code2vec: Learning Distributed Representations of Code

URI ALON, Technion, Israel
MEITAL ZILBERSTEIN, Technion, Israel
OMER LEVY, Facebook AI Research, USA
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 a code snippet as a single fixed-length *code vector*, which can be used to predict semantic properties of the snippet. To this end, code is first decomposed to a collection of paths in its abstract syntax tree. Then, the network learns the atomic representation of each path while *simultaneously* learning how to aggregate a set of them.

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 12M methods. We show that code vectors trained on this dataset can predict method names from files that were unobserved during training. Furthermore, we show that our model learns useful method name vectors that capture semantic similarities, combinations, and analogies.

A comparison of our approach to previous techniques over the same dataset shows an improvement of more than 75%, making it the first to successfully predict method names based on a large, cross-project corpus. Our trained model, visualizations and vector similarities are available as an interactive online demo at http://code2vec.org. The code, data and trained models are available at https://github.com/tech-srl/code2vec.

CCS Concepts: • **Computing methodologies** **Learning latent representations**; • **Software and its engineering** **General programming languages**;

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

ACM Reference Format:
Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.* 3, POPL, Article 40 (January 2019), 29 pages. https://doi.org/10.1145/3290353

1 INTRODUCTION

Distributed representations of words (such as "word2vec") [Mikolov et al. 2013a,b; Pennington et al. 2014], sentences, paragraphs, and documents (such as "doc2vec") [Le and Mikolov 2014] have played a key role in unlocking the potential of neural networks for natural language processing (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*. In these vectors, the "meaning" of an element is distributed across multiple vector components, such that semantically similar objects are mapped to close vectors.

Authors' addresses: Uri Alon, Technion, Israel, urialon@cs.technion.ac.il; Meital Zilberstein, Technion, Israel, mbs@cs.technion.ac.il; Omer Levy, Facebook AI Research, Seattle, USA, omerlevy@gmail.com; Eran Yahav, Technion, Israel, yahave@cs.technion.ac.il.

2475-1421/2019/1-ART40
https://doi.org/10.1145/3290353

```
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.

Goal: The goal of this paper is to learn *code embeddings*, continuous 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-language tasks. In this paper, we use the motivating task of *semantic labeling of code snippets*.

Motivating task: semantic labeling of code snippets. Consider the method in Figure 1. The method contains only low-level assignments to arrays, but a human reading the code may (correctly) label it as performing the *reverse* operation. Our goal is 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.

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.

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 snippets 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 to the application of deep learning in programming languages, in the same way that word embeddings in natural language processing (NLP) are fundamental to the 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 label expresses the semantic property that we wish the network to model, for example, a tag 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 label 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. 2015a; Høst and Østvold 2009].

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

Table 1. Semantic similarities between method names.

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

Like 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 Applications

Embedding a code snippet as a vector has a variety of machine-learning based applications, since machine-learning algorithms usually take vectors as their inputs. In this paper, we examine the following direct applications:

(1) *Automatic code review* - Suggesting better method names when the name given by the developer doesn't match the method's functionality. Better method names prevent naming bugs, improve the readability and maintenance of code, and facilitate the use of public APIs. This application was previously shown to be of significant importance [Allamanis et al. 2015a; Fowler and Beck 1999; Høst and Østvold 2009].

(2) *Retrieval and API discovery* - Semantic similarities enable search in “the problem domain” instead of search “in the solution domain”. For example, a developer might look for a `serialize` method, while the equivalent method of the class is named `toJson` as serialization is performed via `json`. An automatic tool that looks for the *vector* most similar to the requested name among the available methods will find `toJson` (Table 1). Such semantic similarities are difficult to find without our approach. Further, an automatic tool which uses our vectors can easily determine that a programmer is using the method `equals` right after `toLowerCase` and suggest using `equalsIgnoreCase` instead (Table 6).

The code vectors we produce can be used as input to any machine learning pipeline that performs tasks such as code retrieval, captioning, classification and tagging, or as a metric for measuring similarity between snippets of code for ranking and clone detection. The novelty of our approach is in its ability to produce vectors that capture properties of snippets of code, such that similar snippets (according to any desired criteria) are assigned similar vectors. This ability unlocks a variety of applications for working with machine-learning algorithms on code.

We deliberately picked the difficult task of method name prediction, for which prior results were poor [Allamanis et al. 2015a, 2016; Alon et al. 2018], as an evaluation benchmark. Succeeding in this challenging task implies good performance in other tasks such as predicting whether or not a program performs I/O, predicting the required dependencies of a program, and predicting whether a program is a suspected malware. We show that even for this challenging benchmark, our technique dramatically improves the results of previous works.

1.2 Challenges: Representation and Attention

Assigning a semantic label 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, and learning the order of importance of the part.

Representation. NLP methods typically treat text as a linear sequence of tokens. Indeed, many existing approaches also represent source 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.

We note that there is a tradeoff between the degree of program analysis required to extract the representation and the learning effort that follows. Performing no program analysis but learning instead from the program's surface text often incurs a significant learning effort. This learning effort thus requires prohibitive amounts of data because the learning model has to re-learn the syntax and semantics of the programming language from the data. On the other end of the spectrum, performing a deep program analysis to extract the representation may make the learned model language-specific (and even task-specific).

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 find that this representation significantly lowers the learning effort (compared to learning over program text), and is still scalable and general such that it can be applied to a wide range of problems and large amounts of code.

We represent a given code snippet as a bag (multiset) of its extracted paths. The challenges are then *how to aggregate a bag of contexts and which paths to focus on for making a prediction.*

Attention. The problem can be stated informally as the need to learn a correspondence between a bag of path-contexts and a label. Representing each bag of path-contexts *monolithically* will result in 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 utilizing the individual path-contexts and their components (paths, values, etc.) that were observed during training to be parts of other bags.

To address this challenge we use a novel neural 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], speech recognition [Bahdanau et al. 2016; Chorowski et al. 2015] and computer vision [Ba et al. 2014; Mnih et al. 2014; Xu et al. 2015].

Our neural attention mechanism 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, our model is relatively interpretable: 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 focus on at a time. The use of *soft attention* over syntactic paths is the main understanding that leads to the improved results. We compare our model to an equivalent model that uses hard attention in Section 6.2, and show that *soft attention* is more efficient for modeling code.

1.3 Existing Techniques

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

1.4 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 us 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 12M 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), CNNs and CRFs.
- 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 make it possible 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 works such as Alon et al. [2018] and Raychev et al. [2015].

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 observed in its entirely 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 learned 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.

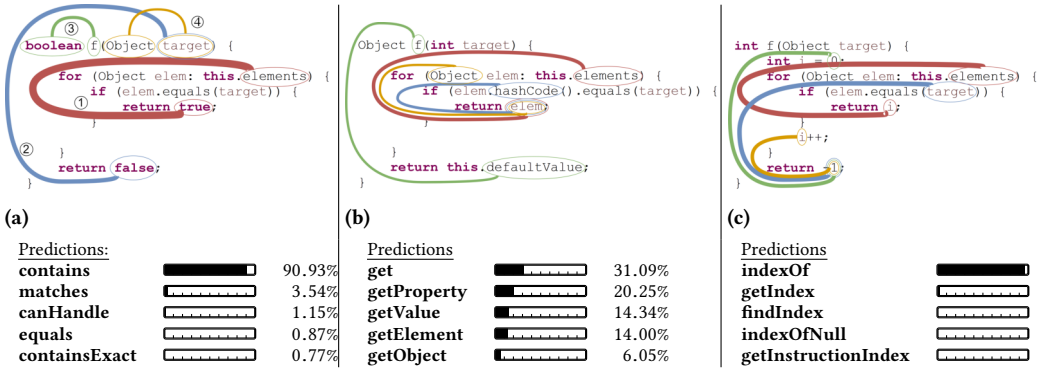

Fig. 2. Examples of three methods that can be easily distinguished by our model despite having similar syntactic structure: our model successfully captures the subtle differences between them and predicts meaningful names. Each method portrays the top-4 paths that were given the most attention by the model. The widths of the colored paths are proportional to the attention that each path was given.

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 in 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` for which `target` *equals 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 respective descriptive method names: `contains`, `get`, and `indexOf`.

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 a tuple we refer to as a *path-context*. These terms are defined formally in Section 3. Figure 3 portrays the top-four path-contexts that were given the most attention by the model, on the AST of the method from Figure 2a, such that the width of each path is proportional to the attention it was given by the model during this prediction.

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 values of 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-context embeddings into a single vector that represents the entire method body. Attention is the mechanism that learns to score each path-context, such that higher attention is reflected in a higher score.

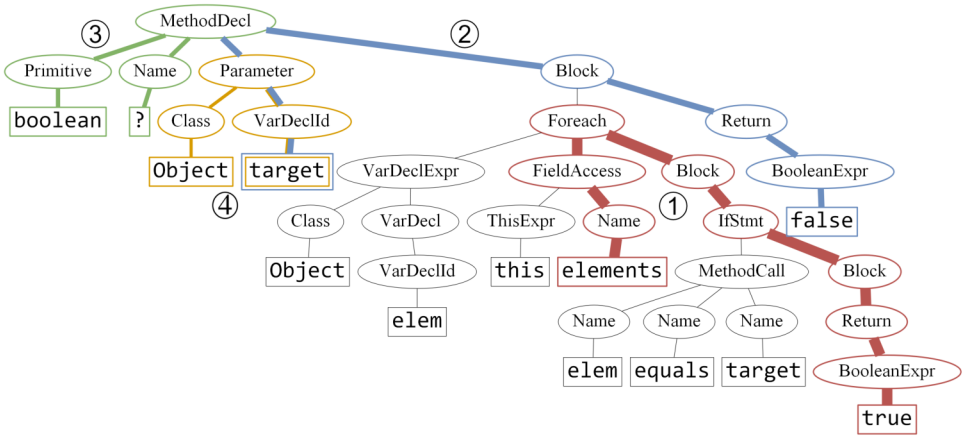

Fig. 3. The top-4 attended paths of Figure 2a, as were learned by the model, shown on the AST of the same snippet. The width of each colored path is proportional to the attention it was given (**red** ①: **0.23**, **blue** ②: **0.14**, **green** ③: **0.09**, **orange** ④: **0.07**).

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. Each code snippet in Figure 2 and Figure 3 highlights the top-four path-contexts that were given the most weight (attention) by the model in each example. The widths of the paths are proportional to the attention score that each of these path-contexts was given. The model has learned how much weight to give every possible path on its own, as part of training on millions of examples. 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 three examples, most of the top suggestions are very similar to each other and all of them are descriptive of 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 is 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. In contrast, a bigram model will represent the expression `Object target` equally whether `target` is a method parameter or a local variable. This shows that a model using a syntactic representation of a code snippet can distinguish between two snippets of code that other representations cannot. By aggregating all the contexts using attention, the model can use subtle differences between snippets to produce a more accurate prediction.

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]. The main difference in this definition is that we define this representation to handle *whole snippets of code*, rather than a single program element (such as a single variable), 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, and 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 $\langle x_s, p, x_t \rangle$ where $x_s = \phi(start(p))$ and $x_t = \phi(end(p))$ 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:

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

To limit the size of the training data and reduce sparsity, it is possible to limit different parameters of the paths. 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 our model.

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 prediction. 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 whose values are learned. The values of this vector capture two distinct notions: (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 them are shown in Table 4 ("hard attention" and "no-attention") to yield poor results.

Our main insight in this work is that *all* context vectors should be used but the model should be allowed 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 a factor of its dot product with another global attention vector. The vector of each context and the global attention vector are trained and learned *simultaneously* using the standard neural approach of backpropagation. Once trained, the neural network is simply a pure mathematical function, which uses algebraic operators to output a code vector given a set of contexts.

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 code snippet into a mathematical object that can be used in a

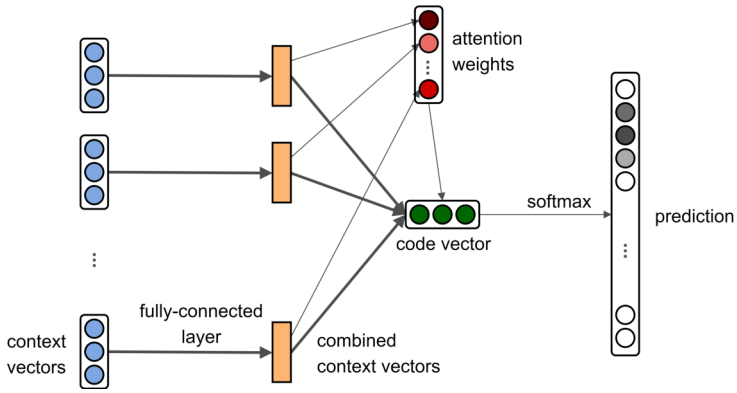

Fig. 4. The architecture of our path-attention network. A *fully connected layer* learns to combine embeddings of each path-context with itself; attention weights are learned using the combined context vectors and used to compute a *code vector*. The code vector is used to predict the label.

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 contain a node and itself):

$$TPairs(C) = \left\{ (term_i, term_j) \mid term_i, term_j \in termNodes(C) \wedge i \neq j \right\}$$

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:

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

That is, C is represented as the set of triplets $\langle x_s, p, x_t \rangle$ such that x_s and x_t 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.

4.2 Path-Attention Model

Overall, the model learns the following components: embeddings for paths and names (matrices $path\_vocab$ and $value\_vocab$), a fully connected layer (matrix W), attention vector ($\boldsymbol{a}$), and embeddings for the tags ($tags\_vocab$). We describe our model 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. An embedding is looked up by 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 width of the matrix W is the embedding size $d \in \mathbb{N}$ – the dimensionality hyperparameter. d is determined empirically, limited by the training time, model complexity, and the GPU memory, and it 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 $b_i = \langle x_s, p_j, x_t \rangle$ be one of these path-contexts, such that $\{x_s, x_t\} \in X$ are values of terminals and $p_j \in P$ is their connecting path. 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*: $\boldsymbol{c_i} \in \mathbb{R}^{3d}$ that represents that path-context:

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

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 $\boldsymbol{c_i}$ 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. This allows the model to give a different attention to every *combination* of paths and values. This combination allows the model the expressivity of giving a certain path more attention when observed with certain values and less attention when the exact same path is observed with other values.

Here, $\tilde{\boldsymbol{c}}_{\boldsymbol{i}}$ 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:

$$\tilde{\boldsymbol{c}}_{\boldsymbol{i}} = tanh\left(W \cdot \boldsymbol{c_i}\right)$$

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 $\tilde{\boldsymbol{c}}_{\boldsymbol{i}}$, and for convenience is the same size (d) as before. In general, the height of W can be different; this will affect the size of the final *code* vector. $tanh$ is the hyperbolic tangent element-wise function, a commonly used monotonic nonlinear activation function which outputs values in the 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 it 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 for each of them. An attention vector $\boldsymbol{a} \in \mathbb{R}^d$ is initialized randomly and learned simultaneously with the network. Given the combined context vectors: $\{\tilde{\boldsymbol{c}}_{\boldsymbol{1}}, ..., \tilde{\boldsymbol{c}}_{\boldsymbol{n}}\}$, the attention weight α_i of each $\tilde{\boldsymbol{c}}_{\boldsymbol{i}}$ is computed as the normalized inner product between the combined context vector and the global attention vector $\boldsymbol{a}$:

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

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 $\boldsymbol{v} \in \mathbb{R}^d$, which represents the whole code snippet, is a linear combination of the combined context vectors $\{\tilde{\boldsymbol{c}}_{\boldsymbol{1}}, ..., \tilde{\boldsymbol{c}}_{\boldsymbol{n}}\}$ factored by their attention weights:

$$\text{code vector } \boldsymbol{v} = \sum_{i=1}^{n} \alpha_i \cdot \tilde{\boldsymbol{c}}_{\boldsymbol{i}} \quad (2)$$

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 $\tilde{\boldsymbol{c}}_{\boldsymbol{i}}$. 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 tag 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. As before, the embedding of tag_i is row i of $tags\_vocab$. For example, looking at Figure 2a again, we see that $tags\_vocab$ 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 $\boldsymbol{v}$ and each of the tag embeddings:

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

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 $\boldsymbol{v}$.

4.3 Training

To train 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 actual tag in the training example and 0 otherwise, the cross-entropy loss for a single example is equivalent to the negative log-likelihood of the true label, and 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. Thus, minimizing this loss is equivalent to maximizing the log-likelihood that the model assigns to the true labels y_{true}.

The network is trained using any gradient descent based algorithm and the standard approach of back-propagating the training error through each of the learned parameters (i.e., deriving the loss with respect to each of the learned parameters and updating the learned parameter's value by a small "step" towards the direction that minimizes the loss).

4.4 Using the Trained Network

A trained network can be used to: (i) perform a downstream task, using the code vector $\boldsymbol{v}$ itself, and (ii) predict tags for new, unseen code.

Using the code vector. An unseen code can be fed into the trained network exactly 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 $\boldsymbol{v}$ 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 $\boldsymbol{v_C}$.

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 in method documentation. In the latter case, the attention vector should be used to re-compute the attention weights after each predicted token, given the previous prediction, as is 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-only context. 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; it can be applied in a similar way to a "bag of contexts" in which the contexts are designed for a specific task, or it can be applied to contexts that were produced using semantic analysis. Specifically, we chose to use a syntactic representation that is similar to that of Alon et al. [2018] because it was shown to be useful as a representation for modeling programming languages in machine learning models. It was also shown to be more expressive than n-grams and manually designed features.

An alternative approach is to include semantic relations as context. Such an approach was taken by Allamanis et al. [2018], who presented a Gated Graph Neural Network in which program elements are graph nodes and semantic relations such as `ComputedFrom` and `LastWrite` are edges in the graph. In their work, these semantic relations were chosen and implemented for specific programming language and tasks. In our work, we wish to explore *how far a syntactic-only approach can go.* Using semantic knowledge has many advantages and might reveal information that is not clearly expressed in a syntactic-only observation. However, using semantic knowledge comes at a cost: (i) an expert is required to choose and design the semantic analyses; (ii) generalizing to new languages is much more difficult, as the semantic analyses need to be implemented differently for every language; and (iii) the designed analyses might not easily generalize to other tasks. In contrast, in our syntactic approach (i) neither expert knowledge of the language nor manual feature designing is required; (ii) generalizing to other languages is accomplished by simply replacing the parser and extracting paths from the new language's AST using the same traversal algorithm; and (iii) the same syntactic paths generalize surprisingly well to other tasks (as was shown by Alon et al. [2018]).

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

Some previous works decomposed the target predictions. Allamanis et al. [2015a, 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. 2015a]. 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 prediction. This results in an *orders-of-magnitude slower prediction rate* (the number of predictions that the model is able to make per second). An empirical comparison of the prediction rate of our model and the models of Allamanis et al. [2016] and Iyer et al. [2016] shows that our model achieves a roughly 200 times faster prediction rate than Iyer et al. [2016] and 10,000 times faster than Allamanis et al. [2016] (Section 6.1).

OoV prediction. The other possible advantage of Allamanis et al. [2016]'s method — the ability to produce out-of-vocabulary (OoV) predictions by means of a copy mechanism and subtoken-by-subtoken decoding — offer only a negligible contribution. 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. The 13% least occurring names are rare names, which did not appear in their entirety in the training data, and are difficult or impossible to predict exactly even with a neologism or copy mechanism. One example is `imageFormatExceptionShouldProduceNotSuccessOperationResultWithMessage`. However, when trained and evaluated on the same corpus as our model, less than 3% of the predictions of each of these baselines were actually neologisms or OoV. Moreover, in most of the cases where the baseline suggested a neologism or OoV, *it could have produced a more accurate prediction using only already seen target names.*

We thus believe that our efforts would be better spent on the prediction of complete names.

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 would also be more time consuming.

Furthermore, 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 slightly 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. 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

Our model uses *distributed representations*, which are representations of elements that are discrete in their nature (e.g., words and names) using vectors and matrices. In distributed representations, the "meaning" of an element is distributed across all of its vector's components. This is in contrast to symbolic representations, where each element is uniquely represented with exactly one component [Allamanis et al. 2017]. Distributed representations have recently become very common in machine learning and NLP because they generalize better while often requiring fewer parameters. Conditional Random Fields (CRFs) are one example of a model that uses symbolic representations. We compare our model to those of Alon et al. [2018] and Raychev et al. [2015], both of which use CRFs. Although CRFs can also use distributed representations [Artieres et al. 2010; Durrett and Klein 2015], for the purpose of this discussion, we refer to CRFs in the sense of Alon et al. [2018] and Raychev et al. [2015].

Generalization ability. Alon et al. [2018] and Raychev et al. [2015] found CRFs to be a powerful method for predicting program properties. However, they are limited to modeling only combinations of values that were seen in the training data. To score 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 that combination's likelihood, even if each of the individual values was observed during training. In contrast, distributed representations can 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 uses algebraic operations to compute the likelihood of their combination.

Trading polynomial complexity for 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: terminal value, path, another terminal value, and the target 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 constant number (128 in our final model). Thus, distributed representations allow us to *trade polynomial complexity for linear*. This is extremely important in these settings, because $|X|$, $|Y|$ and $|P|$ are in the orders of millions (the number of observed values, paths and labels). In fact, using distributed representations of symbols and relations in neural networks allows us to keep *fewer* parameters than the number required for CRFs, and at the same time compute a score for *every* possible combination of observed values, paths and target labels instead of computing it only for observed combinations.

We reproduced the experiments of Alon et al. [2018] for modeling the task of predicting method names with CRFs using ternary factors. In addition to yielding an F1 score of 49.9, which our model improves by 17% (relative), the CRF model required twice the number of parameters, and consumed about 10 times more memory.

6 EVALUATION

The main contribution of our method is 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 functionality 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 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 using no 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)?

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

	Number of methods	Number of files	Size (GB)
Training	12,636,998	1,712,819	30
Validation	371,364	50,000	0.9
Test	368,445	50,000	0.9
Sampled Test	7,454	1,000	0.04

Training process. In our experiments we took the top $1M$ paths — those that occurred the most in the training set. We used the Adam optimization algorithm [Kingma and Ba 2014], an adaptive gradient descent method commonly used in deep learning. We used dropout [Srivastava et al. 2014] of 0.25 on the context vectors. The values of all the parameters were initialized using the initialization heuristic of Glorot and Bengio [2010]. When training on a single Tesla K80 GPU, we achieved 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. Although the attention mechanism can aggregate an arbitrary number of inputs, we randomly sampled up to $k = 200$ path-contexts from each training example. The value $k = 200$ seemed to be enough to "cover" each method, since increasing to $k = 300$ did not seem to improve the results.

Datasets. We wanted to evaluate the ability of the approach to generalize across projects. We used a dataset of 10,072 Java GitHub repositories, originally introduced by Alon et al. [2018]. Following recent work which found a large amount of code duplication in GitHub [Lopes et al. 2017], Alon et al. [2018] used the top-ranked and most popular projects, in which duplication was observed to be less of a problem. Additionally, they filtered out migrated projects and forks of the same project. While it is possible that some duplications are left between the training and test set, in this case the compared baselines could have benefited from them as well. In this dataset, the files from all the projects were shuffled and split to 12,636,998 training, 371,364 validation and 368,445 test methods.

We trained our model on the training set and tuned hyperparameters on the validation set for maximizing F1 score. The number of training epochs was 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.

Evaluation metric. Ideally, we would have liked to manually evaluate the results, but given that manual evaluation is very difficult to scale, we adopted the measure used in previous works [Allamanis et al. 2015a, 2016; Alon et al. 2018], 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 depends mainly on the sub-words used to compose it. For example, for a method called `countLines`, a prediction of `linesCount` is considered as an exact match, a prediction of `count` has full precision but low recall, and a prediction of `countBlankLines` has full recall but low precision. An unknown sub-token in the test label ("UNK") is counted as a false negative, therefore automatically hurting recall.

While there are alternative metrics in the literature, such as accuracy and BLEU score, they are problematic because accuracy counts even mostly correct predictions as completely incorrect, and the BLEU score tends to favor short predictions, which are usually uninformative [Callison-Burch et al. 2006]. We provide a qualitative evaluation including a manual inspection of examples in Section 6.4.

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

Model	Sampled Test Set Precision	Recall	F1	Full Test Set Precision	Recall	F1	prediction 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
Paths+CRFs [Alon et al. 2018]	-	-	-	53.6	46.6	49.9	10
PathAttention (this work)	**63.3**	**56.2**	**59.5**	**63.1**	**54.4**	**58.4**	**1000**

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 prediction rate (Table 3). We note that the F1 score reported here is 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 carefully modified it to be equivalent. We re-trained 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: translation between source code snippets and natural language descriptions, and not specifically for prediction of method names.

Paths+CRFs. — proposed by Alon et al. [2018], using a similar syntactic path representation as this work, with CRFs as the learning algorithm. We evaluate our model on their introduced dataset, and achieve a significant improvement in results, training time and prediction time.

Each baseline was trained on the same training data as our model. We used their default hyperparameters, 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 Alon et al. [2018] seems to perform better than that of Allamanis et al. [2016] and Iyer et al. [2016], while our model achieves significantly better precision and recall than all 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 a natural decrease in performance as the 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 more 52, while the average method length is 7 lines. We used all the methods in the dataset, regardless of their size. This shows the robustness of our model to the length of the

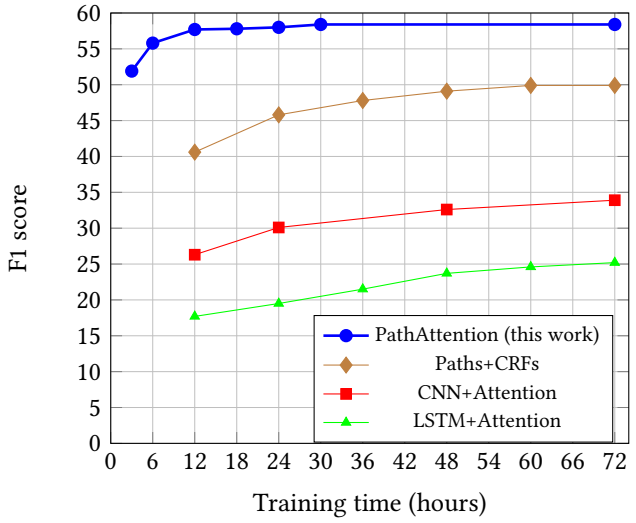

Fig. 5. Our model achieves significantly better results than the baselines and in shorter time.

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, more diverse and sparse, for example: `generateTreeSetHashSetSpoofingSetInteger`, which has 17 lines of code.

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, with both being substantially higher than the best results of the baseline models. Our model achieves its best results after 30 hours.

Table 3 shows the approximate prediction 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 detail here. With 20% of the data, the F1 score of our model drops to 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, there is no real reason to deliberately limit 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 at the opposite extreme:

Table 4. Comparison of model designs.

Model Design	Precision	Recall	F1
No-attention	54.4	45.3	49.4
Hard attention	42.1	35.4	38.5
Train-soft, predict-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**

(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 focusing 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, predict-hard*, in which training is performed using soft attention (as in our ordinary model), and prediction is performed using 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 to focus on the single most important one. *Train-soft, predict-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 prediction, 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, it took 12 hours instead of 6 to reach results that are as 95% as good as the final results, and a few more hours to achieve the final results.

Element-wise soft attention. We also experimented with *element-wise soft attention*. In this design, instead of using a single attention vector $\boldsymbol{a} \in \mathbb{R}^d$ to compute the attention for the whole combined context vector $\tilde{\boldsymbol{c}}_{\boldsymbol{i}}$, there are d attention vectors $\boldsymbol{a_1}, ..., \boldsymbol{a_d} \in \mathbb{R}^d$, 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 $\tilde{\boldsymbol{c}}_{\boldsymbol{i}}$ is: attention weight $\alpha_{i_j} = \frac{\exp(\tilde{\boldsymbol{c}}_{\boldsymbol{i}}{}^T \cdot \boldsymbol{a_j})}{\sum_{k=1}^{n} \exp(\tilde{\boldsymbol{c}}_{\boldsymbol{k}}{}^T \cdot \boldsymbol{a_j})}$. This variation allows the model to compute a different attention score for each *element* in the combined context vector, instead of computing the same attention score for the whole combined context vector. This model achieved an F1 score of 59.3 (on the full test set), which is even higher than our standard soft attention model, but since this model gives 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 at the cost of poor interpretability and slower training.

Table 5. Our model while hiding input components.

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

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 prediction. As the "full" representation is referred to as: $\langle x_s, p, x_t \rangle$, the following experiments were performed:

- "only-values" - using only the values of the terminals for prediction, without paths, and therefore representing each path-context as: $\langle x_s, \_\_, x_t \rangle$.
- "no-values" - using only the path: $\langle \_\_, p, \_\_ \rangle$, without identifiers and keywords.
- "value-path" - allowing the model to use a path and one of its values: $\langle x_s, p, \_\_ \rangle$.
- "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 ($\langle x_s, p, x_t \rangle$) achieves better results than the sum of "only-values" and "no-values", without each of them alone "covering" for the other. This shows the importance of using *both* paths and keywords, and letting the attention mechanism learn how to combine them in every example. The poorer results of "only-values" (compared to the full representation) show the importance of using syntactic paths. As shown in the table, dropping identifiers and keywords hurt the model more than dropping paths, but combining them achieves significantly better results. Better results are obtained for "no-paths" than for "no-values", and "single-identifiers" obtains the worst results.

The poor results of "no-values" 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. Figure 7 and Figure 8 portray the paths on the AST.

The examples of Figure 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 path-contexts that were given the most attention, 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. The average method

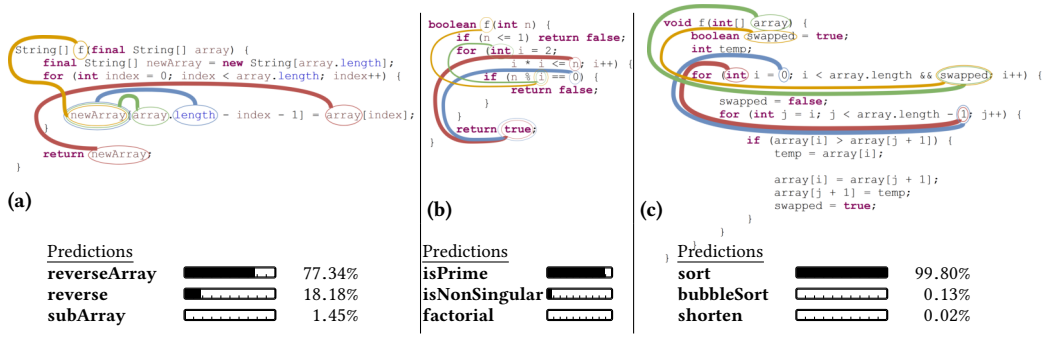

Fig. 6. Example predictions from our model, with the top-4 paths that were given the most attention for each code snippet. The width of each path is proportional to the attention it was given by the model.

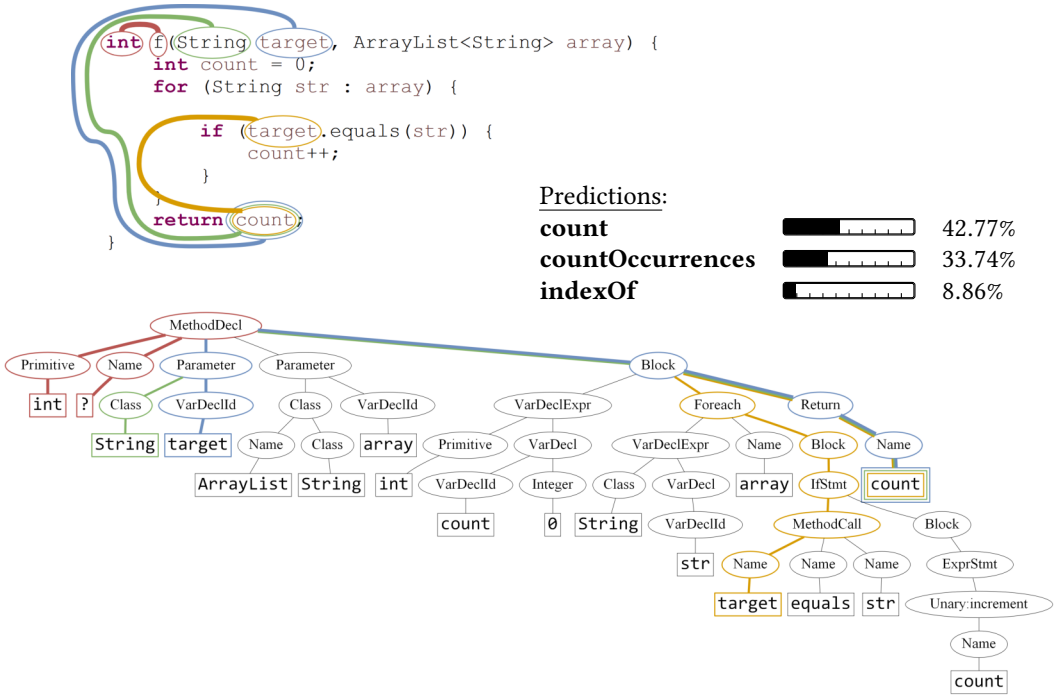

Fig. 7. An example for a method name prediction, portrayed on the AST. The top-four path-contexts were given a similar attention, which is higher than the rest of the path-contexts.

length in our dataset of real-world projects is 7 lines, and the examples presented in this section are longer than this average length.

Figure 7 and Figure 8 show additional predictions of our model, 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. An interactive demo of method name predictions and name vector similarities can be found at: http://code2vec.org. When manually examining the predictions of custom inputs, it is important to note that a machine learning model learns to predict names for examples that are

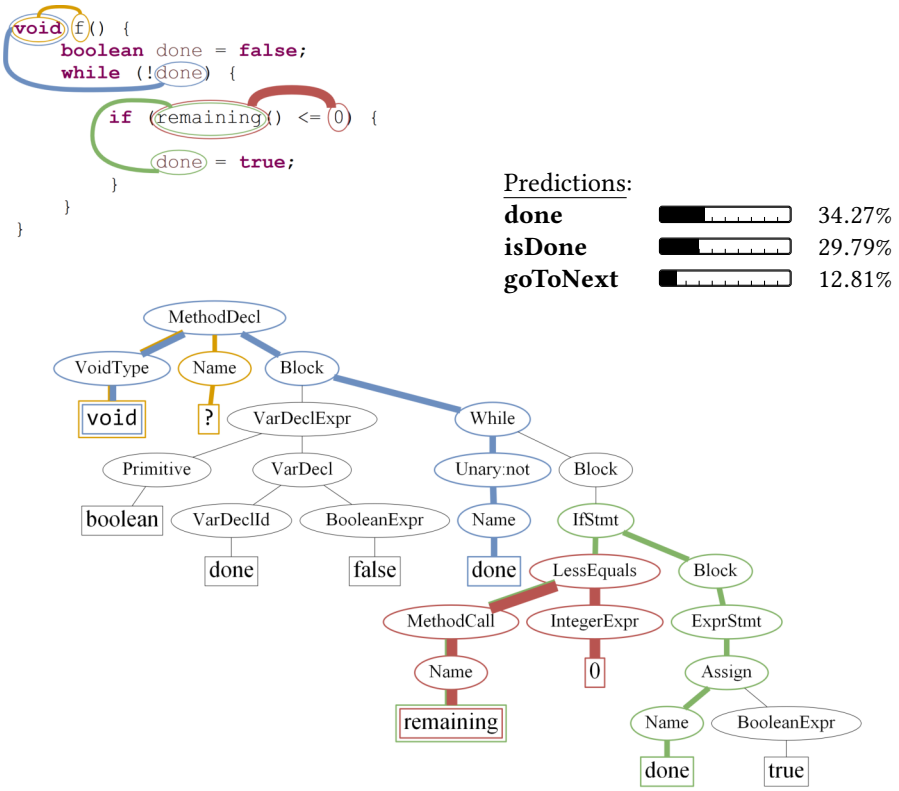

Fig. 8. An example for a method name prediction, portrayed on the AST. The width of each path is proportional to the attention it was given.

Table 6. Semantic combinations of method names.

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	toLowerCase	equalsIgnoreCase

Table 7. Semantic analogies between 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

likely to be observed "in the wild". Thus, it can be misled by confusing adversarial examples that are unlikely to be found in real code.

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 $\boldsymbol{v}$ that maximizes the similarity to two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$:

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

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, Equation (3) can be written as:

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

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

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 $\hat{\boldsymbol{u}}$ as the unit vector of a vector $\boldsymbol{u}$, Equation (4) can be simplified to:

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

since cosine distance between two vectors equals the dot product of their unit vectors. This provides us with a simpler method for finding the above combination of method name similarities:

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

This implies that the model has learned that `equalsIgnoreCase` is the most similar name to `equals` *and* `toLowerCase` combined. Table 6 shows some of these examples.

Just as Mikolov et al. [2013a,c] used vector calculation to express syntactic and semantic word analogies in NLP, the method name vectors learned by our model also express similar syntactic and semantic analogies. For example, $vec\left(\texttt{download}\right)$-$vec\left(\texttt{receive}\right)$+$vec\left(\texttt{send}\right)$ results in a vector whose closest neighbor is the vector for `upload`. This analogy can be read as: "`receive` is to `send` as `download` is to: `upload`". More examples are shown in Table 7.

7 LIMITATIONS OF OUR MODEL

In this section we discuss some limitations of our model and potential future research directions.

Closed labels vocabulary. One of the major limiting factors is the closed label space we use as target: our model is able to predict only labels that were observed as is at training time. This works very well for the vast majority of targets (that repeat across multiple programs), but as the targets become very specific and diverse (e.g., `findUserInfoByUserIdAndKey`) the model is unable to compose such names and usually catches only the main idea (for example: `findUserInfo`). Overall, on a general dataset, our model outperforms the baselines by a large margin even though the baselines are technically able to produce complex names.

Sparsity and Data-hunger. There are three main sources of sparsity in our model:

- Terminal values are represented as whole symbols - e.g., each `newArray` and `oldArray` is a unique symbol that has an embedding of its own, even though they share most of their characters (`Array`).
- AST paths are represented as monolithic symbols - two paths that share most of their AST nodes but differ in only a single node are represented as distinct paths which are assigned distinct embeddings.

- Target nodes are whole symbols, even if they are composed of more common smaller symbols.

This sparsity results in the model consuming a lot of trained parameters to keep an embedding for each observed value. The large number of trained parameters results in large GPU memory consumption at training time, increases the size of the stored model (about 1.4 GB), and requires a lot of training data. Furthermore, modeling source code with a finer granularity of atomic units may allow the model to represent more unseen contexts as compositions of smaller atomic units, thus repeating more atomic units across examples. In the model described in this paper, paths, terminal values or target values that were not observed during training cannot be represented. To address these limitations we train the model on a huge dataset of more than 12M examples, but the model might not perform as well using smaller datasets. Although requiring a lot of GPU memory, training our model on millions of examples fits in the memory of a relatively old Tesla K80 GPU.

An alternative approach for reducing the sparsity of AST paths is to use *path abstractions* where only parts of the path are used in the context (abstracting away certain kinds of nodes, merging certain kinds of nodes, etc.).

Dependency on variable names. Since we trained our model on top-starred open-source projects where variable naming is usually good, the model has learned to leverage variable names to predict the target label. When given uninformative, obfuscated or adversarial variable names, the prediction of the label is usually less accurate. We are considering several approaches to address this limitation in future research. One potential solution is to train the model on a mixed dataset of good and hidden variable names, hopefully reducing model dependency on variable names; another solution is to apply a model that was trained for variable de-obfuscation first (such as [Alon et al. 2018; Raychev et al. 2015]) and feed the predicted variable names into our model.

8 RELATED WORK

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, and therefore can be used as a default representation for any machine learning model for code. Our representation is similar to theirs but can represent *whole snippets of code*. 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*. Their approach can be used only to perform predictions for the exact task it was trained for, while our approach produces *code vectors*, that once trained for a single task, are useful for other tasks as well.

Syntax-based contexts have been used by Raychev et al. [2016b] and Bielik et al. [2016]. Our work targets different tasks. Moreover, 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 at its ends have never been seen by the model before. Another major difference between these works and ours is 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 propose to use *soft attention*, which uses multiple contexts for prediction, with different weights for each. In previous works which used non-neural techniques, soft attention is not even expressible.

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; Gvero and Kuncak 2015; Maddison and Tarlow 2014], n-grams [Allamanis 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. David et al. [2016, 2017] and David and Yahav [2014] use simple models trained on various forms of *tracelets* extracted statically from programs. In Katz et al. [2016, 2018], language models are trained over sequences of API-calls extracted statically from binary code.

Distributed representations of code identifiers were used to predict variable, method, and class names based on token context features [Allamanis et al. 2015a]. A recent work [DeFreez et al. 2018] learns distributed representations of C functions based on the control-flow graph.

Bimodal modeling 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 [Iyer et al. 2016; Murali et al. 2017; 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, when trained for predicting method names instead of description, our model outperformed their model by a large gap. Allamanis et al. [2015b]; 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 token stream.

Allamanis et al. [2016] were the first to consider the problem of predicting method names. However, they used attention over a "sliding window" of tokens, while our model leverages the syntactic structure of code and proposes a simpler architecture which scales to large corpora more easily. 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.

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 focus 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.

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 achieve 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] 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 implicitly factorizes 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 *fewer* parameters than methods based on symbolic representations, and produce vectors for which semantically similar method names are similar in the embedded space.

9 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 a 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.

We demonstrated our approach by predicting method names using a model that was trained on more than 12M 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, all of our code and trained model are publicly available at https://github.com/tech-srl/code2vec.

ACKNOWLEDGMENTS

We would like to thank Guy Waldman for developing the code2vec website (http://code2vec.org). We also thank Miltiadis Allamanis and Srinivasan Iyer for their guidance in the use of their models in the evaluation section, and Yaniv David, Dimitar Dimitrov, Yoav Goldberg, Omer Katz, Nimrod Partush, Vivek Sarkar and Charles Sutton for their fruitful comments.

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7) under grant agreement no. 615688-ERC- COG-PRIME. Cloud computing resources were provided by an AWS Cloud Credits for Research award.

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. 2015a. 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, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *ICLR*.

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. 2015b. Bimodal Modelling of Source Code and Natural Language. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15)*. JMLR.org, 2123–2132. http://dl.acm.org/citation.cfm?id=3045118.3045344

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A General Path-based Representation for Predicting Program Properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 404–419. https://doi.org/10.1145/3192366.3192412

Matthew Amodio, Swarat Chaudhuri, and Thomas W. Reps. 2017. Neural Attribute Machines for Program Generation. *CoRR* abs/1705.09231 (2017). arXiv:1705.09231 http://arxiv.org/abs/1705.09231

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

Chris Callison-Burch, Miles Osborne, and Philipp Koehn. 2006. Re-evaluation the role of bleu in machine translation research. In *11th Conference of the European Chapter of the Association for Computational Linguistics*.

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

Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical Similarity in Binaries. In *PLDI'16: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Yaniv David, Nimrod Partush, and Eran Yahav. 2017. Similarity of Binaries through re-optimization. In *PLDI'17: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Yaniv David and Eran Yahav. 2014. Tracelet-Based Code Search in Executables. In *PLDI'14: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 349–360.

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.

Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. 2018. Path-based Function Embedding and Its Application to Error-handling Specification Mining. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 423–433. https://doi.org/10.1145/3236024.3236059

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

Omer Katz, Ran El-Yaniv, and Eran Yahav. 2016. Estimating Types in Executables using Predictive Modeling. In *POPL'16: Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages*.

Omer Katz, Noam Rinetzky, and Eran Yahav. 2018. Statistical Reconstruction of Class Hierarchies in Binaries. In *ASPLOS'18: Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems*.

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 18th 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

Yanxin Lu, Swarat Chaudhuri, Chris Jermaine, and David Melski. 2017. Data-Driven Program Completion. *CoRR* abs/1705.09042 (2017). arXiv:1705.09042 http://arxiv.org/abs/1705.09042

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).

Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. 2017. Bayesian Sketch Learning for Program Synthesis. *CoRR* abs/1703.05698 (2017). arXiv:1703.05698 http://arxiv.org/abs/1703.05698

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