common activity in software development practices [57, 68]. To implement a certain functionality, for example, to *parse XML files*, developers usually search and reuse previously written code by performing free-text queries over a large-scale codebase.

Many code search approaches have been proposed [13, 15, 29, 31, 32, 35, 44, 45, 47, 62], most of them being based on information retrieval (IR) techniques. For example, Linstead et al. [43] proposed Sourcerer, an information retrieval based code search tool that combines the textual content of a program with structural information. McMillan et al. [47] proposed Portfolio, which returns a chain of functions through keyword matching and PageRank. Lu et al. [44] expanded a query with synonyms obtained from WordNet and then performed keyword matching of method signatures. Lv et al. [45] proposed CodeHow, which combines text similarity and API matching through an extended Boolean model.

A fundamental problem of the IR-based code search is the mismatch between the high-level intent reflected in the natural language queries and low-level implementation details in the source code [12, 46]. Source code and natural language queries are heterogeneous. They may not share common lexical tokens, synonyms, or language structures. Instead, they may only be semantically related. For example, a relevant snippet for the query "*read an object from an xml*" could be as follows:

```
public static < S > S deserialize(Class c, File xml) {
    try {
        JAXBContext context = JAXBContext.newInstance(c);
        Unmarshaller unmarshaller = context.createUnmarshaller();
        S deserialized = (S) unmarshaller.unmarshal(xml);
        return deserialized;
    } catch (JAXBException ex) {
        log.error("Error-deserializing-object-from-XML", ex);
        return null;
    }
}
```

Existing approaches may not be able to return this code snippet as it does not contain keywords such as *read* and *object* or their synonyms such as *load* and *instance*. Therefore, an effective code search engine requires a higher-level semantic mapping between code and natural language queries. Furthermore, the existing approaches have difficulties in query understanding [27, 29, 45]. They cannot effectively handle irrelevant/noisy keywords in queries [27]. Therefore, an effective code search engine should also be able to understand the semantic meanings of natural language queries and source code in order to improve the accuracy of code search.

In our previous work, we introduced the DeepAPI framework [27], which is a deep learning based method that learns the semantics of queries and the corresponding API sequences. However, searching source code is much more difficult than generating APIs, because the semantics of code snippets are related not only to the API sequences but also to other source code aspects such as tokens and method names. For example, DeepAPI could return the same API *ImageIO.write* for the query *save image as png* and *save image as jpg*. Nevertheless, the actual code snippets for answering the two queries are different in terms of source code tokens. Therefore, the code search problem requires models that can exploit more aspects of the source code.

In this paper, we propose a novel deep neural network named CODEnn (Code-Description Embedding Neural Network). To bridge the lexical gap between queries and source code, CODEnn jointly embeds code snippets and natural language descriptions into a high-dimensional vector space, in such a way that code snippet and its corresponding description have similar vectors. With the unified vector representation, code snippets semantically related to a natural language query can be retrieved according to their vectors. Semantically related words can also be recognized and irrelevant/noisy keywords in queries can be handled.

Using CODEnn, we implement a code search tool, DeepCS as a proof of concept. DeepCS trains the CODEnn model on a corpus of 18.2 million Java code snippets (in the form of commented methods) from GitHub. Then, it reads code snippets from a codebase and embeds them into vectors using the trained CODEnn model. Finally, when a user query arrives, DeepCS finds code snippets that have the nearest vectors to the query vector and return them.

To evaluate the effectiveness of DeepCS, we perform code search on a search codebase using 50 real-world queries obtained from Stack Overflow. Our results show that DeepCS returns more relevant code snippets than the two related approaches, that is, CodeHow [45] and a conventional Lucene-based code search tool [5]. On average, the first relevant code snippet returned by DeepCS is ranked 3.5, while the first relevant results returned by CodeHow [45] and Lucene [43] are ranked 5.5 and 6.0, respectively. For 76% of the queries, the relevant code snippets can be found within the top 5 returned results. The evaluation results confirm the effectiveness of DeepCS.

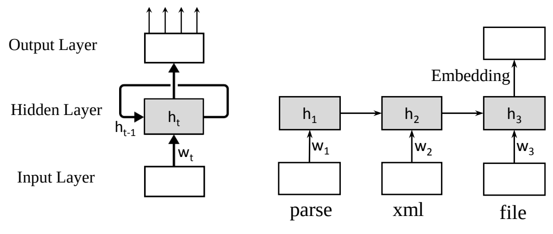
To our knowledge, we are the first to propose deep learning based code search. The main contributions of our work are as follows:

- We propose a novel deep neural network, CODEnn, to learn a unified vector representation of both source code and natural language queries.
- We develop DeepCS, a tool that utilizes CODEnn to retrieve relevant code snippets for given natural language queries.
- We empirically evaluate DeepCS using a large scale codebase.

The rest of this paper is organized as follows. Section 2 describes the background of the deep learning based embedding models. Section 3 describes the proposed deep neural network for code search. Section 4 describes the detailed design of our approach. Section 5 presents the evaluation results. Section 6 discusses our work, followed by Section 7 that presents the related work. We conclude the paper in Section 8.

## 2 BACKGROUND

Our work adopts recent advanced techniques from deep learning and natural language processing [10, 17, 70]. In this section, we discuss the background of these techniques.



**(a) RNN Structure** **(b) RNN for sentence embedding**

**Figure 1: Illustration of the RNN Sentence Embedding**

### 2.1 Embedding Techniques

Embedding (also known as distributed representation [50, 72]) is a technique for learning vector representations of entities such as words, sentences and images in such a way that similar entities have vectors close to each other [48, 50].

A typical embedding technique is word embedding, which represents words as fixed-length vectors so that similar words are close to each other in the vector space [48, 50]. For example, suppose the word *execute* is represented as [0.12, -0.32, 0.01] and the word *run* is represented as [0.12, -0.31, 0.02]. From their vectors, we can estimate their distance and identify their semantic relation. Word embedding is usually realized using a machine learning model such as CBOW and Skip-Gram [48]. These models build a neural network that captures the relations between a word and its contextual words. The vector representations of words, as parameters of the network, are trained with a text corpus [50].

Likewise, a sentence (i.e., a sequence of words) can also be embedded as a vector [59]. A simple way of sentence embedding is, for example, to view it as a bag of words and add up all its word vectors [39].

### 2.2 RNN for Sequence Embedding

We now introduce a widely-used deep neural network, the Recurrent Neural Networks (RNN) [49, 59] for the embedding of sequential data such as natural language sentences. The Recurrent Neural Network is a class of neural networks where hidden layers are recurrently used for computation. This creates an internal state of the network to record dynamic temporal behavior. Figure 1a shows the basic structure of an RNN. The neural network includes three layers, an input layer which maps each input to a vector, a recurrent hidden layer which recurrently computes and updates a hidden state after reading each input, and an output layer which utilizes the hidden state for specific tasks. Unlike traditional feedforward neural networks, RNNs can embed sequential inputs such as sentences using their internal memory [25].

Consider a natural language sentence with a sequence of $T$ words $s$=$w_1, ..., w_T$, RNN embeds it through the following computations: it reads words in the sentence one by one, and updates a hidden state at each time step. Each word $w_t$ is first mapped to a $d$-dimensional vector $\boldsymbol{w}_t \in \mathbb{R}^d$ by a one-hot representation [72] or word embedding [50]. Then, the hidden state (values in the hidden layer) $\boldsymbol{h}_t$ is updated at time $t$ by considering the input word $\boldsymbol{w}_t$ and the preceding hidden state $\boldsymbol{h}_{t-1}$:

$$\boldsymbol{h}_t = \tanh(\boldsymbol{W}[\boldsymbol{h}_{t-1}; \boldsymbol{w}_t]), \forall t = 1, 2, ..., T \quad (1)$$

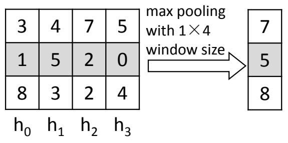


**Figure 2: Illustration of max pooling**

where $[a;b]\in\mathbb{R}^{2d}$ represents the concatenation of two vectors, $\boldsymbol{W}\in\mathbb{R}^{2d\times d}$ is the matrix of trainable parameters in the RNN, while *tanh* is a non-linearity activation function of the RNN. Finally, the embedding vector of the sentence is summarized from the hidden states $\boldsymbol{h}_1, ..., \boldsymbol{h}_T$. A typical way is to select the last hidden state $\boldsymbol{h}_T$ as the embedding vector. The embedding vector can also be summarized using other computations such as the maxpooling [36]:

$$s = \text{maxpooling}([\boldsymbol{h}_1, ..., \boldsymbol{h}_T]) \tag{2}$$

Maxpooling is an operation that selects the maximum value in each fixed-size region over a matrix. Figure 2 shows an example of maxpooling over a sequence of hidden vectors $\boldsymbol{h}_1, ..., \boldsymbol{h}_T$. Each column represents a hidden vector. The window size of each region is set to $1\times T$ in this example. The result is a fixed-length vector whose elements are the maximum values of each row. Maxpooling can capture the most important feature (one with the highest value) for each region and can transform sentences of variable lengths into a fixed-length vector.

Figure 1b shows an example of how RNN embeds a sentence (e.g., *parse xml file*) into a vector. To facilitate understanding, we expand the recurrent hidden layer for each time step. The RNN reads words in the sentence one by one, and updates a hidden state at each time step. When it reads the first word *parse*, it maps the word into a vector $\boldsymbol{w}_1$ and computes the current hidden state $\boldsymbol{h}_1$ using $\boldsymbol{w}_1$. Then, it reads the second word *xml*, embeds it into $\boldsymbol{w}_2$, and updates the hidden state $\boldsymbol{h}_1$ to $\boldsymbol{h}_2$ using $\boldsymbol{w}_2$. The procedure continues until the RNN receives the last word *file* and outputs the final state $\boldsymbol{h}_3$. The final state $\boldsymbol{h}_3$ can be used as the embedding $\boldsymbol{c}$ of the whole sentence.

The embedding of the sentence, i.e., the sentence vector, can be used for specific applications. For example, one can build a language model conditioning on the sentence vector for machine translation [17]. One can also embed two sentences (a question sentence and an answer sentence) and compare their vectors for answer selection [21, 71].

## 2.3 Joint Embedding of Heterogeneous Data

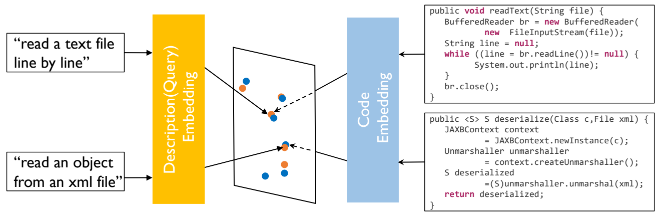
Suppose there are two heterogeneous data sets $\mathcal{X}$ and $\mathcal{Y}$. We want to learn a correlation between them, namely,

$$f : \mathcal{X} \rightarrow \mathcal{Y} \tag{3}$$

For example, suppose $\mathcal{X}$ is a set of images and $\mathcal{Y}$ is a set of natural language sentences, $f$ can be the correlation between the images and the sentences (i.e., image captioning). Since the two data sources are heterogeneous, it is difficult to discover the correlation $f$ directly. Thus, we need a bridge to connect these two levels of information.

Joint Embedding, also known as multi-modal embedding [78], is a technique to jointly embed/correlate heterogeneous data into a unified vector space so that semantically similar concepts across the two modalities occupy nearby regions of the space [33]. The joint embedding of $\mathcal{X}$ and $\mathcal{Y}$ can be formulated as:

$$\mathcal{X} \xrightarrow{\phi} V_{\mathcal{X}} \rightarrow J(V_{\mathcal{X}}, V_{\mathcal{Y}}) \leftarrow V_{\mathcal{Y}} \xleftarrow{\psi} \mathcal{Y} \tag{4}$$


"read a text file line by line"
"read an object from an xml file"
Description(Query) Embedding
Code Embedding

```
public void readText(String file) {
   BufferedReader br = new BufferedReader(
           new  FileInputStream(file));
   String line = null;
   while ((line = br.readLine())!= null) {
         System.out.println(line);
   }
   br.close();
}
```

```
public <S> S deserialize(Class c,File xml) {
   JAXBContext context
           = JAXBContext.newInstance(c);
   Unmarshaller unmarshaller
           = context.createUnmarshaller();
   S deserialized
           =(S)unmarshaller.unmarshal(xml);
   return deserialized;
}
```

**Figure 3: An example showing the idea of joint embedding for code and queries. The yellow points represent query vectors while the blue points represent code vectors.**

where $\phi$:$\mathcal{X}\rightarrow\mathbb{R}^d$ is an embedding function to map $\mathcal{X}$ into a d-dimensional vector space $V$; $\psi$:$\mathcal{Y}\rightarrow\mathbb{R}^d$ is an embedding function to map $\mathcal{Y}$ into the same vector space $V$; $J(\cdot,\cdot)$ is a similarity measure (e.g., cosine) to score the matching degrees of $V_{\mathcal{X}}$ and $V_{\mathcal{Y}}$ in order to learn the mapping functions. Through joint embedding, heterogeneous data can be easily correlated through their vectors.

Joint embedding has been widely used in many tasks [22, 74, 78]. For example, in computer vision, Karpathy and Li [33] use a Convolutional Neural Network (CNN) [22], a deep neural network as the $\phi$ and an RNN as the $\psi$, to jointly embed both image and text into the same vector space for labeling images [33].

## 3 A DEEP NEURAL NETWORK FOR CODE SEARCH

Inspired by existing joint embedding techniques [21, 22, 33, 78], we propose a novel deep neural network named CODEnn (Code-Description Embedding Neural Network) for the code search problem. Figure 3 illustrates the key idea. Natural language queries and code snippets are heterogeneous and cannot be easily matched according to their lexical tokens. To bridge the gap, CODEnn jointly embeds code snippets and natural language descriptions into a unified vector space so that a query and the corresponding code snippets are embedded into nearby vectors and can be matched by vector similarities.

### 3.1 Architecture

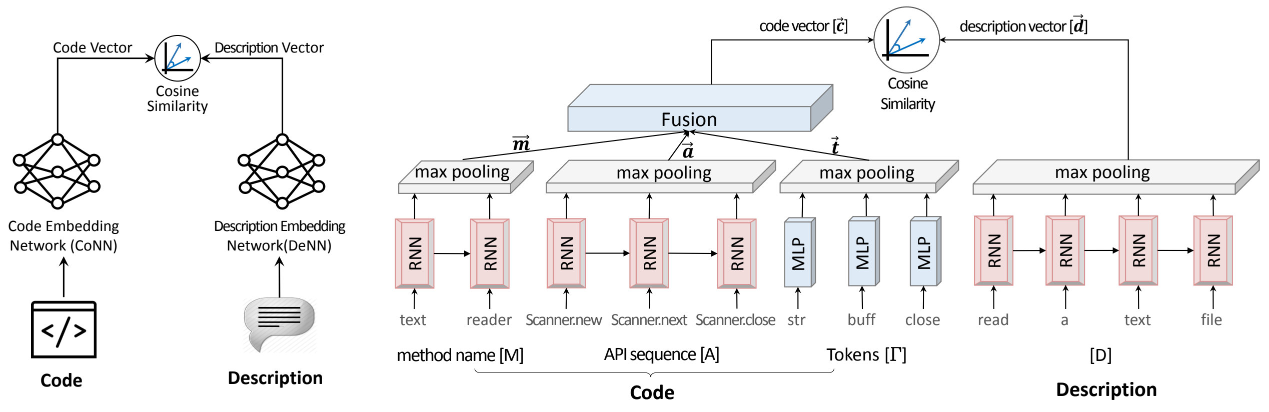
As introduced in Section 2.3, a joint embedding model requires three components: the embedding functions $\phi$:$\mathcal{X}\rightarrow\mathbb{R}^d$ and $\psi$:$\mathcal{Y}\rightarrow\mathbb{R}^d$, as well as the similarity measure $J(\cdot,\cdot)$. CODEnn realizes these components with deep neural networks.

Figure 4 shows the overall architecture of CODEnn. The neural network consists of three modules, each corresponding to a component of joint embedding:

- a code embedding network (CoNN) to embed source code into vectors.
- a description embedding network (DeNN) to embed natural language descriptions into vectors.
- a similarity module that measures the degree of similarity between code and descriptions.

The following subsections describe the detailed design of these modules.

*3.1.1 Code Embedding Network.* The code embedding network embeds source code into vectors. Source code is not simply plain text. It contains multiple aspects of information such as tokens, control flows and APIs [46]. In our model, we consider three aspects



(a) **Overall Architecture** (b) **Detailed Structure**

**Figure 4: The structure of the Code-Description Embedding Neural Network**

of source code: the method name, the API invocation sequence, and the tokens contained in the source code. They are commonly used in existing code search approaches [19, 27, 41, 44, 45]. For each code snippet (at the method level), we extract these three aspects of information. Each is embedded individually and then combined into a single vector representing the entire code.

Consider an input code snippet $C$=[$M, A, \Gamma$], where $M$=$w_1$,..., $w_{N_M}$ is the method name represented as a sequence of $N_M$ camel split tokens [1]; $A$=$a_1, ..., a_{N_A}$ is the API sequence with $N_A$ consecutive API method invocations, and $\Gamma$=$\{\tau_1, ..., \tau_{N_\Gamma}\}$ is the set of tokens in the snippet. The neural network embeds the three aspects as follows: for the method name $M$, it embeds the sequence of camel split tokens using an RNN with maxpooling:

$$\begin{aligned} \boldsymbol{h}_t &= \tanh(\boldsymbol{W}^M[\boldsymbol{h}_{t-1}; \boldsymbol{w}_t]), \forall t = 1, 2, \ldots, N_M \\ \boldsymbol{m} &= \text{maxpooling}([\boldsymbol{h}_1, \ldots, \boldsymbol{h}_{N_M}]) \end{aligned} \tag{5}$$

where $\boldsymbol{w}_t \in \mathbb{R}^d$ is the embedding vector of token $w_t$, [$a$;$b$]$\in\mathbb{R}^{2d}$ represents the concatenation of two vectors, $\boldsymbol{W}^M \in \mathbb{R}^{2d \times d}$ is the matrix of trainable parameters in the RNN, *tanh* is the activation function of the RNN. A method name is thus embedded as a $d$-dimensional vector $\boldsymbol{m}$.

Likewise, the API sequence $A$ is embedded into a vector $\boldsymbol{a}$ using an RNN with maxpooling:

$$\begin{aligned} \boldsymbol{h}_t &= \tanh(\boldsymbol{W}^A[\boldsymbol{h}_{t-1}; \boldsymbol{a}_t]), \forall t = 1, 2, \ldots, N_A \\ \boldsymbol{a} &= \text{maxpooling}([\boldsymbol{h}_1, \ldots, \boldsymbol{h}_{N_A}]) \end{aligned} \tag{6}$$

where $\boldsymbol{a}_t \in \mathbb{R}^d$ is the embedding vector of API $a_t$, $\boldsymbol{W}^A$ is the matrix of trainable parameters in the RNN.

For the tokens $\Gamma$, as they have no strict order in the source code, they are simply embedded via a multilayer perceptron (MLP), i.e., the conventional fully connected layer [52]:

$$\boldsymbol{h}_i = \tanh(\boldsymbol{W}^\Gamma \boldsymbol{\tau}_i), \forall i = 1, 2, \ldots, N_\Gamma \tag{7}$$

where $\boldsymbol{\tau}_i \in \mathbb{R}^d$ represents the embedded representation of the token $\tau_i$, $\boldsymbol{W}^\Gamma$ is the matrix of trainable parameters in the MLP, $\boldsymbol{h}_i, i$=1, ..., $N_\Gamma$ are the embedding vectors of all individual tokens. The individual vectors are also summarized to a single vector $\boldsymbol{t}$ via maxpooling:

$$\boldsymbol{t} = \text{maxpooling}([\boldsymbol{h}_1, \ldots, \boldsymbol{h}_{N_\Gamma}]) \tag{8}$$

Finally, the vectors of the three aspects are fused into one vector through a fully connected layer:

$$\boldsymbol{c} = \tanh(\boldsymbol{W}^C[\boldsymbol{m}; \boldsymbol{a}; \boldsymbol{t}]) \tag{9}$$

where [$a$;$b$;$c$] represents the concatenation of three vectors, $\boldsymbol{W}^C$ is the matrix of trainable parameters in the MLP. The output vector $\boldsymbol{c}$ represents the final embedding of the code snippet.

### 3.1.2 *Description Embedding Network.*

The description embedding network (DeNN) embeds natural language descriptions into vectors. Consider a description $D$=$w_1, ..., w_{N_D}$ comprising a sequence of $N_D$ words. DeNN embeds it into a vector $\boldsymbol{d}$ using an RNN with maxpooling:

$$\begin{aligned} \boldsymbol{h}_t &= \tanh(\boldsymbol{W}^D[\boldsymbol{h}_{t-1}; \boldsymbol{w}_t]), \forall t = 1, 2, \ldots, N_D \\ \boldsymbol{d} &= \text{maxpooling}([\boldsymbol{h}_1, \ldots, \boldsymbol{h}_{N_D}]) \end{aligned} \tag{10}$$

where $\boldsymbol{w}_t \in \mathbb{R}^d$ represents the embedded representation of the description word $w_t$, $\boldsymbol{W}^D$ is the matrix of trainable parameters in the RNN, $\boldsymbol{h}_t, t$=1, ...$N_D$ are the hidden states of the RNN.

### 3.1.3 *Similarity Module.*

We have described the transformations that map the code and description into vectors (i.e., the $\boldsymbol{c}$ and $\boldsymbol{d}$). Since we want the vectors of code and description to be jointly embedded, we measure the similarity between the two vectors.

We use the cosine similarity for the measurement, which is defined as:

$$\cos(\boldsymbol{c}, \boldsymbol{d}) = \frac{\boldsymbol{c}^T \boldsymbol{d}}{\|\boldsymbol{c}\| \|\boldsymbol{d}\|} \tag{11}$$

where $\boldsymbol{c}$ and $\boldsymbol{d}$ are the vectors of code and a description respectively. The higher the similarity, the more related the code is to the description.

Overall, CODEnn takes a ⟨code, description⟩ pair as input and predicts their cosine similarity cos($\boldsymbol{c}, \boldsymbol{d}$).

## 3.2 Model Training

Now we present how to train the CODEnn model to embed both code and descriptions into a unified vector space. The high-level goal of the joint embedding is: if a code snippet and a description have similar semantics, their embedded vectors should be close to each other. In other words, given an arbitrary code snippet $C$ and an arbitrary description $D$, we want it to predict a high similarity if $D$ is a correct description of $C$, and a little similarity otherwise.

At training time, we construct each training instance as a triple $\langle C, D+, D-\rangle$: for each code snippet $C$, there is a positive description $D+$ (a correct description of $C$) as well as a negative description (an incorrect description of $C$) $D-$ randomly chosen from the pool of all $D+$'s. When trained on the set of $\langle C, D+, D-\rangle$ triples, the CODEnn predicts the cosine similarities of both $\langle C, D+\rangle$ and $\langle C, D-\rangle$ pairs and minimizes the ranking loss [18, 22]:

$$\mathcal{L}(\theta) = \sum_{<C, D+, D->\in P} max(0, \epsilon - cos(\boldsymbol{c}, \boldsymbol{d}+) + cos(\boldsymbol{c}, \boldsymbol{d}-)) \quad (12)$$

where $\theta$ denotes the model parameters, $P$ denotes the training dataset, $\epsilon$ is a constant margin. $\boldsymbol{c}$, $\boldsymbol{d}+$ and $\boldsymbol{d}-$ are the embedded vectors of $C$, $D+$ and $D-$, respectively. A small, fixed $\epsilon$ value of 0.05 is used in all the experiments. Intuitively, the ranking loss encourages the cosine similarity between a code snippet and its correct description to go up, and the cosine similarities between a code snippet and incorrect descriptions to go down.

## 4 DEEPCS: DEEP LEARNING BASED CODE SEARCH

In this section, we describe DeepCS, a code search tool based on the proposed CODEnn model. DeepCS recommends top K most relevant code snippets for a given natural language query. Figure 5 shows the overall architecture. It includes three main phases: offline training, offline code embedding, and online code search.

We begin by collecting a large-scale corpus of code snippets, i.e., Java methods with corresponding descriptions. We extract sub-elements (including method names, tokens, and API sequences) from the methods. Then, we use the corpus to train the CODEnn model (the offline training phase). For a given codebase from which users would like to search for code snippets, DeepCS extracts code elements for each Java method in the search codebase, and computes a code vector using the CoNN module of the trained CODEnn model (the offline embedding phase). Finally, when a user query arrives, DeepCS first computes the vector representation of the query using the DeNN module of the CODEnn model, and then returns code snippets whose vectors are close to the query vector (the online code search phase).

In theory, our approach could search for source code written in any programming languages. In this paper, we limit our scope to the Java code. The following sections describe the detailed steps of our approach.

### 4.1 Collecting Training Corpus

As described in Section 3, the CODEnn model requires a large-scale training corpus that contains code elements and the corresponding descriptions, i.e., the ⟨method name, API sequence, tokens, description⟩ tuples. Figure 6 shows an excerpt of the training corpus.

We build the training tuples using Java methods that have documentation comments[1] from open-source projects on GitHub [3]. For each Java method, we use the method declaration as the code element and the first sentence of its documentation comment as its natural language description. According to the Javadoc guidance[2], the first sentence is usually a summary of a method. To prepare the data, we download Java projects from GitHub created from August, 2008 to June, 2016. To remove toy or experimental programs, we exclude any projects without a star. We select only the Java methods that have documentation comments from the downloaded projects. Finally, we obtain a corpus comprising 18,233,872 commented Java methods.

Having collected the corpus of commented code snippets, we extract the ⟨method name, API sequence, tokens, description⟩ tuples as follows:

**Method Name Extraction:** For each Java method, we extract its name and parse the name into a sequence of tokens according to camel case [1]. For example, the method name *listFiles* will be parsed into the tokens *list* and *files*.

**API Sequence Extraction:** We extract an API sequence from each Java method using the same procedures as described in DeepAPI [27] – parsing the AST using the Eclipse JDT compiler [2] and traversing the AST. The API sequences are produced as follows [27]:

- For each constructor invocation *new C*(), we produce *C.new* and append it to the API sequence.
- For each method call $o.m()$ where $o$ is an instance of class $C$, we produce $C.m$ and append it to the API sequence.
- For a method call passed as a parameter, we append the method before the calling method. For example, $o_1.m_1(o_2.m_2(),o_3.m_3())$, we produce a sequence $C_2.m_2$-$C_3.m_3$-$C_1.m_1$, where $C_i$ is the class of the instance $o_i$.
- For a sequence of statements $s_1; s_2;...;s_N$, we extract the API sequence $a_i$ from each statement $s_i$, concatenate them to the API sequence $a_1$-$a_2$-...-$a_N$.
- For conditional statements such as if($s_1$){$s_2$;}else{$s_3$;}, we create a sequence from all possible branches, that is, $a_1$-$a_2$-$a_3$, where $a_i$ is the API sequence extracted from the statement $s_i$.
- For loop statements such as while($s_1$){$s_2$;}, we produce a sequence $a_1$-$a_2$, where $a_1$ and $a_2$ are API sequences extracted from the statement $s_1$ and $s_2$, respectively.

**Token Extraction:** To collect tokens from a Java method, we tokenize the method body, split each token according to camel case [1], and remove the duplicated tokens. We also remove stop words (such as *the* and *in*) and Java keywords as they frequently occur in source code and are not discriminative.

**Description Extraction:** To extract the documentation comment, we use the Eclipse JDT compiler [2] to parse the AST from a Java method and extract the *JavaDoc Comment* from the AST.

Figure 7 shows an example of code elements and documentation comments extracted from a Java method $DateUtils.toCalendar$[3] in the *Apache commons-lang* library.

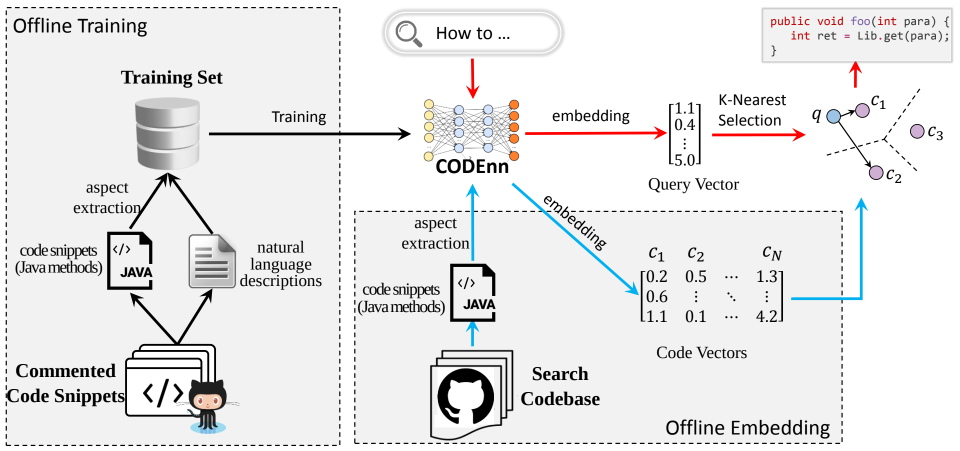
### 4.2 Training CODEnn Model

We use the large-scale corpus described in the previous section to train the CODEnn model, following the method described in Section 3.2.

[1]A documentation comment in JAVA starts with slash-asterisk-asterisk (/**) and ends with asterisk-slash (*/)

[2]http://www.oracle.com/technetwork/articles/java/index-137868.html

[3]https://github.com/apache/commons-lang/blob/master/src/main/java/org/apache/commons/lang3/time/DateUtils.java



```
public void foo(int para) {
   int ret = Lib.get(para);
}
```

$$\begin{bmatrix}1.1\\0.4\\\vdots\\5.0\end{bmatrix}$$

$$\begin{array}{cccc} c_1 & c_2 & & c_N \\ \end{array}$$
$$\begin{bmatrix}0.2 & 0.5 & \cdots & 1.3\\0.6 & \vdots & \ddots & \vdots\\1.1 & 0.1 & \cdots & 4.2\end{bmatrix}$$

**Figure 5: The overall workflow of DeepCS**

| | Method Name | API Sequence | Tokens | Description (English) |
|---|---|---|---|---|
| 1 | file reader | InputStream.read→OutputStream.write | input, output, stream, write | copy a file from an inputstream |
| 2 | open | URL.new→URL.openConnection | url, open, conn | open a url |
| 3 | test exists | File.new→File.exists | file, create, exists | test file exists |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

**Figure 6: An excerpt of training tuples**

```
/**
 * Converts a Date into a Calendar.
 * @param date the date to convert to a Calendar
 * @return the created Calendar
 * @throws NullPointerException if null is passed in
 * @since 3.0
 */
public static Calendar toCalendar(final Date date) {
      final Calendar c = Calendar.getInstance();
      c.setTime(date);
      return c;
}
```

⇩

**Method Name:** to calendar
**API sequence:** Calendar.getInstance⟶Calendar.setTime
**Tokens:** calendar, get, instance, set, time, date
**Description:** converts a date into a calendar.

**Figure 7: An example of extracting code elements from a Java method *DateUtils.toCalendar*[3]**

The detailed implementation of the CODEnn model is as follows: we use the bi-directional LSTM [70], a state-of-the-art kind of RNN for the RNN implementation. All LSTMs have 200 hidden units in each direction. We set the dimension of word embedding to 100. The CODEnn has two types of MLPs, the embedding MLP for embedding individual tokens and the fusion MLP to combine the embeddings of different aspects. We set the number of hidden units as 100 for the embedding MLP and 400 for the fusion MLP.

The CODEnn model is trained via the mini-batch Adam algorithm [37, 40]. We set the batch size (i.e., the number of instances per batch) as 128. For training the neural networks, we limit the size of the vocabulary to 10,000 words that are most frequently used in the training dataset.

We build our model on Keras [4] and Theano [6], two open-source deep learning frameworks. We train our models on a server with one Nvidia K40 GPU. The training lasts ~50 hours with 500 epochs.

## 4.3 Searching Code Snippets

Given a user's free-text query, DeepCS returns the relevant code snippets through the trained CODEnn model. It first computes the code vector for each code snippet (i.e., a Java method) in the search codebase. Then, it selects and returns the code snippets that have the top K nearest vectors to the query vector.

More specially, before a search starts, DeepCS embeds all code snippets in the codebase into vectors using the trained CoNN module of CODEnn in an off-line manner. During the on-line search, when a developer enters a natural language query, DeepCS first embeds the query into a vector using the trained DeNN module of CODEnn. Then, it estimates the cosine similarities between the query vector and all code vectors using Equation 11. Finally, the top K code snippets whose vectors are most similar to the query vector are returned as the search results. $K$ is set to 10 in our experiments.

## 5 EVALUATION

In this section, we evaluate DeepCS through experiments. We also compare DeepCS with related code search approaches.

## 5.1 Experimental Setup

*5.1.1 Search Codebase.* To better evaluate DeepCS, our experiments are performed over a search codebase, which is different from the training corpus. Code snippets that match a user query are retrieved from the search codebase. In practice, the search codebase could be an organization's local codebase or any codebase created from open source projects.

To construct the search codebase, we choose the Java projects that have at least 20 stars in GitHub. Different from the training corpus, they are considered in isolation and contain all code (including those do not have Javadoc comments). There are 9,950 projects in total. We select all 16,262,602 methods from these projects. For each Java method, we extract a ⟨method name, API sequence, tokens⟩ triple to generate its code vector.

*5.1.2 Query Subjects.* To select code search queries for the evaluation, we adopt a systematic procedure used in [41][4]. We build a benchmark of queries from the top 50 voted Java programming questions in Stack Overflow. To achieve so, we browse the list of Java-tagged questions in Stack Overflow and sort them according to the votes that each one receives[5]. We manually check the sorted list sequentially, and add questions that satisfy the following conditions to the benchmark:
(1) The question is a concrete Java programming task. We exclude questions about problems, knowledge, configurations, experience and questions whose descriptions are vague and abstract. For example, *Failed to load the JNI Library*, *What is the difference between StringBuilder and StringBuffer?*, and *Why does Java have transient fields?*. (2) The accepted answer to the question contains a Java code snippet. (3) The question is not a duplicate of the previous questions. We filter out questions that are tagged as "duplicated".

The full list of the 50 selected queries can be found in Table 1. For each query, two developers manually inspect the top 10 results returned by DeepCS and label their relevance to the query. Then they discuss the inconsistent labels and relabel them. The procedure repeats until a consensus is reached.

*5.1.3 Performance Measure.* We use four common metrics to measure the effectiveness of code search, namely, FRank, SuccessRate@k, Precision@k, and Mean Reciprocal Rank (MRR). They are widely used metrics in information retrieval and the code search literature [41, 45, 62, 79].

The FRank (also known as *best hit rank* [41]) is the rank of the first hit result in the result list [62]. It is important as users scan the results from top to bottom. A smaller FRank implies lower inspection effort for finding the desired result. We use FRank to assess the effectiveness of a single code search query.

The *SuccessRate@k* (also known as *success percentage at k* [41]) measures the percentage of queries for which more than one correct result could exist in the top $k$ ranked results [35, 41, 79]. In our evaluations it is calculated as follows:

$$\text{SuccessRate@k} = \frac{1}{|Q|}\sum_{q=1}^{Q}\delta(\text{FRank}_q \leq k) \tag{13}$$

where $Q$ is a set of queries, $\delta(\cdot)$ is a function which returns 1 if the input is true and 0 otherwise. *SuccessRate@k* is important because a better code search engine should allow developers to discover the needed code by inspecting fewer returned results. The higher the metric value, the better the code search performance.

The *Precision@k* [45, 57] measures the percentage of relevant results in the top $k$ returned results for each query. In our evaluations it is calculated as follows:

$$\text{Precision@k} = \frac{\#\text{relevant results in the top k results}}{k} \tag{14}$$

*Precision@k* is important because developers often inspect multiple results of different usages to learn from [62]. A better code search engine should allow developers to inspect less noisy results. The higher the metric values, the better the code search performance. We evaluate *SuccessRate@k* and *Precision@k* when $k$'s value is 1, 5, and 10. These values reflect the typical sizes of results that users would inspect [41].

The MRR [45, 79] is the average of the reciprocal ranks of results of a set of queries $Q$. The reciprocal rank of a query is the inverse of the rank of the first hit result [26]. MRR is calculated as follows:

$$MRR = \frac{1}{|Q|}\sum_{q=1}^{|Q|}\frac{1}{FRank_q} \tag{15}$$

The higher the MRR value, the better the code search performance.

*5.1.4 Comparison Methods.* We compare the effectiveness of our approach with CodeHow [45] and a conventional Lucene-based code search tool [5].

CodeHow is a state-of-the-art code search engine proposed recently. It is an information retrieval based code search tool that incorporates an extended Boolean model and API matching. It first retrieves relevant APIs to a query by matching the query with the API documentation. Then, it searches code by considering both plain code and the related APIs. Like DeepCS, CodeHow also considers multiple aspects of source code such as method name and APIs. It combines multiple aspects using an Extended Boolean Model [45]. The facts that CodeHow also considers APIs and is also built for large-scale code search make it an ideal baseline for our experiments.

Lucene is a popular, conventional text search engine behind many existing code search tools such as Sourcerer [43]. Sourcerer combines Lucene with code properties such as FQN (full qualified name) of entities and code popularity to retrieve the code snippets. In our implementation of the Lucene-based code search tool, we consider the heuristic of FQN. We did not include the code popularity heuristic (computed using PageRank) as it does not significantly improve the code search performance [43].

We use the same experimental setting for CodeHow and the Lucene-based tool as used for evaluating DeepCS.

## 5.2 Results

Table 1 shows the evaluation results of DeepCS and related approaches for each query in the benchmark. The column Question ID shows the original ID of the question in Stack Overflow where the query comes from. The column FRank shows the FRank result of each approach. The symbol 'NF' stands for *Not Found* which means that no relevant result has been returned within the top $K$ results ($K$=10).

The results show that DeepCS produces generally more relevant results than Lucene and CodeHow. Figure 8a shows the statistical summary of FRank for the three approaches. The symbol '+' indicates the average FRank value achieved by each approach. We conservatively treat the FRank as 11 for queries that fail to obtain relevant results within the top 10 returned results. We observe that DeepCS achieves more relevant results with an average FRank of

[4]http://taoxie.cs.illinois.edu/racs/subjects.html
[5]http://stackoverflow.com/questions/tagged/java?sort=votes&pagesize=15

**Table 1: Benchmark Queries and Evaluation Results (NF: Not Found within the top 10 returned results LC:Lucene CH:CodeHow DCS:DeepCS)**

| No. | Question ID | Query | FRank | | |
|---|---|---|---|---|---|
| | | | LC | CH | DCS |
| 1 | 309424 | convert an inputstream to a string | 2 | 1 | 1 |
| 2 | 157944 | create arraylist from array | NF | NF | 2 |
| 3 | 1066589 | iterate through a hashmap | NF | 4 | 1 |
| 4 | 363681 | generating random integers in a specific range | NF | 6 | 2 |
| 5 | 5585779 | converting string to int in java | NF | 10 | 1 |
| 6 | 1005073 | initialization of an array in one line | NF | 4 | 1 |
| 7 | 1128723 | how can I test if an array contains a certain value | 6 | 6 | 1 |
| 8 | 604424 | lookup enum by string value | 1 | NF | 10 |
| 9 | 886955 | breaking out of nested loops in java | NF | NF | NF |
| 10 | 1200621 | how to declare an array | NF | NF | 4 |
| 11 | 41107 | how to generate a random alpha-numeric string | NF | 1 | 1 |
| 12 | 409784 | what is the simplest way to print a java array | 6 | NF | 1 |
| 13 | 109383 | sort a map by values | NF | 1 | 3 |
| 14 | 295579 | fastest way to determine if an integer's square root is an integer | NF | NF | NF |
| 15 | 80476 | how can I concatenate two arrays in java | NF | 1 | 1 |
| 16 | 326369 | how do I create a java string from the contents of a file | 8 | NF | 5 |
| 17 | 1149703 | how can I convert a stack trace to a string | 3 | 1 | 2 |
| 18 | 513832 | how do I compare strings in java | 1 | 3 | 1 |
| 19 | 3481828 | how to split a string in java | 1 | 1 | 1 |
| 20 | 2885173 | how to create a file and write to a file in java | 2 | 1 | NF |
| 21 | 507602 | how can I initialise a static map | 7 | 1 | 2 |
| 22 | 223918 | iterating through a collection, avoiding concurrentmodificationexception when removing in loop | 3 | 3 | 2 |
| 23 | 415953 | how can I generate an md5 hash | 1 | 3 | 6 |
| 24 | 1069066 | get current stack trace in java | 3 | 1 | 1 |
| 25 | 2784514 | sort arraylist of custom objects by property | 1 | 1 | 1 |
| 26 | 153724 | how to round a number to n decimal places in java | 1 | 1 | 4 |
| 27 | 473282 | how can I pad an integers with zeros on the left | NF | 3 | 1 |
| 28 | 529085 | how to create a generic array in java | NF | NF | 3 |
| 29 | 4716503 | reading a plain text file in java | 4 | NF | 7 |
| 30 | 1104975 | a for loop to iterate over enum in java | NF | NF | NF |
| 31 | 3076078 | check if at least two out of three booleans are true | NF | NF | NF |
| 32 | 4105331 | how do I convert from int to string | 2 | 1 | NF |
| 33 | 8172420 | how to convert a char to a string in java | 5 | 10 | 3 |
| 34 | 1816673 | how do I check if a file exists in java | 1 | 2 | 1 |
| 35 | 4216745 | java string to date conversion | 6 | NF | 1 |
| 36 | 1264709 | convert inputstream to byte array in java | 7 | 5 | 1 |
| 37 | 1102891 | how to check if a string is numeric in java | 1 | NF | 2 |
| 38 | 869033 | how do I copy an object in java | 2 | 1 | 1 |
| 39 | 180158 | how do I time a method's execution in java | NF | NF | 2 |
| 40 | 5868369 | how to read a large text file line by line using java | 1 | 1 | 1 |
| 41 | 858572 | how to make a new list in java | 2 | 1 | 1 |
| 42 | 1625234 | how to append text to an existing file in java | 3 | 1 | 1 |
| 43 | 2201925 | converting iso 8601-compliant string to date | 3 | 1 | 1 |
| 44 | 122105 | what is the best way to filter a java collection | NF | 9 | 2 |
| 45 | 5455794 | removing whitespace from strings in java | NF | 3 | 1 |
| 46 | 225337 | how do I split a string with any whitespace chars as delimiters | 1 | 1 | 2 |
| 47 | 52353 | in java, what is the best way to determine the size of an object | NF | NF | NF |
| 48 | 160970 | how do I invoke a java method when given the method name as a string | 3 | 1 | 2 |
| 49 | 207947 | how do I get a platform dependent new line character | 1 | NF | 10 |
| 50 | 1026723 | how to convert a map to list in java | 6 | NF | 1 |

3.5, which is smaller than the average FRank achieved by CodeHow (5.5) and Lucene (6.0). The FRank values of DeepCS concentrate on the range from 1 to 4, while CodeHow and Lucene produce larger variance and many less relevant results. Figure 8b, 8c and 8d show the statistics of Precision@k for the three approaches when *k* is 1, 5 and 10, respectively. We observe that DeepCS achieves better overall precision values than CodeHow and the Lucene-based tool.

To test the statistical significance, we apply the Wilcoxon signed-rank test ($p<0.05$) for the comparison of FRank and Precision@k between DeepCS and the two related approaches for all the queries. We conservatively treat the FRank as 11 for queries that fail to obtain relevant results within the top 10 returned results. The p-values for the comparisons of DeepCS with Lucene and CodeHow are all less than 0.05, indicating the statistical significance of the improvement of DeepCS over the related approaches.

Table 2 shows the overall performance of the three approaches, measured in terms of SuccessRate@k, Precision@k and MRR. The columns R@1, R@5 and R@10 show the results of SuccessRate@k when *k* is 1, 5 and 10, respectively. The columns P@1, P@5 and P@10 show the results of the average Precision@k over all queries when *k* is 1, 5 and 10, respectively. The column MRR shows the MRR values of the three approaches. The results show that DeepCS returns more relevant code snippets than CodeHow and Lucene. For example, the R@5 value is 0.76, which means that for 76% of the queries, the relevant code snippets can be found within the top 5 return results. The P@5 value is 0.5, which means that 50% of the top 5 results are deemed accurate. For the SuccessRate@k, the improvements to CodeHow are 21%, 31% and 30%, respectively. For the Precision@k, the improvements to CodeHow are 21%, 72% and 75%, respectively. For the MRR, the improvement to CodeHow is 33%. Overall, our approach improves the accuracy of related techniques on all metrics.

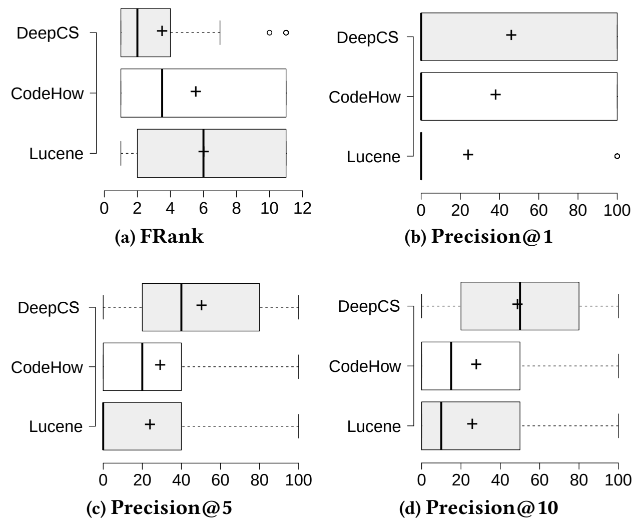


**Figure 8: The statistical comparison of FRank and Precison@k for three code search approaches**

**Table 2: Overall Accuracy of DeepCS and the Related Approaches**

| Tool | R@1 | R@5 | R@10 | P@1 | P@5 | P@10 | MRR |
|---|---|---|---|---|---|---|---|
| Lucene | 0.24 | 0.48 | 0.62 | 0.24 | 0.24 | 0.26 | 0.35 |
| CodeHow | 0.38 | 0.58 | 0.66 | 0.38 | 0.29 | 0.28 | 0.45 |
| DeepCS | 0.46 | 0.76 | 0.86 | 0.46 | 0.50 | 0.49 | 0.60 |

## 5.3 Examples of Code Search Results

We now provide concrete examples of code search results that demonstrate the advantages of DeepCS.

Figure 9a and 9b show the results for two queries: *queue an event to be run on the thread* and *run an event on a thread queue*. The two queries have the same set of keywords with different word sequences. The keyword *queue* in the two queries have different meanings and it could be difficult for an IR-based approach to distinguish. Still, DeepCS can understand the meaning of the two queries and return relevant snippets. Apparently, DeepCS has the ability to recognize query semantics.

The ability of query understanding enables DeepCS to perform a more robust code search. Its search results are less affected by

```java
public boolean enqueue(EventHandler handler, Event event) {
    synchronized(monitor) {
        ......
        handlers[tail] = handler;
        events[tail] = event;
        tail++;
        if (handlers.length <= tail)
            tail = 0;
        monitor.notify();
    }
    return true;
}
```

**(a) The third result of the query "queue an event to be run on the thread"**

```java
public void run() {
    while (!stop) {
        DynamicModelEvent evt;
        while ((evt = eventQueue.poll()) != null) {
            for (DynamicModelListener l: listeners.toArray(
                                            new DynamicModelListener[0]))
                l.dynamicModelChanged(evt);
        }
        ......
    }
}
```

**(b) The first result of the query "run an event on the thread queue"**

**Figure 9: Examples showing the query understanding**

irrelevant or noisy keywords. For example, the query *get the content of an input stream as a string using a specified character encoding* contains 9 keywords. CodeHow returns many snippets that are related to less relevant keywords such as *specified* and *character*. DeepCS, on the other hand, can successfully identify the importance of different keywords and understand the key point of the query (Figure 10).

Another advantage of DeepCS relates to associative search. That is, it not only seeks snippets with matched keywords but also recommends those without matched keywords but are semantically related. This is important because it significantly increases the search scope especially when the codebase is small. Besides, developers need snippets of multiple usages [62]. The associative search provides more options of code snippets for developers to learn from. Figure 11a shows the first result of the query *read an object from an xml file*. As discussed in Section 1, traditional IR-based approaches may only match snippets that contain keywords such as *xml*, *object* and *read*. However, as shown in the figure, DeepCS successfully recognizes the query semantic and returns results of *xml deserialize*, even the keywords do not exist in the result. By contrast, CodeHow only returns snippets containing *read*, *object* and *xml*, narrowing down the search scope. The example indicates that DeepCS searches code by understanding the semantics instead of just matching keywords. Similarly, the query *initialization of an arraylist in one line* in Table 1 returns snippets containing "new ArrayList⟨⟩" although the snippet does not include the keyword *initialization*. Figure 11b shows another example of the associative search. When searching *play a song*. DeepCS not only returns snippets with matching keywords but also recommends results with semantically related words such as *audio* and *voice*.

```java
public static String toStringWithEncoding(
            InputStream inputStream, String encoding) {
    if (inputStream == null)
        throw new IllegalArgumentException(
                "inputStream-should-not-be-null");
    char[] buffer = new char[BUFFER_SIZE];
    StringBuffer stringBuffer = new StringBuffer();
    BufferedReader bufferedReader = new BufferedReader(
            new InputStreamReader(inputStream, encoding), BUFFER_SIZE);
    int character = -1;
    ......
    return stringBuffer.toString();
}
```

**Figure 10: An example showing the search robustness – The first result of the query "get the content of an input stream as a string using a specified character encoding"**

```java
public static < S > S deserialize(Class c, File xml) {
    try {
        JAXBContext context = JAXBContext.newInstance(c);
        Unmarshaller unmarshaller = context.createUnmarshaller();
        S deserialized = (S) unmarshaller.unmarshal(xml);
        return deserialized;
    } catch (JAXBException ex) {
        log.error("Error-deserializing-object-from-XML", ex);
        return null;
    }
}
```

**(a) The first result of the query "read an object from an xml file"**

```java
public void playVoice(int clearedLines) throws Exception {
    int audiosAvailable = audioLibrary.get(clearedLines).size();
    int audioIndex = rand.nextInt(audiosAvailable);
    audioLibrary.get(clearedLines).get(audioIndex).play();
}
```

**(b) The second result of the query "play a song"**

**Figure 11: Examples showing the associative search**

## 6 DISCUSSIONS

### 6.1 Why does DeepCS Work?

We have identified three advantages of DeepCS that may explain its effectiveness in code search:

**A unified representation of heterogeneous data** Source code and natural language queries are heterogeneous. By jointly embedding source code and natural language query into the same vector representation, their similarities can be measured more accurately.

**Better query understanding through deep learning** Unlike traditional techniques, DeepCS learns queries and source code representations with deep learning. Characteristics of queries, such as semantically related words and word orders, are considered in these models [27]. Therefore, it can recognize the semantics of query and code better. For example, it can distinguish the query *queue an event to be run on the thread* from the query *run an event on the event queue*.

**Clustering snippets by natural language semantics** An advantage of our approach is that it embeds semantically similar code snippets into vectors that are close to each other. Semantically similar code snippets are grouped according to their semantics. Therefore, in addition to the exact matching snippets, DeepCS also recommends the semantically related ones.

### 6.2 Limitation of DeepCS

Despite the advantages such as associative search, DeepCS could still return inaccurate results. It sometimes ranks partially relevant results higher than the exact matching ones. Figure 12 shows the result for the query *generate md5*. The exactly matching result is

```java
public static byte[] generateRandom256() {
    byte[] randomSeed1 = ByteUtils.longToBytes(System.nanoTime());
    byte[] randomSeed2 = (new SecureRandom()).generateSeed(KEY_SIZE_BYTES);
    byte[] bh1 = ByteUtils.concatenate(randomSeed1, randomSeed2);
    Thread.sleep(100L);
    byte[] randomSeed3 = UUID.randomUUID().toString().getBytes();
    byte[] randomSeed4 = ByteUtils.longToBytes(System.nanoTime());
    byte[] bh2 = ByteUtils.concatenate(randomSeed3, randomSeed4);
    return simpleHash256(ByteUtils.concatenate(bh1, bh2));
}
```

**Figure 12: An example showing the inaccurate results – The first result of the query "generate md5"**

ranked 7 in the result list, while partially related results such as *generate checksum* are recommended before the exact results. This is because DeepCS ranks results by just considering their semantic vectors. In future work, more code features (such as programming context) [58] could be considered in our model to further adjust the results.

### 6.3 Threats to Validity

Our goal is to improve the performance of code search over GitHub, thus both training and search are performed over GitHub corpus. There is a threat of overlap between the training and search codebases. To mitigate this threat, in our experiments, the training and search codebases are constructed to be significantly different. The training codebase only contains code that has corresponding descriptions, while the search codebase is considered in isolation and contains all code (including those do not have descriptions). We believe the threat of overfitting for this overlap is not significant as our training codebase considers a vast majority of code in Github. The most important goal of our experiments is to evaluate DeepCS in a real-world code search scenario. For that, we used 50 real queries collected from Stack Overflow to test the effectiveness of DeepCS. These queries are not descriptions/comments of Java methods and are not used for training.

In our experiments, the relevancy of returned results were manually graded and could suffer from subjectivity bias. To mitigate this threat, (i) the manual analysis was performed independently by two developers and (ii) the developers performed an open discussion to resolve conflict grades for the 50 questions. In the future, we will further mitigate this threat by inviting more developers for the grading.

In the grading of relevancy, we consider only the top 10 results. Queries that fail are identically assigned with an FRank of 11 and could be biased from the real relevancy of code snippets. However, we believe that the setting is reasonable. In real-world code search, developers usually inspect the top K results and ignore the remaining. That means it does not make much difference if a code snippet appears at rank 11 or 20 if K is 10.

Like related work (e.g., [14, 41]), we evaluate DeepCS with popular Stack Overflow questions. SO questions may not be representative to all possible queries for code search engines. To mitigate this threat, (i) DeepCS is not trained on SO questions but on large scale Github corpus. (ii) We select the most frequently asked questions which might be also commonly asked by developers in other search engines. In the future, we will extend the scale and scope of test queries.

## 7 RELATED WORK

### 7.1 Code Search

In code search, a line of work has investigated marrying state-of-the-art information retrieval and natural language processing techniques [13–15, 32, 35, 41, 45–47, 61, 81, 82]. Much of the existing work focuses on query expansion and reformulation [29, 31, 44]. For example, Hill et al. [30] reformulated queries with natural language phrasal representations of method signatures. Haiduc et al. [29] proposed to reformulate queries based on machine learning. Their method trains a machine learning model that automatically recommends a reformulation strategy based on the query properties. Lu et al. [44] proposed to extend a query with synonyms generated from WordNet. There is also much work that takes into account code characteristics. For example, McMillan et al. [47] proposed Portfolio, a code search engine that combines keyword matching with PageRank to return a chain of functions. Lv et al. [45] proposed CodeHow, a code search tool that incorporates an extended Boolean model and API matching. Ponzanelli et al. [61] proposed an approach that automatically retrieves pertinent discussions from Stack Overflow given a context in the IDE. Recently Li et al. [41] proposed RACS, a code search framework for JavaScript that considers relationships (e.g., sequencing, condition, and callback relationships) among the invoked API methods.

As described in Section 6, DeepCS differs from existing code search techniques in that it does not rely on information retrieval techniques. It measures the similarity between code snippets and user queries through joint embedding and deep learning. Thus, it can better understand code and query semantics.

As the keyword based approaches are inefficient on recognizing semantics, researchers have drawn increasing attention on semantics based code search [34, 65, 69]. For example, Reiss [65] proposed the semantics-based code search, which uses user specifications to characterize the requirement and uses transformations to adapt the searching results. However, Reiss's approach differs significantly from DeepCS. It does not consider the semantics of natural language queries. Furthermore, it requires users to provide not only natural language queries but also other specifications such as method declarations and test cases.

Besides code search, there have been many other information retrieval tasks in software engineering [8, 9, 16, 23, 24, 29, 51, 55, 63, 67] such as bug localization [66, 73, 80], feature localization [19], traceability links recovery [20] and community Question Answering [11]. Ye et al. [80] proposed to embed words into vector representations to bridge the lexical gap between source code and natural language for SE-related text retrieval tasks. Different from DeepCS, the vector representations learned by their method are at the level of individual words and tokens instead of the whole query sentences. Their method is based on a bag-of-words assumption, and word sequences are not considered.

### 7.2 Deep Learning for Source Code

Recently, researchers have investigated possible applications of deep learning techniques to source code [7, 38, 53, 56, 60, 64, 75, 76]. A typical use of deep learning is code generation [42, 54]. For example, Mou et al. [54] proposed to generate code from natural language user intentions using an RNN Encoder-Decoder model. Their results show the feasibility of applying deep learning techniques to code generation from a highly homogeneous dataset (simple programming assignments). Gu et al. [27] applies deep learning for API learning, that is, generating API usage sequences for a given natural language query. They also apply deep learning to migrate APIs between different programming languages [28]. Deep learning is also applied to code completion [64, 77]. For example, White et al. [77] applied the RNN language model to source code files and showed its effectiveness in predicting software tokens. Recently, White et al. [76] also applied deep learning to code clone detection. Their framework automatically links patterns mined at the lexical level with patterns mined at the syntactic level. In our work, we explore the application of deep learning to code search.

## 8 CONCLUSION

In this paper, we propose a novel deep neural network named CODEnn for code search. Instead of matching text similarity, CODEnn learns a unified vector representation of both source code and natural language queries so that code snippets semantically related to a query can be retrieved according to their vectors. As a proof-of-concept application, we implement a code search tool DeepCS based on the proposed CODEnn model. Our experimental study has shown that the proposed approach is effective and outperforms the related approaches. Our source code and data are available at: https://github.com/guxd/deep-code-search

In the future, we will investigate more aspects of source code such as control structures to better represent high-level semantics of source code. The deep neural network we designed may also benefit other software engineering problems such as bug localization.

## 9 ACKNOWLEDGMENT

The authors would like to thank Dongmei Zhang at Microsoft Research Asia for her support for this project and insightful comments on the paper.

## REFERENCES

[1] Camel case, https://en.wikipedia.org/wiki/camelcase.
[2] Eclipse JDT. http://www.eclipse.org/jdt/.
[3] Github. https://github.com.
[4] Keras. https://keras.io/.
[5] Lucene. https://lucene.apache.org/.
[6] Theano, http://deeplearning.net/software/theano/.
[7] M. Allamanis, H. Peng, and C. Sutton. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning (ICML)*, 2016.
[8] J. Anvik and G. C. Murphy. Reducing the effort of bug report triage: Recommenders for development-oriented decisions. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(3):10, 2011.
[9] A. Bacchelli, M. Lanza, and R. Robbes. Linking e-mails and source code artifacts. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 375–384. ACM, 2010.
[10] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.
[11] O. Barzilay, C. Treude, and A. Zagalsky. Facilitating crowd sourced software engineering via stack overflow. In *Finding Source Code on the Web for Remix and Reuse*, pages 289–308. Springer, 2013.
[12] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–82, 1994.
[13] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 513–522. ACM, 2010.
[14] B. A. Campbell and C. Treude. NLP2Code: Code snippet content assist via natural language tasks. *arXiv preprint arXiv:1701.05648*, 2017.
[15] W.-K. Chan, H. Cheng, and D. Lo. Searching connected API subgraph via text phrases. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 10. ACM, 2012.
[16] O. Chaparro and A. Marcus. On the reduction of verbose queries in text retrieval based software maintenance. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 716–718. ACM, 2016.
[17] K. Cho, B. Van Merriënboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, Oct. 2014. Association for Computational Linguistics.
[18] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011.
[19] C. S. Corley, K. Damevski, and N. A. Kraft. Exploring the use of deep learning for feature location. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 556–560. IEEE, 2015.
[20] B. Dagenais and M. P. Robillard. Recovering traceability links between an api and its learning resources. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 47–57. IEEE, 2012.
[21] M. Feng, B. Xiang, M. R. Glass, L. Wang, and B. Zhou. Applying deep learning to answer selection: A study and an open task. In *2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, pages 813–820. IEEE, 2015.
[22] A. Frome, G. S. Corrado, J. Shlens, S. Bengio, J. Dean, T. Mikolov, et al. DeViSE: A deep visual-semantic embedding model. In *Advances in neural information processing systems*, pages 2121–2129, 2013.
[23] X. Ge, D. C. Shepherd, K. Damevski, and E. Murphy-Hill. Design and evaluation of a multi-recommendation system for local code search. *Journal of Visual Languages & Computing*, 2016.
[24] G. Gousios, M. Pinzger, and A. v. Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355. ACM, 2014.
[25] A. Graves, M. Liwicki, S. Fernández, R. Bertolami, H. Bunke, and J. Schmidhuber. A novel connectionist system for unconstrained handwriting recognition. *IEEE transactions on pattern analysis and machine intelligence*, 31(5):855–868, 2009.
[26] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby. A search engine for finding highly relevant applications. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 475–484. IEEE, 2010.
[27] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep API learning. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE'16)*, 2016.
[28] X. Gu, H. Zhang, D. Zhang, and S. Kim. DeepAM: Migrate APIs with multi-modal sequence to sequence learning. In *Proceedings of the Twenty-Sixth International Joint Conferences on Artifical Intelligence (IJCAI'17)*, 2017.
[29] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies. Automatic query reformulations for text retrieval in software engineering. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 842–851. IEEE Press, 2013.
[30] E. Hill, L. Pollock, and K. Vijay-Shanker. Improving source code search with natural language phrasal representations of method signatures. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 524–527. IEEE Computer Society, 2011.
[31] E. Hill, M. Roldan-Vega, J. A. Fails, and G. Mallet. NL-based query refinement and contextualized code search results: A user study. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 34–43. IEEE, 2014.
[32] R. Holmes, R. Cottrell, R. J. Walker, and J. Denzinger. The end-to-end use of source code examples: An exploratory study. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 555–558. IEEE, 2009.
[33] A. Karpathy and L. Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3128–3137, 2015.
[34] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun. Repairing programs with semantic code search (T). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 295–306. IEEE, 2015.
[35] I. Keivanloo, J. Rilling, and Y. Zou. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*, pages 664–675. ACM, 2014.
[36] Y. Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.
[37] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
[38] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 476–481. IEEE, 2015.

[39] Q. Le and T. Mikolov. Distributed representations of sentences and documents. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 1188–1196, 2014.

[40] M. Li, T. Zhang, Y. Chen, and A. J. Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 661–670. ACM, 2014.

[41] X. Li, Z. Wang, Q. Wang, S. Yan, T. Xie, and H. Mei. Relationship-aware code search for JavaScript frameworks. In *Proceedings of the ACM SIGSOFT 24th International Symposium on the Foundations of Software Engineering*. ACM, 2016.

[42] W. Ling, E. Grefenstette, K. M. Hermann, T. Kocisky, A. Senior, F. Wang, and P. Blunsom. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*, 2016.

[43] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi. Sourcerer: mining and searching internet-scale software repositories. *Data Mining and Knowledge Discovery*, 18:300–336, 2009.

[44] M. Lu, X. Sun, S. Wang, D. Lo, and Y. Duan. Query expansion via wordnet for effective code search. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 545–549. IEEE, 2015.

[45] F. Lv, H. Zhang, J. Lou, S. Wang, D. Zhang, and J. Zhao. CodeHow: Effective code search based on API understanding and extended boolean model. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*. IEEE, 2015.

[46] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering*, 38(5):1069–1087, 2012.

[47] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 111–120. IEEE, 2011.

[48] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[49] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur. Recurrent neural network based language model. In *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010*, pages 1045–1048, 2010.

[50] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[51] I. J. Mojica, B. Adams, M. Nagappan, S. Dienst, T. Berger, and A. E. Hassan. A large scale empirical study on software reuse in mobile apps. *IEEE Software*, 31(2):78–86, 2014.

[52] D. J. Montana and L. Davis. Training feedforward neural networks using genetic algorithms. In *IJCAI*, volume 89, pages 762–767, 1989.

[53] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 1287–1293. AAAI Press, 2016.

[54] L. Mou, R. Men, G. Li, L. Zhang, and Z. Jin. On end-to-end program generation from user intention by deep neural networks. *arXiv*, 2015.

[55] A. Nederlof, A. Mesbah, and A. v. Deursen. Software engineering for the web: the state of the practice. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 4–13. ACM, 2014.

[56] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen. Exploring api embedding for api usages and applications. In *Proceedings of the 39th International Conference on Software Engineering*, pages 438–449. IEEE Press, 2017.

[57] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li. Query expansion based on crowd knowledge for code search. *IEEE Transactions on Services Computing*, 9(5):771–783, 2016.

[58] H. Niu, I. Keivanloo, and Y. Zou. Learning to rank code examples for code search engines. *Empirical Software Engineering*, pages 1–33, 2016.

[59] H. Palangi, L. Deng, Y. Shen, J. Gao, X. He, J. Chen, X. Song, and R. K. Ward. Deep sentence embedding using the long short term memory network: Analysis and application to information retrieval. *CoRR*, abs/1502.06922, 2015.

[60] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin. Building program vector representations for deep learning. In *Proceedings of the 8th International Conference on Knowledge Science, Engineering and Management - Volume 9403*, KSEM 2015, pages 547–553, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

[61] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Mining stackoverflow to turn the ide into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 102–111. ACM, 2014.

[62] M. Raghothaman, Y. Wei, and Y. Hamadi. SWIM: synthesizing what I mean: code search and idiomatic snippet synthesis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 357–367. ACM, 2016.

[63] M. Rahimi and J. Cleland-Huang. Patterns of co-evolution between requirements and source code. In *2015 IEEE Fifth International Workshop on Requirements Patterns (RePa)*, pages 25–31. IEEE, 2015.

[64] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *In Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2014.

[65] S. P. Reiss. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*, pages 243–253. IEEE Computer Society, 2009.

[66] M. Renieres and S. P. Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 30–39, Oct 2003.

[67] P. C. Rigby and M. P. Robillard. Discovering essential code elements in informal documentation. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 832–841. IEEE Press, 2013.

[68] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *CASCON First Decade High Impact Papers*, pages 174–188. IBM Corp., 2010.

[69] K. T. Stolee, S. Elbaum, and D. Dobos. Solving the search for source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(3):26, 2014.

[70] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[71] M. Tan, B. Xiang, and B. Zhou. Lstm-based deep learning models for non-factoid answer selection. *arXiv preprint arXiv:1511.04108*, 2015.

[72] J. Turian, L. Ratinov, and Y. Bengio. Word representations: a simple and general method for semi-supervised learning. In *Proceedings of the 48th annual meeting of the association for computational linguistics*, pages 384–394. Association for Computational Linguistics, 2010.

[73] Y. Uneno, O. Mizuno, and E.-H. Choi. Using a distributed representation of words in localizing relevant files for bug reports. In *Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on*, pages 183–190. IEEE, 2016.

[74] J. Weston, S. Bengio, and N. Usunier. Wsabie: scaling up to large vocabulary image annotation. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Three*, pages 2764–2770. AAAI Press, 2011.

[75] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. *arXiv preprint arXiv:1707.04742*, 2017.

[76] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk. Deep learning code fragments for code clone detection. In *Proceedings of the 31th IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*, 2016.

[77] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk. Toward deep learning software repositories. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 334–345. IEEE, 2015.

[78] R. Xu, C. Xiong, W. Chen, and J. J. Corso. Jointly modeling deep video and compositional text to bridge vision and language in a unified framework. In *AAAI*, pages 2346–2352. Citeseer, 2015.

[79] X. Ye, R. Bunescu, and C. Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 689–699. ACM, 2014.

[80] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the 38th International Conference on Software Engineering*, pages 404–415. ACM, 2016.

[81] H. Zhang, A. Jain, G. Khandelwal, C. Kaushik, S. Ge, and W. Hu. Bing developer assistant: Improving developer productivity by recommending sample code. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 956–961. ACM, 2016.

[82] J. Zhou and R. J. Walker. API Deprecation: A retrospective analysis and detection method for code examples on the web. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE'16)*. ACM, 2016.