Learning Programs from Noisy Data

Veselin Raychev Pavol Bielik Martin Vechev Andreas Krause

Department of Computer Science
ETH Zurich
{veselin.raychev, pavol.bielik, martin.vechev, krausea}@inf.ethz.ch

Abstract

We present a new approach for learning programs from noisy datasets. Our approach is based on two new concepts: a *regularized program generator* which produces a candidate program based on a small sample of the entire dataset while avoiding overfitting, and a *dataset sampler* which carefully samples the dataset by leveraging the candidate program's score on that dataset. The two components are connected in a continuous feedback-directed loop.

We show how to apply this approach to two settings: one where the dataset has a bound on the noise, and another without a noise bound. The second setting leads to a new way of performing approximate empirical risk minimization on hypotheses classes formed by a discrete search space.

We then present two new kinds of program synthesizers which target the two noise settings. First, we introduce a novel regularized bitstream synthesizer that successfully generates programs even in the presence of incorrect examples. We show that the synthesizer can detect errors in the examples while combating overfitting – a major problem in existing synthesis techniques. We also show how the approach can be used in a setting where the dataset grows dynamically via new examples (e.g., provided by a human).

Second, we present a novel technique for constructing statistical code completion systems. These are systems trained on massive datasets of open source programs, also known as "Big Code". The key idea is to introduce a domain specific language (DSL) over trees and to learn functions in that DSL directly from the dataset. These learned functions then condition the predictions made by the system. This is a flexible and powerful technique which generalizes several existing works as we no longer need to decide a priori on what the prediction should be conditioned (another benefit is that the learned functions are a natural mechanism for explaining the prediction). As a result, our code completion system surpasses the prediction capabilities of existing, hard-wired systems.

Categories and Subject Descriptors I.2.5 [*Artificial Intelligence*]: Programming Languages and Software; I.2.2 [*Artificial Intelligence*]: Automatic Programming; D.2.3 [*Software Engineering*]: Coding Tools and Techniques

Keywords Program Synthesis, Big Code, Statistical Code Completion, Anomaly Detection, Regularization, Noisy Data

POPL' 16, January 20-22, 2016, St. Petersburg, FL, USA.

http://dx.doi.org/10.1145/2837614.2837671

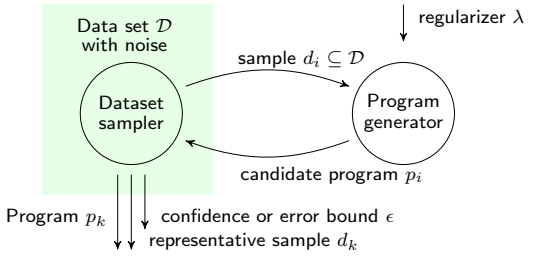

Figure 1. Learning programs from noisy data.

1. Introduction

This paper presents a new approach for learning programs from noisy datasets. While there has been substantial work over the last few decades on learning programs from examples (e.g., [18, 19]), these approaches cannot adequately deal with incorrect input as they attempt to satisfy *all* examples, thus overfitting to the data. This means that when the dataset is noisy, they either fail to return a program or produce the wrong program. Post-mortem ranking techniques do not help as they simply end up ranking incorrect solutions. Other synthesizers have limited support to discover incorrect examples when the synthesis procedure on all examples fails [12], but are specific to a particular domain of programs. The field of machine learning (ML) has extensively studied the problem of learning functions from noisy data. However, the space of programs (hypotheses in ML terms) is vastly more complex than the usual parametric models (e.g., linear separators, neural networks) typically considered in machine learning. When the hypothesis space is simpler, one can afford to perform optimization over the (continuous, and potentially convex) parameter space and the entire dataset, an optimization task that is computationally infeasible in the (discrete) setting of complex program predictions.

Our work The approach presented in this work, illustrated in Fig. 1, enables learning of programs from a noisy dataset $\mathcal{D}$ of arbitrary size. The technique is based on several new concepts. First, a *dataset sampler* carefully selects (small) samples with specific properties from $\mathcal{D}$. Second, a regularized *program generator* produces a program given the selected sample in a way which controls the complexity of the solution and avoids over-fitting. The two components are linked together in a feedback directed way, iterating until the desired solution p_k is found. As we show in the paper, this approach can serve as a basis for constructing a range of prediction engines given a noisy dataset $\mathcal{D}$.

Quantifying noise To systematically instantiate our approach, we consider both cases for quantifying the noise in the dataset: the case where we have a bound on the noise and the case where the noise is arbitrary. In the first case, we also provide optimality guarantees on the learned program. In the second case, we approach the learning

problem with a fast, scalable algorithm for performing approximate empirical risk minimization (ERM) [21], bridging the fields of applied program synthesis and machine learning. Our setting in fact raises new challenges from a machine learning perspective as here, ERM is performed not on the whole data at once as in traditional ML, but on carefully selected (small) samples of it.

Synthesis with noise To illustrate how our concepts apply, we present two new kinds of synthesizers targeting the different noise settings. First, we present a synthesizer for bit-stream programs, called BITSYN, where we dynamically add examples, possibly with some errors, until we produce the desired program. Second, for the case of large datasets, we present DEEPSYN, a new kind of statistical synthesizer that learns probabilistic models from "Big Code" (a dataset of programs available in repositories such as GitHub [10]) and makes predictions (i.e., code completion) based on this model. Our prediction engine generalizes several existing efforts (e.g., [16, 30]) and is able to make predictions beyond the capability of these systems. Importantly, as our predictions are conditioned on program elements, they are easy to understand and justify to a programmer using DEEPSYN, a capability missing in approaches where the prediction is based on weights and feature functions, and is not human understandable.

Detecting noise While not the primary goal, we believe this work represents a new way for performing *anomaly detection*: besides the learned program, the algorithm in Fig. 1 can return the set of examples d_k the program does not satisfy (these represent the potential anomalies). Our approach is unlike prior work which either assumes the program is already provided [5] or require statistical assumptions on the data [6].

Main contributions Our main contributions are:

- A new approach for learning programs from noisy datasets of arbitrary size. The key concepts are a dataset sampler which carefully samples the dataset, and a regularized program generator which avoids overfitting, with both components linked in a feedback directed loop.
- An instantiation of the approach to two settings: one where we place a bound on the noise in the dataset, and another where the error bound is unknown.
- Our second instantiation leads to a new way of performing approximate ERM on hypotheses classes consisting of a discrete search space (traditionally, ML considers hypotheses classes defined via a continuous parameterization).
- Two variants of a bit-stream synthesizer, illustrating how to use our techniques in an interactive programming-by-example setting as well as anomaly detection.
- A new method for constructing statistical code completion engines from "Big Code" (which deal with the unbounded noise setting). This method introduces a DSL over trees and learns loop-free functions in the DSL. The learned functions control the predictions (and thus, can explain a prediction), generalizing much prior work where the prediction is "hard-wired".
- A complete implementation and evaluation of BITSYN and DEEPSYN, showing their effectiveness in handling a variety of noise settings, a capability beyond that of existing approaches.

We believe this is the first comprehensive work that deals with the problem of learning programs from noisy datasets, and represents an important step in understanding the trade-offs arising when trying to build program synthesis engines that deal with noise. Based on the techniques presented here, one can investigate how to adapt and extend many of the existing programming-by-example and synthesis engines to deal with noise.

Paper outline The rest of the paper is structured as follows. In Section 2 we introduce our synthesis algorithm and show some of its important properties. Then, the paper continues with two possible settings for the noise: (i) sections 3 and 4 apply our approach to synthesis with bounded noise on bit-stream programs and (ii) sections 5, 6 and 7 apply our approach to synthesis with unbounded noise for constructing statistical code completion engines from "Big Code". In Section 8 we discuss related work. The paper concludes with Section 9.

We provide additional resources such as source code and test data online at `http://www.srl.inf.ethz.ch/noise.php`.

2. General Approach

In this section we present our general approach for learning programs from datasets with incorrect (noisy) examples. We show how to instantiate this approach in later sections.

2.1 Problem Formulation

Let $\mathcal{D}$ be a dataset consisting of a set of examples and $\mathbb{P}$ be the set of all possible programs. The objective is to discover a program in $\mathbb{P}$ which satisfies the examples in $\mathcal{D}$. In practice however, the dataset $\mathcal{D}$ may be imperfect and contain errors, that is, contain examples which the program should not attempt to satisfy. These errors can arise for various reasons, for instance, the user inadvertently provided an incorrect example in the dataset $\mathcal{D}$, or the dataset already came with noise (of which the user may be unaware of).

Fundamentally, because we are not dealing with the binary case of correct/incorrect programs and need to deal with errors, we introduce a form of a cost (risk) function associated with the program to be learned from the noisy dataset.

Let $r\colon \mathcal{P}(\mathcal{D}) \times \mathbb{P} \to \mathbb{R}$ be a cost function that given a dataset and a program, returns a non-negative real value that determines the inferiority of the program on the dataset. In machine learning terms, we can think of this function as a generalized form of empirical risk (e.g., error rate) associated with the data and the function. In the special case typically addressed by PBE systems (e.g., [18, 19]), the function returns either 0 or 1, that is, the program either produces the desired output for all inputs in the given dataset, or it does not. Later in the paper, we discuss several possibilities for the r function depending on the particular application.

Problem statement The learning problem is the following:

$$\text{find a program } \; p_{best} = \arg\min_{p \in \mathbb{P}} r(\mathcal{D}, p)$$

That is, the goal of learning is to find a program whose cost on the entire dataset is lowest (e.g., makes the least number of errors, or minimizes empirical risk as in Section 5). We note that while in general there could be many programs with an equal (lowest) cost, for our purposes it suffices to find one of these. It is easy to instantiate this problem formulation to the specific, binary case of synthesis from examples, where r returns 1 if some example in $\mathcal{D}$ is not satisfied and 0 otherwise. A challenge which arises in solving the above problem is that the dataset $\mathcal{D}$ may be prohibitively large, or simply infinite (e.g., may need to continually ask a user for samples of the dataset) and thus trying to directly learn the optimal program p_{best} that satisfies the dataset $\mathcal{D}$ may be infeasible.

2.2 Our Solution: Iterative Synthesis Algorithm

The key idea of our solution is to start with a small sample of the dataset $\mathcal{D}$ and to iteratively and carefully grow this sample in a way which allows finding a good solution with a few, small-sized samples. Our solution consists of two separate components: a *program generator* and a *dataset sampler*. We continually iterate between these two components until we reach a fixed point and the desired program is found.

Input: Dataset $\mathcal{D}$, initial (e.g. random) dataset $\emptyset \subset d_1 \subseteq \mathcal{D}$
Output: Program p
begin
  $progs \leftarrow \emptyset$
  $i \leftarrow 0$
  repeat
    $i \leftarrow i + 1$
    // *Dataset sampling* step
    if $i > 1$ **then**
      $d_i \leftarrow ds(progs, |d_{i-1}| + 1)$
    end
    // *Program generation* step
    $p_i \leftarrow gen(d_i)$
    if *found_program*(p_i) **then**
      return p_i
    end
    $progs \leftarrow progs \cup \{p_i\}$
  until $d_i = \mathcal{D}$;
  return "No such program exists"
end

Algorithm 1: Program Synthesis with Noise

Program generator For a finite dataset $d \subseteq \mathcal{D}$, a program generator is a function $gen\colon \mathcal{P}(\mathcal{D}) \to \mathbb{P}$ defined as follows:

$$gen(d) = \arg\min_{p \in \mathbb{P}} r(d, p)$$

To reduce the (expensive) invocation of $gen(d)$, in our prediction algorithm, we aim for a size of the dataset d that is as small as possible.

Dataset sampler The second component of our approach is what we refer to as the *dataset sampler* $ds\colon \mathcal{P}(\mathbb{P}) \times \mathbb{N} \to \mathcal{P}(\mathcal{D})$:

$$ds(progs, n) = d' \texttt{ with } |d'| \geq n$$

That is, a dataset sampler takes as input a set of programs (and a bound on the minimum size of the returned sample) and produces a set of examples which are then fed back into the generator. We will see several instantiations of the data sampler later in the paper.

Iterative sampling We connect the program generator and data sampler components in an iterative loop. The resulting algorithm is shown in Algorithm 1. At every iteration of the loop, the algorithm checks if the current program p_i is a satisfactory solution and can be returned (in later sections, we discuss instantiations of *found_program*). If the current program p_i is not the right one, we sample from the dataset $\mathcal{D}$ using the current set of explored programs $progs$, obtaining the next dataset d_i. Note that while the size of the sample d_i is greater than the size of the previous sample d_{i-1}, there is no requirement that d_i is a superset of d_{i-1} (i.e., the sets may be non-comparable). Once we have obtained our new sample d_i, we use the sample to generate the new candidate program p_i. In case the program p_i is not the desired one, the algorithm continues and adds p_i to the $progs$ set and continues iterating.

2.3 Reduction of Search Space

First, note that Algorithm 1 always terminates if the dataset $\mathcal{D}$ is finite. This is because the size of the dataset d_i increases at every step until it eventually reaches the full dataset. However, our goal is to discover a good program using only a small dataset d_i.

To achieve this, we leverage the dataset sampler to carefully pick small datasets. Consider the first dataset d_1. Since this dataset may be random, let us assume that any possible program p_1 can be returned as a result. If p_1 was not the desired program, we would like to select the next dataset d_2 to be such that p_1 cannot be returned at the next step by $gen(d_2)$. In general, we would like that at step i, $gen(d_i) \notin \{p_j\}_{j=1}^{i-1}$. We illustrate this scenario in Fig. 2 (a). Here, we have three explored programs p_1, p_2 and p_3 which are pruned away by the current dataset. The figure also shows the space of remaining candidate programs (in) that can possibly be generated by gen. This space excludes all three generated programs as well as any programs removed as a result of pruning these three. Indeed, existing synthesis approaches that do not deal with noise (e.g., [18]) typically prune the search space as shown in Fig. 2(a).

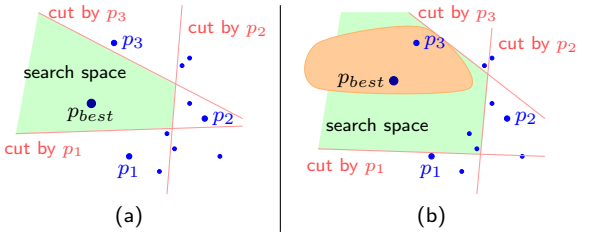

Figure 2. Trimming the space of programs (a) for noise-free synthesis and (b) for synthesis with noise.

Unlike the noise-free setting where a binary criteria for pruning a generated program p exists, when the data contains noise, we cannot immediately decide whether to prune p. The reason is that even though p may make a mistake on a given example, at an intermediate point in the algorithm we may not know whether another, even better program on $\mathcal{D}$ exists. What this uncertainty means is that we may need to keep p in the candidate program space for longer than a single algorithmic iteration. This raises the following question: which programs are kept and which ones are removed from the candidate set?

Reduction of search space To address this question, at every iteration of the synthesis algorithm, we aim to prune some of the generated programs (and conversely, keep the remaining ones). In particular, we keep a generated program p if it is within ϵ distance of p_{best}, that is, $r(\mathcal{D}, p) \leq r(\mathcal{D}, p_{best}) + \epsilon$. In Fig. 2(b), the area shaded with around p_{best} denotes the set of programs within distance ϵ of p_{best}. The space reduction process is illustrated in Fig. 2(b). Here, the first two explored programs p_1 and p_2 fall outside the accepted area and are thus permanently pruned from the candidate space. The score of the latest generated program p_3, however, is within ϵ of p_{best} and is thus kept as a viable candidate to be returned. At this point, the algorithm can return p_3 or keep searching further, hoping to find better scoring programs than p_3. We also require that $\epsilon \geq 0$ ensuring completeness: we always keep the best program p_{best} in the candidate space of programs.

In what follows, we describe dataset samplers which enable pruning of the search space in the manner described above.

2.4 Hard Dataset Sampler (ds^H)

We introduce an instance of the dataset sampler ds used in Algorithm 1 as follows:

Definition 2.1 (Hard dataset sampler)**.** *A hard dataset sampler is a function ds^H such that for $Q \subseteq \mathbb{P}$, $d' = ds^H(Q, min\_size)$, it holds that $\forall p \in Q.\ r(\mathcal{D}, p) \leq r(d', p)$ and $|d'| \geq min\_size$.*

Note that the hard dataset sampler always exists as we can trivially set $d' = \mathcal{D}$. For our synthesizer, we always invoke the hard dataset sampler with $Q = progs$ (the current set of generated programs). The meaning of the hard dataset sampler is that for all programs in Q, the cost on the returned dataset d is higher or equal than on the full dataset $\mathcal{D}$.

In principle, this definition generalizes the concept of providing more examples in noise-free synthesizers. Without noise, $r(d,p)$ simply returns 0 if the program p satisfies all examples in d and 1 if p does not satisfy some example in d. The hard dataset sampler in the noise-free case generates (e.g., by asking questions to the user) a dataset d' such that for all explored programs $progs$, an unsatisfied example is in d' if an unsatisfied example exists in $\mathcal{D}$.

Using the hard dataset sampler, we now state a theorem which ensures that the generated program p_i at step i does not appear in a subset of the explored programs outside certain range beyond p_{best}. That is, p_i cannot be the same as any previously generated p_j that is outside of the area in Fig. 2(b).

Theorem 2.2. *Let $E = \{p_1, \ldots, p_{i-1}\}$ be the set of programs generated up to iteration i of Algorithm 1, where the dataset sampler ds satisfies Definition 2.1.*
If $\epsilon \geq r(d_i, p_{best}) - r(\mathcal{D}, p_{best})$, then $p_i = gen(d_i) \notin E'$ where $E' = \{\, p \in E \mid r(\mathcal{D}, p) > r(\mathcal{D}, p_{best}) + \epsilon \,\}$

Proof: Let $p \in E'$. Because $r(\mathcal{D}, p) > r(\mathcal{D}, p_{best}) + \epsilon$ and from the definition of $\epsilon : \epsilon + r(\mathcal{D}, p_{best}) \geq r(d_i, p_{best})$, we know that $r(\mathcal{D}, p) > r(d_i, p_{best})$. Now, because $d_i = ds(E, \_)$ and ds satisfies Definition 2.1, $r(d_i, p) \geq r(\mathcal{D}, p)$. Then, we have shown that $r(d_i, p) > r(d_i, p_{best})$. Because $p_i = gen(d_i)$ and $gen(d_i) = \arg\min_{p' \in \mathbb{P}} r(d_i, p')$, it follows that $p_i \neq p$. Thus, we prove that $p_i \notin E'$. □

The above theorem is useful when we know the bound ϵ on the best program p_{best}. If we can show a smaller value of ϵ, the areas marked in and around p_{best} will also be smaller and we can then provably trim the search space further. Using this theorem, we can stop the synthesis algorithm as soon as we generate a program p_i that is already in the explored set. As a result of the theorem, we will know that such a p_i is within a distance of at most ϵ from p_{best}.

In Section 3 we consider a case where $\epsilon = 0$ and then Theorem 2.2 provides even stronger guarantees at every step of Algorithm 1: all previously generated candidate programs that are not p_{best} are eliminated from future consideration. For cases where we cannot obtain bounds on the best program p_{best}, we next define a different dataset sampler.

2.5 Representative Dataset Sampler (ds^R)

First, we define a measure of representativeness for dataset d with respect to the full dataset $\mathcal{D}$ on a set of programs $Q \subseteq \mathbb{P}$.

Definition 2.3 (Representativeness measure)**.**

$$repr(Q, \mathcal{D}, d) = \max_{p \in Q} |r(\mathcal{D}, p) - r(d, p)|$$

The measure of representativeness says how close are the costs of the programs in Q on the dataset d with respect to their costs on the full dataset $\mathcal{D}$. The metric is set to the maximum difference in costs since our goal for the dataset d is to be a representative of $\mathcal{D}$ for *all* programs. Then, we define a dataset sampler as follows:

Definition 2.4 (Representative dataset sampler)**.**

$$ds^R(Q, size) = \operatorname*{arg\,min}_{d \subseteq \mathcal{D}, |d| = size} repr(Q, \mathcal{D}, d)$$

Similar to the hard dataset sampler (Definition 2.1), in step i of Algorithm 1 we always use the set of programs $Q = progs$.

Analysis Note that the $repr$ measure is a non-negative function that is minimized by ds^R. If $d' = ds^R(Q, size)$ is such that $repr(Q, \mathcal{D}, d') = 0$ then the produced dataset is perfectly representative. In this case ds^R is a hard dataset sampler, because $\forall p \in Q.\, r(\mathcal{D}, p) = r(d', p)$.

The question then is: why do we attempt to achieve $r(\mathcal{D}, p) = r(d', p)$ instead of $r(\mathcal{D}, p) \leq r(d', p)$ as in Definition 2.1? If we perform our analysis using Theorem 2.2, then we must find as small as possible value $\epsilon \geq 0$ such that $\epsilon \geq r(d_i, p_{best}) - r(\mathcal{D}, p_{best})$. However, instead of minimizing $r(d_i, p_{best}) - r(\mathcal{D}, p_{best})$, in ds^R we minimize $|r(d_i, p_j) - r(\mathcal{D}, p_j)|$ for programs p_j already explored up to step i (i.e. $j \in 1..i-1$). Arguably, the incorrect examples may behave similarly on all programs, but in the general case we cannot find a bound ϵ for Theorem 2.2.

Instead, we give a different argument about eliminating *some*, but not *all* of the already explored programs.

Theorem 2.5. *Let $E = \{p_1, \ldots, p_{i-1}\}$ be the set of programs generated up to iteration i of Algorithm 1.*
Let $p_k = \arg\min_{p \in E} r(\mathcal{D}, p)$ be the best program explored so far.
Let $\delta = repr(Q, \mathcal{D}, d_i)$ be the representativeness measure of d_i.
Then $p_i = gen(d_i) \notin E'$ where:
$E' = \{\, p \in E \mid r(\mathcal{D}, p) > r(\mathcal{D}, p_k) + 2\delta \,\}$

Proof: Let $p \in E'$. From Definition 2.3, $|r(d_i, p) - r(\mathcal{D}, p)| \leq \delta$ and thus $r(d_i, p) \geq r(\mathcal{D}, p) - \delta$. Using a similar argument for p_k, we obtain that $r(d_i, p_k) \leq r(\mathcal{D}, p_k) + \delta$. Then $r(d_i, p_k) < r(d_i, p)$ and thus $p \neq \arg\min_{p' \in \mathbb{P}} r(d_i, p')$ and $p \neq p_i = gen(d_i)$. □

Note that the set E' has the same shape as in Theorem 2.2 except that here we consider p_k (best program so far) instead of p_{best} (best program globally), and instead of ϵ we have 2δ.

What this theorem says is that the programs $E' \subseteq E$ that were already generated and are worse than $p_k \in E$ by more than twice the representativeness measure δ of the dataset d_i cannot be generated at step i of Algorithm 1. We can also instantiate the condition for cutting the space discussed earlier: $r(\mathcal{D}, p) \leq r(\mathcal{D}, p_{best}) + \epsilon$ and visualize Theorem 2.5 in Fig. 2(b) as follows: take $p_3 = p_k$ and let $x = r(p_3, \mathcal{D}) - r(p_{best}, \mathcal{D})$ be the distance between p_k and p_{best}. Then take $\epsilon = x + 2\delta$. Thus, programs p_1 and p_2 are worse than p_{best} by more than ϵ and are permanently removed from the program search space.

In case $\delta = 0$, we can see that all programs in E worse than the (locally) best program $p_k \in E$ will be eliminated. Still, this is a weaker guarantee than for the case where $\epsilon = 0$ in Theorem 2.2. Later we will show that ds^R works well in practice, but in general it is theoretically possible that Algorithm 1 with ds^R makes no progress until a dataset of a certain size is accumulated.

2.6 Cost Functions and Regularization

So far, we have placed few restrictions on the cost function r and we defined the synthesis problem to be such that lower cost is better. We now list concrete cost functions considered later in the paper:

- $num\_errors(d, p)$ returns the number of errors a program p does on a dataset of examples d.
- $error\_rate(d, p) = \frac{num\_errors(d,p)}{|d|}$ is the fraction of the examples with an error. A related metric used in machine learning is the *accuracy*, which is $1 - error\_rate$.
- Other measures weigh the errors done by the program p on the dataset d according to their kind (e.g., perplexity is one possible such measure).

Regularization We also use a class of cost functions known as regularized cost metrics. If r is a cost metric, its regularized version is $r_{reg}(d, p) = r(d, p) + \lambda \cdot \Omega(p)$. Here, λ is a real-valued constant and $\Omega(p)$ is a function referred to as a regularizer. The goal of the regularizer is to penalize programs which are too complex. Note that the regularizer *does not* have access to the dataset d, but only to the given program p. In practice, using regularization means we may not necessarily return the program with the least number of errors if a much simpler program with slightly more errors exists. In Section 5.1, we justify the use of regularization in the context of empirical risk minimization.

3. The Case of Bounded Noise

In this section, we show how to instantiate Algorithm 1 for the case where we can define a bound on the noise that the best program p_{best} exhibits.

Definition 3.1 (Noise Bound)**.** *We say that ϵ_k is a noise bound for samples of size k if for the program p_{best}:*

$$\forall d \subseteq \mathcal{D}.|d| = k \Rightarrow \epsilon_k \geq r(d, p_{best}) - r(\mathcal{D}, p_{best})$$

For example, if $r \triangleq error\_rate$ and $\mathcal{D}$ contains at most one incorrect example, then $\epsilon_{10} = 0.1$ is a noise bound, because for any sample $d \subseteq \mathcal{D}$ of size $|d| = 10$, the error rate is at most 0.1, but the error rate on the entire dataset $\mathcal{D}$ may be lower. Another interesting case is if $r \triangleq num\_errors$ and p_{best} has at most K errors on the examples in $\mathcal{D}$. Then a noise bound for any k is $\epsilon_k = 0$ because no dataset $d \subseteq \mathcal{D}$ has more errors than the full dataset $\mathcal{D}$. Note that using regularization is an orthogonal issue and does not affect the noise bound, because the regularizer $\Omega(p_{best})$ cancels out in the inequality of Definition 3.1.

We can easily instantiate Theorem 2.2 when a noise bound ϵ_k is available by setting $\epsilon = \epsilon_k$ in the theorem's precondition $\epsilon \geq r(d_i, p_{best}) - r(\mathcal{D}, p_{best})$ (here, $k = |d_i|$).

Derived termination criterion Using Theorem 2.2 and the hard dataset sampler allows us to derive a possible termination criterion for Algorithm 1. In particular, if our desired program $p_{desired}$ is such that $r(\mathcal{D}, p_{desired}) \leq r(\mathcal{D}, p_{best}) + \epsilon_{desired}$ (i.e., it is worse than the best program by at most $\epsilon_{desired}$), then if the following stopping criterion triggers:

$$\mathit{found\_program}(p_i) \triangleq (p_i \in progs) \wedge \epsilon_{|d_i|} \leq \epsilon_{desired}$$

the algorithm will produce $p_{desired}$.

This criterion follows from Theorem 2.2 because if a program $p_i \in progs$ (i.e., it was already explored previously), then p_i was not excluded from the search space and thus it must be that $r(\mathcal{D}, p_i) \leq r(\mathcal{D}, p_{best}) + \epsilon_{|d_i|}$ (i.e., $p_i \in$).

Bound on the number of errors We next consider an interesting special case where we know that for the best program p_{best}, there are at most K incorrect examples in $\mathcal{D}$ that it does not satisfy. Note that we only need to know a bound, not the exact number of errors the best program makes. In this case, we propose to use the following cost function:

$$r_K(d, p) = \min(num\_errors(d, p), K + 1)$$

That is, we count the number of unsatisfied examples and cap the cost at $K + 1$, thus we do not distinguish programs or datasets with more than K errors. Since we know that the best program has at most K errors, in Definition 3.1, we can show that $\epsilon_k = 0$ (for any k) is a valid bound. In this case, we can also obtain a stopping criterion with $\epsilon_{desired} = 0$ by using:

$$\mathit{found\_program}(p_i) \triangleq p_i \in progs$$

Thus, we get the stronger guarantees as in Fig. 2 (a) and ensure that upon termination the algorithm produces the program p_{best}.

Discussion We note the meaning of the hard dataset sampler when $r \triangleq num\_errors$. Then, the requirement for a program p of $r(d, p) \geq r(\mathcal{D}, p)$ from Definition 2.1 means that sample d_i must contain *all* errors in $\mathcal{D}$ – naturally, this may lead to a dataset that is too big.

By knowing the cap K and using a cost function $r \triangleq r_K$, we restrict the sampler to always include exactly $K + 1$ unsatisfied examples in order to eliminate p_i as a candidate for the next step, because we know that p_{best} has at most K errors. In this setting, knowing a bound K in advance allows the dataset sampler to insert only the necessary number of samples in the small datasets d_i.

In the next section, we present an implementation of a synthesizer for bitstream programs with a bound on the number of errors as considered here.

4. BitSyn: Bitstream Programs from Noisy Data

In this section we show how to instantiate the approach presented earlier to the problem of building a programming-by-example (PBE) engine able to deal with up to K incorrect input/output examples in its input data set for the best program p_{best}. To illustrate the process, we chose the domain of bitstream programs as they are well understood and easy to implement, allowing us to focus on studying the behavior of Algorithm 1 in a clean manner. We believe many synthesis engines are good candidates for being extended to deal with noise (e.g., synthesis of floating point functions [28] or data extraction [20]).

The setting We consider two scenarios: (1) the dataset $\mathcal{D}$ is obtained dynamically and the noise is bounded (i.e., up to K errors), and (2) the dataset $\mathcal{D}$ is present in advance and may contain an unknown number of errors. Interestingly, the second scenario is useful beyond synthesizing programs, in this case, for anomaly detection.

We created a synthesizer called BitSyn that generates loop-free bit manipulating code from input/output examples. The programs generated by BitSyn are similar to those produced in Jha et al [18]. We use a library of instructions for addition, bitwise logical operations, equality, less than comparison and combine them into a program that takes 32-bit integers as input and outputs one 32-bit integer. The program may use registers to store intermediate values. The goal of the synthesizer is to take a number of input/output examples and generate a program.

4.1 Program Generator with Errors

A key quality of BitSyn is that it includes a program generator that may not satisfy all provided input/output examples. This may serve multiple purposes as we discuss later. Let us consider the following example input/output pairs:

$$d_1 = \{\{2 \to 3\}, \{5 \to 6\}, \{10 \to 11\}, \{15 \to 16\}, \{-2 \to -2\}\}$$

All examples except for $\{-2 \to -2\}$ describe a function that increments its input. A problem with existing PBE engines in this case is that they *succeed* in generating a program even if it was not the desired one, e.g. by producing the following code:

$$p_a = \texttt{return input + 1 + (input >> 8)}$$

Note that providing more examples would not necessarily help discover or solve this problem. The user may in fact get *lucky* by getting into a situation where the synthesizer fails to produce a program, however if the hypothesis space of programs (e.g., which operators is the engine allowed to use) is not very constrained, this program can overfit to the incorrect examples. Later we quantify this problem. The problem of overfitting to the data (i.e., input/output examples) occurs in multiple mathematical and machine learning problems where the provided specification does not permit exactly one solution, for example when dealing with noise.

We combat overfitting by introducing regularization to the cost. We define a function $\Omega\colon \mathbb{P} \to \mathbb{R}^+$ that punishes overly complex programs p by returning the number of instructions used. For our example, $\Omega(p_a) = 3$ since p_a has three instructions (two `+` and one `>>`). Then, we create a program generator that minimizes:

$$r_{reg}(d, p) = error\_rate(d, p) + \lambda \cdot \Omega(p).$$

The value $\lambda \in \mathbb{R}$ is a *regularization constant* that we choose in evaluation. The higher the regularization constant is, the more importance we place on producing small programs. In our example, if $\lambda > 0.1$, the cost r_{reg} of the following p_b program will be lower

	Number of instructions	Number of errors (K)																
		0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	9
Program	↓	Number of input/output examples needed										Synthesis time (seconds)						
P1	2	4	4	10	7	9	11	14	16	17	22	1.11	1.17	1.98	1.51	1.80	7.33	102.76
P2	2	5	6	6	7	11	12	15	19	20	22	1.21	1.48	1.79	2.70	2.45	12.96	72.37
P3	3	4	4	9	10	8	13	15	16	17	21	1.75	1.81	4.42	8.63	9.20	40.62	156.09
P4	2	2	4	7	8	9	10	13	15	17	19	1.05	1.19	1.56	3.07	4.01	11.34	12.30
P5	2	3	3	9	9	10	10	14	16	20	22	1.08	1.10	1.84	3.45	9.38	11.64	139.75
P6	2	4	5	10	9	10	11	13	17	20	22	1.18	1.51	2.70	3.50	10.60	12.44	91.49
P7	3	5	5	7	9	11	12	15	19	20	22	1.80	2.20	2.77	5.15	12.65	21.62	117.16
P8	3	5	5	10	10	8	12	13	16	20	19	1.90	2.44	4.41	4.47	5.15	26.62	41.46
P9	3	3					timeout					2.58	timeout	timeout	timeout	timeout	timeout	timeout

Table 1. Number of input/output examples needed by BITSYN to synthesize the correct program (program taken from [18, 35]) depending on the number of errors in the examples as well as the synthesis time with the respective number of errors.

than the cost of p_a on the dataset d_1:

$$p_b = \texttt{return input + 1}$$

Implementation We implemented BITSYN using the Z3 SMT solver [9]. At each query to the SMT solver, we encode the set of input/output examples $d = \{x_i\}_{i=1}^{n}$ in a formula based on the techniques described in [18]. In each formula given to the SMT solver, we encode the length of the output program (called Ω later) and we additionally encode a constraint for the number of allowed errors. Let χ_i be a formula that is true iff example $x_i \in d$ is satisfied. Then, to encode a constraint that allows up to T errors, we must satisfy the following formula:

$$\Upsilon \equiv T \geq \sum_{i=1}^{n} \text{if } \chi_i \text{ then 0 else 1}$$

To find the best scoring solution, we make multiple calls to the SMT solver to satisfy Υ by iterating over the lengths of programs and the number of allowed incorrect input/output examples T ordered according to the cost of the solution and then return the first obtained satisfiable assignment of instructions.

4.2 Case 1: Examples in $\mathcal{D}$ are provided dynamically

A common scenario for programming-by-example engines is when the input/output examples are obtained dynamically, either by interactively asking the user or by querying an automated reasoning engine. Ultimately, this means that the entire dataset $\mathcal{D}$ is not directly observable by the program generator (in fact, the dataset may be infinite). In this case, the synthesizer starts with a space of candidate programs and narrows that space by dynamically obtaining more examples.

For this setting, we designed a *hard dataset sampler* using the cost function r_K as described in Section 3. Our dataset sampler attempts to create a dataset d_{i+1} with $K+1$ unsatisfied examples for each program in the set of candidate programs explored so far $progs = \{p_j\}_{j=1}^{i-1}$. Generally, incorrect examples can be readily obtained automatically from an SMT solver (or another tool). When the tool is used interactively, the user needs to answer questions until the desired number of errors is reached (that is, here, the user takes an active part in the work of the dataset sampler).

Evaluation Our goal was to check if BITSYN can synthesize the correct program in the presence of errors. Towards this, we implemented a simulated user that provides examples using a hard dataset sampler with a known bound on the incorrect examples. We aimed to answer the following research questions: (1) up to how many errors does BITSYN scale for synthesizing solutions?, and (2) how many (more) examples does BITSYN need in order to compensate for the incorrect examples? For our evaluation, we took a number of programs from [18], on which an existing synthesizer without noise could generate solutions within a few seconds. These are the programs P1-P9 from the Hacker's Delight book [35], also evaluated in previous works [13, 18].

We summarize our results in Table 1. For each program, we tried settings with different numbers of incorrect examples. We first supplied the incorrect examples and then started supplying correct examples. In each cell in the left part of Table 1, we list the *total* number of input/output examples needed to obtain the correct result. In the right part of Table 1, we list the time needed to complete each synthesis task. Our results can be summarized in two areas: (1) overall, adding incorrect input/output examples complicates the program synthesis task. For tasks $P1-P8$, each synthesis task completes within our timeout of 300 seconds. Task $P9$ did not scale since it needs bit-shift operations and their presence leads to difficult formulas for the Z3 solver, and (2) the number of necessary input/output examples overall increases with an increased number of errors, but only slightly. Further, in some cases, the number of needed examples stays constant when introducing more errors. This motivated us to ask the question explored next, which is whether the tool is useful beyond synthesis, but also for detecting incorrect examples (i.e., anomaly detection).

4.3 Case 2: All examples in $\mathcal{D}$ are given in advance

As a side question not directly related to Algorithm 1, we wanted to understand how well the regularized program generator in BITSYN detects incorrect examples in the setting where the dataset $\mathcal{D}$ is fully available. Here, we provide all our examples to a regularized generator and ask if the unsatisfied examples are exactly the incorrect ones. Such a setting of finding a model that describes data and then detects outliers in the model is called anomaly detection [6]. Note that our approach is very different from recent work [5] which considers a more restricted case where the program is already fully available before the process of anomaly detection starts.

Evaluation We evaluate the anomaly detection capability of BITSYN depending on the regularization constant λ and the number of samples present in $\mathcal{D}$. For this experiment, we created data sets $\mathcal{D}$ of various sizes and introduced an incorrect example in each of them. Then, we looked at how often the synthesized program did not satisfy exactly the incorrect examples. We summarize the results in Fig. 3. The figure visualizes the conditions under which the anomaly detection is effective on our test programs (P1-P9). Every cell in the diagram of Fig. 3 says how often a given output occurs. The cells with ▨ denote that the synthesizer successfully satisfied the provided examples, including the incorrect one. This case typically occurs with no or low regularization. This means that synthesizers that fail to take noise into account will easily overfit to the incorrect example and return valid, but incorrect programs.

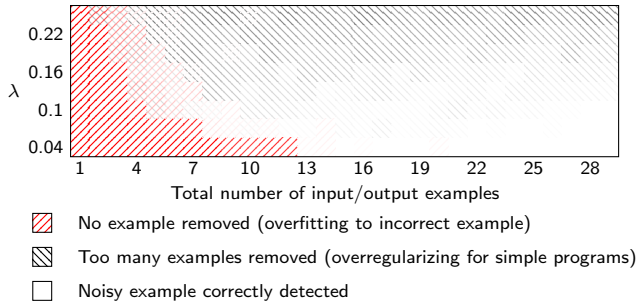

Figure 3. Ability of BITSYN to detect an incorrect example for programs (P1-P9) depending on total number of examples and regularization constant λ.

On the other hand, using too much regularization may bias BITSYN towards producing too simple programs that do not satisfy even some of the correct examples. We denote this with the pattern ▨. The darker a pattern, the more often the corresponding issue occurs. In the white areas of the graph, BITSYN reliably discovers the incorrect input/output example. For this to happen reliably, our results show that we need a dataset with more than 10 examples and a regularization constant λ between 0.05 and 0.1.

5. Handling Unbounded Noise

In this section, we investigate the case where we have no bound on the amount of noise in the dataset $\mathcal{D}$. The key idea is to view the problem of learning a program from a noisy dataset as an empirical risk minimization (ERM) task over a discrete search space of programs. In fact, our approach can be seen as a new way of performing approximate ERM on a discrete search space.

Next, we first review the concept of ERM and the guarantees provided by statistical learning theory. In this case, we restrict $\mathcal{D}$ to be a set obtained from a probability distribution $\mathcal{S}$ over examples $\mathcal{W}$, that is, $\mathcal{D} \subseteq \mathcal{W}$. In what follows, our cost function r is set to be the empirical risk function (discussed below). Then, in Section 5.2, we present our (novel) approach to performing approximate ERM.

5.1 Empirical Risk Minimization

Let $\ell\colon \mathbb{P} \times \mathcal{W} \to \mathbb{R}_{\geq 0}$ be a function, such that for $p \in \mathbb{P}$ and $w \in \mathcal{W}$, $\ell(p, w)$ quantifies the loss (amount of inaccuracy) when applying program p to example w. Later in Section 6.4 we show an example of a loss function. Our task is to synthesize a program $p^* \in \mathbb{P}$ that minimizes the expected loss on example w drawn i.i.d. from distribution $\mathcal{S}$ (we assume w.l.o.g. $\mathcal{S}(w) > 0$ for all $w \in \mathcal{W}$). I.e., we seek to minimize the risk (defined in terms of the expectation of the function):

$$R(p) = \mathbb{E}_{w \sim \mathcal{S}}[\ell(p, w)],$$

i.e. find the program:

$$p^* = \arg\min_{p \in \mathbb{P}} R(p)$$

As a concrete example, $\ell(p, w)$ could be 0 if p produces the "correct" output for w and 1 if it is incorrect. In this setting, $R(p)$ corresponds to the expected number of mistakes that p makes on random example w. Moreover, $R(p) = 0$ iff it is "correct" (i.e., produces the correct behavior on all examples in $\mathcal{S}$). As another example, p could produce real-valued outputs, and $\ell(p, w)$ could measure the squared error between the correct output and the actual output.

There are two problems with computing p^* using the above approach. First, since $\mathcal{S}$ is unknown, the risk $R(p)$ cannot even be evaluated. Second, even if we could evaluate it, finding the best program is generally intractable. To address these concerns, we make two assumptions. First, we assume we are given a *sample* $\mathcal{D}$ of examples drawn i.i.d. from $\mathcal{S}$. We can approximate the risk $R(p)$ by the *empirical risk*, i.e.,

$$r_{emp}(d, p) = \frac{1}{|d|} \sum_{w \in d} \ell(p, w)$$

Then, we assume (for now) that we have an "oracle", an algorithm that can solve the *empirical risk minimization (ERM)* problem

$$p_{best} = \arg\min_{p \in \mathbb{P}} r_{emp}(\mathcal{D}, p).$$

The above equation is in fact an instance of the problem stated in Section 2.1.

Guarantees Standard arguments from statistical learning theory [21] now guarantee that, for any $\varepsilon, \delta > 0$, if our dataset of examples $\mathcal{D}$ is big enough with respect to the space of programs ([23, Chapters 7.3,7.4]), then it holds for the solution p_{best} that $R(p_{best}) \leq R(p^*) + \varepsilon$, with probability at least $1 - \delta$ (over the random sampling of the dataset) [23]. Hence, the best-performing program on the dataset is close (in risk) to the best program over all of $\mathcal{S}$. This is because under the above conditions, the empirical risk approximates the true risk uniformly well, i.e., for all $p \in \mathbb{P}$ it holds that $|R(p) - r_{emp}(\mathcal{D}, p)| \leq \varepsilon$.

Regularization ERM solution can overfit if the dataset is not large enough [21]. Overfitting means that $R(p^*) \ll R(p_{best})$, i.e., the ERM solution has much higher risk than the optimal program. This often happens when the space of programs $\mathbb{P}$ under consideration is very complex, i.e., the solution could overfit by "memorizing" the training data and fail on other examples. As a remedy, a common approach is to apply *regularization*: i.e., instead of minimizing the empirical risk, one modifies the objective function by:

$$r_{reg}(d, p) = r_{emp}(d, p) + \lambda \Omega(p)$$

Hereby, $\Omega : \mathbb{P} \to \mathbb{R}_{\geq 0}$ is a function (called *regularizer*), which prefers "simple" programs. For example, $\Omega(p)$ could count the number of instructions in p, i.e., a program is "simpler" if it contains fewer instructions. Note that the regularizer does not depend on the data set, it only depends on the program. The *regularization parameter* λ, which controls the strength of our simplicity bias, is usually optimized over using a process called *cross-validation*. In the following, we use the notation r_{emp} and refer to minimizing it as ERM, whether or not we are applying regularization.

5.2 Using Representative Dataset Sampler

The complexity of solving ERM is heavily dependent on the size of the dataset $\mathcal{D}$. This is due to the fact that evaluating r_{emp} or r_{reg} gets more expensive (since we need to sum over more examples).

To enable ERM on the large dataset $\mathcal{D}$, we use Algorithm 1 with a representative dataset sampler ds^R and a program generator that solves ERM on small datasets. Our goal here is to sample subsets $d_1, d_2, \ldots d_m \subseteq \mathcal{D}$, with the property that solving ERM on these subsets leads to good solutions in terms of the (intractably large) dataset $\mathcal{D}$. Our goal upon termination of the synthesis procedure is to obtain a program p_m for which:

$$r_{emp}(\mathcal{D}, p_m) \in [r_{emp}(\mathcal{D}, p_{best}), r_{emp}(\mathcal{D}, p_{best}) + \varepsilon]$$

Then, recall that $R(p_{best}) \leq R(p^*) + \varepsilon$ to obtain that the resulting solution p_m will have risk at most 2ε more than p^*. On the other hand, by exploiting the fact that we can solve ERM much faster on small datasets d_i, we can find such a solution much more efficiently than solving the ERM problem on the full dataset $\mathcal{D}$ (which can also be practically infeasible). This instantiation is a new approach of performing approximate ERM over discrete search spaces.

6. DEEPSYN: Learning Statistical Code Completion Systems

In this section we present a new approach for constructing statistical code completion systems. Such statistical code completion systems are typically trained on a large corpus of programs (i.e., "Big Code") and are used to generate a (probabilistically) likely completion of a given input program. Currently, the predictions made by existing systems (e.g., [16, 24, 30]) are "hard-wired" (see Section 8 for further discussion), limiting their expressiveness and precision and requiring changes to this hard-wired strategy for different kinds of predictions. The approach presented here cleanly generalizes these existing approaches.

While not obvious, we will see that the problem of synthesizing a program from noisy data appears in this setting as well, and thus the general framework of synthesis with noise discussed so far also applies here. However, unlike the first-order setting described in Section 4 where the data is simply a set of input-output examples and the learned program tries to explain these examples and predict new examples, the learned program in this section is second-order. This means that the learned program does not predict its output directly from the input, but instead is used as part of a probabilistic model that performs the final prediction seen by the developer.

6.1 Preliminaries

To fix terminology, we will refer to the program that is to be completed as a *tree* (a shortcut for Abstract Syntax Trees). The reason we choose trees as a representation of the program is because trees provide a reasonable way to navigate over the program elements. We begin with a standard definition of context-free grammars (CFGs), trees and parse trees.

Definition 6.1 (CFG). *A context-free grammar (CFG) is the quadruple (N, Σ, s, R) where N is a set of non-terminal symbols, Σ is a set of terminal symbols, $s \in N$ is a start symbol, R is a finite set of production rules of the form $\alpha \to \beta_1 ... \beta_n$ with $\alpha \in N$ and $\beta_i \in N \cup \Sigma$ for $i \in [1..n]$.*

In the whole exposition, we will assume that we are given a fixed CFG: $G = (N, \Sigma, s, R)$.

Definition 6.2 (Tree). *A tree T is a tuple (X, x_0, ξ) where X is a finite set of nodes, $x_0 \in X$ is the root node and $\xi \colon X \to X^*$ is a function that given a node returns a list of its children. A tree is acyclic and connected graph: every node except the root appears exactly once in all the lists of children. Finally, no node has the root as a child.*

Definition 6.3 (Partial parse tree). *A parse tree is a triple (T, G, σ) where $T = (X, x_0, \xi)$ is a tree, $G = (N, \Sigma, s, R)$ is a CFG, and $\sigma \colon X \to \Sigma \cup N$ attaches a terminal or non-terminal symbol to every node of the tree such that: if $\xi(x) = x_{a_1} ... x_{a_n}$ $(n > 1)$, then $\exists(\alpha \to \beta_1 ... \beta_n) \in R$ with $\sigma(x) = \alpha$ and $\forall i \in 1..n . \sigma(x_{a_i}) = \beta_i$.*

Note that the condition for a partial parse tree requires that the tree follows the grammar production rules, but does not require all leaves to be terminal symbols. Let the set of all partial parse trees be PT. Next, we define what is a tree completion query.

Definition 6.4 (Tree completion query). *A tree completion query is a triple $(ptree, x_{comp}, rules)$ where $ptree = (T, G, \sigma)$ is a partial tree (with $T = (X, x_0, \xi)$), $x_{comp} \in X$ is a node labeled with a non-terminal symbol where a completion will be performed, and $rules = \{\sigma(x_{comp}) \to \boldsymbol{\beta}^i\}_{i=1}^n$ is the set of available rules that one can apply at the node x_{comp}.*

Using the above definitions, we can now state the precise problem that is solved by this section.

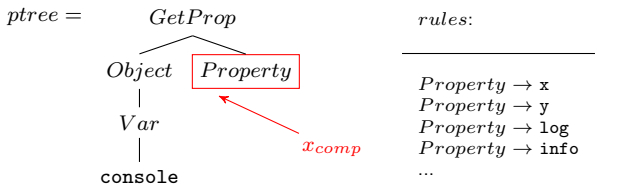

Figure 4. A tree completion query $(ptree, x_{comp}, rules)$ corresponding to completion for the code: "`console`".

Problem statement The code completion problem we are solving can now be stated as follows:

Given a query, select the most likely rule from the set of available rules and complete the partial tree ptree with it.

For the completions we consider, the right hand side $\boldsymbol{\beta}$ of each rule is a terminal symbol (e.g., a single API). In principle, one can make longer completions by iteratively chaining smaller ones.

Example: Field/API completion Consider the following partial JavaScript code "`console.`" which the user is interested in completing. The goal of a completion system is to predict the API call `log`, which is probably the most likely one for `console`. Now consider a simplified CFG that can parse such programs (to avoid clutter, we only list the grammar rules):

$$
\begin{array}{rcl}
GetProp & \to & Object\ \ Property \\
Object & \to & Var \mid GetProp \\
Var & \to & \texttt{console} \mid \texttt{document} \mid \text{... (other variables)} \\
Property & \to & \texttt{info} \mid \texttt{log} \mid \text{... (other properties incl. APIs)}
\end{array}
$$

The tree completion query for this example is illustrated in Fig. 4.

6.2 Our Method: Second-order learning

The key idea of our solution is to *synthesize a program which conditions the prediction*. That is, rather than statically hard-wiring the context on which the prediction depends on as in prior work (e.g., [16, 30]), we use the program to *dynamically* determine the context for the particular query. For our example, given a partial tree $ptree$ and a position x_{comp}, the program determines that the prediction of the API call should depend on the context `console`.

In our setting, a context $c \in Context$ is a sequence ranging over terminal and non-terminal symbols seen in the tree, as well as integers. That is, $Context = (N \cup \Sigma \cup \mathbb{N})^*$. We next describe our method in a step-by-step manner and then elaborate on some of the steps in more detail.

Step 1: Learn a conditioning program $p_{\approx best}$ As a first step, we will begin by learning the best (conditioning) program $p_{\approx best}$. Indeed, it is in this first step of learning where the key concepts of dataset sampling and program generation discussed earlier, arise.

Let $d = \{X^i, Y^i\}_{i=1}^n$ be the training dataset of queries $X^i = (ptree^i, x_{comp}^i, rules)$ along with their corresponding completions $Y^i \in rules$. We assume that all examples in d solve the same task (e.g., API completion) and thus they share $rules$. The goal of this step is to synthesize the (approximately) best conditioning program $p_{\approx best} \in PT \times X \to Context$ that given a query returns the context on which to condition the prediction. For instance, for the example in Fig. 4, a possible program p could produce $p(ptree, x_{comp}) = [\texttt{console}]$. In Section 6.3, we present a domain-specific language from which the conditioning program is drawn while in Section 6.4 we elaborate on this step in detail.

Step 2: Learn a probabilistic model $P(rule \mid context)$ After $p_{\approx best}$ is learned, we use the resulting program to train a probabilistic model. Given our training dataset d as described above, we

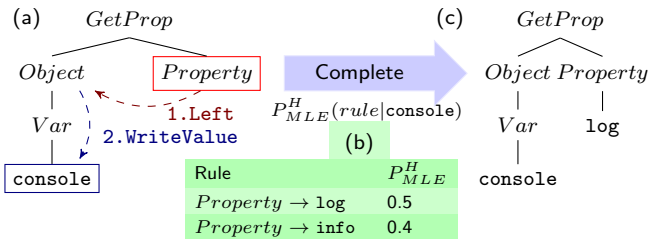

Figure 5. (a) TCOND program p^a = `Left WriteValue` executed on a partial tree producing [`console`], (b) rules with their probabilities conditioned on [`console`], (c) the final completion.

```
   Ops ::=  ϵ | Op Ops
    Op ::= WriteOp | MoveOp
WriteOp ::= WriteValue | WritePos | WriteAction
MoveOp ::= Up | Left | DownFirst | DownLast | PrevDFS |
           PrevLeaf | PrevNodeType | PrevActor
```

Figure 6. The TCOND language for extracting context from trees.

next apply $p_{\approx best}$ to every query in the training data, obtaining a new data set (here $Q_i = (ptree^i, x^i_{comp})$):

$$H(d, p_{\approx best}) = \{(p_{\approx best}(Q_i), Y^i) \mid ((Q_i, rules), Y^i) \in d\}$$

The derived data set consists of a number of pairs where each pair $\{(c_i, r_i)\}$ indicates that rule r_i is triggered by context $c_i \in Context$. Based on this derived set, we can now train a probabilistic model using MLE training (maximum likelihood estimation) which estimates the true probability $P(r \mid c)$. The MLE estimation is standard and is computed as follows:

$$P^H_{MLE}(r \mid c) = \frac{|\{i \mid (c_i, r_i) \in H, c_i = c, r_i = r\}|}{|\{i \mid (c_i, r_i) \in H, c_i = c\}|}$$

The MLE simply counts the number of times rule r appears in context c and divides it by the number of times context c appears. As we will see later in Section 6.4, MLE learning as described above is also used in step 1.

Step 3: Dynamic prediction Once we have learned the conditioning program $p_{\approx best}$ and the probabilistic model $P(rule \mid context)$, we use *both* components to perform prediction. That is, given a query $(ptree, x_{comp}, rules)$, we first compute the context $ctx = p_{\approx best}(ptree, x_{comp})$. Once the context is obtained, we can use the trained probabilistic model to select the best completion (i.e., the most likely rule) from the set of available rules:

$$rule = \arg\max_{r \in rules} P^H_{MLE}(r \mid ctx)$$

To illustrate the prediction on an example, consider the query shown in Fig. 5 (a) (this is the same query as in Fig. 4, repeated here for convenience). In this example, the program $p_{\approx best}$ consists of two instructions: one moves left in the tree and the other one writes the element at the current position (we will see exact semantics of these instructions in Section 6.3). When applied to the given query, the program produces the context ctx consisting of the sole symbol `console`. Once the context is obtained, we can simply look up the probabilistic model P^H_{MLE} to find the most likely rule given the context (we list some of the rules and their probability in Fig. 5 (b)). Finally, we complete the query as shown in Fig. 5 (c).

6.3 TCOND: Domain Specific Language for Tree Contexts

We now present a domain specific language, called TCOND, for expressing the conditioning function p. The language is loop-free and is summarized in Fig. 6. We next provide an informal introduction to TCOND. Every statement of the language transforms a state $v \in PT \times X \times Context$. The state contains a partial tree, a position in the partial tree and the (currently) accumulated context. The partial tree is not modified during program execution but position and the context may be.

The language has two types of instructions: movement (`MoveOp`) and write instructions (`WriteOp`). The program is executed until the last instruction and the accumulated context is returned as the result of the program.

Move instructions change the node in a state as follows: $(ptree, node, ctx) \xrightarrow{\texttt{MoveOp}} (ptree, node', ctx)$. Depending on the operation, $node'$ is set to either the node on the left of $node$ (for `Left`), to the parent of $node$ (for `Up`), to the first child of $node$ (for `DownFirst`), to the last child of $node$ (for `DownLast`), to the last leaf node in the tree on the left of $node$ (for `PrevLeaf`), to the previous node in depth-first search traversal order of the tree (for `PrevDFS`). When $node$ is a non-terminal symbol, the `PrevNodeType` instruction moves to the previous non-terminal symbol of the same type that is left of $node$.

Write instructions update the context of a state as follows: $(ptree, node, ctx) \xrightarrow{\texttt{WriteOp}} (ptree, node, ctx \cdot x)$, where depending on the instruction, different value x is appended to the context. For the `WriteValue` instruction, the value of the terminal symbol below $node$ is written (if there is one, otherwise $x = -1$). For the `WritePos` instruction, if $parent$ is the parent node of $node$, then x is set to the index of $node$ in the list of the children of $parent$.

The `PrevActor` and `WriteAction` instructions use a simple lightweight static analysis. If $node$ denotes a memory location (field, local or global variable, that we call actor), `PrevActor` moves to the previous mention of the same memory location in the tree. Our static analysis ignores loops, branches and function calls thus previous here refers to the occurrence of the memory location on the left of $node$ in the tree. `WriteAction` writes the name of the operation performed on the object referred by $node$. In case the object referred by $node$ is used for a field access, `WriteAction` will write the field name being read from the object. In case the object $node$ is used with another operation (e.g., `+`), the operation will be recorded in the context. An example of a program execution was already discussed for the example in shown in Fig. 5 (more examples can be seen in Fig. 9 (c), discussed later).

For a program $p \in$ `Ops`, we write $p(ptree, x_{comp}) = ctx$ to denote that $(ptree, x_{comp}, \epsilon) \xrightarrow{p} (ptree, node', ctx)$.

6.4 Learning $p_{\approx best}$

We next describe step 1 of our method in greater detail. The key objective of this step is to synthesize a conditioning program $p_{\approx best}$. In fact, as we will see in Section 7.4, several existing code completion systems [16, 30] can be seen as hard-wired programs p for specific tasks. We now define what it means for a synthesized program p to perform well (i.e. we define our cost function r) and then we describe the program generator and the representative dataset sampler ds^R that we use for Algorithm 1.

Building a probabilistic model As described earlier, we are given a dataset $d = \{X^i, Y^i\}_{i=1}^n$ of queries X^i where $X^i = (ptree^i, x^i_{comp}, rules)$, along with their corresponding completions $Y^i \in rules$. For a program p and a dataset d we can then derive a new data set by applying the program to every query in d, obtaining the resulting context, and storing that context and the

given prediction together as a tuple, i.e., $Q_i = (ptree^i, x^i_{comp})$:

$$H(d,p) = \{(p(Q_i), Y^i) \mid ((Q_i, rules), Y^i) \in d\}$$

Let us partition the given dataset d into two non-overlapping parts – d_{train} and d_{eval}. We then obtain the derived (training) set $H^t(d,p) = H(d_{train}, p)$ from which we build a probability distribution $P^{H^t(d,p)}_{MLE}$ as outlined earlier.

Scoring the probabilistic model To evaluate the probabilistic model, we use the measure of perplexity of the derived (evaluation) set $H^e(d,p) = H(d_{eval}, p)$ on the learned model $P^{H^t(d,p)}_{MLE}$. Log-perplexity is a measure of how many bits we need to encode the evaluation data with the model from the training data and provides insight not only on the error rate, but also how often a result is at a high rank and is produced with high confidence. In an empirical risk minimization setting, we use the log-perplexity to define the loss function on a single example $(ctx, rule)$:

$$\ell_{perp}(p, (ctx, rule)) = -\log_2 P^{H^t(d,p)}_{MLE}(rule \mid ctx)$$

Based on this loss function, we now define regularized empirical risk $r_{regperp}$ as in Section 5.1:

$$r_{regperp}(d,p) = \frac{1}{|H^e(d,p)|} \sum_{w \in H^e(d,p)} \ell_{perp}(p,w) + \lambda \cdot \Omega(p),$$

where Ω is a regularizer function that returns the number of instructions in p. For all our experiments, we use $\lambda = 0.05$.

Program generators and dataset samplers Using the cost function $r_{regperp}(d,p)$, we can now define the rest of the components and plug them into Algorithm 1. For our implementation, we use approximate versions of a program generator and a representative dataset sampler realized with random mutations and genetic programming.

An approximate *program generator* $gen_{\approx}$ takes a dataset d_i and an initial program p_{i-1}. Then, the program generator keeps a list of candidate programs and iteratively updates the list in the following manner. First, $gen_{\approx}$ takes one candidate program, then performs random mutations on the instructions of the program, scores the modified program on the given dataset d_i and then it adds it to the list. Using a genetic-programming like procedure, $gen_{\approx}$ randomly removes from the list candidate programs that score worse than another candidate program. After a fixed number of iterations, $gen_{\approx}$ returns the best scoring program $p_i \approx \arg\min_{p \in \mathbb{P}} r_{regperp}(d_i, p)$.

Our approximate *dataset sampler* $ds^R_{\approx}$ keeps tracks of the costs on the full dataset $\mathcal{D}$ for all programs $\{p_j\}_{j=1}^{i-1}$ generated by $gen_{\approx}$. Once a new program p_j is generated, $ds^R_{\approx}$ computes $r_{regperp}(\mathcal{D}, p_j)$. Then, using a genetic-programming like procedure $ds^R_{\approx}$ keeps a like of candidate dataset samples and iteratively updates the list. At each iteration, $ds^R_{\approx}$ takes a dataset and randomly resamples its elements (such that the mutated dataset is $\subseteq \mathcal{D}$), scores the dataset and adds it to the list. Then $ds^R_{\approx}$ randomly removes from the list candidate datasets that are less representative than another candidate dataset. After a fixed number of iterations $ds^R_{\approx}$ returns the dataset $d_i \subseteq \mathcal{D}$ which is approximately the most representative for $progs = \{p_j\}_{j=1}^{i-1}$ according to Definition 2.4.

Termination condition We have chosen a time-based termination condition. After some fixed time limit for training expires, we return the (approximately) best learned program $p_{\approx best}$ obtained up to that moment. This is, from the programs $p_1, ..., p_m$ produced up to the time limit by Algorithm 1, we return as $p_{\approx best}$ the one that has the best cost on the full dataset $\mathcal{D}$.

Smoothing An important detail for increasing the precision of the system is using smoothing when computing the maximum likelihood estimate at any stage of the system. In our implementation, we use Witten-Bell interpolation smoothing [36]. Smoothing ensures our system performs well even in cases when a context was not seen in the training data. Consider for example looking for the probability of $P_{MLE}(rule \mid c_1c_2c_3)$ which is a complex context of three observations in the tree. If such a context was not seen in the training data, to estimate reasonable probabilities we backoff to a simpler context with only c_1c_2 in it. Then, we expect to see more dense data in estimating $P_{MLE}(rule \mid c_1c_2)$. To enable such backoff in $P_{MLE}(rule \mid c_1c_2)$, at training time P_{MLE} counts the events conditioned on the full context and conditioned on all the prefixes of the full context.

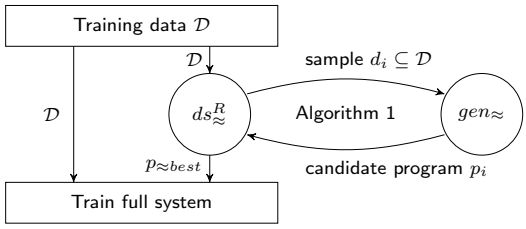

Figure 7. Overall diagram of learning a statistical code completion system using DEEPSYN.

6.5 Summary of Approach

To summarize, our approach to learning consists of two steps summarized in Fig. 7. In the first step, we learn the conditioning program $p_{\approx best}$. That is, given a dataset $\mathcal{D}$, we first use Algorithm 1, $gen_{\approx}$ and $ds^R_{\approx}$ to find a program $p_{\approx best}$ for which the cost $r_{regperp}(\mathcal{D}, p)$ is minimized. Then, in the second step, we a learn a probabilistic model $P^{H(\mathcal{D}, p_{\approx best})}_{MLE}$ over the full dataset $\mathcal{D}$ and the program $p_{\approx best}$ (discussed earlier).

This model, together with $p_{\approx best}$ can then be used to answer field/API completion queries from the programmer. We show in Section 7.1 that the speed-up from Algorithm 1 equipped with a representative dataset sampler lead to high accuracy of the resulting statistical synthesizer.

7. Evaluation of DEEPSYN

Based on Section 6, we created a statistical code completion system for JavaScript programs. This is a particularly hard setting for code completion systems as unlike in other languages (e.g., Java) where type information is easily available and is of significant help (e.g., as in [30]), in JavaScript, obtaining precise aliasing and type information is a difficult task (stressing the prediction capabilities of the system in the presence of noise, even further). The concrete task we consider is "dot" completion: given a receiver object, the completion should predict the field name or the API name that should be used on the object. All our experiments were performed on a 2.13 GHz Intel Xeon E7-4830 32-core machine with 256 GB of RAM, running 64-bit Ubuntu 14.04. To benefit from the amount of cores, we implemented the $gen_{\approx}$ and $ds^R_{\approx}$ procedures to evaluate multiple candidate programs or datasets in parallel.

To train and evaluate our system, we collected JavaScript programs from GitHub, removed duplicate files or project forks (copy of another existing repository) and kept only programs that parse and are not obfuscated. As a result, we obtained $150,000$ JavaScript files. We used two thirds of the data for learning, and the last one third only for evaluation.

7.1 Learning $p_{\approx best}$

We now discuss our training procedure. The first question we may ask is if using Algorithm 1 with a representative dataset sampler

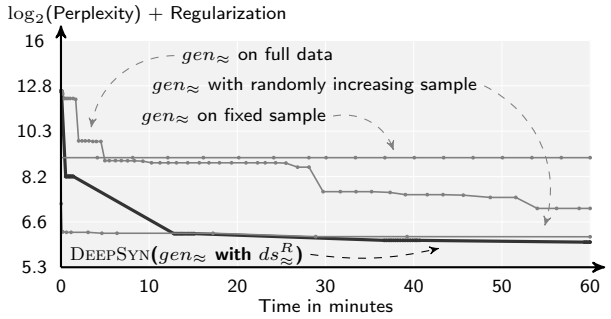

Figure 8. Effect of various data sampling policies used to find TCond programs.

($ds^R_{\approx}$) is of any benefit to the speed or precision of the system. To answer this question, we designed a number of system variants:

- DeepSyn($gen_{\approx}$ with $ds^R_{\approx}$) is our system as described Fig. 7. For this system, we start with a random sample of 100 programs and then through the loop of Algorithm 1, we modify the sample to be more representative.
- $gen_{\approx}$ on full data is a system that directly optimizes the program on the full dataset $\mathcal{D}$. Because each evaluation takes a long time, this system can only try a smaller number of candidate programs and finds only very imprecise programs.
- $gen_{\approx}$ on fixed sample is a system that starts with a random sample of 100 programs and optimizes the program only on the small sample. This policy has the time budget to explore a large number of programs, but is unable to provide reasonable score for them. It quickly reaches a cap on what can be learned on the small dataset.
- $gen_{\approx}$ with randomly increasing sample is a system that starts with a small random sample and iteratively increases its size. It performs no optimization of the sample for representativeness, just adds more elements to it.

The results summarized in Fig. 8 illustrate the effectiveness of each of the four approaches. Each plot gives the $r_{regperp}$ of the candidate program $p_{\approx best}$ found by a system at a given time. Note that lower values of log-perplexity are better and in fact are exponentially better. The graph shows that our full DeepSyn system initially spends time to find the best sample, but then reaches the best program of all systems. If provided with infinite time, $gen_{\approx}$ on full data will approach the optimal program, but in practice it is prohibitively slow and has far worse performance.

We note that the synthesizer with randomly increasing sample appears to find a reasonable program faster than when using $ds^R_{\approx}$. The reason for this is that this procedure did not evaluate its result on the full dataset $dataset$ (we evaluated the programs after the procedure completed). If we include the time to evaluate the candidate programs, this setting would not be as fast as it appears.

Next, we take the best program obtained after one hour of computation by DeepSyn and analyze it in detail.

7.2 Precision of DeepSyn

Once we obtain the best TCond program, we create a completion system based on it, train it on our learning dataset and evaluate it on the evaluation dataset. For each file in the evaluation dataset we randomly selected 100 method calls and queried our system to predict the correct completion one at a time. Given the evaluation dataset of $50,000$ programs, this resulted in invoking the prediction

Task	Size of training data (files) 1K	10K	100K
DOM APIs on document object			
correct completion at position 1	63.2%	69.2%	**77.0%**
correct completion in top 3	90.1%	84.6%	89.9%
correct completion in top 8	83.5%	88.6%	92.9%
Unrestricted API completion			
correct completion at position 1	22.6%	34.2%	**50.4%**
correct completion in top 3	30.8%	44.5%	61.9%
correct completion in top 8	33.6%	47.7%	64.9%
Field (non-API) completion			
correct completion at position 1	21.0%	29.7%	**38.9%**
correct completion in top 3	26.3%	37.0%	48.9%
correct completion in top 8	28.0%	38.8%	51.4%

Table 2. Accuracy of API method and object field completion depending on the task and the amount of training data.

of $2,537,415$ methods for any API and $48,390$ methods when predicting method calls on DOM document object.

The accuracy results are summarized in Table 2. The columns of the table represent systems trained on different amounts of training data. The right-most column trains on all $100,000$ JavaScript programs in the training set and the columns on the left use a subset of this data. Different rows on Table 2 include information for different tasks. On the task of predicting DOM APIs on the document object, the APIs are shared across all projects and the accuracy is higher – the correct completion is the first suggestion in 77% of the cases. When we extend the completion to any APIs, including APIs local to each project that the model may not know about, the accuracy drops to 50.4%. Finally, when used on non-API property completions, our completion system predicts the correct field name as a first suggestion in 38.9% of the cases.

7.3 Interpreting $p_{\approx best}$

A key aspect of TCond programs is that they are easily readable by a human. The best program is listed with its instructions in Fig. 9 (a). Let us illustrate what this program does on a JavaScript code completion query from Fig. 9 (b). Going instruction by instruction, Fig. 9 (c) shows the execution. First, the program moves to the left of the completion position in the tree (i.e., to the receiver object of the completion). Then, it moves to the previous usage of the same object (`PrevActor`), writes the action being done on the project (i.e., the name of the field of API invoked). Next, it writes the name of the variable (if any) and moves to the previous usage of the same object, etc. Finally, at instruction 9, it cannot move anymore and the program stops and returns the accumulated sequence so far. For our example, the program accumulates the following context:

```
querySelectorAll document querySelectorAll show
```

7.4 Comparison to Existing Systems

We note that previous works essentially use a hard-coded tree conditioning program. In some of the works immediate context is used – the non-terminals directly preceding the completion location [1, 16]. Other works hardcode the move to the previous API on the receiver object and condition on the API. Such knowledge is key in [30] and [24]. In fact, we mimic the behavior of [30] with a TCond program and show that our best synthesized program is superior. The context in [24] is more complex than a sequence (it is a graph), but in their experiments, even in a simpler setting within one project they report lower accuracy than our system.

Note that these statistical code completion systems target Java, which makes it much easier to perform static program analysis (e.g., type analysis, alias analysis). Despite these differences, we

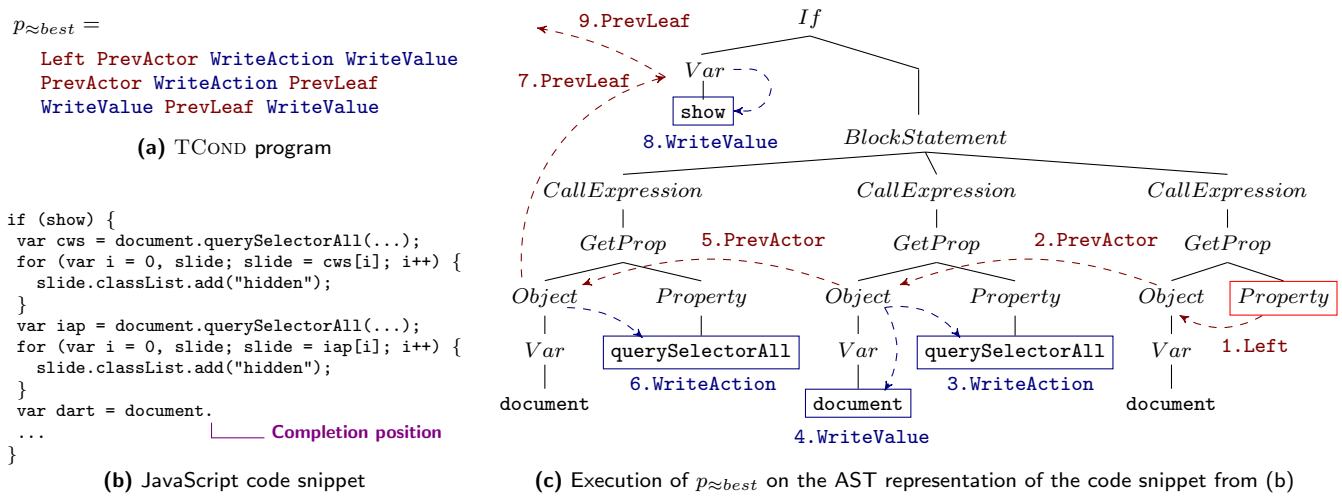

Figure 9. The (a) $p_{\approx best}$ TCOND program describing which parts of the code to condition on for an API method completion task, (b) a JavaScript code snippet on which we want to perform API completion, and (c) a relevant part of the AST representation of (b) where the execution of the TCOND program from (a) is illustrated.

created TCOND programs that mimic the behavior of some of these for JavaScript API completions. Next, we review these programs and compare their accuracy to DEEPSYN using the best program $p_{\approx best}$ in Fig. 9 (a). Recall that the accuracy on predicting JavaScript APIs for this program is 50.4%.

Hindle et al. [16] use an 3-gram language model on the tokenized program without doing any semantic analysis. This results in a predictor for JavaScript APIs with accuracy of 22.2%. The program that corresponds to this model is:

```
PrevDFS WriteValue PrevDFS WriteValue
```

Raychev et al. [30] perform a more complicated conditioning that depends on the previous APIs of the same object as well as whether the object was used as a receiver or as a method parameter. This conditioning is captured in the following program:

```
Left   PrevActor WriteAction WritePos
       PrevActor WriteAction WritePos
```

The accuracy of this program for JavaScript APIs is 30.4%.

***Variants of* DEEPSYN** In addition to the best possible program $p_{\approx best}$, we include the programs generated by our tool with different data sampling procedures.

- $gen_{\approx}$ on full data operates on the full data and within one hour it could not learn to condition on previous APIs. The program learned by this synthesis procedure is

  ```
  Left DownLast WriteValue PrevNodeType DownLast
  PrevDFS PrevLeaf WriteValue DownLast PrevLeaf
  Left WriteAction PrevNodeType WriteValue
  ```

  and results in 46.3% accuracy.
- $gen_{\approx}$ on fixed sample only learns a program from a small sample of 100 programs and results in a simple conditioning that only depends on a single previous API. The resulting accuracy for JavaScript API completion is 18.8% with the program

  ```
  PrevDFS PrevActor WriteAction
  ```

- $gen_{\approx}$ with randomly increasing sample iteratively increases the sample and results in a program that has accuracy of 47.5% which is lower than our best accuracy of 50.4%. The program generated by randomly increasing the sample size conditions on one previous API, but the rest of the program is not optimal

  ```
  Left PrevActor WriteAction
  DownLast PrevActor PrevNodeType WriteAction
  WriteValue Left PrevLeaf WriteValue
  ```

Summary Overall, we have shown that our approach of learning a tree conditioning program in a domain specific language (TCOND) exceeds the accuracy of systems where an expert hard-wired the prediction, as used in previous systems. The accuracy is achieved thanks to our general approach – a program generator that predicts a program from a small dataset ($gen_{\approx}$) and a representative dataset sampler that makes the small set behave similarly to the large training data ($ds^{R}_{\approx}$).

8. Related work

As our work touches on several areas, below we survey some of the existing results that are most closely related to ours.

Boolean program synthesis Over the last few years, there has been an increased interest in various forms of synthesis. Examples of recent techniques include synthesis from examples [18], partial programs [32] and synchronization [34]. A more detailed survey of the various approaches can be found here [11]. Generally, however, these are approaches which attempt to satisfy *all* provided examples and constraints. Thus, they typically over-fit to the data, and as a result, incorrect examples in the data set will lead to incorrectly learned programs (or no programs at all). Some approaches come with various fine-tuned ranking functions which assign preference to the (potentially many) synthesized programs. However, regardless of how good the ranking function is, if the data set contains even one wrong example, then the ranking function will be of little use, as it will simply rank incorrect programs. In contrast, our approach deals with noise and is able to synthesize desirable programs even in the presence of incorrect examples. It achieves that via both, the usage of regularizers which combat over-fitting, and smart, iterative sampling of the entire data set. As we showed in the paper, a standard all-or-nothing synthesis approach can be extended to incorporate and benefit from our techniques.

The work of Menon et al. [22] proposes to speed-up the synthesis process by using machine learning features that guide the search over candidate programs. This approach enables a faster decision

procedure for synthesizing a program, but fundamentally requires all provided input/examples to be satisfied.

Quantitative program synthesis Another line of work is that of synthesis with quantitative objectives [7, 33]. Here, it is possible to specify a quantitative specification (e.g., a probabilistic assertion) and to synthesize a program that satisfies that weaker specification while maximizing some quantitative objective. In our setting, one can think of the dataset $\mathcal{D}$ as being the specification, however, we essentially learn how to relax the spec, and do not require the user to provide it (which can be difficult). Further, our entire setting is very different, from the iterative sampling loop, to the fact that even if the specification can be fully satisfied, our approach need not satisfy it (e.g., due to regularization constraints). In the future, it may be useful to think of ways to bridge these directions.

Statistical code completion An emerging research area and one related to our work is that of learning probabilistic models from a large set of programs (i.e., "Big Code") and using those models to make statistical predictions for how a partial code fragment should be completed. Existing systems are typically hard-wired to work with a specific prediction strategy and are based on a simple token completion via the n-gram language model [16], on tokens with added annotations [25], other API calls [24, 30], expressions [14], or tree substitutions in the AST [2]. Unfortunately, these systems either have very low precision (e.g., [16]) or target restricted scenarios (e.g., APIs in [30]). Unlike these techniques, in our approach the prediction can be conditioned on functions that are expressed in a DSL and are learned from data, leading to better precision, and better support for different kinds of completions (as we share the same learning mechanism for any function in the DSL). Further, because the predictions are conditioned on the context built from evaluating a function, the justification behind a prediction is understandable to a programmer using the system. Thus, our approach generalizes some of the existing works, is experimentally more precise and the predictions are explainable to end users.

Discriminative learning In contrast to the generative learning performed in this work, some applications may enjoy a much better speed and precision with a discriminative model that performs classification without the need to build a probability distribution. One such case is a model tailored specifically to predicting annotations or variable names [29]. However, [29] cannot explain its predictions in terms of a program and requires manually provided feature functions in advance whose weights it learns. Despite not performing discriminative training, our approach essentially learns one class of feature functions (or one model). A possible way to extend our work is to enable learning a model that is a linear combination of several of the best models that we currently learn. The weights of such a linear combination can be then trained using structured support vector machine or another discriminative training procedure. We note that using such a linear combinations of models was applied in machine translation and is part of the current state-of-the-art systems [27]. We believe that extending this work to generate a linear combination of programs is an interesting future work item.

Core sets A core set is a concept in machine learning used to summarize a large data set into a smaller data set that preserves its properties. Core sets were successfully applied in the context of clustering such as k-means [15]. Obtaining core sets from a data set is a procedure that is manually tailored to the particular problem at hand (e.g., k-means), that is, there are currently no universal techniques for constructing core sets for arbitrary programs. In contrast, our work is not based on a specific algorithm or property such as k-means. In fact, an ideal outcome of our sampling step is to compute or approximate a core set of the training data. An interesting question for future work is to explore the connection between core sets and our iterative sampling algorithm.

Genetic algorithms Genetic programming has been proposed as a general approach to explore a large set of candidates in order to discover a solution that maximizes an objective function [4, 31]. The following work [8] discusses the language design decisions to encode a program synthesis problem from input/output examples into genetic programming. Some of the problems studied here in terms of selecting subset of the evaluation data to score instances were considered in the context of genetic programming. A technique known as stochastic sampling [3] reduces the number of evaluations by only considering random subsets of the training data. In our experiments, however, we show that using our strategy of representative sampling is superior than using random sampling.

Dataset cleaning A different approach for dealing with noisy data is to clean up the data beforehand either with a statistical model [6], or with an already given program [5]. These approaches, however need additional statistical assumptions about the data or specification of another program. In contrast, in this work, we simultaneously build the cleaned dataset and the program.

Probabilistic programs A recent synthesizer PSketch [26] performs synthesis of probabilistic programs by approximating the program by a mixture of Gaussians. Fundamentally, probabilistic programs interpret program executions as distributions and then for the synthesis task is fits parameters to these distributions. Instead, with our approach we learn deterministic programs that approximate a dataset well. In general, our idea of a dataset sampler should be applicable also to learning probabilistic programs from data, but we leave this as a future work.

Automatic configuration of algorithms ParamILS [17] picks a configuration of an algorithm based on its performance on a dataset. Similar to our approach, ParamILS attempts to speed-up the evaluation of a configuration by running on a small subset of the full dataset, but only does so by picking a random sample.

9. Conclusion

We presented a new approach for learning programs from noisy datasets of arbitrary size. We instantiated our approach to two important noise settings: the setting where we can place a bound on the noise and the setting where the dataset contains unbounded noise. We showed that the second setting leads to a new way of performing approximate empirical risk minimization over hypotheses classes formed by discrete search spaces. We then illustrated how to instantiate the different noise settings for building practical synthesizers that are able to deal with noisy datasets.

We first presented a synthesizer for bit-stream programs, called BITSYN. Our experimental results with BITSYN indicate that in the setting of bounded noise, our system returns the correct program even in the presence of incorrect examples (and with only a small increase in the number of necessary examples compared to the setting without noise). We also showed that BITSYN is useful for automatically detecting anomalies, without requiring an input program or statistical assumptions on the data.

Second, we presented a new technique for constructing statistical code completion engines based on "Big Code" which generalizes several existing works. The core idea is to define a DSL over trees and to learn functions in this DSL from the dataset. These learned functions then control the prediction made by the statistical synthesizer. We implemented our technique in a code completion system for JavaScript, called DEEPSYN, and showed that its predictions are more precise than existing works.

We believe this is the first comprehensive work that deals with the problem of learning programs from noisy datasets, and can serve as a basis for building new kinds of prediction engines that can deal with uncertainty.

References

[1] ALLAMANIS, M., AND SUTTON, C. Mining source code repositories at massive scale using language modeling. In *MSR* (2013).

[2] ALLAMANIS, M., AND SUTTON, C. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (New York, NY, USA, 2014), FSE 2014, ACM, pp. 472–483.

[3] BAKER, J. E. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application* (Hillsdale, NJ, USA, 1987), L. Erlbaum Associates Inc., pp. 14–21.

[4] BANZHAF, W., FRANCONE, F. D., KELLER, R. E., AND NORDIN, P. *Genetic Programming: An Introduction: on the Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[5] BAROWY, D. W., GOCHEV, D., AND BERGER, E. D. Checkcell: Data debugging for spreadsheets. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (New York, NY, USA, 2014), OOPSLA '14, ACM, pp. 507–523.

[6] CHANDOLA, V., BANERJEE, A., AND KUMAR, V. Anomaly detection: A survey. *ACM Comput. Surv. 41*, 3 (July 2009), 15:1–15:58.

[7] CHAUDHURI, S., CLOCHARD, M., AND SOLAR-LEZAMA, A. Bridging boolean and quantitative synthesis using smoothed proof search. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2014), POPL '14, ACM, pp. 207–220.

[8] CRAMER, N. L. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the 1st International Conference on Genetic Algorithms* (Hillsdale, NJ, USA, 1985), L. Erlbaum Associates Inc., pp. 183–187.

[9] DE MOURA, L., AND BJØRNER, N. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Berlin, Heidelberg, 2008), TACAS'08/ETAPS'08, Springer-Verlag, pp. 337–340.

[10] Github code search. https://github.com/search.

[11] GULWANI, S. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming* (New York, NY, USA, 2010), PPDP '10, ACM, pp. 13–24.

[12] GULWANI, S. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2011), POPL '11, ACM, pp. 317–330.

[13] GULWANI, S., JHA, S., TIWARI, A., AND VENKATESAN, R. Synthesis of loop-free programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2011), PLDI '11, ACM, pp. 62–73.

[14] GVERO, T., KUNCAK, V., KURAJ, I., AND PISKAC, R. Complete completion using types and weights. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2013), PLDI '13, ACM, pp. 27–38.

[15] HAR-PELED, S., AND MAZUMDAR, S. On coresets for k-means and k-median clustering. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 2004), STOC '04, ACM, pp. 291–300.

[16] HINDLE, A., BARR, E. T., SU, Z., GABEL, M., AND DEVANBU, P. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering* (Piscataway, NJ, USA, 2012), ICSE '12, IEEE Press, pp. 837–847.

[17] HUTTER, F., HOOS, H. H., LEYTON-BROWN, K., AND STÜTZLE, T. Paramils: An automatic algorithm configuration framework. *J. Artif. Int. Res. 36*, 1 (Sept. 2009), 267–306.

[18] JHA, S., GULWANI, S., SESHIA, S. A., AND TIWARI, A. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* (New York, NY, USA, 2010), ICSE '10, ACM, pp. 215–224.

[19] LAU, T. A. *Programming by Demonstration: A Machine Learning Approach*. PhD thesis, 2001. AAI3013992.

[20] LE, V., AND GULWANI, S. Flashextract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2014), PLDI '14, ACM, pp. 542–553.

[21] LUXBURG, U. V., AND SCHOELKOPF, B. Statistical learning theory: Models, concepts, and results. In *Inductive Logic*. 2011, pp. 651–706.

[22] MENON, A. K., TAMUZ, O., GULWANI, S., LAMPSON, B. W., AND KALAI, A. A machine learning framework for programming by example. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013* (2013), pp. 187–195.

[23] MITCHELL, T. M. *Machine Learning*, 1 ed. McGraw-Hill, Inc., New York, NY, USA, 1997.

[24] NGUYEN, A. T., AND NGUYEN, T. N. Graph-based statistical language model for code. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Piscataway, NJ, USA, 2015), ICSE '15, IEEE Press, pp. 858–868.

[25] NGUYEN, T. T., NGUYEN, A. T., NGUYEN, H. A., AND NGUYEN, T. N. A statistical semantic language model for source code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2013), ESEC/FSE 2013, ACM, pp. 532–542.

[26] NORI, A. V., OZAIR, S., RAJAMANI, S. K., AND VIJAYKEERTHY, D. Efficient synthesis of probabilistic programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2015), PLDI 2015, ACM, pp. 208–217.

[27] OCH, F. J. Minimum error rate training in statistical machine translation. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1* (Stroudsburg, PA, USA, 2003), ACL '03, Association for Computational Linguistics, pp. 160–167.

[28] PANCHEKHA, P., SANCHEZ-STERN, A., WILCOX, J. R., AND TATLOCK, Z. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015* (2015), pp. 1–11.

[29] RAYCHEV, V., VECHEV, M., AND KRAUSE, A. Predicting program properties from "big code". In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2015), POPL '15, ACM, pp. 111–124.

[30] RAYCHEV, V., VECHEV, M., AND YAHAV, E. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (New York, NY, USA, 2014), PLDI '14, ACM, pp. 419–428.

[31] SMITH, S. F. *A Learning System Based on Genetic Adaptive Algorithms*. PhD thesis, Pittsburgh, PA, USA, 1980. AAI8112638.

[32] SOLAR-LEZAMA, A., TANCAU, L., BODÍK, R., SESHIA, S. A., AND SARASWAT, V. A. Combinatorial sketching for finite programs. In *ASPLOS* (2006), pp. 404–415.

[33] ČERNÝ, P., AND HENZINGER, T. A. From boolean to quantitative synthesis. In *Proceedings of the Ninth ACM International Conference on Embedded Software* (New York, NY, USA, 2011), EMSOFT '11, ACM, pp. 149–154.

[34] VECHEV, M., YAHAV, E., AND YORSH, G. Abstraction-guided synthesis of synchronization. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 2010), POPL '10, ACM, pp. 327–338.

[35] WARREN, H. S. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

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