When Deep Learning Met Code Search

Jose Cambronero
MIT CSAIL
jcamsan@mit.edu

Hongyu Li
Facebook, Inc.
hongyul@fb.com

Seohyun Kim
Facebook, Inc.
skim131@fb.com

Koushik Sen
EECS Department, UC Berkeley
ksen@cs.berkeley.edu

Satish Chandra
Facebook, Inc.
satch@fb.com

ABSTRACT

There have been multiple recent proposals on using deep neural networks for code search using natural language. Common across these proposals is the idea of *embedding* code and natural language queries, into real vectors and then using vector distance to approximate semantic correlation between code and the query. Multiple approaches exist for learning these embeddings [15, 19, 24, 26], including *unsupervised* techniques, which rely only on a corpus of code examples, and *supervised* techniques, which use an *aligned* corpus of paired code and natural language descriptions. The goal of this supervision is to produce embeddings that are more similar for a query and the corresponding desired code snippet.

Clearly, there are choices in whether to use supervised techniques at all, and if one does, what sort of network and training to use for supervision. This paper is the first to evaluate these choices systematically. To this end, we assembled implementations of state-of-the-art techniques to run on a common platform, training and evaluation corpora. To explore the design space in network complexity, we also introduced a new design point that is a *minimal* supervision extension to an existing unsupervised technique.

Our evaluation shows that: 1. adding supervision to an existing unsupervised technique can improve performance, though not necessarily by much; 2. simple networks for supervision can be more effective that more sophisticated sequence-based networks for code search; 3. while it is common to use docstrings to carry out supervision, there is a sizeable gap between the effectiveness of docstrings and a more query-appropriate supervision corpus.

ACM Reference Format:
Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When Deep Learning Met Code Search. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Query 1: How can I convert a stack trace to a string?

```java
public synchronized static String getStackTrace(Exception e) {
  e.fillInStackTrace();
  StringBuffer buffer = new StringBuffer();
  buffer.append(e.getMessage() + "-");
  for (StackTraceElement el: e.getStackTrace()) {
    buffer.append(el.toString() + "-");
  }
  return buffer.toString();
}
```

https://github.com/Dynatrace/Dynatrace-AppMon-REST-Monitor-Plugin/blob/master/src/com/realdolmen/dynatrace/restmonitor/RestMonitor.java

Query 2: How do I get a platform-dependent new line character?

```java
public static String getPlatformLineSeparator() {
  return System.getProperty("line.separator");
}
```

https://github.com/nutritionfactsorg/daily-dozen-android/blob/master/app/src/main/java/org/nutritionfacts/dailydozen/Common.java

Table 1: Example code search results. Each is selected from the top 1 result found by the *UNIF* model that we introduce. The existing code search interface of github.com does not return any relevant code snippets in the top 10 results for these queries.

1 INTRODUCTION

We have recently seen a significant uptick in interest in code search. The goal of code search is to retrieve code fragments from a large code corpus that most closely match a developer's intent, which is expressed in natural language. Being able to examine existing code that is relevant to a developer's intent is a fundamental productivity tool. Sites such as Stack Overflow are popular because they are easy to search for code relevant to a user's question expressed in natural language.

It has typically been harder to search directly over public code repositories, such as GitHub, as well as private repositories internal to companies. Proprietary code repositories in particular pose a challenge, as developers can no longer rely on public sources such as Google or Stack Overflow for assistance, as these may not capture the required organization-specific API and library usage. However, recent works from both academia and industry [15, 19, 21, 26] have taken steps towards enabling more advanced code search using deep learning. We call such methods *neural* code search. See Table 1 for some examples of code snippets retrieved based on natural language queries: it is evident that the state of the technology has become promising indeed. The type of queries presented in Table 1, and the accompanying results, also highlight the difficulties associated with tackling this task purely based on simple approaches such as regular-expression matching.

Conference'17, July 2017, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

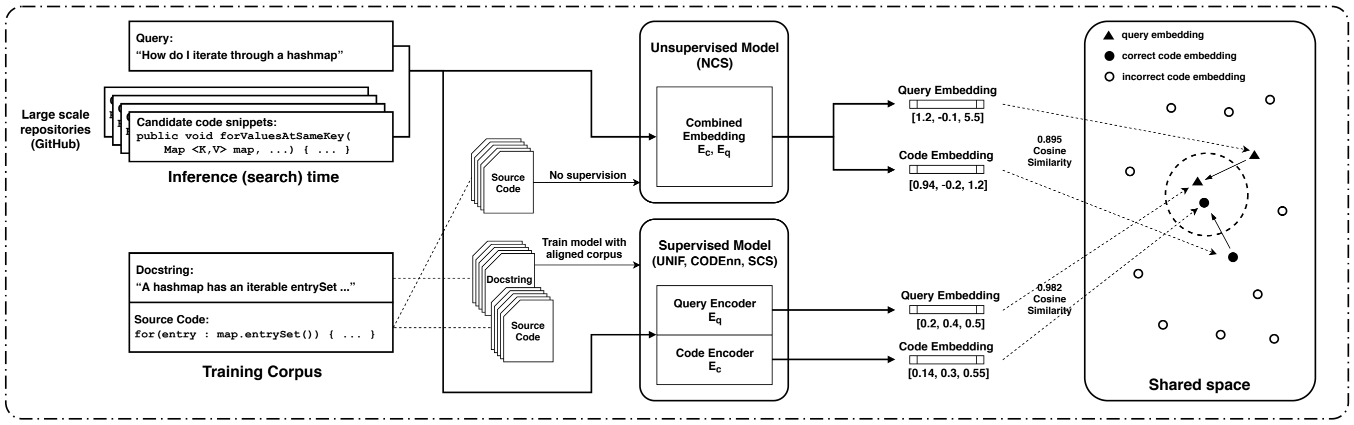

Figure 1: When using embeddings for code search, the query and the candidate code snippets are mapped to a shared vector space, using functions E_q and E_c, respectively. Search then corresponds to maximizing a similarity measure, such as cosine similarity, between the query embedding and code embeddings. These vector representations can be learned in an *unsupervised* manner, which just uses code, or in an *supervised* manner, which exploits an *aligned* corpus of code snippets and their corresponding natural language descriptions.

Figure 1 provides a general overview of neural code search and outlines different techniques, which we address in detail through this paper. The core abstraction in neural code search systems is the notion of *embeddings*, which are vector representations of inputs in a shared vector space. By computing a vector similarity measure, such as cosine similarity [31], over these embeddings, search can retrieve code fragments that are semantically related to a user query. For example, in Figure 1, the query *"How do I iterate through a hashmap?"* is mapped to the vector $\langle 1.2, -0.1, 5.5 \rangle$ by one possible technique (*NCS*). Candidate code snippets are also mapped to vectors using the same technique. In Figure 1, one such code snippet, `public void forValuesAtSameKey...`, for example, is mapped to the vector $\langle 0.94, -0.2, 1.2 \rangle$. The candidate code snippets then can be ranked using vector similarity. A key challenge in neural code search is to learn these embeddings in a way that vector similarity coincides with semantic relatedness.

As shown in Figure 1, the models used to learn these representations can be broadly grouped into *unsupervised* and *supervised*. In our journey to explore the advantages of neural techniques, we started with *NCS*, an effective, unsupervised, neural code search technique we previously built in [26]. Because *NCS* showed promising results, we wanted to experiment with the possibility of improving upon this baseline through additional design enhancements. In particular, recent work [15, 19] presented promising supervised neural code search techniques, labeled *CODEnn* and *SCS* respectively, that successfully learned code and natural language embeddings using corpora of source code and docstrings.

The goal of this supervision is to learn a mapping that produces more similar vectors for user queries and the corresponding desired code. In Figure 1, this goal is depicted by the red and blue arrows, which move the embeddings for the query and correct code fragment, respectively, closer together when mapped using a supervised model.

With so many techniques to choose from, how does anyone trying to design and deploy a code search solution make an informed choice? What are the trade-offs involved? For instance, supervision sounds like a good idea, but how much benefit does it provide, relative to the overhead of obtaining the supervision data? How much value, if any, do the more sophisticated networks – which have many more parameters – bring compared to a simpler network, one of which we introduce in this work (*UNIF* in Figure 1, described further below)? Does model supervision carried out using docstrings (as is common practice in work on bimodal embeddings, including *CODEnn* and [21]), potentially limit performance when models are applied to real user queries?

In this work, we attempt to understand these tradeoffs quantitatively. To do so, we formulate experiments in the context of the code search techniques mentioned above, and use open-source evaluation corpora, described later. Three of these techniques are exactly as in previous work, and are state-of-the-art at this time:

- ***NCS*** An unsupervised technique for neural code search developed at Facebook [26], which uses only word embeddings derived from a code corpus.
- ***CODEnn*** A supervised technique from a recent paper on code search using deep neural networks [15], which uses multiple sequence-to-sequence-based networks, and was shown to outperform other state-of-the-art code search techniques. We use the implementation provided by the authors [16].
- ***SCS*** A supervised neural code search system using multiple sequence-to-sequence networks. We use the implementation provided by the authors in a blog post [19, 20].

Because we wanted to understand the extent to which the complex sequence-of-words based networks help, we also developed a minimal extension to the *NCS* technique, just adding supervision and nothing else:

- ***UNIF*** A supervised extension of the base *NCS* technique of our own creation. *UNIF* uses a bag-of-words-based network, which has significantly lower complexity compared to sequence-of-words-based networks. This simple model is a new contribution of this paper.

Our evaluation is structured using the following three research questions; we also give the summary of our findings along with the question.

Research Question 1. : Does extending an effective unsupervised code search technique with supervision based on a corpus of paired code and natural language descriptions improve performance?

Our results show that *UNIF* performed better than *NCS* on our benchmark queries, though the improvement was not seen uniformly across all data sets.

Research Question 2. : Do more sophisticated networks improve performance in supervised neural code search?

Our results show that *UNIF*, a simple, bag-of-words-based network, outperformed the sequence-of-words-based *CODEnn* and *SCS*. The additional sophistication did not add value.

Research Question 3. : How effective is supervision based on docstrings as the natural language component of the training corpus?

We found that supervision based on docstrings – which is commonly the natural language component of an aligned corpus – did not always improve performance, contrary to expectations. To understand the possible room for improvement, we constructed an ideal alternate training corpus, where the code snippets and natural language, while disjoint from our benchmark queries, were drawn from the same source.

When trained on this corpus, all supervised techniques improved significantly, showing that, as a proof of concept, if given a training corpus that matches expected user evaluation, these techniques can provide impressive search performance.

Contributions. 1. We believe this is the first paper to compare recent neural code search systems running on the same platform and evaluation using the same corpora.
2. We present a new design point in the spectrum of neural code search systems: *UNIF*, an extension to *NCS* that minimally adds supervision and nothing else.
3. Our findings are that *UNIF* outperforms some of the more sophisticated network designs (*CODEnn* and *SCS*) as well as the *NCS*, the unsupervised technique. Moreover, the choice of the aligned corpus used in supervision is extremely pertinent: an idealized training corpus shows that supervised techniques can deliver impressive performance, and highlight the differences in performance that may not be immediately evident from training on the typical code and docstring aligned corpora.

These findings have implications for anyone considering designing and deploying a neural code search system in their organization. The findings also have implications for researchers, who should consider simple baseline models in evaluating their designs.

Outline. The rest of the paper is organized as follows. Section 2 introduces the core idea of embeddings and their use in neural code search. Section 3 details each of the techniques explored in this paper. Section 4 presents our evaluation methodology. Section 5 provides results supporting our research questions' answers. Section 6 and Section 7 discuss threats to validity and related work, respectively. Finally, Section 8 concludes with the main takeaways and implications for neural code search system designers.

2 EMBEDDINGS FOR CODE

An *embedding* refers to a real-valued vector representation for an input. An embedding function $E : \mathcal{X} \rightarrow \mathbb{R}^d$ takes an input x in the domain of $\mathcal{X}$ and produces its corresponding vector representation ink a d-dimensional vector space. This vector is said to be *distributed* [6], where each dimension of the vector is not attributed to a specific hand-coded feature of the input, but rather the "meaning" of the input is captured by the vector as a whole.

Embeddings present multiple appealing properties. They are more expressive than local representations, such as one hot encodings, as values along each dimension are continuous [6]. Embeddings can also be learned, which makes them applicable to different domains where we have example data. One possibility is to learn these embeddings using a neural network, such that the function E uses a network's learned weights.

2.1 Running Example

We present a running example to illustrate some of the key concepts for the use of embeddings in code search. Suppose we want to produce a vector that can successfully represent the code snippet below.

```
for (entry : map.entrySet()) {
  System.out.println(entry);
}
```

One possible approach is to treat this source code as text, and *tokenize* this input into a collection of individual words. The extent of tokenization (and filtering certain words) may depend on the specific model design. For this example, we will tokenize based on standard English conventions (e.g. white-space, punctuation) and punctuation relevant to code (e.g. snake and camel case). The code snippet can now be treated as the collection of words.

```
for entry map entry set system out println entry
```

Given a corpus of multiple code examples, such as the source code corpus in Figure 1, we can tokenize all code examples and learn an embedding for each token.

One approach to learning token embeddings is with an unsupervised model. One popular technique is *word2vec*, which implements a skip-gram model [7, 24]. In the skip-gram model, the embedding for a *target* token is used to predict embeddings of *context* tokens within a fixed window size. In our example, given the embedding for the token `set` and a window size 2, the skip-gram model would learn to predict the embeddings for the tokens `map`, `entry`, `system`, and `out`. The objective of this process is to learn an embedding matrix T, where each row corresponds to the embedding for a token in the vocabulary, and where two embeddings are similar if the corresponding tokens often occur in similar contexts.

At this point, we can map each word in our tokenized code snippet to its corresponding embedding. For example, `for` may be represented by $\langle 0.2, -1.0, 3.8 \rangle$, and `entry` may be represented by $\langle 0.8, 0.9, -2.0 \rangle$.

2.2 Bags and sequences of embeddings

The next step in our procedure will be to combine the token-level embeddings for our code snippet into a single embedding that appropriately represents the snippet as a whole. We discuss two possible approaches to doing so using standard neural network architectures.

So far we have not discussed the impact of the token order in our snippet. We can decide to treat the words as a bag, occasionally called a multiset, and ignore order. In such a case, our example

```
for entry map entry set system out println entry
```

would be equivalent to every other permutation, such as

```
entry for map println entry set out entry system
```

A corresponding bag-based neural network would compute the representation for our code snippet without regard to token order. One simple example of such a bag-of-words-based architecture is one where we use T, the matrix of learned token embeddings, to look up the embedding for each word in the tokenized example and then average (either simple or weighted) these vectors to obtain a single output vector. network.

In contrast, a neural network may instead consume the tokens in an input as a sequence, such that the ordering of elements is significant. We provide details on one common approach to handling sequence-based inputs: recurrent neural networks (RNN) [12].

An RNN starts with an initial hidden state, often initialized randomly, represented as h_0, and processes the words in the input sequentially one by one. In our example, the two permutations of the tokenized code snippet are no longer equivalent.

After processing each word, the RNN updates the hidden state. If the t^{th} word in the sequence is w_t and the hidden state after processing the words before w_t is h_{t-1}, then the next hidden state after processing w_t is obtained as follows:

$$h_t = \tanh(W.[h_{t-1}; w_t]) \tag{1}$$

where W is a matrix whose parameters are learned, $[x; y]$ is the vector obtained by concatenating the vector x and y, and $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ is a non-linear activation function which ensures that the value of $\tanh(x)$ lies between 1 and -1.

There are multiple approaches to obtain a snippet-level embedding using this RNN. For example, one model might take the last hidden state h_n as the snippet embedding. Another could collect the hidden states h_i and apply a reduction operation such as dimension-wise max or mean to produce the snippet embedding.

The network described above is a simple RNN; in practice, an RNN is implemented by using a more complex function on h_{t-1} and w_t. An example of such an RNN is long-short term memory (LSTM) [18].

2.3 Bi-Modal embeddings

So far, we have only discussed how to produce a representative vector given a code snippet. However, neural code search uses embeddings for both code snippets and the user's natural language query. This means that our embedding approach must be able to represent both the code `for (entry : map.entrySet) ...` , and the query *"how to iterate through a hashmap"*, which is expressed in natural language. Such embeddings that relate two different kinds of data are called bi-modal embeddings [2].

The computation of bi-modal embeddings of a code snippet and its natural language description can be abstractly formulated as two functions: $E_c : C \to \mathbb{R}^d$ and $E_q : Q \to \mathbb{R}^d$, where C is the domain of code snippets, Q is the domain of natural language descriptions, $\mathbb{R}^d$ is a real-valued vector of length d, E_c is an embedding function that maps a code snippet to a d-dimensional vector, and E_q is an embedding function that maps a natural language description to a vector in the same vector space. The goal is to learn the functions E_c and E_q such that for some similarity measure *sim*, such as cosine similarity [31], $sim(E_c(c), E_q(q))$ is maximized for a code snippet c and its corresponding natural language description q. Alternatively, for unsupervised models, such as *NCS*, E_c and E_q may be instantiated with the same token-level embedding matrix T, as shown in Figure 1.

2.4 Applying embeddings to code search

Given E_c and E_q, code search can be performed given the user query and a code corpus.

Figure 1 illustrates how embeddings are used in code search. The code embedding function E_c is used to convert each candidate code snippet in our search corpus into a vector.

For example, given the code snippet

```
public void forValuesAtSameKey(Map <K, V> map, ...) {
  ...
}
```

in our search index, an unsupervised E_c (labeled *NCS* in the figure) returns the vector representation $\langle 0.94, -0.2, 1.2 \rangle$. All the snippets in a corpus can be embedded in a similar fashion and used to construct an index that allows for fast lookups based on a similarity metric.

The user query can be similarly embedded using E_q. For example, *"How do I iterate through a hashmap?"* is mapped to the vector $\langle 0.2, 0.4, 0.5 \rangle$. To retrieve relevant code snippets, the code embeddings index can be searched based on similarity to the query embedding. The top N results based on this similarity are returned.

There are a number of possible neural architectures used to learn E_q and E_c, and we will explore several of them in this paper.

3 NEURAL CODE SEARCH MODELS

We now introduce each of the neural techniques explored in this paper.

3.1 *NCS*

In *NCS* [26], a specific technique named after the general concept of *Neural Code Search*, the embedding functions E_c and E_q are implemented using a combination of token-level embeddings using fastText [7], which is similar to *word2vec* [24], and conventional information retrieval techniques, such as TF–IDF [11, 27]. As such this technique does not use conventional deep neural networks nor supervised training. *NCS* computes an embedding matrix $T \in \mathbb{R}^{|V_c| \times d}$, where $|V_c|$ is the size of the code token vocabulary, d is the chosen dimensionality of token embeddings, and the k^{th} row in T is the embedding for the k^{th} word in V_c. Once the matrix T has been computed using fastText, it is *not* further modified using supervised training.

NCS applies the *same* embedding matrix T to *both* the code snippets and the query as follows. Let $c = \{c_1, \ldots, c_n\}$ and $q = \{q_1, \ldots, q_m\}$ represent the code snippet and query, respectively, as

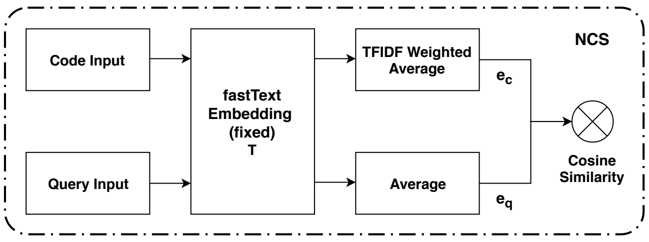

Figure 2: *NCS* embeds the code and query input with the fastText [7] embeddings. The code sentence embedding $\mathbf{e_c}$ is computed from the bag of code embeddings with TF-IDF weights. The query sentence embedding $\mathbf{e_q}$ is produced by averaging the bag of query embeddings.

a multiset (i.e. order insensitive) of tokens. *NCS* generates a bag of embedding vectors $\{T[c_1], \ldots, T[c_n]\}$ for the code snippet and $\{T[q_1], \ldots, T[q_m]\}$ for the query, where $T[w]$ is the embedding vector in the matrix T for the token w.

To combine the bag of code token embeddings into a single code vector e_c, *NCS* sums the embeddings for the set of unique tokens weighed by their corresponding TF–IDF weight. The TF–IDF weight is designed to increase the weight of tokens that appear frequently in a code snippet, and decrease the weight of tokens that appear too frequently globally across all of the code corpus. For example, in a NLP (natural language processing) corpus, a document with many repetitions of a common word such as "the" will have a high TF (term-frequency) for that token, but its weight will be scaled down by the IDF value (inverse document-frequency) as that token likely also appears in many other documents. We elide the classical TF–IDF weighing formula here for brevity.

For the query, *NCS* averages the bag of query token embeddings into a single query vector e_q.[1]

The high level architecture of the *NCS* model is illustrated in Figure 2.

3.2 *UNIF*: a supervised extension of *NCS*

We will introduce *UNIF* next, as it is a supervised minimal extension of the *NCS* technique. In this model, we use supervised learning to modify the initial token embedding matrix T and produce two embedding matrices, T_c and T_q, for code and query tokens, respectively. We also replace the TF-IDF weighing of code token embeddings with a learned attention-based weighing scheme. We refer to this extended approach as *Embedding Unification* (*UNIF*).

We assume that an aligned corpus of code snippets and their natural language descriptions is available for training. We denote this corpus as a collection of (c, q), where c is bag of tokens $c_1, \ldots, c_n$ from a code snippet and q is a bag of tokens from its corresponding natural language description.

The functions E_c and E_q are constructed as follows. Let $T_q \in \mathbb{R}^{|V_q| \times d}$ and $T_c \in \mathbb{R}^{|V_c| \times d}$ be two embedding matrices mapping each word from the natural language description (specifically the docstrings and the query) and code tokens, respectively, to a vector of length d. The two matrices are initialized using the same initial weights, T, and modified separately during training.

[1] The authors of *NCS* also introduce a variant of their model that heuristically extends user queries using code and natural language token co-occurrences. We do not use this heuristic extension in order to directly observe the impact of extending training with natural language supervision.

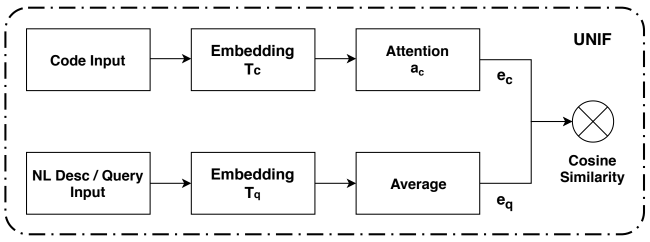

Figure 3: The UNIF network uses attention $\mathbf{a_c}$ to combine per-token embeddings $\mathbf{T_c}$ and produce the code sentence embedding $\mathbf{e_c}$. The query sentence embedding $\mathbf{e_q}$ is produced by averaging the bag of query embeddings $\mathbf{T_q}$. Both $\mathbf{T_c}$ and $\mathbf{T_q}$ are initialized with the fastText [7] embeddings and are further fine-tuned during training.

We apply the respective embedding matrices to each element in the paired corpus, such that for a code snippet c we obtain a bag of embedding vectors $\{T_c[c_1], \ldots, T_c[c_n]\}$, and similarly for each description q. We compute a simple average to combine the query token embeddings into a single vector. The simple averaging is also present in *NCS* and we found it to outperform attention-based weighing during experiments.

To combine each bag of code token vectors into a single code vector that captures the semantic meaning of the corresponding entity, we use an attention mechanism [3] to compute a weighted average. The attention weights, $a_c \in \mathbb{R}^d$, is a d-dimensional vector, which is learned during training. a_c acts as a learned counterpart to the TF-IDF weights in *NCS*.

Given a bag of code token embedding vectors $\{e_1, \ldots, e_n\}$, the attention weight α_i for each e_i is computed as follows:

$$\alpha_i = \frac{\exp(a_c . e_i^\intercal)}{\sum_{i=1}^{n} \exp(a_c . e_i^\intercal)} \tag{2}$$

Here we compute the attention weight for each embedding vector as the softmax over the inner product of the embedding and attention vectors. We normalize each attention weight to make sure that weights add up to 1, and apply exponentiation to make each weight positive and sharp.

The summary code vector of a bag of embedding vectors is then computed as the sum of the embedding vectors weighted by the attention weights α_i:

$$e = \sum_{i=1}^{n} \alpha_i e_i \tag{3}$$

e corresponds to the output of E_c.

Our training process learns parameters T_q, T_c, and a_c using classic backpropagation.

Figure 3 shows a high level diagram of the embedding unification network.

3.3 *CODEnn*

Similar to *UNIF*, *CODEnn* [15] also models both E_c and E_q using neural networks and employs supervised learning; however, the networks used are more sophisticated and deep. We refer to this model as *CODEnn*, short for *CodeDescriptionEmbeddingNeuralNetwork*.

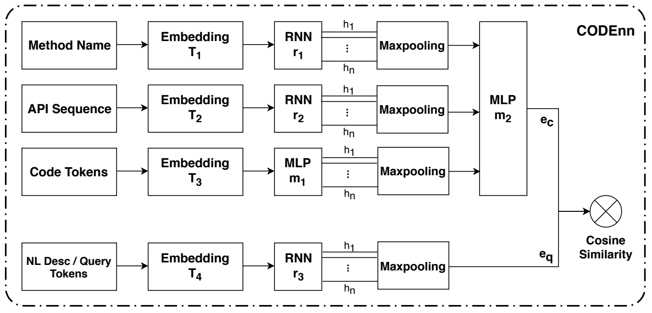

Figure 4: *CODEnn*, the network proposed for code search uses RNNs to embed the method name (r_1), API sequence (r_2), and query (r_3). It uses a feed forward network (MLP) to embed the code body tokens (m_1), and combines this embedding with the method name and API embedding with another MLP (m_2).

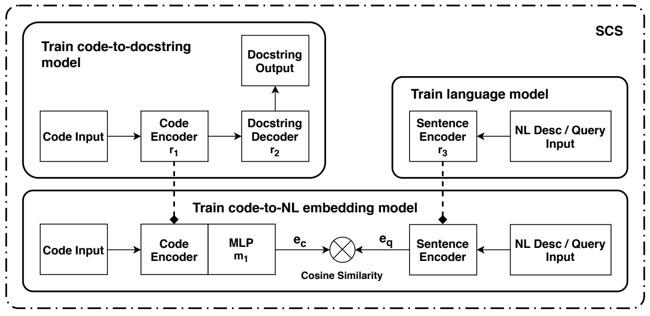

Figure 5: *SCS* uses the encoder portion of the code-to-docstring sequence-based network to embed sequences of code tokens. Separately, it trains a language model to embed sequences of query tokens. A feed forward layer is added to the code encoder to transform code embeddings into query embeddings (derived from the language model).

Instead of treating a code snippet as a bag of tokens, *CODEnn* extracts a sequence of words from the name of the method containing the code snippet, the sequence of API calls in the snippet, and a bag of tokens from the code snippet. The word sequence from a method name is extracted by splitting the method name on camel-case and snake-case.

The method name sequence and API sequence are given as input to two separate bi-directional long-short term memory (bi-LSTM) networks [18].

After applying two separate LSTMs to the method name and API sequences, *CODEnn* obtains two sequences of hidden states. *CODEnn* summarizes each such sequence of hidden states to obtain a single vector. For summarization, *CODEnn* uses the max-pooling function.

Each token in the bag of code tokens is given individually as input to a feed forward dense neural network and the output vectors are max-pooled. A final code embedding is then obtained by concatenating these three vectors (two from the LSTMs and one from the feed-forward network) and feeding them to a dense neural network which produces a single summary vector e_c. All the above networks together implement the function E_c.

CODEnn implements the function E_q using a bi-directional LSTM, which takes as input sequence the description of the code snippet found in the doc string to produce e_q. Figure 4 provides an overview of the architecture.

3.4 *SCS*

We introduce another supervised sequence-based deep neural network for code search, described and implemented by the data science team at GitHub [19]. We will refer to this model as *SCS*, short for *Semantic Code Search*.

SCS is divided into three separate training modules. A sequence-to-sequence gated recurrent unit (GRU) network [8] learns to generate a docstring token sequence given a code token sequence. We refer to this as the code-to-docstring model.

An LSTM network [23] learns a language model for docstrings in the training corpus [29]. This model can be used to embed natural language and compute the probability of a given natural language input.

A final module learns a transformation (in the form of a feed forward layer) to predict a query embedding given a sequence of code tokens. To learn this transformation, the module takes the encoder portion of the code-to-docstring model, freezes its layers, and trains the network on code sequence inputs and the corresponding query embedding produced using the language model. A final training phase fine-tunes the network as a whole by unfreezing the encoder layer for a few epochs. *SCS* uses this fine-tuned encoder portion of the code-to-docstring model as E_c and the language model as E_q. Figure 5 provides an overview of the architecture.

Table 2 provides an overview of the network details for models that employ supervision when learning their E_c and E_q: *UNIF*, *CODEnn*, and *SCS*.

4 EVALUATION METHODOLOGY

Our evaluation uses different datasets and benchmarks. We use the following terminology throughout for clarity:

- ***training corpus*** refers to a dataset of paired code and natural language. An example for natural language could be the code fragment's corresponding docstring. A training corpus is used to *train* the models and may contain duplicate code / natural language pairs. An unsupervised model, such as *NCS*, uses only the code fragments from a dataset.
- ***search corpus*** refers to a dataset of unique code entries[2]. Entries are unique in order to avoid repetition in search results. We apply a trained model to a search corpus to *search* for the top results for a given user query. This dataset is used during the evaluation of the models.
- ***benchmark queries*** refers to a set of evaluation queries used to gauge the performance of trained models. Each query in a benchmark is accompanied by a gold-standard code fragment result, which we use to score results retrieved from a search corpus by a trained model.

Our evaluation uses three different training corpora, two search corpora, and two sets of benchmark queries.

4.1 Training Corpora

We use three training corpora for our experiments.

[2] Dataset is deduplicated after tokenization.

Model	Summary	Parameters
UNIF	Embeds code/query tokens.	Embedding matrices T_c, T_q (Figure 3 $\mathrm{T_c}$, $\mathrm{T_q}$).
	Combines code embeddings with attention.	Attention vector a_c (Figure 3 $\mathrm{a_c}$)
CODEnn	Embeds method name, API sequence and query as sequences.	Embedding matrices T_1, T_2, T_3, T_4 (Figure 4 $\mathrm{T_1}$, $\mathrm{T_2}$, $\mathrm{T_3}$, $\mathrm{T_4}$)
	Embeds a bag of tokens from body.	Bi-directional LSTM parameters for RNNs (Figure 4 $\mathrm{r_1}$, $\mathrm{r_2}$, $\mathrm{r_3}$)
	Combines method name, API, and token embeddings using another layer.	MLP parameters (Figure 4 $\mathrm{m_1}$, $\mathrm{m_2}$)
SCS	Embeds sequence of code tokens.	GRU parameters for RNNs (Figure 5 $\mathrm{r_1}$, $\mathrm{r_2}$)
	Embeds sequence of query tokens.	LSTM parameters for RNN (Figure 5 $\mathrm{r_3}$)
	Transforms code embedding into query token space.	MLP parameters (Figure 5 $\mathrm{m_1}$)

Table 2: Summary of details for models trained with supervision. In terms of network complexity (in terms of parameters, and layers), from least to most complex, we have: *UNIF*, *CODEnn*, and *SCS*.

CODEnn-Java-Train is the dataset publicly released by the authors of *CODEnn*. This corpus consists of approximately 16 million pre-processed Java methods and their corresponding docstrings, intended for training. The dataset includes four types of inputs: method name sequences, API sequences, a bag of method body tokens, and docstring sequences. We additionally derive another input by concatenating the method name sequences to the API sequences and treating this concatenated sequence as a bag of tokens. This derived input is used to train *UNIF* and *SCS*.[3][4]

GitHub-Android-Train is an Android-specific corpus that we built by collecting methods from approximately 26,109 GitHub repositories with the Android tag. We took all methods with an accompanying docstring, approximately 787,000 in total, and used these as training data. Similar to *CODEnn-Java-Train*, we derive the four types of input collections (method name sequences, API sequences, bag of method body tokens, and docstring sequences) necessary to train *CODEnn*. We train *UNIF* and *SCS* on the input sequence generated for *NCS*, which uses a parser to qualify method names with their corresponding class, extracts method invocations, enums, string literals and source code comments, while ignoring variable names, and applies a camel and snake case tokenization [26].[5]

StackOverflow-Android-Train is an Android-specific training corpus that we built by collecting Stack Overflow question titles and code snippets answers. We prepared this dataset by extracting all Stack Overflow posts with an Android tag from a data dump publicly released by Stack Exchange [28]. The dataset is filtered on the following heuristics: (1) The code snippet must not contain XML tags; (2) The code snippet must contain a left paraenthesis '(' to indicate presence of a method call; and (3) The post title must not contain keywords like "gradle", "studio" and "emulator". After filtering, we end up with 451k Stack Overflow title and code snippet pairs. This dataset is *disjoint* from the Android Stackoverflow benchmark queries described later on in Section 4.3.

The goal of *StackOverflow-Android-Train* is to serve as an alternate training corpus that is ideal relative to our evaluation, which leverages Stack Overflow titles and code snippets as benchmark queries and answers, respectively. By training on this corpus, we can measure the possible room for improvement compared to training on a typical aligned corpus, which uses docstrings as natural language. [6]

[3] Supervised models trained on *CODEnn-Java-Train* train for 50 hours on an Nvidia Tesla M40 GPU.

[4] This data has been generously made available the *CODEnn* authors at https://drive.google.com/drive/folders/1GZYLT_lzhlVczXjD6dgwVUvDDPHMB6L7

[5] Supervised models trained on *GitHub-Android-Train* train for 3 hours on an Nvidia Tesla M40 GPU.

4.2 Search Corpora

We use two search corpora during our evaluation.

CODEnn-Java-Search: 4 million *unique* Java methods released by the authors of *CODEnn*.

GitHub-Android-Search: 5.5 million *unique* Android methods collected from GitHub. This corpus is derived from the same 26,109 repositories used to construct *GitHub-Android-Train*, but also includes methods that do not have a docstring available.

4.3 Benchmark Queries

We use two sets of queries as evaluation benchmarks for our models. In both benchmark sets, the queries correspond to Stack Overflow titles and the ground truth answers for each query are the accepted answer or highly voted answer for the corresponding post, which we independently collected. This approach to collecting ground truth answers was borrowed from the original *NCS* paper [26].

Java-50 is a set of 50 queries used to evaluate *CODEnn* in the original paper. These queries correspond to Stack Overflow titles for the top 50 voted Java programming questions. The authors included questions that had a "concrete answer" in Java, included an accepted answer in the thread with code, and were not duplicate questions. When evaluating on this benchmark, models are trained on *CODEnn-Java-Train*.

Android-287 is a set of 287 Android-specific queries used to evaluate *NCS* in the original paper. These questions were chosen by a script with the following criteria: (1) the question title includes "Android" and "Java" tags; (2) there exists an upvoted code answer; and (3) the ground truth code snippet has at least one match in a corpus of GitHub Android repos. When evaluating on this benchmark, models are trained on *GitHub-Android-Train*, unless otherwise specified.

Table 3 provides a summary of the training corpora, search corpora, and benchmark queries used in our evaluation, and what combinations we use for our results.

[6] Supervised models trained on *StackOverflow-Android-Train* train for 3 hours on an Nvidia Tesla M40 GPU.

Dataset	Code/Natural Language	Number of Observations
CODEnn-Java-Train [15]	Method/docstring	16mm
GitHub-Android-Train	Method/docstring	787k
StackOverflow-Android-Train	Forum code snippet/Forum title	451k

(a) Training corpora

Dataset	Number of Entries
CODEnn-Java-Search [15]	4mm
GitHub-Android-Search	5.5mm

(b) Search corpora

Benchmark Queries	Number of Queries
Java-50 [15]	50
Android-287 [26]	287

(c) Benchmark Queries

Table 3: Summary of data used. When evaluating on *Android-287*, we use *GitHub-Android-Search* as search corpus and train on *GitHub-Android-Train* or *StackOverflow-Android-Train*. When evaluating on *Java-50*, we use *CODEnn-Java-Search* as search corpus and train on *CODEnn-Java-Train*.

4.4 Evaluation Pipeline

We found that manually assessing the correctness of search results can be difficult to do in a reproducible fashion, as deciding the relevance or correctness of a code snippet relative to the input query can vary across authors and people trying to reproduce our results. As such, we decided to carry out our evaluation using an automated evaluation pipeline. Our pipeline employs a similarity metric [22] between search results and a ground truth code snippet to assess whether a query was correctly answered. With this pipeline, we can scale our experiments to a much larger set of questions, such as *Android-287*, and assess correctness of results in a reproducible fashion. We use code answers found on Stack Overflow to provide a consistent ground truth for evaluation.[7] This approach was introduced by the authors of [26]. Figure 6 gives an overview of this evaluation pipeline.

The automated pipeline does require that we pick a similarity threshold to decide whether a query has been answered. To decide this value, two authors manually assessed the relevance of the top 10 search results for *Java-50* produced by *CODEnn* and *UNIF*. This assessment was done individually and conflicting decisions were cross-checked and re-assessed. Once a final set of relevant results was determined, we computed the similarity metric for each result with respect to the appropriate ground truth answer. This yielded a distribution of scores that was approximately normal. We took the mean and use this as the similarity threshold in our evaluation. This threshold chosen produces evaluation metrics for *CODEnn* that generally correspond to those in its original paper [15].

In our evaluation, we present the number of questions answered in the top k results. We consider the top 1, 5 and 10 results, and display the corresponding number of questions answered as *Answered@1,5,10*, respectively.

5 RESULTS

We now present our study's results and the answer for each of the research questions posed.

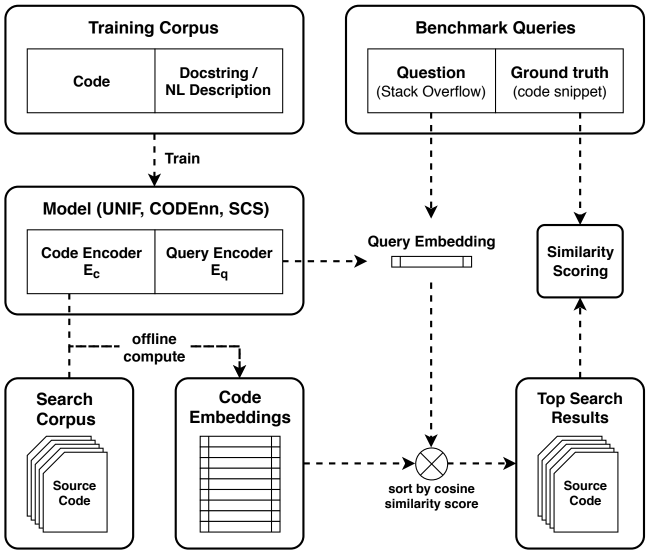

Figure 6: Evaluation pipeline.

5.1 RQ1

Does extending an effective unsupervised code search technique with supervision based on a corpus of paired code and natural language improve performance?

As detailed in Section 3.2, *UNIF* is an extension of *NCS* that adds supervision to modify embeddings during training and replaces the TF-IDF weights used to combine code token embeddings with learned attention. Table 4 shows that *UNIF* answers more questions across the board for *Java-50*. *UNIF* improves the number of answers in the top 10 results for *Android-287* but performs slightly worse for answers in the top 1. We conclude that extending a *NCS*, an unsupervised technique, with supervision improves performance for code search, but not uniformly across our datasets.

5.2 RQ2

Do more sophisticated networks improve performance in supervised neural code search?

[7] We tried to obtain code snippets marked as relevant from the original *CODEnn* authors for completeness, but they were unable to share them [14].

Table 4: Number of queries answered in *Java-50* in the top 1, 5, and 10 results improves when we extend *NCS* (unsupervised) to *UNIF* (supervised). For *Android-287*, supervision increased results in the top 5 and top 10.

Benchmark queries	Model	Answered@1	Answered@5	Answered@10
Java-50	*NCS*	15	29	37
	UNIF	**22**	**39**	**43**
Android-287	*NCS*	**33**	**74**	98
	UNIF	25	**74**	**110**

Table 5: The evaluation results on both benchmarks show that *UNIF* outperforms more sophisticated sequence-based networks such as *CODEnn* and *SCS*.

Benchmark queries	Model	Answered@1	Answered@5	Answered@10
Java-50	*UNIF*	**22**	**39**	**43**
	CODEnn	16	31	39
	SCS	9	17	21
Android-287	*UNIF*	**25**	**74**	**110**
	CODEnn	22	60	82
	SCS	9	19	34

When selecting possible supervised techniques, neural code search system designers may choose to incorporate more sophisticated architectures such as *CODEnn* or *SCS*, or favor a simple architecture, such as that used in *UNIF*. In order to navigate this question, we consider the number of queries answered by different techniques, and their inference speed.

We first compare model performance in terms of the number of queries correctly answered. Table 5 shows that *UNIF*, which uses a simple bag-of-words approach, outperformed both *CODEnn* and *SCS* on both benchmark query sets. *CODEnn* performed better than *SCS* in both cases.

A second consideration in the complexity tradeoff is the speed of inference. More sophisticated networks, in particular those that consume sequences and thus maintain intermediate state, are typically slower. While the time to embed code fragments is less of a concern, as this can be done offline and embeddings produced are typically stored in a performant index, the time to embed a natural language fragment directly relates to the responsiveness of a neural code search system.

We computed the time to embed a sample of code and natural language descriptions from *CODEnn-Java-Train*. Code entries were embedded in a batch of size 1000, while queries were embedded one at a time, to reflect the expected behavior in a code search system, where code is embedded offline and stored in an index and queries are embedded in real-time.

Table 6 shows the ratios of inference times **relative to *UNIF*** in each column, such that a value above 1.0 indicates slower inference for that column. Sequence-based networks such as *CODEnn* and *SCS* take longer to embed both code and natural language inputs.

Table 6: *Time ratios relative to UNIF* to embed sampled code and natural language from *CODEnn-Java-Train*. Values above 1 represent an inference time longer than *UNIF*'s for that column.

Model	code (CPU)	code (GPU)	query (CPU)	query (GPU)
CODEnn	11.72	58.55	103.83	10.75
SCS	11.92	35.01	105.10	15.94

5.3 RQ3

How effective is supervision based on docstrings as the natural language component of the training corpus?

The supervised techniques presented so far use the same kind of natural language during training: docstrings. Docstrings are used as a proxy for user queries and allow neural code search practitioners to collect sizeable aligned datasets for training. However, Table 8 shows that when training on *GitHub-Android-Train*, search performance did not always improve, contrary to expectations.

In this experiment, we use an alternate idealized training corpus, *StackOverflow-Android-Train*, which is drawn from the same source as our benchmarks *Android-287*, but is still disjoint from the queries. Intuitively, the performance attained with this corpus provides a measure for how much supervised techniques could improve search, given a training corpus that matches eventual user queries.

Table 7 shows that when we train on *StackOverflow-Android-Train*, all supervised techniques improve significantly (with one exception, queries answered in the top 1 using the *SCS*). This highlights the impressive search performance that a supervised technique could deliver, if given access to an ideal training corpus with a natural language component that better matches user queries.

Table 7: The number of *Android-287* answered in the top 1, 5, and 10 when the supervised techniques were trained on our idealized *StackOverflow-Android-Train* corpus. Search performance increases substantially, demonstrating the potential for supervised techniques, when given access to a training corpus that resembles eventual user queries.

Model	Answered@1	Answered@5	Answered@10
UNIF	25 → **104**	74 → **164**	110 → **188**
CODEnn	22 → 36	60 → 91	82 → 117
SCS	9 → 11	19 → 24	34 → 47

Table 8 provides a comprehensive performance summary detailed previously in each research question.

6 THREATS TO VALIDITY

Our evaluation shows that a supervised extension of *NCS* performed better. There may exist other unsupervised techniques which require more in-depth modification to successfully take advantage of supervision. Our goal, however, is not to show that our minimal extension is guaranteed to improve any unsupervised technique, but rather than it improves *NCS* specifically.

UNIF is presented a simple alternative to state-of-the-art architectures. We explored two techniques from current literature and show that *UNIF* outperforms them. More sophisticated architectures may successfully outperform *UNIF* but we believe that our result highlight the importance of exploring parsimonious configurations first.

Our experiments were carried out using three training corpora (*CODEnn-Java-Train*, *GitHub-Android-Train*, and *StackOverflow-Android-Train*) and two benchmark query sets (*Java-50* and *Android-287*). Both benchmark query sets have been used in prior neural code search literature. We believe that this extensive evaluation provides generalizable insight into the performance of these techniques.

Benchmark queries	Search corpus	Training corpus	Model	Answered@1	Answered@5	Answered@10	MRR
Java-50	*CODEnn-Java-Search*	Unsupervised	*NCS*	15	29	37	0.437
		CODEnn-Java-Train	*UNIF*	**22**	**39**	**43**	**0.582**
			CODEnn	16	31	39	0.456
			SCS	9	17	21	0.166
Android-287	*GitHub-Android-Search*	Unsupervised	*NCS*	33	74	98	0.189
		GitHub-Android-Train	*UNIF*	25	74	110	0.178
			CODEnn	22	60	82	0.150
			SCS	9	19	34	0.124
		StackOverflow-Android-Train	*UNIF*	**104**	**164**	**188**	**0.465**
			CODEnn	36	91	117	0.215
			SCS	11	24	47	0.138

Table 8: Summary of evaluation results.

We relied on an automated evaluation pipeline to provide reproducible and scalable evaluation of code search results. With this we scaled evaluation to a much larger set of benchmark queries (*Android-287*). While performance may vary depending on the similarity threshold chosen, we derived our similarity threshold choice through manual evaluation of answers produced by two techniques (*CODEnn* and *UNIF*) and believe it correlates well with human judgment. This threshold produces in evaluation results for *CODEnn* that roughly correspond to those found in its original paper [15].

7 RELATED WORK

Recent works from both academia and industry have explored the realm of code search. *NCS* [26] presented a simple yet effective unsupervised model. *CODEnn* [15] and *SCS* [19] provided a deep learning approach by using sophisticated neural networks. The supervised code search techniques build on the idea of bimodal embeddings of source code and natural language.

Existing work in natural language processing [4, 9, 13] has explored constructing embeddings for two languages with little bilingual data. It is possible that some of these techniques might be applicable to the code search task, and address some of the issues we identified during our analysis. However, in the code search task the embedding alignment we care about is not just at the word (i.e. token) level, but rather should aggregate succesfully to whole code snippets and queries.

Other than code search, a line of work has explored enhancing developing productivity by exploiting an aligned corpus of code and natural language. Allamanis et al. [2] proposes a probablistic model to synthesize a program snippet from a natural language query. Bayou [25] is a system that uses deep neural networks to synthesize code programs from a few API calls and a natural language input. CODE-NN [21] uses LSTM networks to produce natural language descriptions given a code snippet.

Interest in applications of neural networks to software engineering has increased significantly. Existing work has introduced neural networks to identify software defects [10], guide program synthesis [5, 30], enable new representations for program analysis [1, 3], facilitate code reuse [17]. The models we present here make use of these technologies to varying degrees to explore the design space and impact of these choices on code search quality.

8 CONCLUSION

In this paper we questioned some of the assumptions made in previous works (e.g. sequence-based models, docstring supervision), and find that these assumptions are not always justifiable. We compared three state-of-the-art techniques for neural code search with a novel extension of our own, and provided quantitative evidence for key design considerations.

We showed that supervision, shown by extending *NCS* to *UNIF*, can improve performance over an unsupervised technique. We suggest baselining against an unsupervised neural code search system and comparing incremental improvements, which should be weighed against the time and resources required to collect supervision data.

We found that more sophisticated networks did not necessarily payoffs: *UNIF* outperformed both *CODEnn* and *SCS*. With this observation in mind, we suggest evaluating simple architectures before incorporating more sophisticated components such as RNNs into their systems.

Finally, we showed that an ideal training corpus that resembled eventual user queries provided impressive improvements for all supervised techniques. We suggest considering the extent to which a training corpus resembles eventual user queries for optimal results, and exploring the possibility of better training corpora, rather than assuming a code/docstring corpus will provide the best performance.

REFERENCES

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

[2] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. Bimodal modelling of source code and natural language. In *International Conference on Machine Learning*, pages 2123–2132, 2015.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *arXiv preprint arXiv:1803.09473*, 2018.

[4] Mikel Artetxe, Gorka Labaka, and Eneko Agirre. Learning bilingual word embeddings with (almost) no bilingual data. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 451–462, 2017.

[5] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

[6] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

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

[8] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[9] Alexis Conneau, Guillaume Lample, Marc’Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Word translation without parallel data. *arXiv preprint arXiv:1710.04087*, 2017.

[10] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim. A deep tree-based model for software defect prediction. *arXiv preprint arXiv:1802.00921*, 2018.

[11] William Bruce Frakes and Ricardo Baeza-Yates. *Information retrieval: Data structures & algorithms*, volume 331. prentice Hall Englewood Cliffs, NJ, 1992.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[13] Edouard Grave, Armand Joulin, and Quentin Berthet. Unsupervised alignment of embeddings with wasserstein procrustes. *arXiv preprint arXiv:1805.11222*, 2018.

[14] Xiadong Gu. Private Communication, 2018.

[15] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*, pages 933–944. ACM, 2018.

[16] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. Deep code search github repository, 2018. URL: https://github.com/guxd/deep-code-search/#54130b6be41fc5d73c4ebb8422942a7b53ad4024.

[17] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. Semantically enhanced software traceability using deep learning techniques. In *Software Engineering (ICSE), 2017 IEEE/ACM 39th International Conference on*, pages 3–14. IEEE, 2017.

[18] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[19] Hamel Husain and Ho-Hsiang Wu. How to create natural language semantic search for arbitrary objects with deep learning, 2018. URL: https://towardsdatascience.com/semantic-code-search-3cd6d244a39c.

[20] Hamel Husain and Ho-Hsiang Wu. Towards natural language semantic code search, 2018. URL: https://githubengineering.com/towards-natural-language-semantic-code-search/.

[21] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 2073–2083, 2016.

[22] Sifei Luan, Di Yang, Koushik Sen, and Satish Chandra. Aroma: Code recommendation via structural code search. *CoRR*, abs/1812.01158, 2018. URL: http://arxiv.org/abs/1812.01158, `arXiv:1812.01158`.

[23] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and Optimizing LSTM Language Models. *arXiv preprint arXiv:1708.02182*, 2017.

[24] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[25] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Bayesian sketch learning for program synthesis. *CoRR*, abs/1703.05698, 2017. URL: http://arxiv.org/abs/1703.05698, `arXiv:1703.05698`.

[26] Saksham Sachdev, Hongyu Li, Sifei Luan, Seohyun Kim, Koushik Sen, and Satish Chandra. Retrieval on source code: a neural code search. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 31–41. ACM, 2018.

[27] Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information processing & management*, 24(5):513–523, 1988.

[28] Inc. Stack Exchange. datastack exchange data dump, 2018. URL: https://archive.org/details/stackexchange.

[29] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[30] Chenglong Wang, Po-Sen Huang, Alex Polozov, Marc Brockschmidt, and Rishabh Singh. Execution-guided neural program decoding. *arXiv preprint arXiv:1807.03100*, 2018.

[31] Ross Wilkinson and Philip Hingston. Using the cosine measure in a neural network for document retrieval. In *Proceedings of the 14th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR ’91, pages 202–210, New York, NY, USA, 1991. ACM. URL: http://doi.acm.org/10.1145/122860.122880, `doi:10.1145/122860.122880`.