code2vec: Learning Distributed Representations of Code

URI ALON, Technion, Israel
MEITAL ZILBERSTEIN, Technion, Israel
OMER LEVY, University of Washington
ERAN YAHAV, Technion, Israel

We present a neural model for representing snippets of code as continuous distributed vectors ("code embeddings"). The main idea is to represent code as a collection of paths in its abstract syntax tree, and aggregate these paths, in a smart and scalable way, into a single fixed-length *code vector*, which can be used to predict semantic properties of the snippet.

We demonstrate the effectiveness of our approach by using it to predict a method's name from the vector representation of its body. We evaluate our approach by training a model on a dataset of 14M methods. We show that code vectors trained on this dataset can predict method names from files that were completely unobserved during training. Furthermore, we show that our model learns useful method name vectors that capture semantic similarities, combinations, and analogies.

Comparing previous techniques over the same data set, our approach obtains a relative improvement of over 75%, being the first to successfully predict method names based on a large, cross-project, corpus.

Additional Key Words and Phrases: Big Code, Machine Learning, Distributed Representations

1 Introduction

Distributed representations of words [Mikolov et al. 2013a,b; Pennington et al. 2014], sentences, paragraphs, and documents [Le and Mikolov 2014] played a key role in unlocking the potential of neural networks for NLP tasks [Bengio et al. 2003; Collobert and Weston 2008; Glorot et al. 2011; Socher et al. 2011; Turian et al. 2010; Turney 2006]. Methods for learning distributed representations produce low-dimensional vector representations for objects, referred to as *embeddings*, such that semantically similar objects are mapped to close vectors.

Goal: The goal of this paper is to learn *code embeddings*, continuous distributed vectors for representing snippets of code. By learning code embeddings, our long term goal is to enable the application of neural techniques to a wide-range of programming-languages tasks. In this paper, we use the motivating task of *semantic labeling of code snippets*.

```
String[] f(final String[] array) {
    final String[] newArray = new String[array.length];
    for (int index = 0; index < array.length; index++) {
        newArray[array.length - index - 1] = array[index];
    }
    return newArray;
}
```

Predictions

reverseArray	77.34%
reverse	18.18%
subArray	1.45%
copyArray	0.74%

Fig. 1. A code snippet and its predicted labels as computed by our model.

Motivating task: semantic labeling of code snippets Consider the method in Figure 1. The method only contains low-level assignments to arrays, but a human reading the code may (correctly) label it as performing the *reverse* operation. Our goal is to be able to predict such labels automatically. The right hand side of Figure 1 shows the labels predicted automatically using our approach. The most likely prediction (77.34%) is *reverseArray*. Section 6 provides additional examples.

Intuitively, this problem is hard because it requires *learning a correspondence* between the *entire content of a method* and a semantic label. That is, it requires aggregating possibly hundreds of expressions and statements from the method body into a single, descriptive label.

A	≈B	A	≈B
size	getSize, length, getCount, getLength	executeQuery	executeSql, runQuery, getResultSet
active	isActive, setActive, getIsActive, enabled	actionPerformed	itemStateChanged, mouseClicked, keyPressed
done	end, stop, terminate	toString	getName, getDescription, getDisplayName
toJson	serialize, toJsonString, getJson, asJson,	equal	eq, notEqual, greaterOrEqual, lessOrEqual
run	execute, call, init, start	error	fatalError, warning, warn

Table 1. Semantic similarities between method names.

Our approach we present a novel framework for predicting program properties using neural networks. Our main contribution is a neural network that learns *code embeddings* - continuous distributed vector representations for code. The code embeddings allow us to model correspondence between code snippet and labels in a natural and effective manner.

Our neural network architecture uses a representation of code snippets that leverages the structured nature of source code, and learns to aggregate multiple syntactic paths into a single vector. This ability is fundamental for the application of deep learning in programming languages. By analogy, word embeddings in natural language processing (NLP) started a revolution of application of deep learning for NLP tasks.

The input to our model is a code snippet and a corresponding tag, label, caption, or name. This tag expresses the semantic property that we wish the network to model, for example: a tag, name that should be assigned to the snippet, or the name of the method, class, or project that the snippet was taken from. Let C be the code snippet and $\mathcal{L}$ be the corresponding label or tag. Our underlying hypothesis is that *the distribution of labels can be inferred from syntactic paths in* C. Our model therefore attempts to learn the tag distribution, conditioned on the code: $P(\mathcal{L}|C)$.

We demonstrate the effectiveness of our approach for the task of predicting a method's name given its body. This problem is important as good method names make code easier to understand and maintain. A good name for a method provides a high-level summary of its purpose. Ideally, *"If you have a good method name, you don't need to look at the body."* [Fowler and Beck 1999]. Choosing good names can be especially critical for methods that are part of public APIs, as poor method names can doom a project to irrelevance [Allamanis et al. 2015; Høst and Østvold 2009].

Capturing semantic similarity between names During the process of learning code vectors, a parallel vocabulary of vectors of the tags is learned. When using our model to predict method names, the method-name vectors provide surprising semantic similarities and analogies. For example, $vector($`equals`$)+vector($`toLower`$)$ results in a vector that is closest to $vector($`equalsIgnoreCase`$)$.

Similar to the famous NLP example of: $vec(\text{“}king\text{''})-vec(\text{“}man\text{''})+vec(\text{“}woman\text{''}) \approx vec(\text{“}queen\text{''})$ [Mikolov et al. 2013c], our model learns analogies that are relevant to source code, such as: *“*`receive` *is to* `send` *as* `download` *is to* `upload`*”*. Table 1 shows additional examples, and section 6.4 provides a detailed discussion.

1.1 Challenges: Representation and Attention

Assigning a semantic tag to a code snippet (such as a name to a method) is an example for a class of problems that require a compact semantic descriptor of a snippet. The question is how to represent code snippets in a way that captures some semantic information, is reusable across programs, and can be used to predict properties such as a label for the snippet. This leads to two challenges:

- Representing a snippet in a way that enables learning across programs
- Learning which parts in the representation are relevant to prediction of the desired property

Representation NLP methods typically treat text as a linear sequence of tokens. Indeed, many existing approaches also represent code as a token stream [Allamanis et al. 2014, 2016; Allamanis

and Sutton 2013; Hindle et al. 2012; Movshovitz-Attias and Cohen 2013; White et al. 2015]. However, as observed previously [Alon et al. 2018; Bielik et al. 2016; Raychev et al. 2015], programming languages can greatly benefit from representations that leverage the structured nature of their syntax.

Following previous works [Alon et al. 2018; Raychev et al. 2015], we use paths in the program's abstract syntax tree (AST) as our representation. By representing a code snippet using its syntactic paths, we can capture regularities that reflect common code patterns. We then represent the snippet as a bag (multiset) of its extracted paths. The challenge is then *how to aggregate a bag of contexts, and which paths to focus on for making a prediction*.

Attention Intuitively, the problem is to learn a correspondence between a bag of path-contexts and a label. Representing each bag of path-contexts *monolithically* is going to suffer from sparsity – even similar methods will not have the *exact* same bag of path-contexts. We therefore need a *compositional* mechanism that can aggregate a bag of path-contexts such that bags that yield the same label are mapped to close vectors. Such a compositional mechanism would be able to generalize and represent new unseen bags by leveraging the fact that during training it observed the individual path-contexts and their components (paths, values, etc) as parts of other bags.

To address this challenge, which is the focus of this paper, we use a novel attention network architecture. Attention models have gained much popularity recently, mainly for neural machine translation (NMT) [Bahdanau et al. 2014; Luong et al. 2015; Vaswani et al. 2017], reading comprehension [Levy et al. 2017; Seo et al. 2016], image captioning [Xu et al. 2015], and more [Ba et al. 2014; Bahdanau et al. 2016; Chorowski et al. 2015; Mnih et al. 2014].

In our model, neural attention learns how much focus ("attention") should be given to each element in a bag of path-contexts. It allows us to precisely aggregate the information captured in each individual path-context into a single vector that captures information about the entire code snippet. As we show in section 6.4, the weights allocated by our attention mechanism can be visualized to understand the relative importance of each path-context in a prediction. The attention mechanism is *learned simultaneously with the embeddings, optimizing both the atomic representations of paths and the ability to compose multiple contexts into a single code vector*.

Soft and hard attention The terms "soft" and "hard" attention were proposed for the task of image caption generation by Xu et al. [2015]. Applied in our setting, *soft-attention* means that weights are distributed "softly" over all path-contexts in a code snippet, while *hard-attention* refers to selection of a single path-context to attend to at a time.

1.2 Existing Techniques

The problem of predicting program properties by learning from big code has seen tremendous interest and progress in recent years [Allamanis et al. 2014, 2017, 2016; Allamanis and Sutton 2013; Alon et al. 2018; Bielik et al. 2016; Hindle et al. 2012; Mishne et al. 2012; Movshovitz-Attias and Cohen 2013; Raychev et al. 2016a, 2015, 2014; White et al. 2015]. The ability to predict semantic properties of a program without running it, and with little or no semantic analysis at all, has a wide range of applications: predicting names for program entities [Allamanis et al. 2015; Alon et al. 2018; Raychev et al. 2015], code completion [Mishne et al. 2012; Raychev et al. 2014], code summarization [Allamanis et al. 2016], and more (see [Allamanis et al. 2017; Vechev and Yahav 2016] for a survey).

Distributed representations of code identifiers were first suggested by Allamanis et al. [2015], and used to predict variable, method, and class names based on token context features. Allamanis et al. [2016] were also the first to consider the problem of predicting method names. Their technique used a Convolutional Neural Network (CNN) where locality in the model is based on textual locality in source code. While their technique works well when training and prediction are performed within

the scope of the same project, they report poor results when used across different projects (as we reproduce in section 6.1). Thus, the problem of predicting method names based on a large corpus has remained an open problem until now. To the best of our knowledge, our technique is the first to train an effective cross-project model for predicting method names.

Syntax-based contexts have been used by Bielik et al. [2016]; Raychev et al. [2016a]. Other than targeting different tasks, our work differs in two major aspects. First, these works traverse the AST only to identify a *context node*, and do not use the information contained in the path itself. In contrast, our model uses the path itself as an input to the model, and can therefore generalize using this information when a known path is observed, even when the nodes in its ends have never been seen by the model before. Second, our work differs in a major aspect that these models attempt to find a *single most informative context* for each prediction. This approach resembles *hard-attention*, in which the hardness is an inherent part of their model. In contrast, we suggest to use *soft-attention*, that uses multiple contexts for prediction, with different weights for each. In previous works which used non-neural techniques, soft-attention is not even expressible. In section 5 we compare distributed representations ("neural") and symbolic representations ("non-neural") for predicting program properties, and show the advantages of our approach, which uses distributed representations, in terms of generalization ability and space complexity.

The use of soft-attention over syntactic paths is the main understanding that provides this work much better results than previous works. We compare our model with an equivalent model that uses hard-attention in section 6.2, and show that *soft-attention* is more efficient for modeling code.

1.3 Contributions

The main contributions of this paper are:

- A path-based attention model for learning vectors for arbitrary-sized snippet of code. This model allows to embed a program, which is a discrete object, into a continuous space, such that it can be fed into a deep learning pipeline for various tasks.
- As a benchmark for our approach, we perform a quantitative evaluation for predicting cross-project method names, trained on more than 14M methods of real-world data, and compared with previous works. Experiments show that our approach achieves significantly better results than previous works which used Long Short-Term Memory networks (LSTMs) and CNNs (F1 score of 59.5 compared to 33.9).
- A qualitative evaluation that interprets the attention that the model has learned to give to the different path-contexts when making predictions.
- A collection of method name embeddings, which often assign semantically similar names to similar vectors, and even allows to compute analogies using simple vector arithmetic.
- An analysis that shows the significant advantages in terms of generalization ability and space complexity of our model, compared to previous non-neural models.

2 Overview

In this section we demonstrate how our model assigns different vectors to similar snippets of code, *in a way that captures the subtle differences between them*. The vectors are useful for making a prediction about each snippet, even though none of these snippets has been exactly observed in the training data.

The main idea of our approach is to extract syntactic paths from within a code snippet, represent them as a bag of distributed vector representations, and use an attention mechanism to compute a weighted average of the path vectors in order to produce a single *code vector*. Finally, this code vector can be used for various tasks, such as to predict a likely name for the whole snippet.

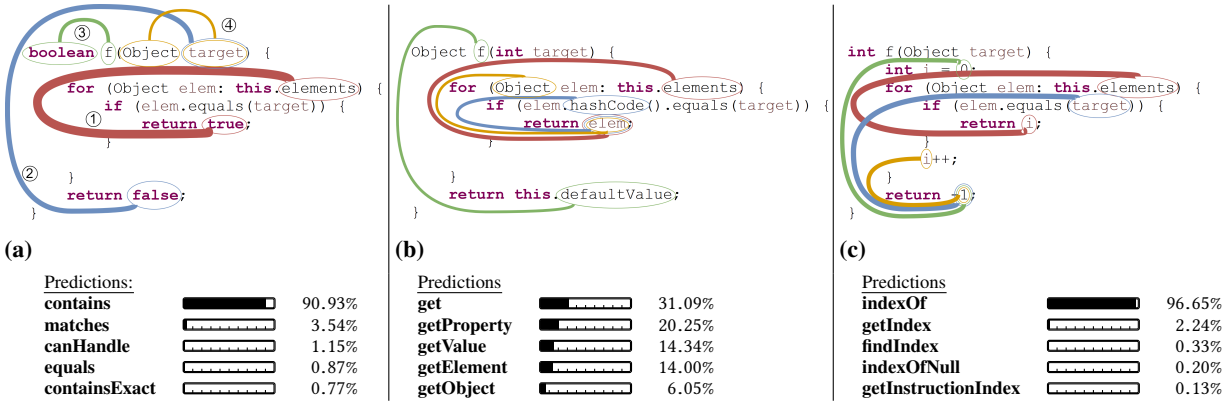

Fig. 2. An example for three methods that have a similar syntactic structure, but our model successfully captures the delicate differences between them and manages to predict meaningful names. The widths of the colored paths are proportional to the attention that each path was given.

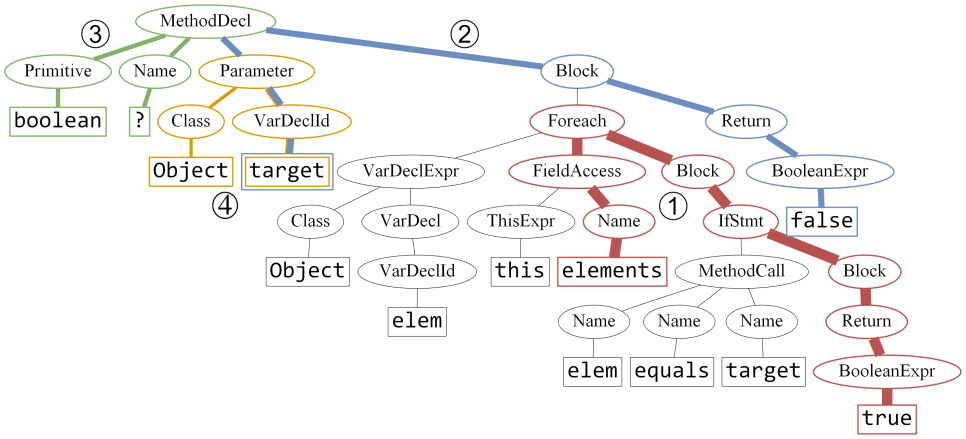

Fig. 3. The paths of Figure 2a, shown on the AST of the same snippet. The width of each colored path is proportional to the attention it was given (**red** ①: **0.23**, **blue** ②: **0.14**, **green** ③: **0.09**, **orange** ④: **0.07**).

2.1 Motivating Example

Since method names are usually descriptive and accurate labels for code snippets, we demonstrate our approach for the task of learning code vectors for method bodies, and predicting the method name given the body. In general, the same approach can be applied to any snippet of code that has a corresponding label.

Consider the three Java methods of Figure 2. These methods share a similar syntactic structure: they all (i) have a single parameter named `target` (ii) iterate over a field named `elements` and (iii) have an `if` condition inside the loop body. The main differences are that the method of Fig. 2a returns `true` when `elements` *contains* `target` and `false` otherwise; the method of Fig. 2b returns the element from `elements` which `target` *equals to its* `hashCode`; and the method of Fig. 2c returns the *index of* `target` in `elements`. Despite their shared characteristics, our model captures

the subtle differences and predicts the descriptive method names: `contains`, `get`, and `indexOf` respectively.

Path extraction First, each query method in the training corpus is parsed to construct an AST. Then, the AST is traversed and syntactic paths between AST leaves are extracted. Each path is represented as a sequence of AST nodes, linked by up and down arrows, which symbolize the up or down link between adjacent nodes in the tree. The path composition is kept with the values of the AST leaves it is connecting, as an (x_s, p, x_f) tuple, which we refer to as a *path-context*. These terms are defined formally in section 3. Figure 3 portrays the path-contexts that were given the most attention by the model, on the AST of the method from Figure 2a.

Distributed representation of contexts Each of the path and leaf-values of a path-context is mapped to its corresponding real-valued vector representation, or its *embedding*. Then, the three vectors of each context are concatenated to a single vector that represents that path-context. During training, the embeddings are learned jointly with the attention parameter and the rest of the network parameters.

Path-attention network The Path-Attention network aggregates multiple path-contexts embeddings into a single vector that represents the whole method body. Attention is the mechanism that learns to score each path-context, such that higher attention is reflected in a higher score. These multiple embeddings are aggregated using the attention scores into a single *code vector*. The network then predicts the probability for each target method name given the code vector. The network architecture is described in section 4.

Path-attention interpretation While it is usually difficult or impossible to interpret specific values of vector components in neural networks, it is possible and interesting to observe the attention scores that each path-context was given by the network. The widths of the paths in Figure 2 and Figure 3 are proportional to the attention score that each of these path-contexts was given. For example, it can be seen in Figure 3 that the **red** ① path-context, which spans from the field `elements` to the return value `true` was given the highest attention. For comparison, the **blue** ② path-context, which spans from the parameter `target` to the return value `false` was given a lower attention.

Consider the **red** ① path-context of Figure 2a and Figure 3. As we explain in section 3, this path is represented as:

(`elements`, Name↑FieldAccess↑Foreach↓Block↓IfStmt↓Block↓Return↓BooleanExpr, `true`)

Inspecting this path node-by-node reveals that this single path captures the main functionality of the method: the method iterates over a field called `elements`, and for each of its values it checks an `if` condition; if the condition is true, the method returns `true`. Since we use soft-attention, the final prediction takes into account other paths as well, such as paths that describe the `if` condition itself, but it can be understood why the model gave this path the highest attention.

Figure 2 also shows the top-5 suggestions from the model for each method. As can be seen in all of the three examples, in many cases most of the top suggestions are very similar to each other and all of them are descriptive regarding the method. Observing the top-5 suggestions in Figure 2a shows that two of them (`contains` and `containsExact`) are very accurate, but it can also be imagined how a method called `matches` would share similar characteristics. A method called `matches` is also likely to have an `if` condition inside a `for` loop, and to return `true` if the condition is true.

Another interesting observation is that the **orange** ④ path-context of Figure 2a which spans from `Object` to `target` was given a lower attention than other path-contexts in the same method, but *higher attention than the same path-context in Figure 2c*. This demonstrates how attention is not constant, but given with respect to the other path-contexts in the code.

Comparison with n-grams The method in Figure 2a shows the four path-contexts that were given the most attention during the prediction of the method name `contains`. Out of them, the **orange**

④ path-context spans between two consecutive tokens: `Object` and `target`. This might create the (false) impression that representing this method as a bag-of-bigrams could be as expressive as syntactic paths. However, as can be seen in Figure 3, the **orange** ④ path goes through an AST node of type *Parameter*, which uniquely distinguishes it from, for example, a local variable declaration of the same name and type. This shows that a model using a syntactic representation of a code snippet can distinguish between two snippets of code that other representations cannot, and is therefore more useful in predicting properties of the whole code snippet.

Key aspects The illustrated examples highlight several key aspects of our approach:

- A code snippet can be efficiently represented as a bag of path-contexts.
- Using a single context is not enough to make an accurate prediction. An attention-based neural network can identify the importance of multiple path-contexts, and aggregate them accordingly to make a prediction.
- Subtle differences between code snippets are easily distinguished by our model, even if the code snippets have a similar syntactic structure and share many common tokens and n-grams.
- Large corpus, cross-project prediction of method names is possible using this model.
- Although our model is based on a neural network, the model is human-interpretable and provides interesting observations.

3 Background - Representing Code using AST Paths

In this section, we briefly describe the representation of a code snippet as a set of syntactic paths in its abstract syntax tree (AST). This representation is based on the general-purpose approach for representing program elements by Alon et al. [2018], which was shown to be useful mainly for predicting properties of a single node in an AST. We extend their representation to handle whole snippet of code, and use it as input to our path-attention neural network.

We start by defining an AST, a path and a path-context.

Definition 1 (Abstract Syntax Tree)**.** An Abstract Syntax Tree (AST) for a code snippet C is a tuple $\langle N, T, X, s, \delta, \phi \rangle$ where N is a set of nonterminal nodes, T is a set of terminal nodes, X is a set of values, $s \in N$ is the root node, $\delta : N \rightarrow (N \cup T)^*$ is a function that maps a nonterminal node to a list of its children, and $\phi : T \rightarrow X$ is a function that maps a terminal node to an associated value. Every node except the root appears exactly once in all the lists of children.

Next, we define AST paths. For convenience, in the rest of this section we assume that all definitions refer to a single AST $\langle N, T, X, s, \delta, \phi \rangle$.

An AST path is a path between nodes in the AST, starting from one terminal, ending in another terminal, passing through an intermediate nonterminal in the path which is a common ancestor of both terminals. More formally:

Definition 2 (AST path)**.** An AST-path of length k is a sequence of the form: $n_1 d_1 ... n_k d_k n_{k+1}$, where $n_1, n_{k+1} \in T$ are terminals, for $i \in [2..k]$: $n_i \in N$ are nonterminals and for $i \in [1..k]$: $d_i \in \{\uparrow, \downarrow\}$ are movement directions (either up or down in the tree). If $d_i = \uparrow$, then: $n_i \in \delta(n_{i+1})$; if $d_i = \downarrow$, then: $n_{i+1} \in \delta(n_i)$. For an AST-path p, we use $start(p)$ to denote n_1 - the starting terminal of p; and $end(p)$ to denote n_{k+1} - its final terminal.

Using this definition we define a *path-context* as a tuple of an AST path and the values associated with its terminals:

Definition 3 (Path-context)**.** Given an AST Path p, its path-context is a triplet $\langle x_s, p, x_f \rangle$ where $x_s = \phi(start(p))$ and $x_f = \phi(end(p))$ are the values associated with the start and end terminals of p.

That is, a path-context describes two actual tokens with the syntactic path between them.

Example 3.1. A possible path-context that represents the statement: "`x = 7;`" would be:

$$\langle \texttt{x}, (NameExpr \uparrow AssignExpr \downarrow IntegerLiteralExpr), \texttt{7} \rangle$$

Practically, to limit the size of the training data and reduce sparsity, it is possible to limit the paths by different aspects. Following earlier works, we limit the paths by maximum *length* — the maximal value of k, and limit the maximum *width* — the maximal difference in child index between two child nodes of the same intermediate node. These values are determined empirically as hyperparameters of the model and following conclusions of previous works.

4 Model

In this section we describe our model in detail. Section 4.1 describes the way the input source code is represented; section 4.2 describes the architecture of the neural network; Section 4.3 describes the training process, and Section 4.4 describes the way the trained model is used for inference. Finally Section 4.5 discusses some of the model design choices, and compares the architecture to prior art.

High-level view At a high-level, the key point is that a code snippet is composed of a bag of contexts, and each context is represented by a vector that its values are learned. The values of this vector capture two distinct goals: (i) the semantic meaning of this context, and (ii) the amount of attention this context should get.

The problem is as follows: given an arbitrarily large number of context vectors, we need to aggregate them into a single vector. Two trivial approaches would be to learn the most important one of them, or to use them all by vector-averaging them. These alternatives will be discussed in section 6.2, and the results of implementing these two alternatives are shown in table 4 ("hard-attention" and "no-attention") to yield poor results.

The main understanding in this work is that *all* context vectors need to be used, but the model should be let to learn how much focus to give each vector. This is done by learning how to average context vectors in a weighted manner. The weighted average is obtained by weighting each vector by its dot product with another global attention vector. The vector of each context and the attention vector are trained and learned *simultaneously*, using the standard neural approach of backpropagation.

4.1 Code as a Bag of Path-Contexts

Our path-attention model receives as input a code snippet in some programming language and a parser for that language.

Representing a snippet of code We denote by Rep the representation function (also known as a feature function) which transforms a real-world object (e.g., code snippet) into a mathematical object that can be used in a learning model. Given a code snippet C and its AST $\langle N, T, X, s, \delta, \phi \rangle$, we denote by $TPairs$ the set of all pairs of AST terminal nodes (excluding pairs that contains a node and itself):

$$TPairs(C) = \left\{ (t_i, t_j) \,|\, t_i, t_j \in termNodes(C) \wedge i \neq j \right\}$$

where $termNodes$ is a mapping between a code snippet and the set of terminal nodes in its AST. We represent C as the set of path-contexts that can be derived from it:

$$Rep(C) = \left\{ (x_s, p, x_f) \;\middle|\; \begin{array}{l} \exists (t_s, t_f) \in TPairs(C) : \\ x_s = \phi(t_s) \wedge x_f = \phi(t_f) \\ \wedge\, start(p) = t_s \wedge end(p) = t_f \end{array} \right\}$$

that is, C is represented as the set of triplets $\langle x_s, p, x_f \rangle$ such that x_s and x_f are values of AST terminals, and p is the AST path that connects them. For example, the representation of the code snippet from Figure 2a contains, among others, the four AST paths of Figure 3.

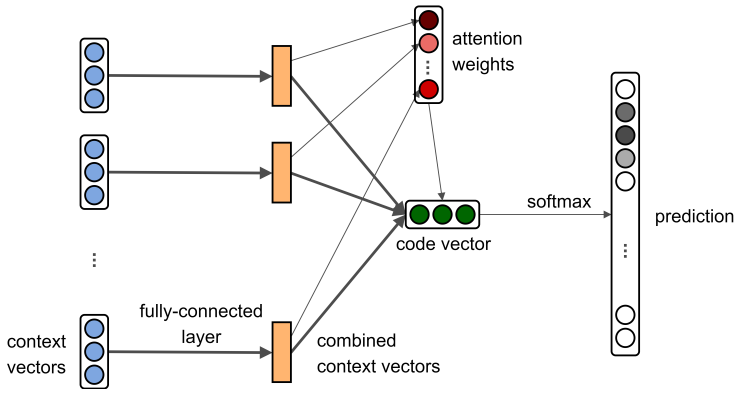

Fig. 4. The architecture of our path-attention network. A *full-connected layer* learns to combine embeddings of each path-contexts with itself; attention weights are learned using the combined context vectors, and used to compute a *code vector*. The code vector is used to predicts the label.

4.2 Path-Attention Model

We describe our model from from left-to-right (Fig. 4). We define two embedding vocabularies: $value\_vocab$ and $path\_vocab$, which are matrices in which every row corresponds to an embedding associated with a certain object:

$$value\_vocab \in \mathbb{R}^{|X| \times d}$$
$$path\_vocab \in \mathbb{R}^{|P| \times d}$$

where as before, X is the set of values of AST terminals that were observed during training, and P is the set of AST paths. Looking up an embedding is simply picking the appropriate row of the matrix. For example, if we consider Figure 2a again, $value\_vocab$ contains rows for each token value such as `boolean`, `target` and `Object`. $path\_vocab$ contains rows which are mapped to each of the AST paths of Figure 3 (without the token values), such as the **red** ① path: $Name \uparrow FieldAccess \uparrow Foreach \downarrow Block \downarrow IfStmt \downarrow Block \downarrow Return \downarrow BooleanExpr$. The values of these matrices are initialized randomly and are learned simultaneously with the network during training.

The embeddings size $d \in \mathbb{N}$ is the dimensionality hyperparameter. d is determined empirically, limited by the training time, model complexity and the GPU memory, and typically ranges between 100-500. For convenience, we refer to the embeddings of both the paths and the values as vectors of the same size d, but in general they can be of different sizes.

A bag of path-contexts $\mathcal{B} = \{b_1, ..., b_n\}$ that were extracted from a given code snippet is fed into the network. Let $b_i = \langle x_s, p_j, x_f \rangle$ be one of these path-contexts, such that $\{x_s, x_f\} \in X$ and $p_j \in P$. Each component of a path-context is looked-up and mapped to its corresponding embedding. The three embeddings of each path-context are concatenated to a single *context vector*: $c_i \in \mathbb{R}^{3d}$ that represents that path-context:

$$c_i = embedding\left(\langle x_s, p_j, x_f \rangle\right) = \left[\ word\_vocab_s\ ;\ path\_vocab_j\ ;\ word\_vocab_f\ \right] \in \mathbb{R}^{3d} \quad (1)$$

For example, for the **red** ① path-context from Figure 3, its context vector would be the concatenation of the vectors of `elements`, the **red** ① path, and `true`.

Fully connected layer Since every context vector c_i is formed by a concatenation of three independent vectors, a fully connected layer learns to *combine* its components. This is done separately for each context vector, using the same learned combination function. Here, $\tilde{c}_i$ is the output of the fully

connected layer, which we refer to as a *combined context vector*, computed for a path-context b_i. The computation of this layer can be described simply as:

$$\tilde{c}_i = tanh\,(W \cdot c_i)$$

where $W \in \mathbb{R}^{d \times 3d}$ is a learned weights matrix and $tanh$ is the hyperbolic tangent function. The height of the weights matrix W determines the size of $\tilde{c}_i$, and for convenience is the same size d as before, but in general it can be of a different size to determines the size of the *target* vector. $tanh$ is the hyperbolic tangent element-wise function, a commonly used monotonic nonlinear activation function which outputs values that range $(-1, 1)$, which increases the expressiveness of the model.

That is, the fully connected layer "compresses" a context vector of size $3d$ into a combined context vector of size d by multiplying it with a weights matrix, and then applies the $tanh$ function to each element of the vector separately.

Aggregating multiple contexts into a single vector representation with attention The attention mechanism computes a weighted average over the combined context vectors, and its main job is to compute a scalar weight to each of them. An attention parameter $a \in \mathbb{R}^d$ is initialized randomly and learned simultaneously with the network. Given the combined context vectors: $\{\tilde{c}_1, ..., \tilde{c}_n\}$, the attention weight α_i of each $\tilde{c}_i$ is computed as the normalized inner product between the combined context vector and the global attention parameter a:

$$\text{attention weight } \alpha_i = \frac{\exp(\tilde{c}_i^T \cdot a)}{\sum_{j=1}^{n} \exp(\tilde{c}_j^T \cdot a)}$$

The exponents in the equations are used only to make the attention weights positive, and they are divided by their sum to have a sum of 1, as a standard softmax function.

The aggregated code vector $v \in \mathbb{R}^d$, which represents the whole code snippet, is a linear combination of the combined context vectors $\{\tilde{c}_1, ..., \tilde{c}_n\}$ factored by their attention weights:

$$\text{code vector } v = \sum_{i=1}^{n} \alpha_i \cdot \tilde{c}_i \tag{2}$$

that is, the attention weights are non-negative and their sum is 1, and they are used as the factors of the combined context vectors $\tilde{c}_i$. Thus, attention can be viewed as a weighted average, where the weights are learned and calculated with respect to the other members in the bag of path-contexts.

Prediction Prediction of the tag is performed using the code vector. We define a tags vocabulary which is learned as part of training:

$$tags\_vocab \in \mathbb{R}^{|Y| \times d}$$

where Y is the set of tag values found in the training corpus. Similarly as before, the embedding of tag_i is row i of $tags\_vocab$. For example, looking at Figure 2a again, $tags\_vocab$ contains rows for each of `contains`, `matches` and `canHandle`. The predicted distribution of the model $q\,(y)$ is computed as the (softmax-normalized) dot product between the code vector v and each of the tag embeddings:

$$for\ y_i \in Y:\ q\,(y_i) = \frac{\exp(v^T \cdot tags\_vocab_i)}{\sum_{y_j \in Y} \exp(v^T \cdot tags\_vocab_j)}$$

that is, the probability that a specific tag y_i should be assigned to the given code snippet C is the normalized dot product between the vector of y_i and the code vector v.

4.3 Training

For training the network we use cross-entropy loss [Rubinstein 1999, 2001] between the predicted distribution q and the "true" distribution p. Since p is a distribution that assigns a value of 1 to the tag that was seen in the training example and 0 otherwise, the cross-entropy loss for a single example can be expressed as:

$$\mathcal{H}(p||q) = -\sum_{y \in Y} p(y) \log q(y) = -\log q(y_{true})$$

where y_{true} is the actual tag that was seen in the example. That is, the loss is the negative logarithm of $q(y_{true})$, the probability that the model assigns to y_{true}. As $q(y_{true})$ tends to 1, the loss approaches zero. The further $q(y_{true})$ goes below 1, the greater the loss becomes.

Training the network is performed using any gradient descent based algorithm, and the standard approach of backpropagating the training error through each of the learned parameters.

4.4 Using the trained network

A trained network can be used for two main purposes: (i) Use the code vector v itself in a down-stream task, and (ii) Use the network to predict tags for new, unseen code.

Using the code vector An unseen code can be fed into the trained network exactly the same as in the training step, up to the computation of the code vector (Eq. (2)). This code embedding can now be used in another deep learning pipeline for various tasks such as finding similar programs, code search, refactoring suggestion, and code summarization.

Predicting tags and names the network can also be used to predict tags and names for unseen code. In this case we also compute the code vector v using the weights and parameters that were learned during training, and prediction is done by finding the closest target tag:

$$prediction(C) = argmax_{\mathcal{L}} P(\mathcal{L}|C) = argmax_{\mathcal{L}} \left\{q_{v_C}(y_{\mathcal{L}})\right\}$$

where q_{v_C} is the predicted distribution of the model, given the code vector v_C.

Scenario-dependant variants For simplicity, we describe a network that predicts a single label, but the same architecture can be adapted for slightly different scenarios. For example, in a multi-tagging scenario [Tsoumakas and Katakis 2006], each code snippet contains multiple true tags as in StackOverflow questions. Another example is predicting a sequence of target words such as method documentation. In the latter case, the attention parameter should be used to re-compute the attention weights after each predicted token, given the previous prediction, as commonly done in neural machine translation [Bahdanau et al. 2014; Luong et al. 2015].

4.5 Design Decisions

Bag of contexts We represent a snippet of code as an unordered bag of path-contexts. This choice reflects our hypothesis that the *existence* of path-contexts in a method body is more significant than their internal location or order.

An alternative representation is to sort path-contexts according to a predefined order (e.g., order of their occurrence). However, unlike natural language, there is no predetermined location in a method where the main attention should be focused. An important path-context can appear anywhere in a method body (and span throughout the method body).

Working with syntactic paths The main contribution of this work is its ability to aggregate multiple contexts into a fixed-length vector in a weighted manner, and use the vector to make a prediction. In general, our proposed model is not bound to any specific representation of the input program, and can be applied in a similar way to a "bag of contexts" in which the contexts are designed for a specific task, or contexts that were produced using semantic analysis. Specifically, we chose to

use a syntactic representation that is similar to Alon et al. [2018] because it was shown to be useful as a representation for modelling programming languages in machine learning models, and more expressive than n-grams and manually-designed features.

Large corpus, simple model Similarly to the approach of Mikolov et al. [2013a] for word representations, we found that it is more efficient to use a simpler model with a large amount of data, rather than a complex model and a small corpus.

Some previous works decomposed the target predictions. Allamanis et al. [2015, 2016] decomposed method names into smaller "sub-tokens" and used the continuous prediction approach to compose a full name. Iyer et al. [2016] decomposed StackOverflow titles to single words and predicted them word-by-word. In theory, this approach could be used to predict new compositions of names that were not observed in the training corpus, referred to as neologisms [Allamanis et al. 2015]. However, when scaling to millions of examples this approach might become cumbersome and fail to train well due to hardware and time limitations. As shown in section 6.1, our model yields significantly better results than previous models that used this approach.

Another disadvantage of subtoken-by-subtoken learning is that it requires a time consuming beam-search during inference. This results in an *orders of magnitude slower inference rate* (the number of predictions that the model is able to make per second). An empirical comparison of the inference rate of our model and the models of Allamanis et al. [2016]; Iyer et al. [2016], shows that our model achieves roughly 200 times higher inference rate than Iyer et al. [2016] and 10,000 times higher than Allamanis et al. [2016] (section 6.1).

OoV prediction The main potential advantage of the models of Allamanis et al. [2016] and Iyer et al. [2016] over our model is the subtoken-by-subtoken prediction, which allows them to predict a neologism, and the copy mechanism used by Allamanis et al. [2016] which allows it to use out-of-vocabulary (OoV) words in the prediction.

An analysis of our test data shows that the top-10 most frequent method names, such as `toString`, `hashCode` and `equals`, which are typically easy to predict, appear in less than 6% of the test examples. Other 13% of the names are rare names, which did not appear as whole in the training data, and are difficult or impossible to predict exactly even with a neologism or copy mechanism, such as: `imageFormatExceptionShouldProduceNotSuccessOperationResultWithMessage`. Therefore, our goal is to maximize our efforts on the remaining of the examples.

Even though the upper bound of accuracy of models which incorporate neologism or copy mechanisms is hypothetically higher than ours, the actual contribution of these abilities is minor. Empirically, when trained and evaluated on the same corpora as our model, only less than 3% of the predictions of each of these baselines were actually neologism or OoV. Further, out of all the cases that the baseline suggested a neologism or OoV, *more predictions could have been exact-matches using an already-seen target name, rather than composing a neologism or OoV*.

Although it is possible to incorporate these mechanisms in our model as well, we chose to predict complete names due to the high cost of training and inference time and the relatively negligible contribution of these mechanisms.

Granularity of path decomposition An alternative approach could decompose the representation of a path to granularity of single nodes, and learn to represent a whole path node-by-node using a recurrent neural network (RNN). This would possibly require less space, but will require more time to train and infer, as training of RNNs is usually more time consuming and not clearly better.

Further, a statistical analysis of our corpus shows that more than 95% *of the paths in the test set were already seen in the training set*. Accordingly, in the trade-off between time and space we chose a little less expressive, more memory-consuming, but fast-to-train approach. This choice leads to results that are as 95% as good as our final results in only 6 hours of training, while significantly

improving over previous works (even when limiting our runtime to 3 hours). Despite our choice of time over space, training our model on millions of examples fits in the memory of common GPUs.

5 Distributed vs. Symbolic Representations

Distributed representations refer to representations of elements that are discrete in their nature (e.g. words and names) as vectors or matrices, such that the meaning of an element is distributed across multiple components. This contrasts with symbolic representations, where each element is uniquely represented with exactly one component [Allamanis et al. 2017]. Distributed representations have recently become extremely common in machine learning and NLP because they generalize better, while often requiring fewer parameters.

We compare our model, which uses distributed representations, with Conditional Random Fields (CRFs) as an example of a model that uses symbolic representations. In general, CRFs can also use distributed representations [Artieres et al. 2010; Durrett and Klein 2015], but for the purpose of this discussion, "CRFs" refers to a CRFs with symbolic representations as used by Raychev et al. [2015].

Generalization ability Using CRFs for predicting program properties was found to be powerful [Raychev et al. 2015], but limited to modeling only combinations of values that were seen in the training data. In their work, in order to assign a score for the likelihood of a combination of values, the trained model keeps a scalar parameter for every combination of three components that was observed in the training corpus: variable name, another identifier, and the relation between them. When an unseen combination is observed in test data, a model of this kind cannot generalize and evaluate its likelihood, even if each of the individual values was observed during training.

In contrast, the main advantage of distributed representations in this aspect is the ability to compute the likelihood of *every* combination of observed values. Instead of keeping a parameter for every observed *combination* of values, our model keeps a small constant number (d) of learned parameters for each atomic value, and use algebraic operations to compute the likelihood of their combination.

Trading polynomial complexity with linear Using symbolic representations can be very costly in terms of the number of required parameters. Using CRFs in our problem, which models the probability of a label given a bag of path-contexts, would require using *ternary* factors, which require keeping a parameter for every observed combination of *four* components: identifer, path, another identifier, and the code label (a ternary factor which is determined by the path, with its three parameters). A CRF would thus have a space complexity of $O\left(X^2 \cdot P \cdot Y\right)$, where X is the set of terminal values, P is the set of paths, and Y is the set of labels.

In contrast, the number of parameters in our model is $O\left(d \cdot (X + P + Y)\right)$, where d is a relatively small constants (128 in our final model). Therefore, distributed representations allow to *trade the polynomial complexity with linear*. In fact, using distributed representations of symbols and relations in neural networks allows to keep *less* parameters than CRFs, and at the same time assign a score to *every* possible combination of observed values and paths, instead of only observed combinations.

Practically, we experimented with modeling the task of predicting method names with CRFs using trenary factors. In addition to yielding an F1 score of 49.9, which our model relatively improves by 17%, the CRF model required 104% more parameters, and about 10 times more memory.

6 Evaluation

The main contribution of our method is in its ability to aggregate an arbitrary sized snippet of code into a fixed-size vector in a way that captures its semantics. Since Java methods are usually short, focused, have a single responsibility and a descriptive name, a natural benchmark of our approach would consider a method body as a code snippet, and use the produced code vector to predict the

	Number of methods	Number of files	Size (GB)
Training	14,162,842	1,712,819	66
Validation	415,046	50,000	2.3
Test	413,915	50,000	2.3
Sampled Test	7,454	1,000	0.04

Table 2. Size of data used in the experimental evaluation.

method name. Succeeding in this task would suggest that the code vector has indeed accurately captured the functionality and semantic role of the method.

Our evaluation aims to answer the following questions:

- How useful is our model in predicting method names, and how well does it measure in comparison to other recent approaches (section 6.1)?
- What is the contribution of the attention mechanism to the model? How well would it perform using *hard*-attention instead, or without attention at all (section 6.2)?
- What is the contribution of each of the path-context components to the model (section 6.3)?
- Is it actually able to predict names of complex methods, or only of trivial ones (section 6.4)?
- What are the properties of the learned vectors? Which semantic patterns do they encode (section 6.4)?

Training process In our experiments we use the Adam optimization algorithm [Kingma and Ba 2014], an adaptive gradient descent method commonly used in deep learning. We use dropout [Srivastava et al. 2014] of 0.25 on the context vectors. The values of all the parameters are initialized using the initialization heuristic of Glorot and Bengio [2010]. When training on a single Tesla K80 GPU, we achieve a training throughput of more than 1000 methods per second. Therefore, a single training epoch takes about 3 hours, and it takes about 1.5 days to completely train a model. Training on newer GPUs doubles and quadruples the speed.

Data sets We are interested in evaluating the ability of the approach to generalize across projects. We collected a set of 10,072 Java GitHub repositories. We shuffle the files from all the projects and split them randomly to 14,162,842 training, 415,046 validation and 413,915 of test methods. The model is trained on the training set, and hyperparameters are tuned on the validation set for maximizing F1 score. The number of training epochs is tuned on the validation set using early stopping. Finally, we report results on the unseen test set. A summary of the amount of data used is shown in table 2.

Reducing duplications Following recent work which found a large amount of code duplication in GitHub [Lopes et al. 2017], we took the top-ranked and most popular projects, in which we observed duplication to be less of a problem (Lopes et al. [2017] measured duplication across all the code in GitHub). We put a lot of effort into filtering out duplicates, migrated projects and forks of the same project from our data sets. These occasions were a few out of thousands. It is possible that some duplications are left between the training and test set. However, in these cases, the compared baselines could have benefited from them as well.

Evaluation metric We adopted the measure used by Allamanis et al. [2016], which measured precision, recall, and F1 score over sub-tokens, case-insensitive. This is based on the idea that the quality of a method name prediction is mostly dependant on the sub-words that were used to compose it, rather than on their order. For example, for a method called `countLines`, a prediction of `linesCount` is considered as an exact match, a prediction of `count` has a full precision but low recall, and a prediction of `countBlankLines` has a full recall but low precision. An unknown sub-token ("UNK") is counted as a false negative, therefore automatically hurting recall.

Model	Sampled Test Set (7454 methods) Precision	Recall	F1	Full Test Set (413915 methods) Precision	Recall	F1	inference rate (examples / sec)
CNN+Attention [Allamanis et al. 2016]	47.3	29.4	33.9	-	-	-	0.1
LSTM+Attention [Iyer et al. 2016]	27.5	21.5	24.1	33.7	22.0	26.6	5
PathAttention (this work)	**63.3**	**56.2**	**59.5**	**63.1**	**54.4**	**58.4**	**1000**

Table 3. Evaluation comparison between our model and previous works.

6.1 Quantitative Evaluation

We compare our model to two other recently-proposed models that address similar tasks:

CNN+attention — proposed by Allamanis et al. [2016] for prediction of method names using CNNs and attention. This baseline was evaluated on a random sample of the test set due to its slow inference rate (table 3). We note that the results reported here are lower than the original results reported in their paper, because we consider the task of learning *a single model that is able to predict names for a method from any possible project*. We do not make the restrictive assumption of having a per-project model, able to predict only names within that project. The results we report for CNN+attention are when evaluating their technique in this realistic setting. In contrast, the numbers reported in their original work are for the simplified setting of predicting names *within the scope of a single project*.

LSTM+attention — proposed by Iyer et al. [2016], originally for translation between StackOverflow questions in English and snippets of code that were posted as answers and vice-versa, using an encoder-decoder architecture based on LSTMs and attention. Originally, they demonstrated their approach for C# and SQL. We used a Java lexer instead of the original C#, and pedantically modified it to be equivalent. We retrained their model with the target language being the methods' names, split into sub-tokens. Note that this model was designed for a slightly different task than ours, of translation between source code snippets and natural language descriptions, and not specifically for prediction of method names.

Each baseline was trained on the same training data as our model. We used their default hyper-parameters, except for the embedding and LSTM size of the LSTM+attention model, which were reduced from 400 to 100, to allow it to scale to our enormous training set while complying with the GPU's memory constraints. The alternative was to reduce the amount of training data, which achieved worse results.

Performance Table 3 shows the precision, recall, and F1 score of each model. The model of Allamanis et al. [2016] seems to perform better than that of Iyer et al. [2016], while our model achieves significantly better precision and recall than both of them.

Short and long methods The reported results are based on evaluation on *all* the test data. Additionally evaluating the performance of our model with respect to the length of a test method, we observe similar results across method lengths, with natural descent as length increases. For example, the F1 score of one-line methods is around 65; for two-to-ten lines 59; and for eleven-lines and further 52, while the average method length is 7 lines. This shows the robustness of our model to the length of the methods. Short methods have shorter names and their logic is usually simpler, while long methods benefit from more context for prediction, but their names are usually longer and more diverse.

Speed Fig. 5 shows the test F1 score over training time for each of the evaluated models. In just 3 hours, our model achieves results that are as 88% as good as its final results, and in 6 hours results that are as 95% as good, while both being substantially higher than the best results of the baseline models. Our model achieves its best results after 30 hours.

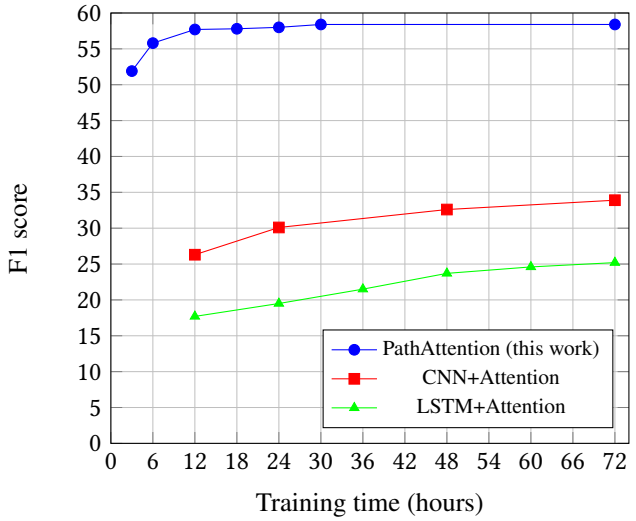

Fig. 5. Our model achieves significantly higher results than the baselines and in shorter time.

Table 3 shows the approximate inference rate of the different models. The syntactic preprocessing time of our model is negligible but is included in the calculation.

As shown, due to their complexity and expensive beam search on prediction, the other models are several orders of magnitude slower than ours, limiting their applicability.

Data efficiency The results reported in table 3 were obtained using our full and large training corpus, to demonstrate the ability of our approach to leverage enormous amounts of training data in a relatively short training time. However, in order to investigate the data efficiency of our model, we also performed experiments using smaller training corpora which are not reported in details here. With 20% of the amounts of data, the F1 score of our model drops in only 50%. With 5% of the data, the F1 score drops only to 30% of our top results. We do not focus on this series of experiments here, since our model can process more than a thousand of examples per second, so there is no significant practical point in deliberately limiting the size of the training corpus.

6.2 Evaluation of Alternative Designs

We experiment with alternative model designs, in order to understand the contribution of each network component.

Attention As we refer to our approach as *soft-attention*, we examine two other approaches which are the extreme alternatives to our approach:

(1) *No-attention* — in which every path-context is given an *equal* weight: the model uses the ordinary average of the path-contexts rather than learning a weighted average.
(2) *Hard-attention* — in which instead of placing the attention "softly" over the path-contexts, all the attention is given to a single path-context, i.e., the network learns to select a *single* most important path-context at a time.

A new model was trained for each of these alternative designs. However, training hard-attention neural networks is difficult, because the gradient of the *argmax* function is zero almost everywhere. Therefore, we experimented with an additional approach: *train-soft, infer-hard*, in which training is performed using soft-attention (as in our ordinary model), and inference is performed using

Model Design	Precision	Recall	F1
No-attention	54.4	45.3	49.4
Hard-attention	42.1	35.4	38.5
Train-soft, infer-hard	52.7	45.9	49.1
Soft-attention	**63.1**	**54.4**	**58.4**
Element-wise soft-attention	**63.7**	**55.4**	**59.3**

Table 4. Comparison of model designs.

Path-context input		Precision	Recall	F1
Full:	$\langle x_s, p, x_f \rangle$	63.1	54.4	58.4
Only-values:	$\langle x_s, \_\_, x_f \rangle$	44.9	37.1	40.6
Value-path:	$\langle x_s, p, \_\_ \rangle$	31.5	30.1	30.7
No-values:	$\langle \_\_, p, \_\_ \rangle$	12.0	12.6	12.3
One-value:	$\langle x_s, \_\_, \_\_ \rangle$	10.6	10.4	10.7

Table 5. Our model while hiding input components.

hard-attention. Table 4 shows the results of all the compared alternative designs. As seen, hard-attention achieves the lowest results. This concludes that when predicting method names, or in general describing code snippets, it is more beneficial to use all the contexts with equal weights than focusing on the single most important one. *Train-soft, infer-hard* improves over hard training, and gains similar results to no-attention. As soft-attention achieves higher scores than all of the alternatives, both on training and inference, this experiment shows its contribution as a "sweet-spot" between no-attention and hard-attention.

Removing the fully-connected layer To understand the contribution of each component of our model, we experiment with removing the fully connected layer (described in section 4.2). In this experiment, soft-attention is applied directly on the *context-vectors* instead of the *combined context-vectors*. This experiment resulted in the same final F1 score as our regular model. Even though its training rate (training examples per second) was faster, it took more actual training time to achieve the same results. For example, instead of reaching results that are as 95% as good as the final results in 6 hours, it took 12 hours, and a few more hours to achieve the final results than the regular model.

Element-wise soft-attention We also experimented with *element-wise soft-attention*. In this design, instead of using a single attention vector $a \in \mathbb{R}^d$ to compute the attention for the whole combined context vector $\tilde{c}_i$, there are d attention vectors $a_1, ..., a_d \in \mathbb{R}^d$, and each of them is used to compute the attention for a different *element*. Therefore, the attention weight for element j of a combined context vector $\tilde{c}_i$ is: attention weight $\alpha_{i_j} = \frac{\exp(\tilde{c}_i^T \cdot a_j)}{\sum_{k=1}^{n} \exp(\tilde{c}_k^T \cdot a_j)}$. This model achieved F1 score of 59.3 (on the full test set) which is even higher than our regular soft-attention model, but since this model gives a different attention to different elements within the same context vector it is more difficult to interpret. Thus, this is an alternative model that gives slightly better results in the cost of losing its interpretability and slower training.

6.3 Data Ablation Study

The contribution of each path-context element To understand the contribution of each component of a path-context, we evaluate our best model on the same test set in the same settings, except that one or more input locations is "hidden" and replaced with a constant "UNK" symbol, such that the model cannot use this element for inference. As the "full" representation is referred to as: $\langle x_s, p, x_f \rangle$, the following experiments were performed:

- "only-values" - using only the values of the terminals for prediction, and therefore representing each path-context as: $\langle x_s, \_\_, x_f \rangle$.
- "value-path" - allowing the model to use a path and one of its values: $\langle x_s, p, \_\_ \rangle$.
- "no-values" - using only the path: $\langle \_\_, p, \_\_ \rangle$.
- "one-value" - using only one of the values: $\langle x_s, \_\_, \_\_ \rangle$.

The results of these experiments are presented in table 5. Interestingly, the "full" representation ($\langle x_s, p, x_f \rangle$) achieves better results than the sum of "no-paths" and "no-values", without each of them

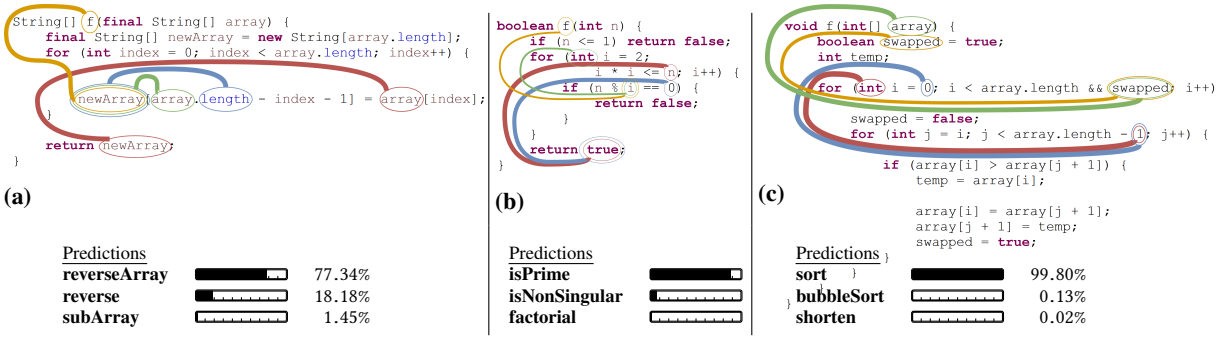

Fig. 6. Example predictions from our model. The width of each path is proportional to its attention.

alone "covering" for the other. "no-paths" gets better results than "no-values", and "single-identifiers" gets the worst results.

The lower results of "only-values" (compared to the full representation) show the importance of using syntactic paths. The low results of "no-words" suggest that predicting names for methods with obfuscated names is a much more difficult task. In this scenario, it might be more beneficial to predict variable names as a first step using a model that was trained specifically for this task, and then predict a method name given the predicted variable names.

6.4 Qualitative Evaluation

6.4.1 Interpreting Attention Despite the "black-box" reputation of neural networks, our model is partially interpretable thanks to the attention mechanism, which allows us to visualize the distribution of weights over the bag of path-contexts. Figure 6 illustrates a few predictions, along with the path-contexts that were given the most attention in each method. The width of each of the visualized paths is proportional to the attention weight that it was allocated. We note that in these figures the path is represented only as a connecting line between tokens, while in fact it contains rich syntactic information which is not expressed properly in the figures. Appendix A portrays the paths on the AST and presents a more informative visualization.

The examples of fig. 6 are particularly interesting since the top names are accurate and descriptive (`reverseArray` and `reverse`; `isPrime`; `sort` and `bubbleSort`) but do not appear explicitly in the method bodies. The method bodies, and specifically the most attended path-contexts describe lower-level operations. Suggesting a descriptive name for each of these methods is difficult and might take time even for a trained human programmer.

6.4.2 Semantic Properties of the Learned Embeddings Surprisingly, the learned method name vectors encode many semantic similarities and even analogies that can be represented as linear additions and subtractions. When simply looking for the closest vector (in terms of cosine distance) to a given method name vector, the resulting neighbors usually contain semantically similar names; e.g. `size` is most similar to `getSize`, `length`, `getCount`, and `getLength`. Table 1 shows additional examples of name similarities.

When looking for a vector that is close to *two* other vectors, we often find names that are semantic combinations of the two other names. Specifically, we can look for the vector v that maximizes the similarity to two vectors a and b:

$$argmax_{v \in V} \left(sim\left(a, v\right) \circledast sim\left(b, v\right) \right) \tag{3}$$

A	+B	≈C
get	value	getValue
get	instance	getInstance
getRequest	addBody	postRequest
setHeaders	setRequestBody	createHttpPost
remove	add	update
decode	fromBytes	deserialize
encode	toBytes	serialize
equals	toLower	equalsIgnoreCase

Table 6. Semantic combinations of method names.

A :	B	C :	D
open :	connect	close :	disconnect
key :	keys	value :	values
lower :	toLowerCase	upper :	toUpperCase
down :	onMouseDown	up :	onMouseUp
warning :	getWarningCount	error :	getErrorCount
value :	containsValue	key :	containsKey
start :	activate	end :	deactivate
receive :	download	send :	upload

Table 7. Semantic analogies between method names.

where $\circledast$ is an arithmetic operator used to combine two similarities, and V is a vocabulary of learned name vectors, *tags_vocab* in our case. When measuring similarity using cosine distance, eq. (3) can be written as:

$$argmax_{v \in V} \left(cos\left(a, v\right) \circledast cos\left(b, v\right) \right) \tag{4}$$

Neither $vec(\texttt{equals})$ nor $vec(\texttt{toLower})$ are the closest vectors to $vec(\texttt{equalsIgnoreCase})$ individually. However, assigning $a = vec\left(\texttt{equals}\right)$, $b = vec\left(\texttt{toLower}\right)$ and using "+" as the operator $\circledast$, results with the vector of `equalsIgnoreCase` as the vector that maximizes eq. (4) for v.

Previous work in NLP has suggested a variety of methods for combining similarities [Levy and Goldberg 2014a] for the task of natural language analogy recovery. Specifically, when using "+" as the operator $\circledast$, as done by Mikolov et al. [2013b], and denoting $\hat{u}$ as the unit vector of a vector u, eq. (4) can be simplified to:

$$argmax_{v \in V} \left(\hat{a} + \hat{b} \right) \cdot \hat{v}$$

Since cosine distance between two vectors equals to the dot product of their unit vectors. Particularly, this can be used as a simpler way to find the above combination of method name similarities:

$$vec\left(\texttt{equals}\right) + vec\left(\texttt{toLower}\right) \approx vec\left(\texttt{equalsIgnoreCase}\right)$$

table 6 shows some of these examples.

Similarly to the way that syntactic and semantic word analogies were found using vector calculation in NLP by Mikolov et al. [2013a,c], the method name vectors that were learned by our model also express similar syntactic and semantic analogies. For example, vec(`keys`) − vec(`key`) + vec(`value`) results in a vector whose closest neighbor is the vector for `values`. This analogy can be read as: "`key` is to `keys` as `value` is to `values`". More examples are shown in table 7.

6.5 Additional Examples

Figure 7 and Figure 8 show additional of our model's predictions, along with the path-contexts that were given the most attention in each example. The path-contexts are portrayed both on the code and on the AST.

7 Related Work

Bimodal modelling of code and natural language Several works have investigated the properties of source code as bimodal: it is at the same time executable for machines, and readable for humans [Allamanis et al. 2015, 2016, [n. d.]; Iyer et al. 2016; Maddison and Tarlow 2014; Zilberstein and Yahav 2016]. This property drives the hope to model natural language conditioned on code and vice-versa. Iyer et al. [2016] designed a token-based neural model using LSTMs and attention for translation between source code snippets and natural language descriptions. As we show in section 6,

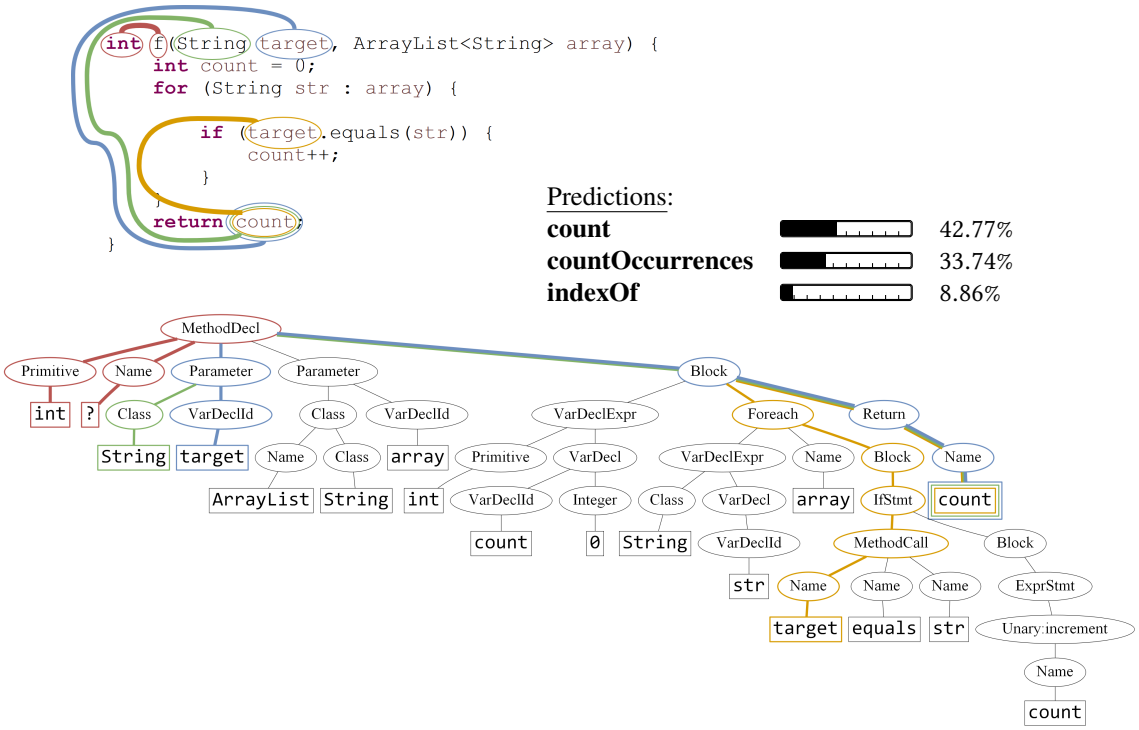

Fig. 7. An example for a method name prediction, portrayed on the AST. The top-four path-contexts were given a similar attention, which is higher than the rest of the path-contexts.

when trained for predicting method names instead of description, our model outperformed their model by a large gap. Allamanis et al. [[n. d.]]; Maddison and Tarlow [2014] also addressed the problem of translating between code and natural language, by considering the syntax of the code rather than representing it as a tokens stream.

Allamanis et al. [2016] have suggested a CNN for summarization of code. The main difference from our work is that they used attention over a "sliding window" of tokens, while our model leverages the syntactic structure of code and propose a simpler architecture which scales to large corpora more easily. Their approach was mainly useful for learning and predicting code from the same project, and had worse results when trained a model using several projects. As we show in section 6, when trained on a multi-project corpus, our model achieved significantly better results.

Representation of code in machine learning models A previous work suggested a general representation of program elements using syntactic relations [Alon et al. 2018]. They showed a simple representation that is useful across different tasks and programming languages: Java, JavaScript, Python and C#, and therefore can be used as a default representation for any machine learning models for code. Our representation is similar to theirs, but can represent *whole snippets of code*. Further, the main novelty of our work is the understanding that *soft-attention over multiple contexts is needed for embedding programs into a continuous space*, and *the use of this embedding to predict properties of a whole code snippet*.

Traditional machine learning algorithms such as decision trees [Raychev et al. 2016a], Conditional Random Fields [Raychev et al. 2015], Probabilistic Context-Free Grammars [Allamanis and Sutton 2014; Bielik et al. 2016; Gvero and Kuncak 2015; Maddison and Tarlow 2014], n-grams [Allamanis

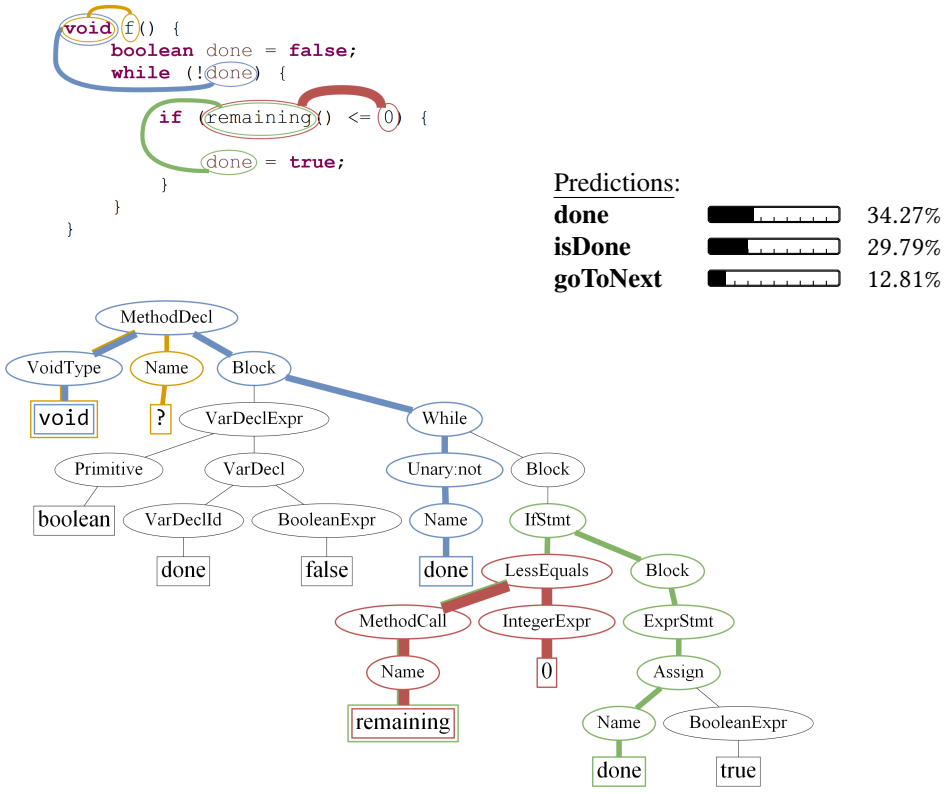

Fig. 8. An example for a method name prediction, portrayed on the AST. The width of each path is proportional to the attention it was given.

et al. 2014; Allamanis and Sutton 2013; Hindle et al. 2012; Nguyen et al. 2013; Raychev et al. 2014] have been used for programming languages in the past.

Attention in machine learning Attention models have shown great success in many NLP tasks such as neural machine translation [Bahdanau et al. 2014; Luong et al. 2015; Vaswani et al. 2017], reading comprehension [Hermann et al. 2015; Levy et al. 2017; Seo et al. 2016], and also in vision [Ba et al. 2014; Mnih et al. 2014], image captioning [Xu et al. 2015], and speech recognition [Bahdanau et al. 2016; Chorowski et al. 2015]. The general idea is to simultaneously learn to concentrate on a small portion of the input data and to use this data for prediction. Xu et al. [2015] proposed the terms "soft" and "hard" attention for the task of image captioning.

Recent works [Bielik et al. 2016; Raychev et al. 2016a,b] used different models to learn a context node for prediction of program properties. This approach is mostly similar to *hard-attention*, as we experiment with in section 6.2, since it uses only the single most important context for every prediction. In this work, we suggest that *soft-attention* is more efficient for modeling code.

Distributed representations The idea of distributed representations of words date back to Deerwester et al. [1990] and even Salton et al. [1975], and are commonly based on the distributional hypothesis of Harris [1954] and Firth [1957], which states that words in similar contexts have similar meaning. These traditional methods included frequency-based methods, and specifically pointwise mutual information (PMI) matrices.

Recently, distributed representations of words, sentences, and documents [Le and Mikolov 2014] were shown to help learning algorithms to achieve a better performance in a variety of tasks [Bengio et al. 2003; Collobert and Weston 2008; Glorot et al. 2011; Socher et al. 2011; Turian et al. 2010; Turney 2006]. Mikolov et al. [2013a,b] has introduced word2vec, a toolkit enabling the training of embeddings. An analysis by Levy and Goldberg [2014b] showed that word2vec's skip-gram model with negative sampling is implicitly factorizing a word-context PMI matrix, linking the modern neural approaches with traditional statistical approaches.

In this work, we use distributed representations of code elements, paths, and method names that are trained as part of our network. Distributed representations make our model generalize better, require *less* parameters than methods based on symbolic representations, and produce vectors with the property that vectors of semantically similar method names are similar in the embedded space.

8 Conclusion

We presented a new attention-based neural network for representing arbitrary-sized snippets of code using a learned fixed-length continuous vector. The core idea is to use soft-attention mechanism over syntactic paths that are derived from the Abstract Syntax Tree of the snippet, and aggregate all of their vector representations into a single vector.

As an example of our approach, we demonstrated it by predicting method names using a model that was trained on more than $14,000,000$ methods. In contrast with previous techniques, our model generalizes well and is able to predict names in files across different projects. We conjecture that the ability to generalize stems from the relative simplicity and the distributed nature of our model. Thanks to the attention mechanism, the prediction results are interpretable and provide interesting observations.

We believe that the attention-based model which uses a structural representation of code can serve as a basis for a wide range of programming language processing tasks. To serve this purpose, we will make all of our trained models publicly available.

References

Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. Learning Natural Coding Conventions. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 281–293. https://doi.org/10.1145/2635868.2635883

Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting Accurate Method and Class Names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 38–49. https://doi.org/10.1145/2786805.2786849

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2017. A Survey of Machine Learning for Big Code and Naturalness. *arXiv preprint arXiv:1709.06182* (2017).

Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. 2016. A Convolutional Attention Network for Extreme Summarization of Source Code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. 2091–2100. http://jmlr.org/proceedings/papers/v48/allamanis16.html

Miltiadis Allamanis and Charles Sutton. 2013. Mining Source Code Repositories at Massive Scale Using Language Modeling. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, Piscataway, NJ, USA, 207–216. http://dl.acm.org/citation.cfm?id=2487085.2487127

Miltiadis Allamanis and Charles Sutton. 2014. Mining Idioms from Source Code. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 472–483. https://doi.org/10.1145/2635868.2635901

Miltiadis Allamanis, Daniel Tarlow, Andrew D. Gordon, and Yi Wei. [n. d.].

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A General Path-Based Representation for Predicting Program Properties. *CoRR* abs/1803.09544 (2018), *to appear in PLDI 2018*. arXiv:1803.09544 http://arxiv.org/abs/1803.09544

Thierry Artieres et al. 2010. Neural conditional random fields. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. 177–184.

Jimmy Ba, Volodymyr Mnih, and Koray Kavukcuoglu. 2014. Multiple object recognition with visual attention. *arXiv preprint arXiv:1412.7755* (2014).

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural Machine Translation by Jointly Learning to Align and Translate. *CoRR* abs/1409.0473 (2014). http://arxiv.org/abs/1409.0473

Dzmitry Bahdanau, Jan Chorowski, Dmitriy Serdyuk, Philemon Brakel, and Yoshua Bengio. 2016. End-to-end attention-based large vocabulary speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*. IEEE, 4945–4949.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. 2003. A Neural Probabilistic Language Model. *J. Mach. Learn. Res.* 3 (March 2003), 1137–1155. http://dl.acm.org/citation.cfm?id=944919.944966

Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2016. PHOG: Probabilistic Model for Code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. 2933–2942. http://jmlr.org/proceedings/papers/v48/bielik16.html

Jan K Chorowski, Dzmitry Bahdanau, Dmitriy Serdyuk, Kyunghyun Cho, and Yoshua Bengio. 2015. Attention-based models for speech recognition. In *Advances in Neural Information Processing Systems*. 577–585.

Ronan Collobert and Jason Weston. 2008. A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning. In *Proceedings of the 25th International Conference on Machine Learning (ICML '08)*. ACM, New York, NY, USA, 160–167. https://doi.org/10.1145/1390156.1390177

Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. 1990. Indexing by latent semantic analysis. *Journal of the American society for information science* 41, 6 (1990), 391.

Greg Durrett and Dan Klein. 2015. Neural CRF Parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, Vol. 1. 302–312.

J.R. Firth. 1957. *A Synopsis of Linguistic Theory, 1930-1955*. https://books.google.co.il/books?id=T8LDtgAACAAJ

Martin Fowler and Kent Beck. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. 249–256.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Domain adaptation for large-scale sentiment classification: A deep learning approach. In *Proceedings of the 28th international conference on machine learning (ICML-11)*. 513–520.

Tihomir Gvero and Viktor Kuncak. 2015. Synthesizing Java Expressions from Free-form Queries. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 416–432. https://doi.org/10.1145/2814270.2814295

Zellig S Harris. 1954. Distributional structure. *Word* 10, 2-3 (1954), 146–162.

Karl Moritz Hermann, Tomáš Kočiský, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. 2015. Teaching Machines to Read and Comprehend. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'15)*. MIT Press, Cambridge, MA, USA, 1693–1701. http://dl.acm.org/citation.cfm?id=2969239.2969428

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 837–847. http://dl.acm.org/citation.cfm?id=2337223.2337322

Einar W. Høst and Bjarte M. Østvold. 2009. Debugging Method Names. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming (Genoa)*. Springer-Verlag, Berlin, Heidelberg, 294–317. https://doi.org/10.1007/978-3-642-03013-0_14

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*. http://aclweb.org/anthology/P/P16/P16-1195.pdf

Diederik Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

Quoc Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, Tony Jebara and Eric P. Xing (Eds.). JMLR Workshop and Conference Proceedings, 1188–1196. http://jmlr.org/proceedings/papers/v32/le14.pdf

Omer Levy and Yoav Goldberg. 2014a. Linguistic regularities in sparse and explicit word representations. In *Proceedings of the eighteenth conference on computational natural language learning*. 171–180.

Omer Levy and Yoav Goldberg. 2014b. Neural Word Embeddings as Implicit Matrix Factorization. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*. 2177–2185.

Omer Levy, Minjoon Seo, Eunsol Choi, and Luke Zettlemoyer. 2017. Zero-Shot Relation Extraction via Reading Comprehension. In *Proceedings of the 21st Conference on Computational Natural Language Learning (CoNLL 2017), Vancouver, Canada, August 3-4, 2017*. 333–342. https://doi.org/10.18653/v1/K17-1034

Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéJàVu: A Map of Code Duplicates on GitHub. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 84 (Oct. 2017), 28 pages.

https://doi.org/10.1145/3133908

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP 2015, Lisbon, Portugal, September 17-21, 2015*. 1412–1421. http://aclweb.org/anthology/D/D15/D15-1166.pdf

Chris J. Maddison and Daniel Tarlow. 2014. Structured Generative Models of Natural Source Code. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32 (ICML'14)*. JMLR.org, II–649–II–657. http://dl.acm.org/citation.cfm?id=3044805.3044965

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013a. Efficient Estimation of Word Representations in Vector Space. *CoRR* abs/1301.3781 (2013). http://arxiv.org/abs/1301.3781

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013b. Distributed Representations of Words and Phrases and Their Compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems (NIPS'13)*. Curran Associates Inc., USA, 3111–3119. http://dl.acm.org/citation.cfm?id=2999792.2999959

Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. 2013c. Linguistic regularities in continuous space word representations.

Alon Mishne, Sharon Shoham, and Eran Yahav. 2012. Typestate-based Semantic Code Search over Partial Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, New York, NY, USA, 997–1016. https://doi.org/10.1145/2384616.2384689

Volodymyr Mnih, Nicolas Heess, Alex Graves, and Koray Kavukcuoglu. 2014. Recurrent Models of Visual Attention. In *Proceedings of the 27th International Conference on Neural Information Processing Systems (NIPS'14)*. MIT Press, Cambridge, MA, USA, 2204–2212. http://dl.acm.org/citation.cfm?id=2969033.2969073

Dana Movshovitz-Attias and William W Cohen. 2013. Natural language models for predicting programming comments. (2013).

Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2013. A Statistical Semantic Language Model for Source Code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM, New York, NY, USA, 532–542. https://doi.org/10.1145/2491411.2491458

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*. 1532–1543. http://www.aclweb.org/anthology/D14-1162

Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016a. Probabilistic Model for Code with Decision Trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 731–747. https://doi.org/10.1145/2983990.2984041

Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. 2016b. Learning Programs from Noisy Data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 761–774. https://doi.org/10.1145/2837614.2837671

Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting Program Properties from "Big Code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 111–124. https://doi.org/10.1145/2676726.2677009

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 419–428. https://doi.org/10.1145/2594291.2594321

Reuven Rubinstein. 1999. The cross-entropy method for combinatorial and continuous optimization. *Methodology and computing in applied probability* 1, 2 (1999), 127–190.

Reuven Y Rubinstein. 2001. Combinatorial optimization, cross-entropy, ants and rare events. *Stochastic optimization: algorithms and applications* 54 (2001), 303–363.

Gerard Salton, Anita Wong, and Chung-Shu Yang. 1975. A vector space model for automatic indexing. *Commun. ACM* 18, 11 (1975), 613–620.

Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. 2016. Bidirectional attention flow for machine comprehension. *arXiv preprint arXiv:1611.01603* (2016).

Richard Socher, Cliff C. Lin, Andrew Y. Ng, and Christopher D. Manning. 2011. Parsing Natural Scenes and Natural Language with Recursive Neural Networks. In *Proceedings of the 26th International Conference on Machine Learning (ICML)*.

Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research* 15, 1 (2014), 1929–1958.

Grigorios Tsoumakas and Ioannis Katakis. 2006. Multi-label classification: An overview. *International Journal of Data Warehousing and Mining* 3, 3 (2006).

Joseph Turian, Lev Ratinov, and Yoshua Bengio. 2010. Word Representations: A Simple and General Method for Semi-supervised Learning. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL '10)*. Association for Computational Linguistics, Stroudsburg, PA, USA, 384–394. http://dl.acm.org/citation.cfm?id=1858681.1858721

Peter D Turney. 2006. Similarity of semantic relations. *Computational Linguistics* 32, 3 (2006), 379–416.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*. 6000–6010.

Martin T. Vechev and Eran Yahav. 2016. Programming with "Big Code". *Foundations and Trends in Programming Languages* 3, 4 (2016), 231–284. https://doi.org/10.1561/2500000028

Martin White, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. 2015. Toward Deep Learning Software Repositories. In *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR '15)*. IEEE Press, Piscataway, NJ, USA, 334–345. http://dl.acm.org/citation.cfm?id=2820518.2820559

Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. 2015. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning*. 2048–2057.

Meital Zilberstein and Eran Yahav. 2016. Leveraging a Corpus of Natural Language Descriptions for Program Similarity. In *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2016)*. ACM, New York, NY, USA, 197–211. https://doi.org/10.1145/2986012.2986013