Probabilistic Model for Code with Decision Trees

Veselin Raychev
Department of Computer Science
ETH Zürich, Switzerland
veselin.raychev@inf.ethz.ch

Pavol Bielik
Department of Computer Science
ETH Zürich, Switzerland
pavol.bielik@inf.ethz.ch

Martin Vechev
Department of Computer Science
ETH Zürich, Switzerland
martin.vechev@inf.ethz.ch

Abstract

In this paper we introduce a new approach for learning precise and general probabilistic models of code based on decision tree learning. Our approach directly benefits an emerging class of statistical programming tools which leverage probabilistic models of code learned over large codebases (e.g., GitHub) to make predictions about new programs (e.g., code completion, repair, etc).

The key idea is to phrase the problem of learning a probabilistic model of code as learning a decision tree in a domain specific language over abstract syntax trees (called TGEN). This allows us to condition the prediction of a program element on a dynamically computed context. Further, our problem formulation enables us to easily instantiate known decision tree learning algorithms such as ID3, but also to obtain new variants we refer to as ID3+ and E13, not previously explored and ones that outperform ID3 in prediction accuracy.

Our approach is general and can be used to learn a probabilistic model of any programming language. We implemented our approach in a system called DEEP3 and evaluated it for the challenging task of learning probabilistic models of JavaScript and Python. Our experimental results indicate that DEEP3 predicts elements of JavaScript and Python code with precision above 82% and 69%, respectively. Further, DEEP3 often significantly outperforms state-of-the-art approaches in overall prediction accuracy.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Statistical methods; I.2.2 [*Artificial Intelligence*]: Program synthesis—Automatic Programming

Keywords Probabilistic Models of Code, Decision Trees, Code Completion

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4444-9/16/11...$15.00
http://dx.doi.org/10.1145/2983990.2984041

1. Introduction

Recent years have seen an increased interest in programming tools based on probabilistic models of code built from large codebases (e.g., GitHub repositories). The goal of these tools is to automate certain programming tasks by learning from the large effort already spent in building a massive collection of existing programs. Example applications targeted by such tools include programming language translation [9, 15], patch generation [18], probabilistic type inference [26], and code completion [7, 10, 21, 25, 27]. However, the probabilistic models used by existing tools are either: (i) carefully crafted with specific features tailored to a particular task [18, 25, 26]; this requires careful manual effort that is difficult to reuse for other tasks, or (ii) the models are general but may be imprecise (e.g., n-gram, PCFGs, [7], [27]).

A fundamental challenge then is coming up with a probabilistic model of code which is both general and precise. However, building such model is a difficult problem and one which has recently attracted the attention of researchers from several areas. Latest attempts to addressing this problem range from using n-gram models [11, 22, 34] to probabilistic grammars [2, 9, 17, 19] to log-bilinear models [3, 4]. All of these however make predictions about program elements based on a *fixed set* of relatively *shallow* features (e.g., the n code tokens that precede the prediction). Unfortunately, such features are often a poor choice because they blindly capture only dependencies that are syntactically local to the element to be predicted by the model. Perhaps more importantly, a key limitation of these techniques is that they use the same features for a very diverse set of predictions: API calls, identifiers, constants, or even expressions. As a result, while general, these models tend to suffer from low precision limiting their practical applicability.

The need for different contexts: an example To motivate our approach and obtain a better intuition, consider the four JavaScript code examples shown in Fig. 1. The goal for each of these examples is to predict the most likely HTTP header property that should be set when performing a HTTP request (marked as ?). An important observation is that even though all four examples predict a property name, the best features needed to make the correct prediction are very different in

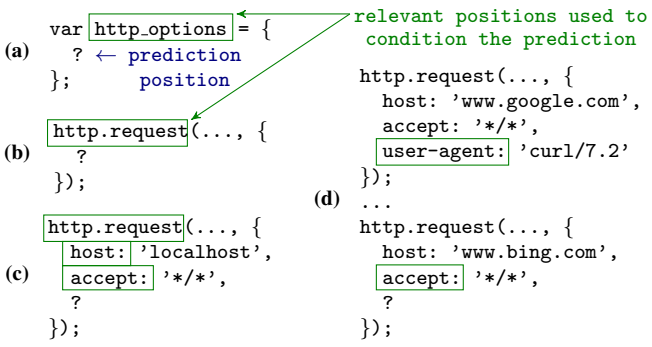


Figure 1. JavaScript code snippets used to motivate the need for a probabilistic model with *dynamically* computed context based on the input. Although each example predicts a property name used for an HTTP request, the best context used to condition the prediction (shown as rectangles) is different for each input.

each example (marked as green boxes). In Fig. 1 (a) and (b), the only available information is the variable name to which the dictionary object is assigned, and the fact that it is used in the `http.request` API, respectively. In Fig. 1 (c), a prediction can be based on the previous two properties that were already set, as well as on the fact that the dictionary is used as a second argument in the `http.request` API. In Fig. 1 (d), another similar call to `http.request` is present in the same program and a probabilistic model may leverage that information. Intuitively, the best features depend on the *context* in which the completion is performed. Thus, ideally, we would like our probabilistic model to automatically discover the relevant features/context for each case.

This work We present a new approach for learning accurate probabilistic models of code which automatically determine the right context when making a prediction. The key technical insight is to (recursively) split the training data in a fashion similar to decision trees [20, §3] and to then learn smaller specialized probabilistic models for each branch of the tree. Our approach is phrased as learning a program p in a domain specific language for ASTs (abstract syntax trees) that we introduce (called TGEN) and then using p and the training data to obtain a probabilistic model. We designed TGEN to be independent of the specific programming language which means that our approach can be used for building a probabilistic model of any programming language.

The learned decomposition enables us to *automatically* produce a set of *deep* features dynamically selected when executing the program p on the given input. As a result, as shown later in the paper, our probabilistic model enables higher prediction accuracy than the state of the art. Importantly, our formulation allows us to cleanly represent (and experiment) with classic decision tree learning algorithms such as ID3 [23] but also permits new, previously unexplored extensions which lead to better precision than ID3.

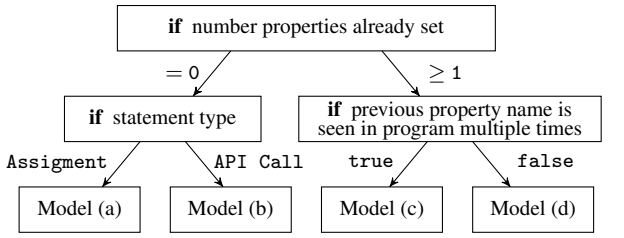


Figure 2. An example of a decision tree learned by our approach. When processed by this tree, each of the inputs in Fig. 1 will end up in a different leaf and thus a different model will be used to answer the queries of each example.

The closest work related to ours is `DeepSyn` [27], which also proposes to use a domain specific language (DSL) for finding the correct context of its predictions. Their DSL, however, does not include branches and therefore `DeepSyn` cannot learn to predict all cases of Fig. 1 simultaneously. This limitation cannot be fixed easily, because simply adding branches to the DSL of `DeepSyn` leads to an exponentially harder synthesis task that is untractable by the existing synthesis procedures – a problem that we address in this paper.

Tree decomposition: an example Our approach can learn a decision tree such as the one shown in Fig. 2 directly from the training data. This tree represents a program p from our TGEN language (discussed later in the paper). Here, the tree contains several tests that check properties of the input code snippet (e.g., whether some properties have been set) in order to select the suitable probabilistic model used for making the prediction for that specific input. In this example, each of the four inputs from Fig. 1 will end up with a different probabilistic model for making the prediction.

Main contributions Our main contributions are:

- A new approach for building probabilistic models of code based on learning decision trees represented as programs in a DSL called TGEN. The DSL is agnostic to the language for which the model is built allowing applicability of the approach to any programming language.
- An instantiation of our approach to several decision tree algorithms, including classic learning such as ID3, but also new variants which combine decision trees with other powerful models.
- A complete and scalable implementation of our model in a system called DEEP3, applied to JavaScript and Python.
- An extensive experimental evaluation of DEEP3 on the challenging tasks of learning probabilistic models of JavaScript and Python. Our experimental results show that our models are significantly more precise than existing work: for JavaScript, the model achieves 82.9% accuracy when predicting arbitrary values used in JavaScript programs and 83.9% in predicting code structure.

Benefits Our approach as well as our implementation can be directly used to learn a probabilistic model of any programming language. Despite its generality, the precision of the model is higher than prior work. For instance, the model can make predictions about program structure such as loops and branches with more than 65% accuracy whereas all evaluated prior models have accuracy of less than 41%. As a result of its generality and higher precision, we believe that DEEP3 can immediately benefit existing works by replacing their probabilistic model [1, 3, 4, 9, 11, 15]. It can also be used as a building block for future tools that rely on a probabilistic model of code (e.g., code repairs, synthesis, etc).

Outline In the remainder of the paper, we first present our general approach based on decision tree learning. We then discuss the domain-specific language TGEN and how to obtain a probabilistic model for that language, followed by a description of the implementation of DEEP3. Our experimental section describes a comprehensive evaluation of DEEP3 on the hard challenge of learning a precise model of JavaScript programs.

2. Learning Decision Trees for Programs

In this section we present our approach for learning probabilistic models of programs. Our probabilistic models are based on generalizing classic decision trees [20] and combining these with other probabilistic models. A key observation of our work is that we can describe a probabilistic model (decision trees, combinations with other models) via a program p in a domain specific language (DSL). This enables us to cleanly instantiate existing decision tree learning algorithms as well as obtain interesting new variants. These variants have not been explored previously yet turn out to be practically useful. To our knowledge, this is the first work to use decision trees in the context of learning programs.

In what follows, we first discuss how to obtain our training data set. We then discuss the general requirements on the DSL so that it is able to represent decision trees and combinations with other models. Finally, we show how to learn different probabilistic models by instantiating the program p in a fragment of DSL. In Section 3 we provide a concrete instantiation of a DSL suitable for learning programs.

2.1 Training Data

Our training data is built by first taking the set of available programs and generating a set of pairs where each pair is a tuple of a partial program (part of the original program) and an element in the original program that we would like the model to predict. More formally, our training dataset is $\mathcal{D} = \{(\boldsymbol{x}^{(j)}, y^{(j)})\}_{j=1}^{n}$ of n samples where $\boldsymbol{x}^{(j)} \in X$ are inputs (partial programs) and $y^{(j)} \in Y$ are outputs (correct predictions for the partial programs). Each possible output is called a label and Y is the set of possible labels. This way of obtaining a data set reflects a key goal of our probabilistic model: it should be able to precisely predict program elements (e.g., statements, expressions, etc) and be useful when making predictions on new, unseen programs in a variety of scenarios (e.g., statistical code synthesis, repair).

An example of a data set is shown in Fig. 3 (a). Our goal is to obtain a probabilistic model $Pr(y \mid \boldsymbol{x})$ that predicts outputs given new inputs. We can then use the Pr model to perform various tasks, for instance, statistical code synthesis can be performed by computing:

$$y' = \arg\max_{y \in Y} Pr(y \mid \boldsymbol{x}')$$

where $\boldsymbol{x}'$ is a partial code fragment written by the user (and possibly not seen in the training data).

2.2 Representing Decision Trees with a DSL

To represent decision trees with a DSL, we require the DSL to include branch statements of the following shape:

$$\textbf{if } (pred(\boldsymbol{x})) \textbf{ then } p_a \textbf{ else } p_b$$

where $pred$ is a predicate on the input $\boldsymbol{x}$ and $p_a, p_b \in$ DSL. Here, p_a and p_b may be branch instructions or other programs in DSL. The semantics of the instruction is standard: check the predicate $pred(\boldsymbol{x})$ and depending on the outcome either execute p_a or p_b.

Example Consider the decision tree in Fig. 3 (b) corresponding to the following program:

$$p \equiv \textbf{if } (\texttt{query } \boldsymbol{x} \texttt{ is console.}) \textbf{ then } \epsilon_{100} \textbf{ else } \epsilon_{101}$$

Here $\epsilon_{100}, \epsilon_{101} \in$ DSL are leaves of the tree and the root checks if the query $\boldsymbol{x}$ is on the `console` object. According to this tree, our six training samples from Fig. 3 (a) are split such that the first two fall in ϵ_{100} and the rest in ϵ_{101}.

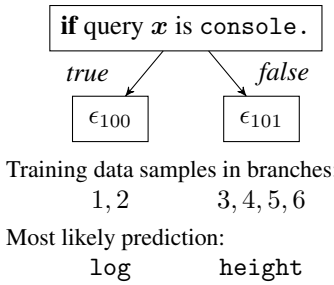
Now, assume that we would like to use this decision tree to make predictions depending on the branch an input $\boldsymbol{x}$ falls into. Given our training data, we observe that all outputs falling in ϵ_{100} are `log`. Thus, we can learn that ϵ_{100} should produce label `log`. The remaining outputs fall into the other branch and are more diverse. Their labels can be either `height`, `left` or `top`. To predict these labels, we either need to refine the tree or keep the tree as is and use a probabilistic model not perfectly confident in its prediction: for example, since half of the samples have label `height`, we can guess with confidence $1/2$ that the label should be `height`. To formalize these probabilistic estimates, we include the empty program in our DSL.

Empty programs We say a program $\epsilon_i \in$ DSL is *empty* if this program encodes an unconditional probabilistic model. A possible way to obtain an unconditional probability estimate is to use maximum likelihood estimation (MLE):

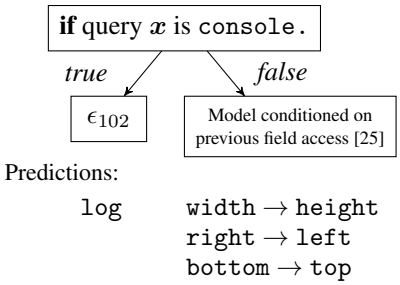
$$Pr(y \mid \boldsymbol{x}) = \frac{|\{(\boldsymbol{x}^{(j)}, y^{(j)}) \in d \mid y^{(j)} = y\}|}{|d|}$$

j	$\boldsymbol{x}^{(j)}$	$y^{(j)}$
1	`console.`	`log`
2	`console.`	`log`
3	`x.width * x.`	`height`
4	`y.width * y.`	`height`
5	`a.right - a.`	`left`
6	`b.bottom - b.`	`top`

(a) Training data



(b) Empty programs in leaves



(c) Model based on a previous field access on the same object in the right leaf.

Figure 3. Example training data for field name prediction and two possible probabilistic models visualized as trees.

Here, the dataset $d \subseteq \mathcal{D}$ includes the samples from $\mathcal{D}$ that reached the empty program. That is, we count the number of times we see label y in the data and divide it by the total number of samples [1]. For Fig. 3 (b), the model of program ϵ_{100} results in probability 1 for label `log` and probability 0 for any other label. The model of program ϵ_{101} assigns probability $1/2$ to `height` and $1/4$ to `left` and `top`. This estimate is called unconditional because the input $\boldsymbol{x}$ is not used when computing $Pr(y \mid \boldsymbol{x})$.

Note that despite the fact that the two programs in the leaves of the tree in Fig. 3 (b) are empty, they encode two different models for each branch. Technically, to achieve this we require that empty programs are uniquely identified – either by the branch taken in branch statements or by an index of the empty program (100 and 101 in our case).

2.3 Representing a Combination of Decisions Trees and Other Models in a DSL

Our DSL approach is not limited to decision trees. In practice, using decision trees for some tasks leads to very large trees and overfitting [20, §3.7] and thus we allow other probabilistic models in the leaves of our trees instead of only empty programs. In Fig. 3 (c) we show the same tree as in Fig. 3 (b) except that in the right subtree we use a probabilistic model from [25] that predicts the name of a field based on the name of the previous field access on the same variable. Using this more advanced model, we learn the mapping shown in the lower-right corner of Fig. 3 (c). The resulting probabilistic model is as follows: if the previous field access on the same variable was `width`, then $Pr(y = \texttt{height} \mid \boldsymbol{x}) = 1$. If the previous field access on the same variable was `right`, then $Pr(y = \texttt{left} \mid \boldsymbol{x}) = 1$, etc. This means that for the input "`z.right - z.`", the predicted field name by our model will be `left` with probability 1 (and not `height`, as the unconditional model would predict).

Discussion Here, we make two additional observations. First, we can use a deeper decision tree instead of the model in Fig. 3 (c) and still achieve similar predictions for our toy example dataset. This deeper tree, however, would be impractically large for realistic probabilistic models. Second, adding decision trees to a model such as the one from [25] increases the power of the resulting model. For example, the model from [25] needs a previous API access to make conditioning, but our model in Fig. 3 (c) can make correct predictions based on the name of the `console` variable even in the absence of other field accesses.

So far we showed how to obtain a probabilistic model $Pr_p(y \mid \boldsymbol{x})$ from a program $p \in \texttt{DSL}$ and a training dataset $\mathcal{D}$. Next, we discuss how the program p is learned from $\mathcal{D}$.

2.4 Learning Programs in DSL from Data

Our goal at learning time is to discover $p \in \texttt{DSL}$ that "explains well" the training data $\mathcal{D}$. One possible and commonly used metric to measure this is the entropy of the resulting probability distribution, estimated as follows:

$$H(\mathcal{D}, p) = -\sum_{(\boldsymbol{x}^{(j)}, y^{(j)}) \in \mathcal{D}} \frac{1}{|\mathcal{D}|} \log_2 Pr_p(y^{(j)} \mid \boldsymbol{x}^{(j)})$$

We use a variant of the entropy metric called cross-entropy where the training data is split in two parts. The first part is used to build the probability distribution Pr_p parametrized on p and the second part to calculate the entropy measure. Then our training goal is to find a program $p \in \texttt{DSL}$ that minimizes $H(\mathcal{D}, p)$. Because the space of programs is very large we use approximations in our minimization procedure.

Our greedy learning algorithm is shown in Algorithm 1. A useful benefit of our approach is that we can instantiate different decision tree learning algorithms by simply varying the fragment of the DSL to which the learned program p belongs. In this way we instantiate existing learning algorithms such as ID3 [23] but also new and interesting variants which have not been explored before.

ID3 decision tree learning ID3 is one of the most commonly used and studied decision tree algorithms [20, §3]. We instantiate ID3 learning as follows. Let $\texttt{DSL}_0$ be a fragment of DSL with programs in the following shape:

$$\textbf{if } (pred(\boldsymbol{x})) \textbf{ then } \epsilon_a \textbf{ else } \epsilon_b$$

[1] In practice, we use a slightly modified equation with smoothing [20, §6.9.1.1] to give non-zero probability to labels outside of the training data.

```
1  def Learn(d, syn)
       Input: Dataset d, local synthesis procedure syn
       Output: Program p ∈ DSL
2      begin
3          if done(d) then
4              return ε_i // generate unique i
5          p ← syn(d)
6          where p ≡ if (pred(x)) then p_a else p_b
7          p_a ← Learn({(x, y) ∈ d | pred(x)}, syn)
8          p_b ← Learn({(x, y) ∈ d | ¬pred(x)}, syn)
9          return p
10     end
```

Algorithm 1: Decision Tree Learning Algorithm.

where ϵ_a and ϵ_b are empty programs. Then, for a dataset d we define $syn_0(d) = \arg\min_{p \in \texttt{DSL}_0} H(d, p)$. We obtain the ID3 learning algorithm by invoking Algorithm 1 with $syn = syn_0$. Then, at each step of the algorithm we synthesize a single branch using syn_0, split the data according to the branch and call the algorithm recursively. As a termination condition we use a function $done(d)$ which in our implementation stops once the dataset is smaller than a certain size. This limits the depth of the tree and prevents overfitting to the training data.

We note that the original formulation of ID3 found in [20, 23] may at first sight look quite different from ours. A reason their formulation is more complicated is because they do not use a DSL with empty programs like we do. Then, the metric they maximize is the *information gain* of adding a branch. The information gain of a program $p \in \texttt{DSL}_0$ for a dataset d is $IG = H(d, p) - H(d, \epsilon)$. Since $H(d, \epsilon)$ is independent of p, our formulation is equivalent to the original ID3.

2.5 Extensions of ID3

Our formulation of ID3 as learning a program in a DSL fragment allows us to extend and improve the algorithm by making modifications of the DSL fragment. We also provide extensions that leverage the capability of the DSL to express probabilistic models other than standard decision trees.

ID3+ decision tree learning Classic ID3 learning results in decision trees (i.e., programs in DSL) with the leaves of the tree being empty programs. Our models, however, allow combining branch instructions with other probabilistic models like in Fig. 3 (c). To handle such combinations, we extend the ID3 algorithm to also generate programs in the leaves of the trees.

Let $\texttt{DSL}_S$ be a fragment of DSL. Programs in this fragment may describe probabilistic models without branches such as language models [1, 29] or other models (e.g., [7, 27]). Let $syn_S(d) = \arg\min_{p \in \texttt{DSL}_S} H(d, p)$ be a synthesis procedure which computes a program that best fits d. Then, the ID3+ learning algorithm extends ID3 by replacing the empty programs returned on Line 4 of Algorithm 1 with $syn_S(d)$. The

```
     TGen ::= SimpleCond | BranchCond
SimpleCond ::= ϵ | WriteOp SimpleCond | MoveOp SimpleCond
   WriteOp ::= WriteValue | WritePos | WriteType
    MoveOp ::= Up | Left | Right | DownFirst | DownLast |
               NextDFS | NextLeaf | PrevDFS | PrevLeaf |
               PrevNodeType | PrevNodeValue
               PrevNodeContext
BranchCond ::= Switch(SimpleCond) {
                   case v_1: TGen| ··· | case v_n: TGen
                   default: TGen
               }
```

Figure 4. The TGen language for extracting structured context from trees with `switch` statements.

branch synthesis of the ID3+ algorithm uses the same syn_0 procedure as ID3 and as a result constructs a tree with the same structure. The only difference in the resulting tree is the programs in the leaves of the tree.

E13 algorithm The ID3+ learning algorithm essentially first runs ID3 to generate branches until too few samples fall into a branch and then creates a probabilistic model for the final samples. Intuitively, there are two potential issues with this approach. First, ID3+ always trains the probabilistic models based on $\texttt{DSL}_S$ only on small datasets. These small datasets may result in learning inaccurate models at the leaves. Second, syn_0 always minimizes information with respect to empty programs, but leaves of the tree may contain non-empty programs.

To address these limitations, we propose to instantiate the learning algorithm so that instead of using syn_0 as ID3 does, it uses the following procedure:

- First, let $syn_S(d) \in \texttt{DSL}$ be the best program that we would synthesize for d if that program was a leaf node in the ID3+ algorithm.
- Let $\texttt{DSL}_A$ be a fragment of DSL with programs in the shape $p \equiv$ **if** $(pred(\boldsymbol{x}))$ **then** p_a **else** p_b where $pred$ is a predicate and p_a and p_b are either the empty program or $syn_S(d)$. This is, we consider that not only empty programs may stay in the branches, but we take into account that the program $syn_S(d)$ may be there.
- Finally, to obtain our E13 algorithm, we set syn to $syn_A = \arg\min_{p \in \texttt{DSL}_A} H(d, p)$ in Algorithm 1.

Summary In this section we showed how to use a decision tree and a dataset to obtain a probability distribution. We also showed how by varying the DSL over which we learn our program, we can instantiate different variants of ID3 learning and (combinations of) probabilistic models. In the next section, we discuss a specific instance of a DSL suitable for learning probabilistic models of programs.

$$t \in Tree \qquad n \in X \qquad ctx \in C \qquad s \in \texttt{TGen}$$

$$\frac{op \in \texttt{MoveOp} \qquad n' = mv(op, t, n)}{\langle op :: s, t, n, ctx\rangle \rightarrow \langle s, t, n', ctx\rangle} \text{ [Move]}$$

$$\frac{op \in \texttt{WriteOp} \qquad c = wr(op, t, n)}{\langle op :: s, t, n, ctx\rangle \rightarrow \langle s, t, n, ctx \cdot c\rangle} \text{ [Write]}$$

$$\frac{op \in \texttt{Switch} \qquad \langle op.cond, t, n, []\rangle \rightarrow \langle \epsilon, t', n', ctx_{cond}\rangle \qquad ctx_{cond} \in op.cases}{\langle op, t, n, ctx\rangle \rightarrow \langle op.cases[ctx_{cond}], t, n, ctx \cdot op.case\_id(ctx_{cond})\rangle} \text{ [Switch]}$$

$$\frac{op \in \texttt{Switch} \qquad \langle op.cond, t, n, []\rangle \rightarrow \langle \epsilon, t', n', ctx_{cond}\rangle \qquad ctx_{cond} \notin op.cases}{\langle op, t, n, ctx\rangle \rightarrow \langle op.default, t, n, ctx \cdot op.case\_id(\bot)\rangle} \text{ [Switch-Def]}$$

Figure 5. TGen language small-step semantics. Each rule is of the type: TGen $\times$ $States \rightarrow$ TGen $\times$ $States$.

3. TGen: a DSL for ASTs

In this section we provide a definition of our domain specific language (DSL), called TGen, which will be used to learn programs that define a probabilistic model of code (as discussed earlier). This language contains operations which traverse abstract syntax trees (ASTs) where the operations are independent of the actual programming language for which the probabilistic model is built. This means that TGen is generally applicable to building probabilistic models for any programming language. Next, we discuss the syntax and semantics of TGen.

Syntax The syntax of TGen is summarized in Fig. 4 and consists of two kinds of instructions – `BranchCond` instructions execute a subprogram depending on a checked condition and `SimpleCond` programs that encode straight line tree traversal programs and consist of two basic types of instructions: `MoveOp` and `WriteOp`. Move instructions facilitate the tree traversal by moving the current position in the tree, while write instructions append facts about the currently visited node to the accumulated context.

Semantics TGen programs operate on a state σ defined as $\sigma = \langle t, n, ctx\rangle \in States$ where $States = Tree \times X \times C$. In a state σ, t is a tree, n is the current position in the tree and ctx is the currently accumulated context. The accumulated context $ctx \in C = (N \cup \Sigma \cup \mathbb{N})^*$ by a TGen program is a sequence of observations on the tree where each observation can be a non-terminal symbol N from the tree, a terminal symbol Σ from the tree or a natural number in $\mathbb{N}$. Initially, execution starts with the empty observation list $[] \in C$ and instructions from the program are executed. We show the small-step semantics of TGen in Fig. 5.

For a program $p \in$ TGen, a tree $t \in Tree$, and node $n \in X$, we say that program $p \in$ TGen computes the context $ctx = p(t, n)$ iff there exists a sequence of transitions from $\langle p, t, n, []\rangle$ to $\langle \epsilon, t, n', ctx\rangle$. That is, ctx is the accumulated context obtained by executing the program p on a tree t starting at node n. That tuple (t, n) of an AST and a prediction position is the input $\boldsymbol{x}$ to our probabilistic models.

3.1 Semantics of Basic Instructions

The basic instructions `MoveOp` and `WriteOp` are used to build straight line programs that traverse a tree and accumulate context.

The semantics of the write instructions are described by the [Write] rule in Fig. 5. Each write accumulates a value c to the conditioning set ctx as defined by the function $c = wr(op, t, n)$ where wr: `WriteOp` $\times Tree \times X \rightarrow N \cup \Sigma \cup \mathbb{N}$. The returned value of wr is defined as follows:

- $wr(\texttt{WriteType}, t, n) = x$ where $x \in N$ is the non-terminal symbol at node n.
- $wr(\texttt{WriteValue}, t, n)$ returns the terminal symbol at node n if one is available or a special value 0 otherwise, and
- $wr(\texttt{WritePos}, t, n)$ returns a number $x \in \mathbb{N}$ that is the index of n in the list of children kept by the parent of n.

Move instructions are described by the [Move] rule in Fig. 5 and use the function mv: `MoveOp` $\times Tree \times X \rightarrow X$. The function mv is defined as follows:

- $mv(\texttt{Up}, t, n) = n'$ where n' is the parent node of n in t or n if n has no parent node in t. Note that the [Move] rule updates the node at the current position to be the parent.
- $mv(\texttt{Left}, t, n) = n'$ where n' is the left sibling of n in t. Similarly, $mv(\texttt{Right}, t, n)$ produces the right sibling.
- $mv(\texttt{DownFirst}, t, n) = n'$ where n' is the first child of n in t. Similarly, $mv(\texttt{DownLast}, t, n)$ produces the last child of n.
- $mv(\texttt{PrevDFS}, t, n) = n'$ where n' is the predecessor of n in t in the left-to-right depth-first search traversal order. Similarly, $mv(\texttt{NextDFS}, t, n)$ returns the successor of n in the left-to-right depth-first search traversal order.
- $mv(\texttt{PrevLeft}, t, n) = n'$ where n' is the first leaf on the left of n. Similarly, $mv(\texttt{NextLeft}, t, n)$ returns the first leaf on the right of n.
- $mv(\texttt{PrevNodeValue}, t, n) = n'$ where n' is the first node on the left of n that has the same non-terminal symbol as n. Similarly, $mv(\texttt{PrevNodeType}, t, n)$ returns the first node on the left of n that has the same terminal symbol as n.
- $mv(\texttt{PrevNodeContext}, t, n) = n'$ where n' is the first node on the left of n that has *both* the same terminal and non-terminal symbol as n, and the parent of n has the same non-terminal symbol as the parent of n'.

j	$\boldsymbol{x}^{(j)}$	$y^{(j)}$
1	`console.`	`log`
2	`console.`	`log`
3	`x.width * x.`	`height`
4	`y.width * y.`	`height`
5	`a.right - a.`	`left`
6	`b.bottom - b.`	`top`

(a) Training data

$p(\boldsymbol{x}^{(j)})$	$Pr(y = \texttt{height} \mid p(\boldsymbol{x}^{(j)}))$
100	0
100	0
101	0.5
101	0.5
101	0.5
101	0.5

$p \equiv$ `Switch(Left WriteValue) {`
`case console` $: \epsilon$ `default` $: \epsilon$`}`

(b) Probabilistic model from Fig. 3 (b)

$p(\boldsymbol{x}^{(j)})$	$Pr(y = \texttt{height} \mid p(\boldsymbol{x}^{(j)}))$
100	0
100	0
101 `width`	1
101 `width`	1
101 `right`	0
101 `bottom`	0

$p \equiv$ `Switch(Left WriteValue) {case console` $: \epsilon$
`default : Left PrevNodeValue Right WriteValue}`

(c) Probabilistic model from Fig. 3 (c)

Figure 6. Probabilistic models for the examples from Fig. 3 along with their TGEN programs.

3.2 Semantics of Switch

The `Switch` instruction provides a means of branching based on observed values in the input tree t. The way branching works is as follows: first, the program of basic instructions (denoted $op.cond$ for $op \in$ `Switch`) in the condition of the switch statement is executed, which accumulates a conditioning context ctx_{cond}. Then, for different values of ctx_{cond}, different branches are taken. Currently, the TGEN language only allows comparing the produced ctx_{cond} to constants given in the program.

The semantics of a switch instruction given by the rules [SWITCH] and [SWITCH-DEF] in Fig. 5 describe the case when a program takes some of the branches given by one `case` or the `default` branch, respectively. For an $op \in$ `Switch`, we use $op.cases[ctx_{cond}]$ to denote the program given by the case where $\boldsymbol{v}_i = ctx_{cond}$. We also assign a unique index to each subprogram in a `Switch` statement that we refer to with $op.case\_id(ctx_{cond}) = i$ if $\boldsymbol{v}_i = ctx_{cond}$, otherwise it is $\perp$.

The current context is updated by appending the index of the taken branch to it. This allows us to distinguish which branch a program has taken even if the programs in the branches end up as empty programs.

We note that our switch statement allows for more than two branches coming out of one statement and is slightly more general than traditional decision trees which typically have at most two branches (e.g., if-then-else).

Summary In this section we presented a domain-specific language for traversing ASTs, called TGEN, that is used (as will be discussed in Section 4) to define a probabilistic model for code. Similar to the DSL used in `DeepSyn` [27], the idea of TGEN is to provide means of navigating over trees and accumulating context with values from a given tree. There are two notable differences between the DSLs. First, our TGEN language does not include any instruction that depends on manually-provided analysis of the programming language that is being learned, whereas the language of `DeepSyn` relies on lightweight static analysis for JavaScript. Second, the TGEN language includes the `Switch` instruction which allows us to express programs with branches and enables learning more powerful probabilistic models that are specialized by the *type* and the particular *context* in which the prediction is performed (as illustrated in Section 1).

4. From TGEN to a Probabilistic Model

Earlier, we had discussed how a program in `DSL` determines a probabilistic model. In this section we concretize this discussion further for the TGEN language, provide examples and also describe an extension to the model which allows it to predict labels not seen in the training data.

4.1 Probabilistic Models with TGEN

Using the program $p \in$ TGEN and a training dataset of examples $\mathcal{D} = \{(\boldsymbol{x}^{(j)}, y^{(j)})\}_{j=1}^{n}$, we can now build a probabilistic model of the outputs y in the data given the inputs $\boldsymbol{x}$. The model $Pr(y \mid \boldsymbol{x})$, equal to $Pr(y \mid p(\boldsymbol{x}))$, is built using maximum likelihood estimation (MLE) as follows:

$$Pr(y \mid p(\boldsymbol{x})) = \frac{|\{(\boldsymbol{x}', y) \in \mathcal{D} \mid p(\boldsymbol{x}') = p(\boldsymbol{x})\}|}{|\{(\boldsymbol{x}', y') \in \mathcal{D} \mid p(\boldsymbol{x}') = p(\boldsymbol{x})\}|}$$

This is, to estimate the probability of observing y given $\boldsymbol{x}$ we remember the cases in the training data where $p(\boldsymbol{x}^{(j)}) = p(\boldsymbol{x})$. From these cases, we count how many times the value of the label $y^{(j)}$ was y and divide it to the number of times where $y^{(j)}$ was any arbitrary y'.

Example Next, we give examples of the probabilistic models in Fig. 3 as described by TGEN programs. The model from Fig. 3 (b) is described by the following program p:

`Switch(Left WriteValue) {case console` $: \epsilon$ `default` $: \epsilon$`}`

The program moves to the left of the prediction position and records the syntactic value at that position. The recorded value is then compared to `console` and the corresponding branch is taken (the branches are indexed 100 and 101, respectively). Consider again the training data $\mathcal{D}$ shown in Fig. 6 (a) (repeated here for convenience). In the first column of Fig. 6 (b) we show the resulting context produced by

the above program p when executing it on each of the six data points $\boldsymbol{x}^{(j)}$. That is, the program accumulates a context which is simply the sequence of taken branches for that input. We show the probability of a label $y =$ `height` given each of the code fragments. If the input $\boldsymbol{x}^{(j)}$ is "`console.`", the computed context is 100 and because there were no cases in the training data $\mathcal{D}$ with context 100 and label `height`, the resulting probability is 0. In the other branch, half of the training data (2 out of 4 samples with context 101) have label `height` and thus the probability estimates are 0.5. These results are summarized in the second column of Fig. 6 (b).

Note how the model in Fig. 6 (b) assigns probability of 0.5 to label `height` for training data samples $3, 4, 5, 6$. This is because the switch statement only records the target branch (in this case 101) but does not keep any history of how it got there (e.g., the actual condition that led to the 101 branch being taken). To address this issue and make the model more precise, we have two choices: (i) either adjust the semantics of the switch statement to record the condition in the context, or (ii) keep the semantics as is, but change the program so that it records more context. In our work, we chose the second option as it allows for greater flexibility. This option is illustrated below on our running example.

Here, the model in Fig. 6 (c) solves the above problem by accumulating context in the case where the call is not on the `console` object. This is achieved via the following `SimpleCond` program:

```
Left PrevNodeValue Right WriteValue
```

This program moves left of the prediction position in the AST (where the variable is, e.g., `x` in sample 3), then moves to the previous use of the same variable (using `PrevNodeValue`), then moves right and records the previous field name used on the same variable. The accumulated contexts using that program and the corresponding probability of label `height` appearing given the context are shown in Fig. 6 (c). This program is an example of combining a `Switch` statement with a straight line program. Note how the `SimpleCond` program encodes the language model from [25] and how the model naturally integrates with the decision tree via the variable sized context computed by p.

In summary, we can see that by simply varying the program in our DSL, we automatically vary the probabilistic model. This is useful as it allows us to quickly build and experiment with powerful probabilistic models that learn from the training data.

4.2 Extension: Predicting Out-of-Vocabulary Labels

A limitation of the probabilistic models described by TGEN programs is that they cannot predict values (i.e., APIs, variable names, etc.) not seen in the training data. We offer a simple mitigation for this limitation by introducing a second TGEN program p_2 that given an input program $\boldsymbol{x}$ whose element y we would like to predict, modifies the probabilistic model so the model can express equality of the output label y to a value already present in $\boldsymbol{x}$ as described below.

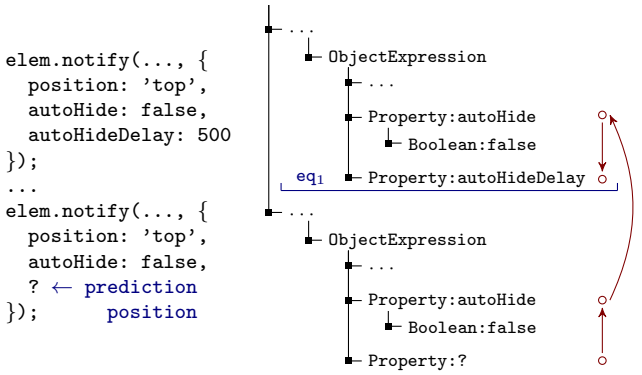


Figure 7. Illustration of relating the predicted label to a value already present in the program.

In practice, we synthesize both p and p_2 together with the same decision tree algorithm and the two programs follow the same branches. Only once we synthesize a program with the syn_S procedure, we synthesize different instructions for the programs p and p_2.

Training First, we discuss the modified training procedure on a training data sample $(\boldsymbol{x}, y)$. Let $ctx' = c_1 c_2 \cdots c_n = p_2(\boldsymbol{x})$ be the context computed by the program p_2. Recall that y is the label to be predicted for the input $\boldsymbol{x}$. We define:

$$y' = \begin{cases} eq_i & \text{if } \exists i. c_i = x \quad \text{(select } i \text{ to be minimal)} \\ y & \text{otherwise} \end{cases}$$

In other words, if any of the accumulated values is equal to the label y we are predicting, we replace the label with a special symbol eq_i and train the probabilistic model as before with y' instead of y.

Prediction If at prediction time for some program $\boldsymbol{x'}$ our probabilistic model returns the special value eq_i, we accumulate a context $ctx' = c_1 c_2 \cdots c_n = p_2(\boldsymbol{x'})$ and then replace the predicted label eq_i with c_i from the program $\boldsymbol{x'}$.

Example Consider the two code snippets shown in Fig. 1 (d) and Fig. 7 that both try to predict the name of next property that the developer should set. In Fig. 7, the value to predict y is a property name `autoHideDelay` that is present in the query program $\boldsymbol{x}$ at another location. Then, if we execute the following TGEN program:

$$p_2 \equiv \texttt{Left PrevNodeContext Right WriteValue}$$

we will obtain context $ctx' =$ `autoHideDelay`. The execution of p_2 is sketched with arrows in Fig. 7.

If we train on Fig. 7 with p set to the empty program and p_2 as described above, we will learn a probability distribution that $Pr(y = eq_1) = 1$. This means that certainly the

value to be predicted is equal to the value determined by the program p_2. Then, given a program at query time (could be another program, e.g., from Fig. 1 (d)), this model will predict that y is equal to the value returned by p_2 (the predicted value will be `user-agent` for the program in Fig. 1 (d)).

Our current semantics for p_2 relate a predicted label y to another value in the input $\boldsymbol{x}$ with an equality predicate. An interesting item for future work is to use other predicates to predict values that are a sum, a concatenation or another function on possibly several values in the input $\boldsymbol{x}$.

5. Implementation

We created a system called DEEP3 that given training data consisting of ASTs learns a probabilistic model that can predict program elements. Although we later evaluate our system on JavaScript and Python (as these are practically relevant), DEEP3 is language independent and is directly applicable to any programming language. DEEP3 consists of multiple components: a learning component that learns a TGEN program (using the ID3+ and E13 algorithms), a component that given a TGEN program builds a probabilistic model and a component to query the probability distribution and make predictions.

Since the other components are mostly standard and similar to what is seen in other probabilistic models (e.g., SRILM for language models [32]), here we focus on describing the component that learns TGEN programs.

5.1 Learning SIMPLECOND Programs

The SIMPLECOND fragment of the TGEN language includes programs without branches. Synthesis of such programs is what we defined as the procedure $syn_S(d) = \arg\min_{p \in \mathtt{DSL}_S} H(d, p)$ in Section 2.4. Our SIMPLECOND synthesis procedure simply enumerates all programs with up to 5 instructions and evaluates their entropy metric $H(d, p)$ on the given dataset d. Then to find longer programs we use genetic programming that keeps the best program discovered to a point plus several other candidate programs. At each iteration, our genetic programming procedure mutates the candidates program randomly by adding, removing or modifying a subset of their instructions. Overall, this synthesis procedure explores $\approx 20,000$ programs out of which the best one is selected.

5.2 Learning Branches

Our instantiation of the synthesis procedures syn_0 and syn_A for learning `Switch` instructions of the TGEN language uses a similar approach to the synthesis of SIMPLECOND programs. Here, we again use the exhaustive enumeration together with genetic programming in order to search for the predicates in the switch statement. In particular, we initialize the genetic search by enumerating all predicates consisting of up to 3 move instructions and 1 write instruction. Afterwards, for a given predicate, we consider the 32 most common values obtained by executing the predicate on the training data as possible branches in the synthesized case statement.

We further restrict the generated program to avoid overfitting to the training data. First, we require that each synthesized branch of a `Switch` instruction contains either more than 250 training data samples or 10% of the samples in the dataset d. Then, if a program in a branch gets 100% accuracy on the training data, we select this split even if some other split would score higher according to the metrics defined in ID3+ or E13.

5.3 Parallel Learning

Our complete learning procedure generally requires significant computational power when processing large datasets. To improve its speed, we parallelize the computation and scale it out to multiple machines in the cloud. Since the branches in a TGEN program can be synthesized independently, we send them to different machines. For learning TGEN programs, we used standard 4-core n1-standard-8 instances on the Google Compute Engine [2]. Our average training time per model was 60 machine hours and we could fit all our training in the free trial provided by Google. We learn two TGEN separate programs per programming language – one for predicting terminal symbols and one for predicting non-terminal symbols. Once a TGEN program is learned, obtaining a probabilistic model is fast and requires a few minutes on a single machine.

5.4 Smoothing

We use standard techniques from machine learning to increase the precision of our probabilistic models by incorporating Laplace smoothing [20, §6.9.1.1] and Witten-Bell interpolation smoothing [33]. Witten-Bell smoothing deals with data sparseness and uses the idea of falling back to conditioning contexts of smaller size (in the worst case falling to the unconditioned prediction) in case a prediction is based only on very few training samples. Laplace smoothing reassigns small amount of probability to labels not seen in training data to ensure they are assigned non-zero probability.

6. Evaluation

This section provides a thorough experimental evaluation of the learning approach presented so far. For the purposes of evaluation, we chose the tasks of learning probabilistic models of code for JavaScript and Python programs. This is a particularly challenging setting due to the dynamic nature of these languages, because it is difficult to extract precise semantic information via static analysis (e.g., type information) especially when working with partial code snippets. To evaluate our probabilistic model, we built a code completion system capable of predicting *any* JavaScript or Python program element (terminal or non-terminal symbol in the AST).

[2] https://cloud.google.com/compute/

Key benefits of our approach We demonstrate the benefits of our decision-tree-based learning approach by showing that:

- Our model essentially manages to learn "how to program" from a large training corpus of code. This is possible as the scalable learning allows for synthesizing large, complex TGEN programs not possible otherwise. Using our probabilistic model based on those programs, we report state-of-the-art accuracy for a number of interesting code prediction tasks including completing API calls, field accesses, loops, or branches in a program.
- The decision trees and probabilistic models synthesized by our approach are interesting beyond the induced probabilistic model. Our synthesized TGEN programs enable us to highlight the program elements used to make each prediction, and can in fact be used to explain and justify the prediction to the user.
- Although our TGEN language includes instructions that only traverse the JavaScript AST purely syntactically, it is interesting that the (large) obtained program has automatically learned how to perform lightweight program analysis to improve the precision of the models.

Experimental comparison of various systems In our experiments we compared the performance of several systems:

- PCFG and N-GRAM: we include two commonly used probabilistic models based on probabilistic context free grammars (PCFGs) and n-gram models (for $n = 3$). Despite their low accuracy, these models are currently used by a number of existing programming tools [1, 3, 4, 9, 11, 15].
- DEEPSYN: we use a previous state-of-the-art system for JavaScript code completion instantiated with our TGEN language (including the extension for predicting the out-of-vocabulary labels). For a fair comparison we implemented the probabilistic model from [27] and trained it on the full Python and JavaScript languages (as opposed to only JavaScript APIs and fields as in [27]) by learning one program for each AST node type (i.e., separate programs for predicting properties, identifiers, etc.).
- ID3+ and E13: these are the learning algorithms proposed in this work and implemented in DEEP3.

JavaScript datasets In our evaluation we use a corpus collected from GitHub repositories containing $150,000$ deduplicated and non-obfuscated JavaScript files that is publicly available at `http://www.srl.inf.ethz.ch/js150` and previously used in [27]. The first $100,000$ of the data are used for training and the last $50,000$ are used as a blind set for evaluation purposes only. From the $100,000$ training data samples, we use the first $20,000$ for learning the TGEN program. Further, in our experiments, we use only files that parse to ASTs with at most $30,000$ nodes, because larger trees tend to contain JSON objects as opposed to code.

The files are stored in their corresponding ASTs formats as defined by the ESTree specification[3]. Each AST node contains two attributes – the type of the node and an optional value. As an example consider the AST node `Property:autoHide` from Fig. 7 where `Property` denotes the type and `autoHide` is the value. The number of unique types is relatively small (44 for JavaScript) and is determined by the non-terminal symbols in the grammar that describe the AST whereas the number of values (10^9 in our corpus) is very large and is a mixture of identifiers, literals and language specified operators (e.g., `+, -, *`).

Python datasets For our evaluation, we also collected a corpus of Python programs from GitHub and made it available as ASTs at `http://www.srl.inf.ethz.ch/py150`. This dataset only includes programs with up to $30,000$ AST nodes and from open source projects with non-viral licenses such as MIT, Apache and BSD. To parse the dataset, we used the AST format for Python 2.7 from the parser included in the Python standard library (we also include the code that we used for parsing the input programs).

The ASTs are stored in a fashion similar to the JavaScript ASTs such that every node includes type and optionally a value. For example, the AST node `attr:path` is of type `attr` and has value `path`. Semantically, this node corresponds to accessing the `path` attribute of a Python object.

Methodology Given the structure of ASTs, we learn two different models for a programming language – one for predicting node type and a second one for predicting the node value. Since both of these define probability distributions, they can be easily combined into a single model of the full programming language.

To train the model for predicting types we generate one training sample for each AST node in the dataset by replacing it with an empty node (i.e., a hole) and removing all the nodes to the right. Following the same procedure we generate training samples for training the model for values, except that we do not remove the type of node to be predicted, but only its value. Using this procedure we obtain our training dataset $\mathcal{D} = \{(\boldsymbol{x}^{(j)}, y^{(j)})\}_{j=1}^{n}$. In total, our JavaScript dataset consists of $10.7 * 10^8$ samples used for training and $5.3 * 10^7$ used for evaluation. Our Python dataset consists of $6.2 * 10^7$ training samples and $3 * 10^7$ evaluation samples.

In our evaluation, we measure the precision of our models by using the *accuracy* metric. Accuracy is the proportion of cases where the predicted label y with the highest probability according to our probabilistic model is the correct one in the evaluated program. Note that our model always provides a prediction and therefore the recall metric is always 100%.

To make sure our predictions also capture the structure of the tree, and not only the labels in the nodes of the tree, we

[3] https://github.com/estree/estree

Applications	PCFG	Prior Work 3-gram	DeepSyn [27]	Our Work: Deep3 ID3+	E13
Value prediction accuracy (Fig. 9)					
Unrestricted prediction	50.1%	71.2%	80.9%	76.5%	**82.9%**
API prediction	0.04%	30.0%	59.4%	54.0%	**66.6%**
Field access prediction	3.2%	32.9%	61.8%	52.5%	**67.0%**
Type prediction accuracy (Fig. 8)					
Unrestricted prediction	51.5%	69.2%	74.1%	**83.9%**	80.0%
Predicting Loop statements	0%	37.5%	0.04%	**65.0%**	28.3%
Predicting Branch statements	0%	40.9%	17.3%	**65.7%**	40.4%

Table 1. Accuracy comparison for selected tasks between JavaScript models used in prior work and our technique.

also predict whether a given node should have any siblings or children. For this purpose, when predicting the type of an AST node, the label further encodes whether the given node has right siblings and children which allows us to expand the tree accordingly.

6.1 Probabilistic Model for JavaScript

Our most interesting setting is one where we are given a large training data corpus of JavaScript programs, and we learn a probabilistic model of JavaScript. That is, the model essentially learns to predict new JavaScript programs by learning from a large corpus of existing JavaScript code. In Table 1 we provide highlights of the accuracy for several interesting prediction tasks.

Each row of Table 1 includes an application we evaluate on and each column includes the accuracy of the corresponding probabilistic model on this task. The tasks of "unrestricted predictions" for both values and types include a wide range of sub-tasks – some of them are easy and others are not. Example of an easy task is to predict that there will be nodes of type `Property` inside an `ObjectExpression` (i.e., properties inside a JSON object). As a result of these easy tasks, even the most trivial baselines such as PCFG succeed in predicting around half of the labels.

We include some of the more difficult tasks as separate rows in Table 1. When faced with these tasks of predicting APIs, field accesses or less frequent statements such as loops and branches, the accuracy of the PCFG model essentially goes down to 0%.

For every task in our experiments, a probabilistic model based on decision trees has higher accuracy than any of previous models – PCFG, the 3-gram language model or the `DeepSyn` model. An interesting observation is that the model obtained from the `ID3+` algorithm is more precise than the one obtained from `E13` when predicting types and in contrast, `E13` produces the most precise model for values. We hypothesize that the reason for this is the relatively smaller number of labels for types – there are 176 unique labels for types and around 10^9 labels for values. In our further evaluation, we take the TGEN program obtained from `ID3+` for types and the TGEN program obtained from `E13` for values.

Prediction Type	Prior Work DeepSyn [27]	Our Work ID3+
`ContinueStatement`	17%	**58%**
`ForStatement`	0%	**60%**
`WhileStatement`	0%	**76%**
`ReturnStatement`	14%	**75%**
`SwitchStatement`	1%	**47%**
`ThrowStatement`	9%	**51%**
`TryStatement`	2%	**54%**
`IfStatement`	18%	**66%**

```
positions used to condition the prediction
for (j = 0; j < groups.length; j++) {
  idsInGroup = groups[j].filter(
    function(id) {
      return ids.indexOf(id) >= 0;
    }
  );
  if (idsInGroup.length === 0) {
    ? ← prediction position
  }
}
```

correct → type

	$Pr_{\text{ID3+}}$	Pr_{DeepSyn}	$Pr_{\text{n-gram}}$
`ContinueStatement`	0.86	0.03	0.00
`ReturnStatement`	0.04	0.11	0.11
`ExpressionStatement`	0.03	0.66	0.61
`VariableDeclaration`	0.02	0.06	0.10

Figure 8. List of new predictions enabled by our decision tree model when predicting type of a statement (top). Example of predicting type of a statement from a code snippet in our evaluation data. We show the top 4 predictions and their probabilities predicted by each system (bottom).

Since each of these coarse-grained tasks in our evaluation include a wide range of easy and more difficult prediction tasks, next we focus on a more detailed evaluation on the predictions for node types and values.

6.1.1 Predicting Types

We first discuss the application of our model for predicting types of AST nodes, that is, learning the structure of the code. There are in total 44 different types of nodes in JavaScript that range over program statements, expressions

as well as constants and identifiers. Our predicted labels for types also include tree structure information whether the node has a right sibling and children. As a result, the total number of different labels that can be predicted is 176.

We provide a detailed list of difficult type predictions for previous models that are now enabled by our decision tree models in Fig. 8 (top). An interesting insight of our evaluation is that there are entire classes of predictions where previous models (such as DeepSyn) fail to make correct predictions about the program structure. For example because `for` statements are less frequent in code than other statements (e.g., assignments), previous models predict such statements with very low accuracy. In contrast, our decision tree models partition the training data into multiple branches and builds precise models for each such case.

Predicting the structure of code is overall a very hard task. Previous models failed to predict a range of statements such as loops, switch statements, if statements and exception handling statements as shown in Fig. 8. We next give an example of a prediction done by DEEP3.

Example completion To Illustrate the difficulty of correctly predicting such queries consider an example shown in Fig. 8 (bottom). Here, the figure shows the original code snippet with the developer querying the code completion system asking it to predict the statement at the position denoted with "?". As can be seen by inspecting at the code, it is not immediately clear which statement should be filled in as it depends on the intended semantics of the developer. However, by training on large enough dataset and conditioning our prediction on appropriate parts of the code, we can hopefully discover some regularities that help us make good predictions.

Indeed, for this example as well as 58% of other queries, DEEP3 successfully predicted a `ContinueStatement` statement. In this case, our model suggests `ContinueStatement` with very high probability of 86%, whereas the second most likely prediction has only 4% probability.

On the other hand, existing models (PCFG, n-gram and DeepSyn) are biased towards predicting the statement `ExpressionStatement` simply because it is three orders of magnitude more frequent than `ContinueStatement`. These models cannot discover proper conditioning to predict the correct statement with high confidence.

Learned program To understand how DEEP3 obtained its high accuracy we examined the program learned during the training and the branch this example fell into. The decision tree performed the following checks for this prediction: i) check whether the query node is the first child of the current scope, ii) test whether the current scope is defined by an `IfStatement`, iii) retrieve what is the type of node that defined previous scope and test if it defines a loop or other constructs such as function or an `IfStatement`. Once all of these conditions were satisfied, the probabilistic model looked at the values in the last `IfStatement` where the query node is defined. In Fig. 8 (bottom), we highlighted all positions in the code on which our probabilistic model conditions in order to make the correct prediction.

Prediction Type	Prior Work DeepSyn [27]	Our Work E13
API name Example: `this.getScrollBottom(inTop)`	59.4%	**66.6%**
API call target Example: `node.removeAttribute(attName)`	63.1%	**67.0%**
Array index identifier Example: `event[prop] = ...`	72.4%	**82.8%**
Assigment variable identifier Example: `result = ...`	66.8%	**70.3%**

API completion query:

```
point_x.applyForce(direction.multiply(...));
point_y.applyForce(direction.? ← prediction position
```

Fragment of the learned TGEN program for predicting APIs:

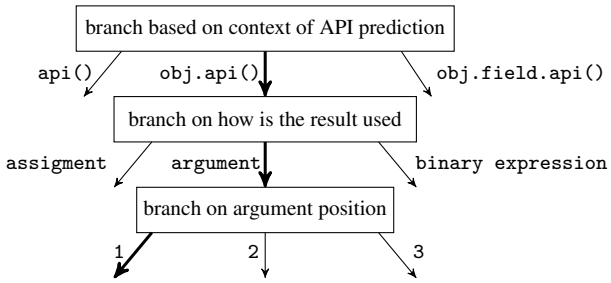


Figure 9. Accuracy of various applications for predicting values in JavaScript (top). Example of an API completion query and visualization of the decomposition learned by our approach (bottom). As can be seen the our approach learns a specialized model that is learn on queries predicting API invoked directly on call target that are used as second argument in another method invocation.

6.1.2 Predicting Values

We now turn attention to evaluating the quality of the learned program trained for predicting values in JavaScript programs. While this task is similar to the task of predicting node types in JavaScript ASTs, some of the predictions are much more challenging because the label set for values is several orders of magnitude larger than the label set for types. In Fig. 9 (top) we show the accuracy for several prediction tasks as well as examples of predictions made for these tasks. For all these tasks we improve the accuracy between 3% to 10% over the accuracy achieved by the DeepSyn model. In addition, for the API and field access prediction tasks, DEEP3 outperforms DeepSyn by 6% and 5%, respectively as shown in Table 1. We expect that these tasks are useful in the context of IDE code completion and

the improvement in accuracy should result in better user experience in the IDE.

Learned program As an interesting example query for value prediction consider the query shown in Fig. 9 (bottom) where the value should be completed with an API call. The goal of this query is to predict the API name at the position denoted by "?". For this case, the learned TGEN program investigates the context in which the API call is done (on a variable, on a field object, on `this` object, etc.), how the result of the call is used and since it is used as an argument in a function call, at what position that argument is. Note that while this program makes sense since it closely identifies the kind of API used, providing all this conditioning manually would require tremendous amount of effort.

6.2 Predicting Out-of-Vocabulary Labels

We next evaluate DEEP3's capability to predict values not seen in the training data with the model described in Section 4.2. To check the effect of this extension (which affects 11% of the learned programs used as leaves in the decision tree), we performed an experiment where we compared the accuracy of the resulting probabilistic models with and without the extension. For a fair comparison, Table 1 summarizes the results for both `DeepSyn` and DEEP3 with this extension enabled.

If we disable the second program p_2 from Section 4.2, the overall accuracy for predicting values decreases by 2%, from 82.9% to 80.9%. This decrease is caused mostly by the lower accuracy of predicting identifiers and properties – these are the two prediction tasks that contain most of user defined values. On the other hand, for predicting types the second program that describes equality does not affect the accuracy. This is intuitive as all the possible labels for types are easily seen in the training data.

6.3 Learned TGEN Programs

Using the learning approach proposed in our work, we discover a TGEN program with a large number of branches that is interesting in itself and provides several benefits beyond providing state-of-the-art code completion system. The programs learned using the `E13` algorithm contain $13,160$ and 307 leaves together with $5,869$ and 157 internal `switch` nodes in their decision trees for values and types respectively. That number of cases is clearly infeasible to conceive or design manually. For example, only for predicting `ContinueStatement`, the learned TGEN program for types uses 30 different leaves in its decision tree. One of the advantages of our approach is that despite the relatively large size of the model, its learned TGEN program can be easily inspected, interpreted and even manually modified by an expert, if needed.

An interesting observation we made by looking at the learned TGEN program for values is that the model learns sequences of instructions that perform traversals to a previous method invocation depending on whether the call target is a simple identifier, another call expression or field access. That is, our synthesized TGEN performs a form of lightweight program analysis and leads to improvements in the prediction accuracy. Such specialized sequences are at the moment learned for performing certain kinds of predictions in some branches of a TGEN program. An interesting future work item is to build a library of such automatically learned program analyses and investigate their applicability for building either more precise probabilistic models for code or for other problems in this space.

Applications	SVM	Our Work: Deep3
Value prediction accuracy	70.5%	**82.9%**
Type prediction accuracy	67.5%	**83.9%**

Table 2. Accuracy comparison of discriminative model trained using SVM and our technique.

Prediction speed Even though the learned programs contain thousands of instructions, executing them is fast. This is due to the fact that for any prediction only a small part of the instructions in a program needs to be executed (since execution traverses only a single branch of the decision tree). As a result, DEEP3 is capable of answering around $15,000$ completion queries per second on a single CPU core.

6.4 Comparison to SVM

The probabilistic model that we construct in DEEP3 is *generative*, meaning that it can assign probabilities to entire programs. In contrast, several recent works build *discriminative* models that only learn to do certain kinds of predictions conditioned on a program without being able to assign probabilities to the program itself. Examples of such discriminative models are presented in [3, 4, 18, 26], which all work by first generating a set of features and then learning the weights of these features on a training dataset.

We compare DEEP3 to a discriminative model based on support vector machine (SVM). A similar SVM model was used in JSNICE [26] for predicting variable names and type annotations. Our SVM model is based on syntactic feature functions that correspond to the types and values of the 10 nodes preceding the completion position in the AST (these features are also similar to the ones used in [4]). The query that we trained our SVM model on, is completing one AST node of a program – the very same query on which we evaluate our probabilistic models for code, except that the SVM model does not return probabilities for its predictions.

To learn the weights for each feature, we used an online support vector machine (SVM) learning algorithm based on hinge loss (following the approach in [26]) and performed grid-search for the parameters that control the regularization (L_∞), the learning rate and the margin of the SVM.

Applications	PCFG	Prior Work 3-gram	DeepSyn [27]	Our Work: Deep3 ID3+	E13
Value prediction accuracy					
Unrestricted prediction	10.2%	63.9%	67.2%	63.7%	**69.2%**
Attribute access prediction	0%	25%	**42%**	27%	**42%**
Numeric constant prediction	22%	44%	40%	39%	**46%**
Name (variable, module) prediction	17%	38%	38%	39%	**51%**
Function parameter name prediction	40%	50%	50%	50%	**57%**
Type prediction accuracy					
Unrestricted prediction	59.0%	63.2%	72.5%	76.1%	**76.3%**
Function call prediction	55%	65%	71%	**74%**	**74%**
Assignment statement prediction	29%	39%	61%	**67%**	66%
Return statement prediction	0%	19%	10%	**41%**	29%
List prediction	23%	46%	52%	**58%**	52%
Dictionary prediction	30%	52%	59%	**61%**	**61%**
Predicting `raise` statements	0%	18%	1%	**27%**	13%
Predicting (presence of) function parameters	54%	47%	70%	75%	**76%**

Table 3. Accuracy comparison for selected tasks between Python models used in prior work and our technique.

The results for predicting values and types of AST nodes are summarized in Table 2. The SVM has worse accuracy compared to DEEP3 and is in fact worse than the 3-gram model shown in Table 1. A reason for the worse performance of these discriminative models is that in contrast to our approach, SVMs always use fixed weights for the given features and cannot express that different features are relevant for different kinds of predictions.

6.5 Probabilistic Model for Python

To build our probabilistic model for Python, we took DEEP3 and fed it with another training dataset of Python ASTs. This means that the only effort necessary for our models to handle that programming language was to provide a parser for Python and to download a large training dataset.

We summarize the accuracy of DEEP3 in Table 3 and compare it to baseline models such as PCFG, n-gram language model and a model synthesized by the algorithm of `DeepSyn` [27]. Overall, the results for Python mimic the ones for JavaScript, but with some nuances:

- Similar to JavaScript, the best Python models learned by DEEP3 outperform the models from previous works. The ID3+ algorithm performs well for predicting the node types, but not as well for predicting node values. The E13 algorithm is the best algorithm for predicting node values.
- The precision of all Python models is lower than the precision of the corresponding model for JavaScript. One possible explanation is the different structure of the Python ASTs, which include less information that is redundant and easily predictable. The lower precision of the PCFG model for Python in comparison to the PCFG model for JavaScript also supports this interpretation of the results.
- In contrast to JavaScript, the Python syntax includes expressions for specifying lists and sets. DEEP3 can predict these expressions with higher accuracy than prior works.

7. Related Work

We next survey some of the work that is most closely related to ours.

Decision tree learning Decision trees are a well studied and widely used approach for learning classifiers. Among the large number of decision tree algorithms notable ones include ID3 [23] and its successor C4.5 [24], along with various general purpose techniques such as bagging and random forests.

Although decision trees are mostly used as black box classification technique for some applications, it is sometimes necessary to extend them in a way that the learning reflects the requirements or the domain knowledge available for the given task at hand. For example, in the context of learning program invariants [8], it is necessary that the decision tree classifies all examples perfectly and includes domain knowledge in form of implication counter-examples.

Similarly, in our work we extend and adapt the classic ID3 learning algorithm so that it takes advantage of the fact that the leafs do not necessary have to correspond to unconditioned maximum likelihood estimate but can be represented using probabilistic models conditioned on complex features (in our case, contexts).

Probabilistic models of code Recently, there has been an increased interest in building probabilistic models of code and using these probabilistic models for various prediction tasks. Several existing approaches build a probabilistic model where the conditioning is hard-wired: via the simple n-gram model and syntactic elements [11], the n-gram and recurrent neural networks with API calls [25], program

expressions [10], tree substitutions in the abstract syntax tree [2] or other [21]. Further, several approaches address the limitation of the n-gram model by providing better semantic program abstractions using program dependence graphs [13] or various extensions including topic models [22] and modeling local and global context [34].

While a good first step, these systems suffer from very low precision (e.g., [11]) or perform well only in restricted scenarios (e.g., APIs in [25]) and for specific languages with strong type information (e.g., Java). For instance, [25] has poor API prediction precision in the case of dynamic languages such as JavaScript (evaluated in [27]).

Regardless of the performance of such models, our approach provides a general framework which can be instantiated by arbitrary existing models used as leafs in the decision tree that is being learned without any changes required to the learning algorithm. That is, our work allows us to leverage some of these models at the leafs.

As discussed throughout the paper, in terms of expressiveness, the recent works of [7, 27] are perhaps most advanced and precise: they condition not only on a fixed context but also on a dynamically learned context. While promising, as we illustrated in the paper, these approaches only learn a single program without branches that must somehow explain a large structured domain such as entire, complex programming languages (e.g., JavaScript and Python) together with the frameworks used by programs written in these languages. As a result, the precision of these approaches is worse than our work.

Another recent work proposes a language model of C# programs based on a log-linear tree-traversal model [19]. This model is only capable of learning a linear combination of a small set of predefined features in contrast to our work where we learn from an exponentially larger set of programs. Further, our learned functions that guide the predictions are interpretable as a program and not only as a set of weights. Because the model of our work is described by a program, it can also be understood, edited by a human and further tuned towards a specific application.

Discriminative learning In addition to generative approaches for code modeling, there have also been several recent works that employ discriminative models including [18, 26]. Such approaches however do not provide valid probability distributions and require user specified feature functions whose weights are then learned. The feature functions can be difficult to discover manually, are designed only for a specific task the tool addresses, and cannot serve as a basis for a general purpose model of complex, rich programming languages such as JavaScript and Python. Further, the combination of weights and feature functions leads to models that are difficult to understand, debug and explain.

Program synthesis A common challenge for many existing program synthesis techniques [5, 14, 30, 31] is scalability: it is still difficult to scale these approaches to the task of synthesizing large, practical programs. To address this challenge, several recent works attempt exploit the structure of the particular task to be synthesized by using techniques such as hierarchical relational specifications [12] or shapes of independent components [6]. Such approaches allow for finding suitable decompositions easily which in turn can significantly speed up the synthesis procedure. Another approach taken by Raza et. al. [28] proposes the use of compositional synthesis guided by examples. Finally, Kneuss et. al. [16] decompose the initial synthesis problem of discovering a recursive function into smaller subproblems.

The main difference between our work and these approaches is that these attempt to satisfy all of the provided input/output examples. This has implications on the scalability as well as on the level of acceptable errors. Further, in our work we consider a very different setting consisting of large and noisy datasets where our goal is to discover a decomposition *without relying on any domain specific knowledge* on the shape of the underlying components and without using guidance from counter-examples.

8. Conclusion

We presented a new approach for learning probabilistic models of code. The key insight of our work is a new decision tree based learning approach that: (i) discovers a suitable decomposition of the entire data set, (ii) learns the best conditioning context for each component, and (iii) allows usage of probabilistic models as leafs in the obtained decision tree.

We generalize our approach by phrasing the problem of decision tree learning in terms of learning a program in a domain specific language (DSL) that includes branch instructions. We instantiate the learning by defining a general purpose and programming language independent DSL, called TGEN. This language allows for traversing abstract syntax trees (ASTs) and provides ways for accumulating a conditioning context obtained from traversing the AST.

We implemented our approach and applied it to the task of building probabilistic models for Python and JavaScript, and developed a statistical code completion system, called DEEP3, based on our model. DEEP3 can provide completions for any program element and achieves precision higher than that of existing techniques.

The key to DEEP3's improved accuracy is the ability of the underlying probabilistic model to discover interesting decompositions of the entire dataset into more than several hundreds specialized components. This is a task which would be infeasible to perform manually, by hand. Crucially, the discovered components can be inspected by a human, are often intuitive, and can also be used to provide a justification for the prediction.

We believe this work represents an important advance in systematically learning powerful probabilistic models of code which will be useful as a core building block for a variety of applications.

References

[1] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. In *MSR*, 2013.

[2] M. Allamanis and C. Sutton. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 472–483, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3056-5.

[3] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8.

[4] M. Allamanis, D. Tarlow, A. D. Gordon, and Y. Wei. Bimodal modelling of source code and natural language. In F. R. Bach and D. M. Blei, editors, *ICML*, volume 37 of *JMLR Proceedings*, pages 2123–2132. JMLR.org, 2015.

[5] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 1–8, 2013.

[6] S. Barman, R. Bodik, S. Chandra, E. Torlak, A. Bhattacharya, and D. Culler. Toward tool support for interactive synthesis. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, pages 121–136, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3688-8.

[7] P. Bielik, V. Raychev, and M. T. Vechev. 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*, pages 2933–2942, 2016. URL `http://jmlr.org/proceedings/papers/v48/bielik16.html`.

[8] P. Garg, D. Neider, P. Madhusudan, and D. Roth. Learning invariants using decision trees and implication counterexamples. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 499–512, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2.

[9] T. Gvero and V. Kuncak. 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, part of SLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 416–432, 2015.

[10] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 27–38. ACM, 2013. ISBN 978-1-4503-2014-6.

[11] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3.

[12] T. Hottelier and R. Bodik. Synthesis of layout engines from relational constraints. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 74–88, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3689-5.

[13] C.-H. Hsiao, M. Cafarella, and S. Narayanasamy. Using web corpus statistics for program analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 49–65, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1.

[14] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 215–224, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-719-6.

[15] S. Karaivanov, V. Raychev, and M. T. Vechev. Phrase-based statistical translation of programming languages. In *Onward! 2014, Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software, part of SLASH '14, Portland, OR, USA, October 20-24, 2014*, pages 173–184, 2014. doi: 10.1145/2661136.2661148.

[16] E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter. Synthesis modulo recursive functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 407–426, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1. doi: 10.1145/2509136.2509555. URL `http://doi.acm.org/10.1145/2509136.2509555`.

[17] P. Liang, M. I. Jordan, and D. Klein. Learning programs: A hierarchical bayesian approach. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*, pages 639–646, 2010. URL `http://www.icml2010.org/papers/568.pdf`.

[18] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 298–312, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2.

[19] C. J. Maddison and D. Tarlow. Structured generative models of natural source code. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014*, pages 649–657, 2014.

[20] T. M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[21] A. T. Nguyen and T. N. Nguyen. Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 858–868, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5.

[22] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 532–542, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2237-9.

[23] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1 (1):81–106, Mar. 1986. ISSN 0885-6125. doi: 10.1023/A:1022643204877.

[24] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. ISBN 1-55860-238-0.

[25] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 419–428, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8.

[26] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from "big code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 111–124. ACM, 2015. ISBN 978-1-4503-3300-9.

[27] V. Raychev, P. Bielik, M. Vechev, and A. Krause. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 761–774, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837671.

[28] M. Raza, S. Gulwani, and N. Milic-Frayling. Compositional program synthesis from natural language and examples. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 792–800, 2015.

[29] R. Rosenfeld. Two decades of statistical language modeling: Where do we go from here. In *Proceedings of the IEEE*, 2000.

[30] A. Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.

[31] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[32] A. Stolcke. SRILM-an Extensible Language Modeling Toolkit. *International Conference on Spoken Language Processing*, 2002.

[33] I. H. Witten and T. C. Bell. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, 37 (4):1085–1094, 1991.

[34] T. Zhaopeng, S. Zhendong, and D. Premkumar. On the localness of software. In *Foundations of Software Engineering*, FSE '14, New York, NY, USA, 2014. ACM.