A General Path-Based Representation for Predicting Program Properties

Uri Alon
Technion
Haifa, Israel
urialon@cs.technion.ac.il

Meital Zilberstein
Technion
Haifa, Israel
mbs@cs.technion.ac.il

Omer Levy
University of Washington
Seattle, WA
omerlevy@cs.washington.edu

Eran Yahav
Technion
Haifa, Israel
yahave@cs.technion.ac.il

Abstract

Predicting program properties such as names or expression types has a wide range of applications. It can ease the task of programming, and increase programmer productivity. A major challenge when learning from programs is *how to represent programs in a way that facilitates effective learning.*

We present a *general path-based representation* for learning from programs. Our representation is purely syntactic and extracted automatically. The main idea is to represent a program using paths in its abstract syntax tree (AST). This allows a learning model to leverage the structured nature of code rather than treating it as a flat sequence of tokens.

We show that this representation is general and can: (i) cover different prediction tasks, (ii) drive different learning algorithms (for both generative and discriminative models), and (iii) work across different programming languages.

We evaluate our approach on the tasks of predicting variable names, method names, and full types. We use our representation to drive both CRF-based and word2vec-based learning, for programs of four languages: JavaScript, Java, Python and C#. Our evaluation shows that our approach obtains better results than task-specific handcrafted representations across different tasks and programming languages.

CCS Concepts • **Software and its engineering → General programming languages**; *Automatic programming*; • **Computing methodologies → Machine learning**;

Keywords Programming Languages, Big Code, Machine Learning, Learning Representations

ACM Reference Format:
Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A General Path-Based Representation for Predicting Program Properties. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3192366.3192412

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5698-5/18/06…$15.00
https://doi.org/10.1145/3192366.3192412

1 Introduction

Leveraging machine learning models for predicting program properties such as variable names, method names, and expression types is a topic of much recent interest [6, 7, 12, 30, 38, 40]. These techniques are based on learning a statistical model from a large amount of code and using the model to make predictions in new programs. A major challenge in these techniques (and in many other machine-learning problems) is how to represent instances of the input space to facilitate learning [42]. Designing a program representation that enables effective learning is a critical task that is *often done manually for each task and programming language.*

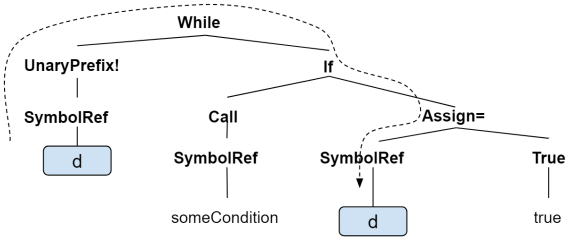
Our approach We present a novel program representation for learning from programs. Our approach uses different path-based abstractions of the program's abstract syntax tree. This family of path-based representations is natural, general, fully automatic, and works well across different tasks and programming languages.

AST paths We define AST paths as paths between nodes in a program's abstract syntax tree (AST). To automatically generate paths, we first parse the program to produce an AST, and then extract paths between nodes in the tree. We represent a path in the AST as a sequence of nodes connected by up and down movements, and represent a program element as the set of paths that its occurrences participate in. Fig. 1a shows an example JavaScript program. Fig. 1b shows its AST, and one of the extracted paths. The path from the first occurrence of the variable `d` to its second occurrence can be represented as:

SymbolRef ↑ UnaryPrefix! ↑ While ↓ If ↓ Assign= ↓ SymbolRef

```
while (!d) {
  if (someCondition()) {
    d = true;
  }
}
```

(a) A simple JavaScript program.



(b) The program's AST, and example to an AST path.

Figure 1. A JavaScript program and its AST, along with an example of one of the paths.

This is an example of a pairwise path between leaves in the AST, but in general the family of path-based representations contains n-wise paths, which do not necessarily span between leaves and do not necessarily contain all the nodes in between. Specifically, we consider several choices of subsets of this family in Section 4.

Using a path-based representation has several major advantages over existing methods:

1. Paths are generated automatically: there is no need for manual design of features aiming to capture potentially interesting relationships between program elements. This approach extracts unexpectedly useful paths, without the need for an expert to design features. The user is required only to choose a subset of our proposed family of path-based representations.
2. This representation is useful for any programming language, without the need to identify common patterns and nuances in each language.
3. The same representation is useful for a variety of prediction tasks, by using it with off-the-shelf learning algorithms or by simply replacing the representation of program elements in existing models (as we show in Section 5.3).
4. AST paths are purely syntactic, and do not require any semantic analysis.

Tasks In this paper, we demonstrate the power and generality of AST paths on the following tasks:

- **Predicting names for program elements** Descriptive and informative names for program elements such as variables and classes play a significant role in the readability and comprehensibility of code. Empirical studies have shown that choosing appropriate names makes code more understandable [44], reduces code maintenance efforts, and leads to fewer bugs [13]. A study in the Psychology of Programming suggests that the ways in which programmers choose names reflect deep cognitive and linguistic influences [28]. A meaningful name describes the role of a program element, carries its semantic meanings, and indicates its usage and behavior throughout the program.
- **Predicting method names** Good method names adequately balance the need to describe the internal implementation of the method and its external usage [21]. When published in a popular library's API, descriptive and intuitive method names facilitate the use of methods and classes, while poorly chosen names can doom a project to irrelevance [6]. Although method names are clearly program elements and can be predicted by the previous task, in this task we assumes that all the other names in the method are given, along with the names of the elements around the method invocation, when available in the same file.
- **Predicting expression types** Statistical type prediction allows (likely) types of expressions to be inferred without the need for type inference, which often requires a global program view (possibly unavailable, e.g., in the case of snippets from sites such as StackOverflow).

Raychev et al. [40] used relations in the AST as features for learning tasks over programs. They defined an explicit grammar to derive features which capture specific relationships between nodes in the AST of JavaScript programs, as well as relations produced by language-specific semantic analysis, such as "may call" and "may access". We show that our *automatic general representation* performs better than their features for their original task, and also generalizes to drive two different learning algorithms and three different prediction tasks, over different programming languages.

Paths in an AST have also been used by Bielik et al. [12] and by Raychev et al. [38, 39] for a different goal: identifying context nodes. These works do not use the paths themselves as a representation of the input, and the prediction is only affected by the context node that was found on the other end of the path. In our work, we use the path itself as a representation of a program element. Therefore, the prediction depends not only on the context node but also on *the way it is related* to the element in question.

Allamanis et al. [6] defined the challenging task of predicting method names, which can be viewed as a form of function summarization [7]. We show that our representation performs better by being able to learn across projects.

We provide a more elaborate discussion of related work, including deep learning approaches, in Section 6.

Contributions The main contributions of this paper are:

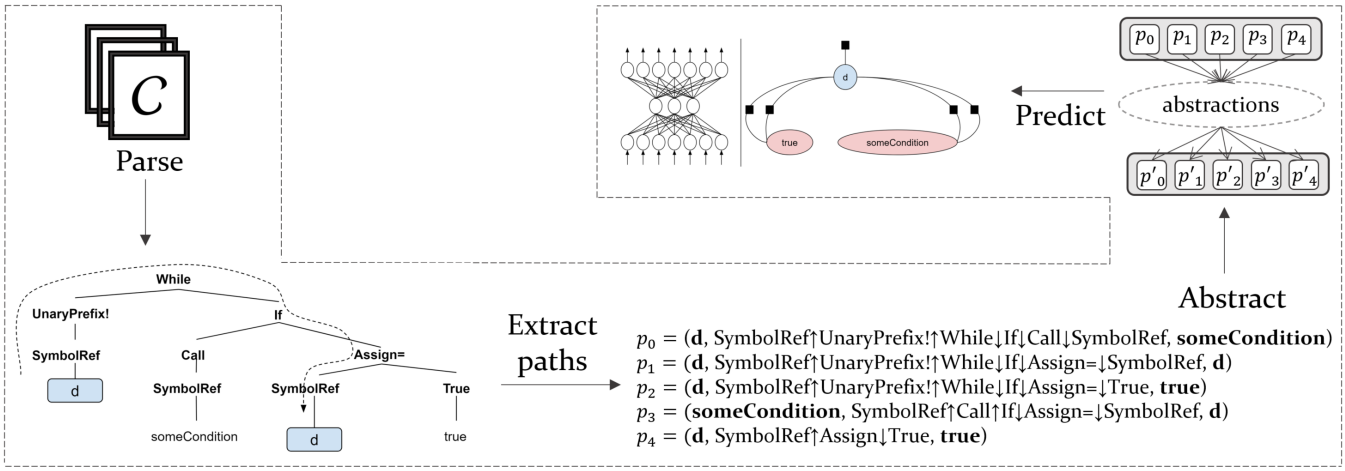


Figure 2. An overview of our approach. We start with a code snippet C, and extract its path representation to be used as an input to machine learning models. The AST and paths were extracted from the example program in Fig. 1a.

- A new, general family of representations for program elements. The main idea is to use AST paths as representations of code.
- A cross-language tool called Pigeon, which is an implementation of our approach for predicting program element names, method names, and types.
- An evaluation on real-world programs. Our experiments show that our approach produces accurate results for different languages (JavaScript, Java, Python, C#), tasks (predicting variable names, method names, types) and learning algorithms (CRFs, word2vec). Furthermore, for JavaScript and Java, where previous methods exist, our automatic approach produces more accurate results.

2 Overview

In this section, we illustrate our approach with a simple JavaScript program for the task of predicting names; as we show in later sections, the same approach also applies to other tasks, other languages, and other learning algorithms.

Given a program with non-descriptive names, our goal is to predict likely names for local variables and function parameters. The non-descriptive names could have been given by an inexperienced programmer, or could have been the result of deliberate stripping. In the latter case, we refer to such a program as a program with *stripped names*. Stripping names can be part of a minification process in JavaScript, or obfuscation in Java and other languages.

Consider the code snippet of Fig. 1a. This simple snippet captures a common programming pattern in many languages. Suppose that we wish to find a better name for the variable `d`.

Program element representation The main idea of our approach is to extract paths from the program's AST and use them to represent an element, such as the variable `d`, in a machine learning model. Fig. 2 shows an overview of this process. First, we parse the query program to construct an AST. Then, we traverse the tree and extract paths between AST nodes. To simplify presentation, in this example we only consider pairwise paths between AST leaves. We assume that a path is represented as a sequence of AST nodes, linked by up and down movements (denoted by arrows). As we describe in Sec. 4, the path can also connect a leaf and a higher nonterminal in the AST, connect several nodes (n-wise path), and can be abstracted in different levels.

Consider the p_1 in Fig. 2, between the two occurrences of the variable `d`:

$$\text{SymbolRef} \uparrow \text{UnaryPrefix!} \uparrow \text{While} \downarrow \text{If} \downarrow \text{Assign=} \downarrow \text{SymbolRef} \quad \text{(I)}$$

The path expresses the fact that the variable `d` is used, with negation, as a stopping condition of a "while" loop, and then assigned a new value if an "if" condition inside the loop evaluates to `true`. This path alone expresses the fact that `d` is the stopping condition of the loop.

The path p_4 in Fig. 2, between the variable `d` and the value `true` is:

$$\text{SymbolRef} \uparrow \text{Assign=} \downarrow \text{True} \quad \text{(II)}$$

This path captures the fact that the assignment changes the value of `d` to `true`, and therefore it is indeed the assignment that stops the loop.

Prediction By observing these two paths, a programmer is likely to name `d` "done", "complete", "stop" or something similar. Indeed, a learning model that was trained using our representation predicts that the most likely name for the variable is `done`, and neither "done", "complete", nor any similar name was predicted by past work for this example.

Learning algorithms The learning model can vary between different algorithms, presenting tradeoffs of efficiency and accuracy. In Section 5.3 we show that both CRFs and word2vec

```
var d = false;
while(!d) {
    doSomething();
    if (someCondition()) {
        d = true;
    }
}
```

(a)

```
someCondition();
doSomething();
var d = false;
d = true;
```

(b)

Figure 3. An example for two code snippets that are indistinguishable by the model of Raychev et al. [40], and are easily distinguishable by AST paths.

can be used for this prediction task. In both of these learning algorithms, using AST paths produces better results than the alternative representations, whether they are manually designed or sequence-based representations.

Path abstractions Automatic generation may produce a prohibitively large number of paths. To control the number of paths, *higher levels of abstraction* can be used. Instead of representing the whole path node-by-node, it can be further abstracted by keeping only parts of it, which results in similar paths being represented equally, as we show in Section 5.6. Another way to control the number of paths is to limit the number of extracted paths. We provide hyper-parameters (i.e., model configurations that are not tuned by the optimization process) that control the maximal length and width of AST paths. The number of extracted paths can be further reduced using downsampling, with minimal impact on accuracy and a significant saving in training time (Sec. 5.3). These methods make the accuracy – training time tradeoff tunable.

The discriminative power of AST paths Examples indistinguishable by manually designed representations will always exist. For example, UnuglifyJS [40] extracts an identical set of relations (and therefore predicts the same name for `d`) for the example in Fig. 3a and for the simplified example in Fig. 3b, even though the variable `d` clearly does not play a similar role in these two examples. In contrast, these two examples are easily distinguishable using our AST paths.

Key aspects The example highlights several key aspects of our approach:

- Useful paths such as path I span multiple lines of the program, but are also supported by shorter paths like path II, which only spans a single program line. Short paths alone are not enough to predict a meaningful name. Making a prediction using all paths that an element participates in provides a rich context for predicting the name of the element.
- No special assumptions regarding the AST or the programming language were made, making the same mechanism useful in other languages in a similar way.
- This representation can be plugged into existing models as a richer representation of the input code, without interfering with the learning algorithm itself.
- AST paths can distinguish between programs that previous works could not.
- In addition to predicting `done`, a model trained with AST paths can propose several semantically similar names, as we demonstrate in Sec. 5.3. This shows that AST paths are strong indicators of the program element's semantics.

3 Background

In this section, we provide necessary background. In Sections 3.1 and 3.2 we present CRFs and word2vec and how they are used to predict program properties.

3.1 Conditional Random Fields

Probabilistic graphical models are a formalism for expressing the dependence structure of entities. Traditionally, graphical models have been used to represent the joint probability distribution $P(y, x)$, where y represents an assignment of attributes for the entities that we wish to predict, and x represents our observed knowledge about the entities [35, 36, 43]. Such models are called *generative* models. However, modeling the joint distribution requires modeling the marginal probability $P(x)$, which can be difficult, computationally expensive, and in our case requires us to estimate the distribution of programs [40].

A simpler solution is to model the conditional distribution $P(y|x)$ directly. Such models are called *discriminative* models. This is the approach taken by *Conditional Random Fields* (CRFs). A CRF is a conditional distribution $P(y|x)$ with an associated graphical structure [25]. They have been used in several fields such as natural language processing, bioinformatics, and computer vision.

Formally, given a variable set Y and a collection of subsets $\{Y_a\}_{a=1}^{A}$ of Y, an *undirected graphical model* is the set of all distributions that can be written as:

$$P(y) = \frac{1}{Z} \prod_{a=1}^{A} \Psi_a(y_a) \tag{1}$$

where each $\Psi_a(y_a)$ represents a *factor*. Each factor $\Psi_a(y_a)$ depends only on a subset $Y_a \subseteq Y$ of the variables. Its value is a non-negative scalar which can be thought of as a measure of how compatible the values y_a are. The constant Z is a normalization factor, also known as a partition function, that ensures that the distribution sums to 1. It is defined as:

$$Z = \sum_{y} \prod_{a=1}^{A} \Psi_a(y_a) \tag{2}$$

A CRF can also be represented as a bipartite undirected graph $G = (V, F, E)$, in which one set of nodes $V = \{1, 2, \ldots, |Y|\}$

represents indices of random variables, and the other set of nodes $F = \{1, 2, ..., A\}$ represents indices of the factors.

Several algorithms and heuristics were suggested for training CRFs and finding the assignment that yields the maximal probability [24, 25, 43]. We will not focus on them here, since our work is orthogonal to the learning model and the prediction algorithm.

Using CRFs to predict program properties was first proposed by Raychev et al. [40], where each node represented an identifier in the code. These include variables, constants, function and property names. The factors represented the relationships or dependencies between those identifiers, and were *defined by an explicit grammar and relations that were produced using semantic analysis.*

To demonstrate the use of AST paths with CRFs, we use CRFs exactly as they were used by Raychev et al. [40] but use AST paths instead of their original factors. Additionally, we introduce the use of unary factors (factors that depend on a single node). Unary factors are derived automatically by AST paths between different occurrences of the same program element throughout the program.

3.2 Neural Word Embeddings

Recently, neural-network based approaches have shown that syntactic and semantic properties of natural language words can be captured using low-dimensional vector representations, referred to as "word embeddings", such that similar words are assigned similar vectors [10, 14, 37]. These representations are trained over large swaths of unlabeled text, such as Wikipedia or newswire corpora, and are essentially unsupervised. The produced vectors were shown to be effective for various NLP tasks [15, 45].

In particular, the skip-gram model trained with the negative sampling objective (SGNS), introduced by Mikolov et al. [32, 33], has gained immense popularity via the word2vec toolkit, and substantially outperformed previous models while being efficient to train.

SGNS works by first extracting the *contexts*: $c_1, ..., c_n$ of each word w. It then learns a latent d-dimensional representation for each word and context in the vocabulary ($\vec{w}, \vec{c} \in \mathbb{R}^d$) by maximizing the similarity of each word w and context c that were observed together, while minimizing the similarity of w with a randomly sampled context c'. In Mikolov et al.'s original implementation, each context c_i is a neighbor of w, i.e., a word that appeared within a fixed-length window of tokens from w. Levy and Goldberg [26] extended this definition to include arbitrary types of contexts.

As shown by Levy and Goldberg [27], this algorithm implicitly tries to encode the pointwise mutual information (PMI) between each word and context via their vector representations' inner products:

$$\vec{w} \cdot \vec{c} = PMI(w, c) = \log \frac{p(w, c)}{p(w)p(c)} \tag{3}$$

where each probability term models the frequency of observing a word w and a context c together (or independently) in the training corpus.

Recently, a simple model has achieved near state-of-the-art results for the lexical substitution task using embeddings that were learned by word2vec [31]. The task requires identifying meaning-preserving substitutes for a target word in a given sentential context. The model in this work uses both word embeddings and context embeddings, and looks for a word out of the entire vocabulary whose embedding is the closest to all the given context embeddings and to the original word. The similarities between the substitute word and each of the contexts and the original word are aggregated by an arithmetic mean or a geometric mean.

In contrast to natural language methods, our method does not use the original word but finds the unknown name by aggregating only the similarities between the candidate vector w and each of the given context vectors $\widetilde{C}$:

$$prediction = argmax_{w \in W} \sum_{c \in \widetilde{C}} (w \cdot c) \tag{4}$$

To demonstrate the use of AST paths with word2vec, we use AST paths as the context of prediction. As we show in Section 5.3, using AST paths as context gives a relative improvement of 96% over treating code as a token-stream and using the surrounding tokens as context.

4 AST Paths Representation

In this section, we formally describe the family of AST paths.

4.1 AST Paths

To learn from programs, we are looking for a representation that captures interesting properties of ASTs while keeping it open for generalization. One way to obtain such a representation is to decompose the AST to parts that repeat across programs but can also discriminate between different programs. One such decomposition is into paths between nodes in the AST. We note that in general we consider n-wise paths, i.e., those that have more than two ends, but for simplicity we base the following definitions on pairwise paths between AST terminals.

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

Definition 4.1 (Abstract Syntax Tree)**.** An Abstract Syntax Tree (AST) for a code snippet C is a tuple $\langle N, T, X, s, \delta, val \rangle$ where N is a set of nonterminal nodes, T is a set of terminal nodes, X is a set of terminal 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 $val : T \rightarrow X$ 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.

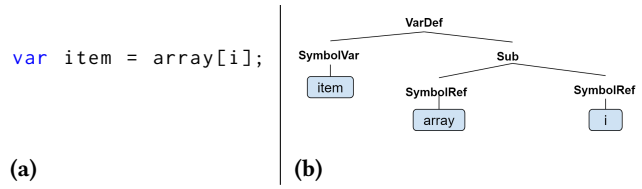


Figure 4. A JavaScript statement and its partial AST.

For convenience, we also define $\pi : (N \cup T) \rightarrow N$, the inverse function for δ. Given a node, this function returns its parent node, such that for every two terminal or nonterminal nodes $y_1, y_2 \in (N \cup T)$, one is the parent node of the other if and only if the latter is in the list of children of the former: $\pi(y_1) = y_2 \iff y_1 \in \delta(y_2)$. In the case of the start symbol, its parent is undefined.

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

An AST pairwise path is a path between two nodes in the AST, formally defined as follows:

Definition 4.2 (AST path)**.** An AST-path of length k is a sequence $n_1 d_1 ... n_k d_k n_{k+1}$, where for $i \in [1..k+1]$: $n_i \in (N \cup T)$ are terminals or 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: $n_{i+1} = \pi(n_i)$; if $d_i = \downarrow$, then: $n_i = \pi(n_{i+1})$. We use $start(p)$ to denote n_1 and $end(p)$ to denote n_{k+1}.

We define a *path-context* as a tuple of an AST path and the values associated with its end nodes: (i.e. n_1 and n_{k+1}). In general, we consider path-contexts which span between arbitrary AST nodes, e.g., a terminal and its ancestor, but for simplicity, we base the following definitions on path-contexts which span between terminals:

Definition 4.3 (Path-context)**.** Given an AST Path p, its path-context is the triplet $\langle x_s, p, x_f \rangle$ where $x_s = val(start(p))$ and $x_f = val(end(p))$ are the values associated with the start and end nodes of p.

That is, a path-context describes two nodes from the AST with the syntactic path between them.

Finally, we define an *Abstract path-context* as an abstraction of concrete path context:

Definition 4.4 (Abstract path-context)**.** Given a path-context $\langle x_s, p, x_f \rangle$ and an abstraction function $\alpha : P \rightarrow \hat{P}$, an abstract path-context is the triplet $\langle x_s, \alpha(p), x_f \rangle$, where P is the set of AST paths, $\hat{P}$ is the set of abstract AST paths, and α is a function that maps a path to an abstract representation of it.

The abstraction function α is any function that transforms a path to a different representation. A trivial abstraction function is α_{id}, which maps a path to itself: $\alpha_{id}(p) = p$.

Example 4.5. For example, consider the JavaScript line of code in Fig. 4a and its partial AST in Fig. 4b. We denote the path between the variable `item` to the variable `array` by p. Using α_{id}, the abstract path-context of p is:

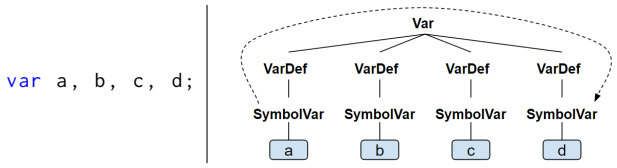


Figure 5. An example statement and its AST, with an example of a path between the *SymbolVar* terminals that represent `a` and `d`. The length of this path is 4, and its width is 3.

$$\langle \texttt{item}, \alpha_{id}(p), \texttt{array} \rangle = \tag{5}$$

$$\langle \texttt{item}, (SymbolVar \uparrow VarDef \downarrow Sub \downarrow SymbolRef), \texttt{array} \rangle \tag{6}$$

Using a different abstraction function yields a different abstract path-context, for example $\alpha_{forget-arrows}$:

$$\langle \texttt{item}, \alpha_{forget-arrows}(p), \texttt{array} \rangle = \tag{7}$$

$$\langle \texttt{item}, (SymbolVar, VarDef, Sub, SymbolRef), \texttt{array} \rangle \tag{8}$$

Naïvely extracting all the paths in the AST and representing each of them uniquely can be computationally infeasible, and as a result of the bias-variance tradeoff [19, p. 37 and 219], can lead to worse prediction results. However, alternative abstraction functions can be used to control the number of distinct extracted paths. In Section 5.6 we describe alternative abstractions that abstract some of the information, and thus allow us to tune the trade-off between accuracy, training time, and model size.

4.2 Limiting the Number of Paths

Another approach for controlling the number of distinct paths is to limit the *number of extracted paths.*

Path length and width We define hyper-parameters that limit the path length and width. We define the following hyper-parameters:

- *max_length*, defined as the maximal *length* of a path, i.e., the maximum value of k.
- *max_width*, defined as the maximal allowed difference between sibling nodes that participate in the same path, as shown in Fig. 5.

When limiting these parameters to certain values, we do not extract longer or wider paths. We tune the optimal values of width and length by grid search of combinations on a validation set of programs and choose the combination that yields the highest accuracy, as described in Section 5. The tuning process of finding the optimal parameter values should be separate for each language and task.

Obviously, setting the values of these parameters to a value that is too low limits the expressiveness of the paths, does not capture enough context for each element, limits the ability to model the training and test data, and therefore produces

```
assert.equal(a,1);
assert.equal(...);
...
assert.equal(b,1);
```

Figure 6. An example of a typical program where the maximal path length is relatively small, but the width can be large.

poor accuracy. Why, then, does limiting the path length and width actually improve accuracy? There are several reasons:

- **Locality** The role of a program element is affected mostly by its surroundings. For example, consider the program in Fig. 6. The width of a path that connects the variable a in the first line to the variable b in the last line is as large as the number of lines in the program. Usually, the names of a and b can be predicted by considering elements within a closer distance. Therefore, using paths between too distant elements can cause noise and pollute the relevant information.
- **Sparsity** Using paths that are too long can cause the representation space to be too sparse. A long path might appear too few times (or even only once) in the training set and cause the model to predict specific labels with high probability. This phenomenon is known as *overfitting*, where the learned AST paths are very specific to the training data and the model fails to generalize to new, unseen data.
- **Performance** There is a practical limit on the amount of data that a model can be trained on. Too much data can cause the training phase to become infeasibly long. There is a tradeoff between how many programs the model can be trained on, and how many paths are extracted from each program. Therefore, it makes sense to limit the number of extracted paths from each program by limiting the paths' length and width, in order to be able to train on a larger and more *varied* training set.

In fact, tuning path length and width is used to control the bias-variance tradeoff. Shorter paths increase the bias error, while longer paths increase the variance error. The relationship between these parameters and results is discussed and demonstrated in Section 5.5.

5 Evaluation

Since the goal of this work is to *provide a representation of program elements*, we compared the effect of different representations on the accuracy of the learning algorithms. To show that our approach can be applied to the representation of the input without modifying the learning algorithm, we used off-the-shelf learning algorithms but represented the input in each experiment using *a different representation* (when possible).

Our evaluation aims to answer the following questions:

- How useful are AST paths compared to existing representations? (Section 5.3)
- How useful are AST paths across different programming languages, tasks and learning algorithms? (Section 5.3)
- Do AST paths just memorize the input, or do they capture deeper semantic regularities? (Section 5.4)
- How long are the useful paths? How do the paths' length and width affect the results? (Section 5.5)
- How important is the concrete representation of paths? Which abstractions can be used to represent paths without reducing accuracy? (Section 5.6)

Leafwise and semi-paths Although the family of representations in this work includes n-wise paths and paths between any kind of AST nodes, for simplicity and feasible training time, we performed most of the experiments using leafwise-paths (paths between AST terminals) and semi-paths — paths between an AST terminal and one of its ancestor nodes in the AST. The idea is that leafwise-paths are more diverse and therefore more expressive than semi-paths, but semi-paths provide more *generalization*. Semi-paths allow us to generalize learning and capture common patterns in different programs, even if the full path does not recur.

An exception is the prediction of full types in Java, in which we predict types of expressions which are not necessarily terminals. In this case, we also used paths between terminals to the nonterminal in question.

5.1 Prototype Implementation

We implemented a cross-language tool called Pigeon. The tool consists of separate modules that parse and traverse the AST of a program in each different language, but the main algorithm is the same across all languages. Currently Pigeon contains modules for Java, JavaScript, Python and C#, and it can be easily extended to any other language.

AST construction and path extraction For Java we used JavaParser; for JavaScript we used UglifyJS for parsing and traversing the AST, along with additional modifications from UnuglifyJS; for Python we used the Python internal parser and AST visitor; and for C# we used Roslyn.

Learning algorithms We experiment with two learning algorithms: Conditional Random Fields, based on the implementation of Nice2Predict [40], and the word2vec based implementation of Levy and Goldberg [26].

To support our representation in the learning engine side and produce a qualitative evaluation, we introduced minor extensions to the Nice2Predict framework:

- *Support unary factors.* Previously, Nice2Predict supported only pairwise feature functions, and we implemented support for unary factors to express the relationship between different occurrences of the same identifier. Note that this is required because different

AST nodes for the same identifier are merged into a single node in the CRF. Hence, a path between these nodes in the AST becomes a unary-factor in the CRF. This extension increases accuracy by about 1.5%.

- *Top-k candidates suggestion.* CRFs output a single prediction for each program element. We implemented an additional API that receives a parameter k and suggests the *top-k* candidate names for each program element (this extension was adopted into Nice2Predict). This allowed us to manually investigate the quality of results (Section 5.4). When all top-k predictions for a variable name captured similar notions, it increased our confidence that the model performs stable predictions.

5.2 Experimental Setting

Data sets For each language, we collected source code from public GitHub projects, and split it randomly to training, validation and test sets. Our data included the top ranked projects of each language and the projects that were forked the most. Table 1 shows the amount of data used for each language. Java required an order of magnitude more data than the other languages: we had to keep enlarging our Java dataset to achieve results that were close to the other languages.

Following recent work which found a large amount of code duplication in GitHub [29], we devoted much effort into filtering duplicates from our dataset, and especially the JavaScript dataset. To filter duplicates, we used file names, directory names (such as "node_modules"), and md5 of files. In Java and Python, which do not commit dependencies, duplication is less severe (as also observed by Lopes et al. [29]). Furthermore, in our setting, we took the top-ranked and most popular projects, in which we observed duplication to be less of a problem (Lopes et al. [29] measured duplication across *all* the code in GitHub).

Evaluation metric For simplicity, in all the experiments we measured the percentage of exact match predictions, case-insensitive and ignoring differences in non-alphabetical characters. For example, this metric considers `totalCount` as an exact match to `total_count`. An exception is the comparison to Allamanis et al. [7], who optimized their Java method name prediction model to maximize the F1 score over sub-tokens. In this case, we compared their model with ours on both exact match and F1 score. An unknown test label ("UNK") was always counted as an incorrect prediction, or as a possibly partial prediction when using the F1 score, and our model never suggests "UNK". For example, if the true test label is `get<UNK>`, our model could get partial precision and partial recall for predicting `getFoo`.

Table 1. The amounts of data used for the experimental evaluation of each language.

Language	Total repos	Training Set File	Training Set Size (GB)	Test set File	Test set Size (MB)
Java	10, 081	1, 717, 016	16	50, 000	1001
JavaScript	6, 863	159, 038	3.4	38, 103	130
Python	8, 565	458, 771	5.4	39, 941	588
C#	1, 000	262, 774	4.7	50, 000	1208

5.3 Quantitative Evaluation

We conducted several experiments to evaluate the usefulness of AST paths in different tasks and programming languages. We performed the following quantitative experiments:

- *Prediction of variable names across all four languages.* Variable names have sufficient training data in all languages to produce meaningful results. In this experiment we used both CRFs and word2vec. As baselines we used the work of Raychev et al. [40], CRFs with token-based n-grams as factors, and a simple rule-based baseline. For JavaScript with word2vec, we used word2vec with linear token context as a baseline and show that path representations yield dramatic improvement.
- *Prediction of method names across JavaScript, Java and Python.* We compared our general approach for method name prediction with Allamanis et al. [7], who used a convolutional neural network with attention.
- *Prediction of full types in Java.* For Java, we compared our results to a synthetic (straw-man) baseline that predicts all types to be `java.lang.String`. This baseline shows that despite the prevalence of the `String` type, the task of type prediction is still very challenging.

In all of the following CRF experimental setups, "no-path" refers to a "bag-of-words" baseline, in which we used the same CRF learning algorithm, but used a single symbol to represent all relations. In this baseline, path information was hidden from the model during training and testing, and therefore it always assigned the same likelihood for each specific pair of identifiers, regardless of the syntactic relation between them. This baseline can be seen as a "bag of near identifiers" that uses the neighbors' names without their syntactic relation and therefore without considering the *way* program elements are related.

5.3.1 Predicting Variable Names

To predict variable names, we used both CRFs and word2vec.

Evaluation with CRFs We present our evaluation results with CRFs for names in the top part of Table 2. For JavaScript, where a tool that uses predefined features exists, we evaluated the other tool with the exact same datasets and settings, and the same AST terminals as CRF nodes, which makes the input representation (AST paths vs. their features) the

Table 2. Accuracy comparison for variable name prediction, method name prediction, and full type prediction using CRFs.

Task	Previous works		AST Paths (this work)	Params (length/width)
Variable name prediction				
JavaScript	24.9% (no-paths)	60.0% (UnuglifyJS)	**67.3%**	7/3
Java	23.7% (rule-based)	50.1% (CRFs+4-grams)	**58.2%**	6/3
Python	35.2% (no-paths)		**56.7%** (top-7)	7/4
C#			**56.1%**	7/4
Method name prediction				
JavaScript	44.1% (no-paths)		**53.1%**	12/4
Java	16.5%, F1: 33.9 (Allamanis et al. [7])		**47.3%**, F1: **49.9**	6/2
Python	41.6% (no-paths)		**51.1%** (top-7)	10/6
Full type prediction				
Java	24.1% (naïve baseline)		**69.1%**	4/1

Table 3. Accuracy comparison for the variable name prediction task that was evaluated using word2vec in JavaScript.

Model	Names Accuracy
linear token-stream + word2vec	20.6%
path-neighbors, no-paths + word2vec	23.2%
AST Paths (this work) + word2vec	**40.4%**

only difference between the two experiments. Using our representations yields 7.6% higher accuracy than the previous work.

For Java, we compared the results with two baselines:

- *CRFs + n-grams* - this baseline uses the same CRF nodes as the path-based model, except that the relations between them are the *sequential* n-grams. We chose $n = 4$ as the value that maximizes accuracy on the validation set, such that the produced model consumes approximately the same amount of memory and disk as the path-based model.
- *Rule-based* - Since Java is a typed language which has a rich type system, and typical code tends to use a lot of classes and interfaces, we wonder whether the task of predicting variable names is easier in Java than in other languages and can be solved using traditional rule-based (non-learning) approaches. Our rule-based baseline predicts variable names based on the following pattern heuristics and statistics of the training corpus:
  - `for(int i = ...) {`
  - `this.<fieldName> = <fieldName>;`
  - `catch (... e) {`
  - `void set<fieldName>(... <fieldName>) {`
  - Otherwise: use the type: `HttpClient client`.

As shown, using CRFs with AST paths yields higher results than the baselines, in all the languages, showing that our representation yields higher results than manually defined features, n-grams, and rule-based approaches.

Evaluation with word2vec We present our evaluation results with a word2vec based implementation in Table 3. For comparison, we use two alternative approaches to represent the context for prediction:

- The *linear token-stream* approach uses the surrounding tokens to predict a variable name. Surrounding tokens (e.g., values, keywords, parentheses, dots and brackets) may implicitly hint at the syntactic relations, without AST paths. This is the type of context usually used in NLP, in the original implementation of word2vec, and in many works in programming languages.
- The *path-neighbors, no-paths* approach uses the same surrounding AST nodes for contexts as AST paths, except that the path itself is hidden, and only the identity of the surrounding AST nodes is used. The goal of using this baseline is to show that the advantage of AST paths over token-stream is not only in their wider *span*, but in the representation of the path itself.

Using word2vec with AST paths produces much better results compared to these baselines. This shows the advantage of using AST paths as context over token-stream based contexts, and the significance of using a representation of the paths for prediction.

Limitations of evaluation We noticed that our models often predict names that are very similar but not identical to the original name, such as `message` instead of `msg`, or synonyms such as `complete` instead of `done`; these are counted as incorrect predictions. Moreover, we noticed that our models sometimes predict *better* names than the original names. Therefore, the accuracy results are an underapproximation of the ability of AST paths to predict meaningful names.

Another limitation lies in the inability of CRFs and word2vec to predict out-of-vocabulary (OoV) names. As was previously observed [6, 7], there are two main types of OoV names in programs: names that did not appear in the training corpus but can be composed of known names (neologisms), and entirely new names. The total OoV rate among our various

datasets and tasks varied between 5 – 15%, and specifically 7% for predicting variable names in JavaScript, and 13% for Java method names. Several techniques were suggested to deal with each type of OoV [6, 7], which we did not consider here and are out of scope of this work.

Discussion We note that the accuracy for Java is lower than for JavaScript. We have a few possible explanations: (i) The JavaScript training set contains projects that are rather domain specific, mostly client and server code for web systems (for example, the terms `request` and `response` are widely used across all projects). In contrast, the Java code is much more varied in terms of domains. (ii) The Java naming scheme makes extensive use of compound names (e.g., `multithreadedHttpConnectionManager`), and this is amplified by the type-based name suggestions for variables provided by modern Java IDEs. In contrast, the JavaScript variable names are typically shorter and are not an amalgamation of multiple words (e.g., `value`, `name`, `elem`, `data` are frequent names).

The accuracy of C# is similar to Java, but using significantly less training data. We believe that C# naming is more structured because the commonly used C# IDE (VisualStudio), suggests variable names based on their types.

The accuracy for Python is lower than that of JavaScript. Manual examination of the training data shows that Python programs vary widely in code quality, making the training set more noisy than that of other languages. In addition, the variety of domains and IDEs for Python makes variable names less standard. Finally, Python is easy to write, even for non-programmers, and thus there is a wide variety of non-professional Python code. The low accuracy for Python is also consistent with Raychev et al. [38].

Comparison of CRFs and word2vec We observe that the accuracy of Pigeon + CRFs is higher than that of Pigeon + word2vec, as can be seen in Table 2. One reason is that, unlike CRFs, word2vec was not designed exactly for this prediction task. Originally, word2vec was intended to produce meaningful word embeddings: given a set of query path-contexts, the vectors of all of them are assigned the same weight for predicting the unknown value.

Moreover, CRFs are relatively more interpretable. The weights assigned to each factor can be observed and explain a prediction posteriori. However, word2vec was faster to train and much more memory efficient. In our evaluation, the memory required for training was over 200GB for CRFs and only 10GB with word2vec. Further, the training time of CRFs was up to 100 hours, where word2vec required at most 5 hours.

The goal here is not to provide a fair comparison between CRFs and word2vec, as their prediction tasks are slightly different; our observations in this regard are merely anecdotal. The main goal is to compare *different representations for the same learning algorithm* and show that each of the learning algorithms separately can be improved by plugging in our simple representation.

5.3.2 Predicting Method Names

We present our evaluation results for predicting method names in Table 2. Accuracy was similar for all languages ($\sim$ 50%).

Good method names balance the need to describe the internal implementation of the method and its external usage [21]. For predicting method names, we use mostly the paths from within a method to its name, but when available in the same file, we also use paths from invocations of the method to the method name. Ideally, one would use paths from different files (and for library methods, even across projects), but this requires a non-local view, which we would like to avoid for efficiency reasons.

We use the internal paths from the leaf that represents the method name to other leaves within the method AST (which capture the method implementation) and the external paths from references of the method to their surrounding leaves (which represent the usage of the method). However, we observed that using only internal paths yields only 1% lower accuracy.

In Java, CRFs with AST paths are compared to the model of Allamanis et al. [7], which we trained on the same training corpus. Since their model is optimized to maximize the F1 score over sub-tokens, Table 2 presents both exact accuracy and F1 score for method name prediction in Java. The table shows that CRFs with AST paths significantly improve over the previous work in both metrics.

5.3.3 Predicting Full Types

Our results for predicting full types in Java using CRFs are shown in the bottom part of Table 2. Our goal is to predict the full type even when it explicitly appears in the code (e.g., `com.mysql.jdbc.Connection`, rather than `org.apache.http.Connection`). Here we also use paths from leaves to nonterminals which represent expressions. The evaluated types were only those that could be solved by a global type inference engine. Therefore, accuracy is the percent of correct predictions out of the results that are given by type inference.

Although a type inference engine still produces more accurate results than our learning approach, our results using AST paths are surprisingly good, especially considering the relative simplicity of our representation. We also note that type inference is a global task, and our approach reconstructs types locally without considering the global scope of the project.

CRFs with AST paths achieved 69.1% accuracy when predicting full type for Java. We contrast this result with a naïve baseline that uniformly predicts the type `java.lang.String` for all expressions. This naive baseline yields an accuracy

Stripped names	AST paths + CRFs

`def sh3(c):`
`  p = Popen(c, stdout=PIPE,`
`  stderr=PIPE, shell=True)`
`  o, e = p.communicate()`
`  r = p.returncode`
`  if r:`
`  raise CalledProcessError(r, c)`
`  else:`
`  return o.rstrip(), e.rstrip()`
	`def sh3(cmd):`
	`  process = Popen(cmd, stdout=PIPE,`
	`  stderr=PIPE, shell=True)`
	`  out, err = process.communicate()`
	`  retcode = process.returncode`
	`  if retcode:`
	`  raise CalledProcessError(retcode, cmd)`
	`  else:`
	`  return out.rstrip(), err.rstrip()`

Figure 7. Example of a Python program with stripped names and with predictions produced using our AST paths.

Stripped Names	AST Paths + CRFs	nice2predict.org

`function f(a, b, c) {`
`  b.open('GET', a, false);`
`  b.send(c);`
`}`
	`function f(url, request, callback) {`
	`  request.open('GET', url, false);`
	`  request.send(callback);`
	`}`
		`function f(source, req, n) {`
		`  req.open("GET", source, false);`
		`  req.send(n);`
		`}`

Figure 8. Example of a JavaScript program with stripped names, with predictions produced using our AST paths and an online version of UnuglifyJS at nice2predict.org. This is the default example shown at nice2predict.org.

Stripped names	AST paths + CRFs

`boolean d = false;`
`while (!d) {`
`  if (someCondition()) {`
`  d = true;`
`  }`
`}`
	`boolean done = false;`
	`while (!done) {`
	`  if (someCondition()) {`
	`  done = true;`
	`  }`
	`}`

`int count(List<Integer> x, int t) {`
`  int c = 0;`
`  for (int r: x) {`
`  if (r == t) {`
`  c++;`
`  }`
`  }`
`  return c;`
`}`
	`int count(List<Integer> values, int value) {`
	`  int count = 0;`
	`  for (int v: values) {`
	`  if (v == value) {`
	`  count++;`
	`  }`
	`  }`
	`  return count;`
	`}`

Figure 9. Examples of Java programs with stripped names and with predictions produced using our AST paths. We deliberately selected challenging examples in which the prediction cannot be aided by specific classes and interfaces.

of 24.1%, which shows the task is nontrivial, even when factoring out the most commonly used Java type.

5.4 Qualitative Evaluation

Our qualitative evaluation includes:

- An anecdotal study of name prediction in different languages. For JavaScript we also compared our predictions to those of Raychev et al. [40] in interesting cases.
- An anecdotal study of top-k predictions for some examples, showing semantic similarities between predicted names as captured by the trained model.

5.4.1 Prediction Examples

Fig. 7 shows an example of a Python program predicted using AST paths. It can be seen that all the names predicted using AST paths were renamed with meaningful names such as `process`, `cmd` and `retcode`.

Fig. 8 shows the default JavaScript example from nice2predict.org, predicted using AST paths and an online version of UnuglifyJS at nice2predict.org. We note that their online model was not trained on the same dataset as our model. The model which was trained using UnuglifyJS and our dataset yielded worse results. It can be seen that our model produced more meaningful names such as `url` (instead of `source`) and `callback` (instead of `n`).

Fig. 9 shows examples of Java programs. To demonstrate the expressiveness of AST paths, we deliberately selected challenging examples in which the prediction cannot be aided by the informative class and interface names that Java code usually contains (as in: `HttpClient client`). Instead,

Table 4. Semantic similarities between names.

(a) Candidates for prediction of the variable d from the example program of Fig. 1a.

	Candidate
1.	done
2.	ended
3.	complete
4.	found
5.	finished
6.	stop
7.	end
8.	success

(b) Examples of semantic similarities between names found among the top-10 candidates.

Semantic Similarities
req ~ request ~ client
items ~ values ~ objects ~ keys ~ elements
array ~ arr ~ ary ~ list
item ~ value ~ key ~ target
element ~ elem ~ el
count ~ counter ~ total
res ~ result ~ ret
i ~ j ~ index

our model had to leverage the syntactic structure to predict the meaningful names: `done`, `values`, `value` and `count`.

5.4.2 Semantic Similarity between Names

It is interesting to observe the other *candidates* that our trained model produces for program elements. As Table 4a shows, the next candidates after `done` (in Fig. 1a) are: `ended`, `complete`, `found`, `finished`, `stop` and `end`, which are all semantically similar (in programs, not necessarily in natural language). In many cases, AST paths capture the *semantic similarity* between names, for example `req~request` and `list~array`, as shown in Table 4b. This supports our hypothesis that AST paths capture the semantic role of a program element.

5.5 Impact of Parameter Values

In Section 4 we introduced and discussed the importance of the *max_length* and *max_width* parameters. For each language we experimented with different combinations of values for *max_length* and *max_width* on its validation set. We chose the values that produced the highest accuracy while still being computationally feasible when evaluating the model with the test set.

Accuracy with different path length and width We experimented with tuning the path parameters and observed their effect on the accuracy. The best parameter values for each prediction are shown in Table 2.

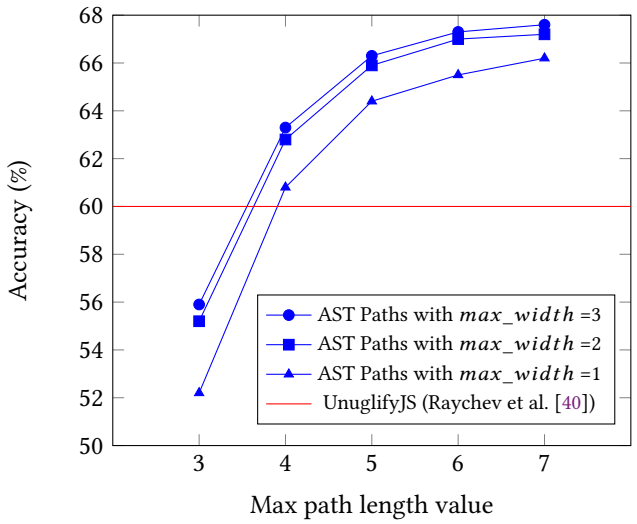


Figure 10. Accuracy results of AST paths with CRFs, for the task of variable naming in JavaScript, for different combination values of *max_length* and *max_width* (UnuglifyJS is presented here for comparison).

For the task of name prediction, for all languages, the best path length is 6-7, and the best width is 3-4. The variations in path length stem from minor differences in the structure of the AST. For example, despite the similarity in source level between Java and C#, the C# AST is slightly more elaborate than the one we used for Java.

A drill-down of the accuracy given different parameter values for variable name prediction in JavaScript is shown in Fig. 10. We observe that the *max_length* parameter has a significant positive effect, while the contribution of a larger *max_width* is positive but minor. This observation affirms our initial hypothesis that our long-distance paths are fundamental and crucial to the accuracy of the prediction. It also confirms our belief that an automatic representation of code (rather than manually defined) is essential, since the long-distance paths are very unlikely to have been designed manually.

For the task of method name prediction, since there are significantly fewer paths, we could afford to set a high parameter value without too much tuning and still keep the training time and resources feasible. We therefore set the length in this case to 12 for JavaScript, 10 for Python, and just 6 for Java.

For the task of predicting full types in Java, we used length 4 and width 1, which yielded an accuracy of 69.1%. The intuition for the short path length is that in many cases the type of an expression can be inferred locally from other neighboring types, often from an explicit type declaration.

Higher values for *max_length* and *max_width* resulted in higher training times, but combined with the *downsampling* approach, it is possible to maintain a shorter training

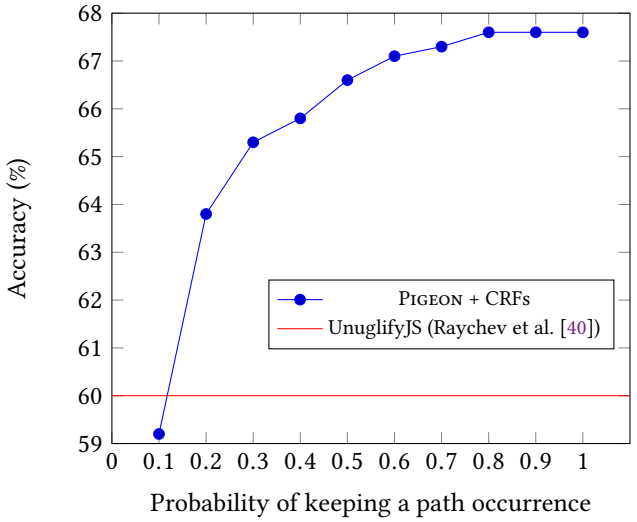


Figure 11. Downsampling: accuracy results of AST paths with CRFs, for the task of variable naming in JavaScript, for different values of p - the probability of keeping an AST path occurrence. UnuglifyJS was evaluated with all the training data and is presented here for comparison.

time while increasing the parameter values, and control the tradeoff between accuracy and training time.

Downsampling We wanted to measure the effect of training data size on accuracy and training time. Inspired by Grover and Leskovec [18], who used random walks to learn representations for neighborhoods of nodes in a network, we experimented with randomly omitting a fraction of the extracted path-contexts. After extracting path-contexts from all the programs in the training set, we randomly omitted each occurrence of a path-context in probability of $1-p$ (i.e., p is the probability to keep it) and trained a model only on the remaining paths. As can be seen in Fig. 11, randomly dropping contexts can significantly reduce training time, with a minimal effect on accuracy. For example, for $p = 0.8$ we observed exactly the same accuracy as for the complete set of paths ($p = 1$), while training time was reduced by about 25%. Moreover, decreasing p down to 0.2 still yielded higher accuracy than UnuglifyJS but reduced training time by about 80% (compared to $p = 1.0$).

5.6 Abstractions of AST Paths

In order to evaluate the full expressiveness of AST paths, the previously reported experiments were performed using no abstraction, i.e. α_{id}. However, it is also possible to use a higher level of abstraction. Instead of representing the whole path node-by-node with separating up and down arrows, it is possible to keep only parts of this representation. This abstraction results in less expressive paths and might represent two different paths as equal, but it enables decreasing the number of distinct paths, thus reducing the number of model parameters. Training will be faster as a result.

Different levels of path abstractions also allow us to evaluate the importance of different components of AST paths, or *which components of AST paths contribute to their usefulness the most*. We experimented with several levels of abstraction:

- "No-arrows" - using the full path encoding, except the up and down symbols $\{\uparrow, \downarrow\}$.
- "Forget-order" - using paths without arrows and without order between the nodes: instead of treating a path as a *sequence* of nodes, treat it as a *bag* of nodes.
- "First-top-last" - keeping only the first, top and last nodes of the path. The "top" node refers to the node that is hierarchically the highest, from which the direction of the path changes from upwards to downwards.
- "First-last" - keeping only the first and last nodes.
- "Top" - keeping only the top node.
- "No-paths" - using no paths at all, and treating all relations between program elements as the same. The name of an element is predicted by using the bag of surrounding identifiers, without considering the syntactic relation to each of them.

All of the following experiments were performed using CRFs for variable names prediction, on the Java corpus and on the same hardware. In every experiment, the training corpus and the rest of the settings were identical. The number of training iterations was fixed.

Fig. 12 shows the accuracy of each abstraction compared to the consumed training time. As shown, as more information is kept, accuracy is increased, with the cost of a longer training time. An interesting "sweet-spot" is "first-top-last", which reduces training time by half compared to the full representation, with accuracy that is as 95% as good.

We also observe that the arrows and the order of the nodes in the path contribute about 1% accuracy.

6 Related Work

Naming in Software Engineering Several studies about naming in code have been conducted [6, 13, 21, 44]. Some of them applied neural network approaches for various applications. An approach for inferring method and class names was suggested by Allamanis et al. [6], by learning the similarity between names; however, they used features that were manually designed for a given task. A recent work presents a convolutional attention model [7] and evaluates it by predicting method names. In Section 5, we show that using our general approach yields better results.

Several works have studied the use of NLP techniques in programming languages, for applications such as estimating code similarity [46], naming convention recommendations [5], program synthesis [16], translation between programming languages [23] and code completion [20, 30, 41]. A

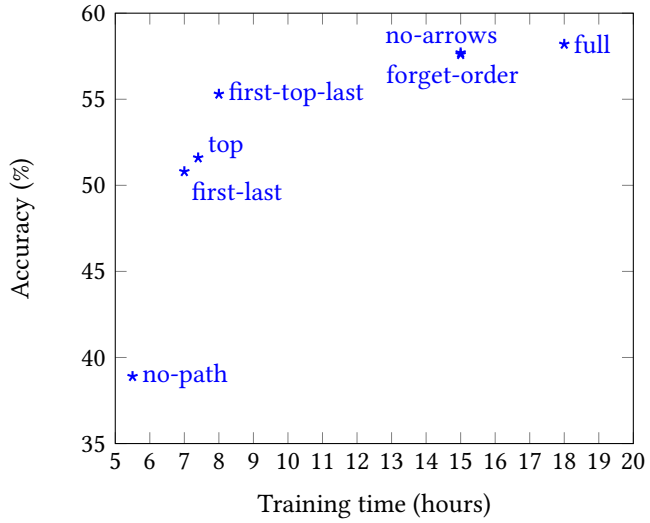


Figure 12. The accuracy of each abstraction method compared to the consumed training time, for the task of variable naming in Java

bimodal modeling of code and natural language was suggested by Allamanis et al. [9], for tasks of code retrieval given a natural language query, and generating a natural language description given a code snippet.

A recent work presented a model that uses LSTM for code summarization, which is interpreted as generating a natural language description for given program [22]. This work presents impressive results, but requires a very large amount of human-annotated data.

Predicting program properties using probabilistic graphical models CRFs have been used for structured prediction of program properties in JavaScript and Android Java code [11, 40]. The method has been shown to produce good results for prediction of names and types, by modeling programs with CRFs. Raychev et al. [40] defined relationships between program elements using an explicit grammar specific to JavaScript. The possible relationships span only a single statement, and do not include relationships that involve conditional statements or loops. Bichsel et al. [11] also defined several types of features and the conditions in which each of them is used. These works motivate a representation that is extractable automatically and can be applied to different languages. Instead of defining the relationships between program elements in advance, we learn them from the training data in an automatic process that is similar for different languages.

Parse Tree Paths An approach which resembles ours is Parse Tree Paths (PTPs) in Natural Language Processing. PTPs were mainly used in Semantic Role Labeling (SRL) — the NLP task of identifying semantic roles such as Message, Speaker or Topic in a natural language sentence. PTPs were first suggested by Gildea and Jurafsky [17] for automatic labeling of semantic roles, among other linguistic features. The paths were extracted from a target word to each of the constituents in question, and the method remains very popular in SRL and general NLP systems. The rich and unambiguous structure of programming languages renders AST paths even more important as a representation of program elements than PTPs in natural language.

Code completion using PCFGs Probabilistic Context Free Grammar (PCFG) for programming languages has been used for several tasks, such as finding code idioms [8]. PCFGs were generalized by Bielik et al. [12] by learning a relative context node on which each production rule depends, allowing conditioning of production rules beyond the parent nonterminal, thus capturing richer context for each production rule. Even though these works use paths in an AST, they differ from our work in that the path is only used to find a context node. In our work, the path itself is part of the representation, and therefore the prediction depends not only on the context node but also on the way it is related to the element in question.

Mou et al. [34] used a tree-based representation to learn snippets of code using a tree-convolutional neural network, for tasks of code category classification. Our representation differs from their mainly in that we decompose the AST into paths, which better capture data-flow properties, whereas their representation decomposes into sub-trees.

7 Conclusion

We presented a simple and general approach for learning from programs. The main idea is to represent a program using paths in its abstract syntax tree (AST). This allows a learning model to leverage the structured nature of source code rather than treating it as a flat sequence of tokens.

We show that this representation can be useful in a variety of programming languages and prediction tasks, and can improve the results of different learning algorithms without modifying the learning algorithm itself.

We believe that since the representation of programs using AST paths is fundamental to programming languages, it can be used in a variety of other machine learning tasks, including different applications and different learning models.

Acknowledgments

We would like to thank Eytan Singher for implementing the C# module of Pigeon. We also thank Miltiadis Allamanis, Veselin Raychev and Pavol Bielik for their guidance in the use of their tools in the evaluation section.

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 a Microsoft Azure for Research award and an AWS Cloud Credits for Research award.

References

[1] [n. d.]. JavaParser. http://javaparser.org.

[2] [n. d.]. Roslyn. https://github.com/dotnet/roslyn.

[3] [n. d.]. UglifyJS. https://github.com/mishoo/UglifyJS.

[4] [n. d.]. UnuglifyJS. https://github.com/eth-srl/UnuglifyJS.

[5] 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

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

[7] 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

[8] 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

[9] Miltiadis Allamanis, Daniel Tarlow, Andrew D. Gordon, and Yi Wei. 2015. Bimodal Modelling of Source Code and Natural Language. In *Proceedings of the 32nd International Conference on Machine Learning (JMLR Proceedings)*, Vol. 37. JMLR.org, 2123–2132.

[10] 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

[11] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. 2016. Statistical Deobfuscation of Android Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 343–355. https://doi.org/10.1145/2976749.2978422

[12] 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

[13] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. 2009. Relating Identifier Naming Flaws and Code Quality: An Empirical Study. In *2009 16th Working Conference on Reverse Engineering*. 31–35. https://doi.org/10.1109/WCRE.2009.50

[14] 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

[15] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural language processing (almost) from scratch. *Journal of Machine Learning Research* 12, Aug (2011), 2493–2537.

[16] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Sailesh R, and Subhajit Roy. 2016. Program Synthesis Using Natural Language. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 345–356. https://doi.org/10.1145/2884781.2884786

[17] Daniel Gildea and Daniel Jurafsky. 2002. Automatic Labeling of Semantic Roles. *Comput. Linguist.* 28, 3 (Sept. 2002), 245–288. https://doi.org/10.1162/089120102760275983

[18] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

[19] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2001. *The Elements of Statistical Learning*. Springer New York Inc., New York, NY, USA.

[20] 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

[21] 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

[22] 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

[23] Svetoslav Karaivanov, Veselin Raychev, and Martin Vechev. 2014. Phrase-Based Statistical Translation of Programming Languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. ACM, New York, NY, USA, 173–184. https://doi.org/10.1145/2661136.2661148

[24] D. Koller, N. Friedman, L. Getoor, and B. Taskar. 2007. Graphical Models in a Nutshell. In *Introduction to Statistical Relational Learning*, L. Getoor and B. Taskar (Eds.). MIT Press.

[25] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. 2001. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 282–289. http://dl.acm.org/citation.cfm?id=645530.655813

[26] Omer Levy and Yoav Goldberg. 2014. Dependency-Based Word Embeddings.. In *ACL (2)*. Citeseer, 302–308.

[27] Omer Levy and Yoav Goldberg. 2014. 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.

[28] Ben Liblit, Andrew Begel, and Eve Sweeser. 2006. Cognitive Perspectives on the Role of Naming in Computer Programs. In *Proceedings of the 18th Annual Psychology of Programming Workshop*. Psychology of Programming Interest Group, Sussex, United Kingdom.

[29] 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

[30] 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

[31] Oren Melamud, Omer Levy, and Ido Dagan. 2015. A Simple Word Embedding Model for Lexical Substitution. In *Proceedings of the 1st Workshop on Vector Space Modeling for Natural Language Processing, VS@NAACL-HLT 2015, June 5, 2015, Denver, Colorado, USA*. 1–7. http://aclweb.org/anthology/W/W15/W15-1501.pdf

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

[33] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. 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

[34] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI'16)*. AAAI Press, 1287–1293. http://dl.acm.org/citation.cfm?id=3015812.3016002

[35] Judea Pearl. 2011. Bayesian networks. (2011).

[36] Judea Pearl. 2014. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Elsevier.

[37] 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

[38] Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. 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

[39] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. 2016. 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

[40] 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

[41] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. *SIGPLAN Not.* 49, 6 (June 2014), 419–428. https://doi.org/10.1145/2666356.2594321

[42] Shai Shalev-Shwartz and Shai Ben-David. 2014. *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press, New York, NY, USA.

[43] Charles Sutton and Andrew McCallum. 2012. An Introduction to Conditional Random Fields. *Found. Trends Mach. Learn.* 4, 4 (April 2012), 267–373. https://doi.org/10.1561/2200000013

[44] Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. 1996. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Prog. Lang.* 4, 3 (1996), 143–167. http://compscinet.dcs.kcl.ac.uk/JP/jp040302.abs.html

[45] 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

[46] 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