PROGRAM SYNTHESIS FOR CHARACTER LEVEL LANGUAGE MODELING

Pavol Bielik, Veselin Raychev & Martin Vechev
Department of Computer Science, ETH Zürich, Switzerland
{pavol.bielik,veselin.raychev,martin.vechev}@inf.ethz.ch

ABSTRACT

We propose a statistical model applicable to character level language modeling and show that it is a good fit for both, program source code and English text. The model is parameterized by a program from a domain-specific language (DSL) that allows expressing non-trivial data dependencies. Learning is done in two phases: (i) we synthesize a program from the DSL, essentially learning a good representation for the data, and (ii) we learn parameters from the training data – the process is done via counting, as in simple language models such as n-gram.

Our experiments show that the precision of our model is comparable to that of neural networks while sharing a number of advantages with n-gram models such as fast query time and the capability to quickly add and remove training data samples. Further, the model is parameterized by a program that can be manually inspected, understood and updated, addressing a major problem of neural networks.

1 INTRODUCTION

Recent years have shown increased interest in learning from large datasets in order to make accurate predictions on important tasks. A significant catalyst for this movement has been the ground breaking precision improvements on a number of cognitive tasks achieved via deep neural networks. Deep neural networks have made substantial inroads in areas such as image recognition (Krizhevsky et al., 2012) and natural language processing (Józefowicz et al., 2016) thanks to large datasets, deeper networks (He et al., 2016) and substantial investments in computational power Oh & Jung (2004).

While neural networks remain a practical choice for many applications, they have been less effective when used for more structured tasks such as those concerning predictions about programs (Allamanis et al., 2016; Raychev et al., 2014). Initially targeting the programming languages domain, a new method for synthesizing probabilistic models proposed by Bielik et al. (2016), without a neural network, has shown to be effective for modeling source code, and has gained traction.

In this work, we investigate the applicability of this new method to tasks which have so far been addressed with recurrent neural networks and n-gram language models. The probabilistic models we propose are defined by a program from a domain-specific language (DSL). A program in this DSL describes a probabilistic model such as n-gram language models or a variant of it - e.g. trained on subsets of the training data, queried only when certain conditions are met and specialized in making specific classes of predictions. These programs can also be combined to produce one large program that queries different specialized submodels depending on the context of the query.

For example, consider predicting the characters in an English text. Typically, the first character of a word is much more difficult to predict than other characters and thus we would like to predict it differently. Let f be a function that takes a prediction position t in a text x and returns a list of characters to make prediction on. For example, let f be defined as follows:

$$f(t, x) = \textbf{if } x_{t-1} \textbf{ is whitespace then } x_s \textbf{ else } x_{t-1}x_{t-2}$$

where x_s is the first character of the word preceding the predicted word at position t. Now, consider a model that predicts a character by estimating each character x_t from a distribution $P(x_t|f(t, x))$. For positions that do not follow a whitespace, this distribution uses the two characters preceding x_t and thus it simply encodes a trigram language model (in this example, without backoff, but we

consider backoff separately). However, $P(x_t|f(t,x))$ combines the trigram model with another interesting model for the samples in the beginning of the words where trigram models typically fail.

More generally, f is a function that describes a probabilistic model. By simply varying f, we can define trivial models such as n-gram models, or much deeper and interesting models and combinations. In this work, we draw f from a domain-specific language (DSL) that resembles a standard programming language: it includes if statements, limited use of variables and one iterator over the text, but overall that language can be further extended to handle specific tasks depending on the nature of the data. The learning process now includes finding f from the DSL such that the model $P(x_t|f(t,x))$ performs best on a validation set and we show that we can effectively learn such functions using Markov chain Monte Carlo (MCMC) search techniques combined with decision tree learning.

Advantages An advantage of having a function f drawn from a DSL is that f becomes humanly readable, in contrast to neural networks that generally provide non-human readable matrices (Li et al., 2016). Further, the training procedure is two-fold: first, we synthesize f from the DSL, and then for a given f, we estimate probabilities for $P(x_t|f(t,x))$ by counting in the training data. This gives us additional advantages such as the ability to synthesize f and learn the probability distribution P on different datasets: e.g., we can easily add and remove samples from the dataset used for computing the probability estimate P. Finally, because the final model is based on counting, estimating probabilities $P(x_t|f(t,x))$ is efficient: applying f and looking up in a hashtable to determine how frequently in the training data, x_t appears in the resulting context of $f(t,x)$.

Before we continue, we note an important point about DSL-based models. In contrast to deep neural networks that can theoretically encode all continuous functions (Hornik, 1991), a DSL by definition targets a particular application domain, and thus comes with restricted expressiveness. Increasing the expressibility of the DSL (e.g., by adding new instructions) can in theory make the synthesis problem intractable or even undecidable. Overall, this means that a DSL should balance between expressibility and efficiency of synthesizing functions in it (Gulwani, 2010).

Contributions This work makes two main contributions:

- We define a DSL which is useful in expressing (character level) language models. This DSL can express n-gram language models with backoff, with caching and can compose them using if statements in a decision-tree-like fashion. We also provide efficient synthesis procedures for functions in this DSL.
- We experimentally compare our DSL-based probabilistic model with state-of-the-art neural network models on two popular datasets: the *Linux Kernel* dataset (Karpathy et al., 2015) and the *Hutter Prize Wikipedia* dataset (Hutter, 2012).

2 A DSL FOR CHARACTER LEVEL LANGUAGE MODELING

We now provide a definition of our domain specific language (DSL) called `TChar` for learning character level language models. At a high level, executing a program $p \in$ `TChar` at a position $t \in \mathbb{N}$ in the input sequence of characters $x \in X$ returns an `LMProgram` that specifies language model to be used at position t. That is, $p \in$ `TChar` : $\mathbb{N} \times X \rightarrow$ `LMProgram`. The best model to use at position t is selected depending on the current program state (updated after processing each character) and conditioning on the *dynamically* computed context for position t of the input text x. This allows us to train specialized models suited for various types of prediction such as for comments vs. source code, newlines, indentation, opening/closing brackets, first character of a word and many more. Despite of the fact that the approach uses many specialized models, we still obtain a valid probability distribution as all of these models operate on disjoint data and are valid probability distributions. Subsequently, the selected program $f \in$ `LMProgram` determines the language model that estimates the probability of character x_t by building a probability distribution $P(x_t|f(t,x))$. That is, the probability distribution is conditioned on the context obtained by executing the program f.

The syntax of `TChar` is shown in Fig. 1 and is designed such that it contains general purpose instructions and statements that operate over a sequence of characters. One of the advantages of our approach is that this language can be further refined by adding more instructions that are specialized for a given domain at hand (e.g., in future versions, we can easily include set of instructions specific

Program State $state \in \mathbb{N}$ Accumulated Context $v \in \mathbb{N}^*$ Backoff Threshold $d \in \langle 0, 1 \rangle$

`TChar` ::= `SwitchProgram` | `StateProgram` | **`return`** `LMProgram`

`StateProgram` ::= `StateUpdate`; `StateSwitch`

`StateUpdate` ::=

```
switch SimpleProgram
  case v1 then INC          (state = state + 1)
  case v2 then DEC          (state = max(0, state − 1))
  default SKIP              (state = state)
```

`StateSwitch` ::=

```
switch state
  case v1 then TChar
  ···
  case vn then TChar
  default TChar
```

`SwitchProgram` ::=

```
switch SimpleProgram
  case v1 or ··· or vk then TChar
  ···
  case vj or ··· or vn then TChar
  default TChar
```

`SimpleProgram` ::= ϵ | `Move`; `SimpleProgram` | `Write`; `SimpleProgram`

`Move` ::= `LEFT` | `RIGHT` | `PREV_CHAR` | `PREV_POS`

`Write` ::= `WRITE_CHAR` | `WRITE_HASH` | `WRITE_DIST`

`LMProgram` ::= `SimpleProgram` | `SimpleProgram` **`backoff`** d; `LMProgram` | (`SimpleProgram`, `SimpleProgram`).

Figure 1: Syntax of `TChar` language for character level language modeling. Program semantics are given in the appendix.

to modeling C/C++ source code). We now informally describe the general `TChar` language of this work. We provide a formal definition and semantics of the `TChar` language in the appendix.

2.1 SIMPLEPROGRAMS

The `SimpleProgram` is a basic building block of the `TChar` language. It describes a loop-free and branch-free program that accumulates context with values from the input by means of navigating within the input (using `Move` instructions) and writing the observed values (using `Write` instructions). The result of executing a `SimpleProgram` is the accumulated context which is used either to condition the prediction, to update the program state or to determine which program to execute next.

Move Instructions We define four basic types of `Move` instructions – `LEFT` and `RIGHT` that move to the previous and next character respectively, `PREV_CHAR` that moves to the most recent position in the input with the same value as the current character x_t and `PREV_POS` which works as `PREV_CHAR` but only considers positions in the input that are partitioned into the same language model. Additionally, for each character c in the input vocabulary we generate instruction `PREV_CHAR(c)` that navigates to the most recent position of character c. We note that all `Move` instructions are allowed to navigate only to the left of the character x_t that is to be predicted.

Write Instructions We define three `Write` instructions – `WRITE_CHAR` that writes the value of the character at current position, `WRITE_HASH` that writes a hash of all the values seen between the current position and the position of last `Write` instruction, and `WRITE_DIST` that writes a distance (i.e., number of characters) between current position and the position of last `Write` instruction. In our implementation we truncate `WRITE_HASH` and `WRITE_DIST` to a maximum size of 16.

Example With `Write` and `Move` instructions, we can express various programs that extract useful context for a given position t in text x. For example, we can encode the context used in trigram language model with a program `LEFT WRITE_CHAR LEFT WRITE_CHAR`. We can also express programs such as `LEFT PREV_CHAR RIGHT WRITE_CHAR` that finds the previous occurrence of the character on the left of the current position and records the character following it in the context.

2.2 SWITCHPROGRAMS

A problem of using only one `SimpleProgram` is that the context it generates may not work well for the entire dataset, although combining several such programs can generate suitable contexts for the different types of predictions. To handle these cases, we introduce `SwitchProgram` with `switch` statements that can conditionally select appropriate subprograms to use depending on the context of the prediction. The checked conditions of `switch` are themselves program $p_{guard} \in$ `SimpleProgram` that accumulate values that are used to select the appropriate branch that should be executed next. During the learning the goal is then to synthesize best program p_{guard} to be used as a guard, the values $v_1, \cdots, v_n$ used as branch conditions as well as the programs to be used in each branch. We note that we support disjunction of values within branch conditions. As a result, even if the same program is to be used for two different contexts v_1 and v_2, the synthesis procedure can decide whether a single model should be trained for both (by synthesizing a single branch with a disjunction of v_1 and v_2) or a separate models should be trained for each (by synthesizing two branches).

Example We now briefly discuss some of the `BranchProgram` synthesized for the *Linux Kernel* dataset in our experiments described in Section 3. By inspecting the synthesized program we identified interesting `SimplePrograms` building blocks such as `PREV_CHAR`(␣) `RIGHT` `WRITE_CHAR` that conditions on the first character of the current word, `PREV_CHAR`(\n) `WRITE_DIST` that conditions on the distance from the beginning of the line or `PREV_CHAR`(_) `LEFT` `WRITE_CHAR` that checks the preceding character of a previous underscore (useful for predicting variable names). These are examples of more specialized programs that are typically found in the branches of nested switches of a large `TChar` program. The top level switch of the synthesized program used the character before the predicted position (i.e. **`switch`** `LEFT` `WRITE_CHAR`) and handles separately cases such as newline, tabs, special characters (e.g., !#@.∗), upper-case characters and the rest.

2.3 STATEPROGRAMS

A common difficulty in building statistical language models is capturing long range dependencies in the given dataset. Our `TChar` language partially addresses this issue by using `Move` instructions that can jump to various positions in the data using `PREV_CHAR` and `PREV_POS` instructions. However, we can further improve by explicitly introducing a state to our programs using `StateProgram`. The `StateProgram` consists of two sequential operations – updating the current state and determining which program to execute next based on the value of the current state. For both we reuse the `switch` construct defined previously for `SwitchProgram`. In our work we consider integer valued *state* that can by either incremented, decremented or left unmodified after processing each input character. We note that other definitions of the state, such as stack based state, are possible.

Example As an example of a `StateProgram` consider the question of detecting whether the current character is inside a comment or is a source code. These denote very different types of data that we might want to model separately if it leads to improvement in our cost metric. This can be achieved by using a simple state program with condition `LEFT` `WRITE_CHAR` `LEFT` `WRITE_CHAR` that increments the state on '/*', decrements on '*/' and leaves the state unchanged otherwise.

2.4 LMPROGRAMS

The `LMProgram` describes a probabilistic model trained and queried on a subset of the data as defined by the branches taken in the `SwitchPrograms` and `StatePrograms`. The `LMProgram` in `TChar` is instantiated with a language model described by a `SimpleProgram` plus backoff. That is, the prediction is conditioned on the sequence of values returned by executing the program, i.e. $P(x_t|f(t,x))$.

Recall that given a position t in text x, executing a `SimpleProgram` returns context $f(t,x)$. For example, executing `LEFT` `WRITE_CHAR` `LEFT` `WRITE_CHAR` returns the two characters $x_{t-1}x_{t-2}$ preceding x_t. In this example $P(x_t|f(t,x))$ is a trigram model. To be effective in practice, however, such models should support smoothing or backoff to lower order models. We provide backoff in two ways. First, because the accumulated context by the `SimpleProgram` is a sequence we simply backoff to a model that uses a shorter sequences by using Witten-Bell backoff (Witten & Bell, 1991). Second, in the `LMProgram` we explicitly allow to backoff to other models specified by a `TChar` pro-

gram if the probability of the most likely character from vocabulary V according to the $P(x_t|f(t,x))$ model is less than a constant d. Additionally, for some of our experiments we also consider backoff to a cache model (Kuhn & De Mori, 1990).

Predicting Out-of-Vocabulary Labels Finally, we incorporate a feature of the language models as proposed by Raychev et al. (2016a) that enables us not only to predict characters directly but instead to predict a character which is equal to some other character in the text. This is achieved by synthesising a pair of $(\texttt{SimpleProgram}, \texttt{SimpleProgram})$. The first program is called equality program and it navigates over the text to return characters that may be equal to the character that we are trying to predict. Then, the second program f describes $P(x_t|f(t,x))$ as described before, except that a possible output is equality to one of the characters returned by the equality program.

2.5 SYNTHESISING TChar PROGRAMS

The goal of the synthesizer is given a set of training and validation samples D, to find a program:

$$p_{best} = \arg\min_{p \in \texttt{TChar}} cost(D, p)$$

where $cost(D,p) = -logprob(D,p) + \lambda \cdot \Omega(p)$. Here $logprob(D,p)$ is the log-probability of the trained models on the dataset D and $\Omega(p)$ is a regularization that penalizes complex functions to avoid over-fitting to the data. In our implementation, $\Omega(p)$ returns the number of instructions in p.

The language TChar essentially consists of two fragments: branches and straight-line SimplePrograms. To synthesize branches, we essentially need a decision tree learning algorithm that we instantiate with the ID3+ algorithm as described in Raychev et al. (2016a). To synthesize SimplePrograms we use a combination of brute-force search for very short programs (up to 5 instructions), genetic programming-based search and Markov chain Monte Carlo-based search. These procedures are computationally feasible, because each SimpleProgram consists of only a small number of moves and writes. We provide more details about this procedure in Appendix B.1.

3 EXPERIMENTS

Datasets For our experiments we use two diverse datasets: a natural language one and a structured text (source code) one. Both were previously used to evaluate character-level language models – the *Linux Kernel* dataset Karpathy et al. (2015) and *Hutter Prize Wikipedia* dataset Hutter (2012). The *Linux Kernel* dataset contains header and source files in the C language shuffled randomly, and consists of $6,206,996$ characters in total with vocabulary size 101. The *Hutter Prize Wikipedia* dataset contains the contents of Wikipedia articles annotated with meta-data using special mark-up (e.g., XML or hyperlinks) and consists of $100,000,000$ characters and vocabulary size 205. For both datasets we use the first 80% for training, next 10% for validation and final 10% as a test set.

Evaluation Metrics To evaluate the performance of various probabilistic language models we use two metrics. Firstly, we use the *bits-per-character* (BPC) metric which corresponds to the negative log likelihood of a given prediction $\mathbb{E}[-\log_2 p(x_t \mid x_{<t})]$, where x_t is character being predicted and $x_{<t}$ denotes characters preceding x_t. Further, we use *error rate* which corresponds to the ratio of mistakes the model makes. This is a practical metric that directly quantifies how useful is the model in a concrete task (e.g., completion). As we will see, having two different evaluation metrics is beneficial as better (lower) BPC does not always correspond to better (lower) error rate.

3.1 LANGUAGE MODELS

We compare the performance of our trained DSL model, instantiated with the TChar language described in Section 2, against two widely used language models – n-gram model and recurrent neural networks. For all models we consider character level modeling of the dataset at hand. That is, the models are trained by feeding the input data character by character, without any knowledge of higher level word boundaries and dataset structure.

Linux Kernel Dataset (Karpathy et al., 2015)					
Model	**Bits per Character**	**Error Rate**	**Training Time**	**Queries per Second**	**Model Size**
LSTM (Layers×Hidden Size)					
2×128	2.31	40.1%	≈28 hours	4 000	5 MB
2×256	2.15	37.9%	≈49 hours	1 100	15 MB
2×512	2.05	38.1%	≈80 hours	300	53 MB
***n*-gram**					
4-gram	2.49	47.4%	1 sec	46 000	2 MB
7-gram	2.23	37.7%	4 sec	41 000	24 MB
10-gram	2.32	36.2%	11 sec	32 000	89 MB
15-gram	2.42	35.9%	23 sec	21 500	283 MB
DSL model (This Work)					
$\texttt{TChar}_{\text{w/o cache \& backoff}}$	1.92	33.3%	≈8 hours	62 000	17 MB
$\texttt{TChar}_{\text{w/o backoff}}$	1.84	31.4%	≈8 hours	28 000	19 MB
$\texttt{TChar}_{\text{w/o cache}}$	1.75	28.0%	≈8.2 hours	24 000	43 MB
TChar	1.53	23.5%	≈8.2 hours	3 000	45 MB

Table 1: Detailed comparison of LSTM, *n*-gram and DSL models on *Linux Kernel* dataset.

***N*-gram** We use the *n*-gram model as a baseline model as it has been traditionally the most widely used language model due to it simplicity, efficient training and fast sampling. We note that *n*-gram can be trivially expressed in the `TChar` language as a program containing a sequence of `LEFT` and `WRITE` instructions. To deal with data sparseness we have experimented with various smoothing techniques including Witten-Bell interpolation smoothing Witten & Bell (1991) and modified Kneser-Ney smoothing Kneser & Ney (1995); Chen & Goodman (1998).

Recurrent Neural Networks To evaluate the effectiveness of the DSL model we compare to a recurrent network language model shown to produce state-of-the-art performance in various natural language processing tasks. In particular, for the *Linux Kernel* dataset we compare against a variant of recurrent neural networks with Long Short-Term Memory (LSTM) proposed by Hochreiter & Schmidhuber (1997). To train our models we follow the experimental set-up and use the implementation of Karpathy et al. (2015). We initialize all parameters uniformly in range $[-0.08, 0.08]$, use mini-batch stochastic gradient descent with batch size 50 and RMSProp Dauphin et al. (2015) per-parameter adaptive update with base learning rate 2×10^{-3} and decay 0.95. Further, the network is unrolled 100 time steps and we do not use dropout. Finally, the network is trained for 50 epochs (with early stopping based on a validation set) and the learning rate is decayed after 10 epochs by multiplying it with a factor of 0.95 each additional epoch. For the *Hutter Prize Wikipedia* dataset we compared to various other, more sophisticated models as reported by Chung et al. (2017).

DSL model To better understand various features of the `TChar` language we include experiments that disable some of the language features. Concretely, we evaluate the effect of including cache and backoff. In our experiments we backoff the learned program to a *7*-gram and *3*-gram models and we use cache size of 800 characters. The backoff thresholds d are selected by evaluating the model performance on the validation set. Finally, for the *Linux Kernel* dataset we manually include a `StateProgram` as a root that distinguishes between comments and code (illustrated in Section 2.3). The program learned for the *Linux Kernel* dataset contains $\approx$ 700 `BranchPrograms` and $\approx$ 2200 `SimplePrograms` and has over 8600 `Move` and `Write` instructions in total. We provide an interactive visualization of the program and it's performance on the *Linux Kernel* dataset online at:

`www.srl.inf.ethz.ch/charmodel.html`

Hutter Prize Wikipedia Dataset (Hutter, 2012)						
Metric	***n*-gram**	**DSL model**	**Stacked LSTM**	**MRNN**	**MI-LSTM**	**HM-LSTM†**
	$n = 7$	This Work	Graves (2013)	Sutskever et al. (2011)	Wu et al. (2016)	Chung et al. (2017)
BPC	1.94	1.62	1.67	1.60	1.44	1.34

Table 2: Bits-per-character metric for various neural language models (as reported by Chung et al. (2017)) achieved on *Hutter Prize Wikipedia* dataset where the `TChar` model achieves competitive results. †State-of-the-art network combining character and word level models that are learned from the data.

3.2 Model Performance

We compare the performance of the DSL model, *n*-gram and neural networks for the tasks of learning character level language models by discussing a number of relevant metrics shown in Table 1 and Table 2.

Precision In terms of model precision, we can see that as expected, the *n*-gram model performs worse in both BPC and error rate metrics. However, even though the best BPC is achieved for a 7-gram model, the error rate decreases up to 15-gram. This suggests that none of the smoothing techniques we tried can properly adjust to the data sparsity inherent in the higher order *n*-gram models. It is however possible that more advanced smoothing techniques such as one based on Pitman-Yor Processes (Teh, 2006) might address this issue. As the DSL model uses the same smoothing technique as *n*-grams, any improvement to smoothing is directly applicable to it.

As reported by Karpathy et al. (2015), the LSTM model trained on the *Linux Kernel* dataset improves BPC over the *n*-gram. However, in our experiments this improvement did not translate to lower error rate. In contrast, our model is superior to *n*-gram and LSTM in all configurations, beating the best other model significantly in both evaluation metrics – decreasing BPC by over 0.5 and improving error rate by more than 12%.

For the *Hutter Prize Wikipedia* dataset, even though the dataset consists of natural language text and is much less structured than the *Linux Kernel*, our model is competitive with several neural network models. Similar to the results achieved on *Linux Kernel*, we expect the error rate of the DSL model for *Hutter Prize Wikipedia* dataset, which is 30.5%, to be comparable to the error rate achieved by other models. However, this experiment shows that our model is less suitable to unstructured text such as the one found on Wikipedia.

Training Time Training time is dominated by the LSTM model that takes several days for the network with the highest number of parameters. On the other hand, training *n*-gram models is extremely fast since the model is trained simply by counting in a single pass over the training data. The DSL model sits between these two approaches and takes ≈ 8 hours to train. The majority of the DSL training time is spent in the synthesis of `SwitchPrograms` where one needs to consider a massive search space of possible programs from which the synthesis algorithm aims to find one that is approximately the best (e.g., for *Linux Dataset* the number of basic instructions is 108 which means that naive enumeration of programs up to size 3 already leads to 108^3 different candidate programs). All of our experiments were performed on a machine with Intel(R) Xeon(R) CPU E5-2690 with 14 cores. All training times are reported for parallel training on CPU. Using GPUs for training of the neural networks might provide additional improvement in training time.

Query (Sampling) Time Sampling the *n*-gram is extremely fast, answering $\approx 46,000$ queries per second, as each query corresponds to only a several hash look-ups. The query time for the DSL model is similarly fast, in fact can be even faster, by answering $\approx 62,000$ queries per second. This is because it consists of two steps: (i) executing the `TChar` program that selects a suitable language model (which is very fast once the program has been learned), and (ii) querying the language model. The reason why the model can be faster is that there are fewer hash lookups which are also faster due to the fact that the specialized language models are much smaller compared to the *n*-gram (e.g.,

$\approx 22\%$ of the models simply compute unconditioned probabilities). Adding backoff and cache to the DSL model decreases sampling speed, which is partially because our implementation is currently not optimized for querying and fully evaluates all of the models even though that is not necessary.

Model Size Finally, we include the size of all the trained models measured by the size in MB of the model parameters. The models have roughly the same size except for the *n*-gram models with high order for which the size increases significantly. We note that the reason why both the *n*-gram and DSL models are relatively small is that we use hash-based implementation for storing the prediction context. That is, in a 7-gram model the previous 6 characters are hashed into a single number. This decreases the model size significantly at the expense of some hash collisions.

4 Related Work

Program Synthesis Program synthesis is a well studied research field in which the goal is to automatically discover a program that satisfies a given specification that can be expressed in various forms including input/output examples, logical formulas, set of constraints or even natural language description.In addition to the techniques that typically scale only for smaller programs and attempt to satisfy the specification completely (Alur et al., 2013; Solar-Lezama et al., 2006; Solar-Lezama, 2013; Jha et al., 2010) a recent line of work considers a different setting consisting of large (e.g., millions of examples) and noisy (i.e., no program can satisfy all examples perfectly) datasets. This is the setting that needs to be considered when synthesizing programs in our setting and in a language such as `TChar`. Here, the work of Raychev et al. (2016b) showed how to efficiently synthesize straight-line programs and how to handle noise, then Raychev et al. (2016a) showed how to synthesize branches, and thework of Heule et al. (2015) proposed a way to synthesize loops.

In our work we take advantage of these existing synthesis algorithms and use them to efficiently synthesize a program in `TChar`. We propose a simple extension that uses MCMC to sample from a large amount of instructions included in `TChar` (many more than prior work). Apart from this, no other modifications were required in order to use existing techniques for the setting of character level language modeling we consider here.

Recurrent Neural Networks Recent years have seen an emerging interest in building a neural language models over words (Bengio et al., 2003), characters (Karpathy et al., 2015; Sutskever et al., 2011; Wu et al., 2016) as well as combination of both (Chung et al., 2017; Kim et al., 2015; Mikolov et al., 2012). Such models have been shown to achieve state-of-the-art performance in several domains and there is a significant research effort aimed at improving such models further.

In contrast, we take a very different approach, one that aims to explain the data by means of synthesizing a *program* from a domain specific language. Synthesising such programs efficiently while achieving competitive performance to the carefully tuned neural networks creates a valuable resource that be used as a standalone model, combined with existing neural language models or even used for their training. For example, the context on which the predictions are conditioned is similar to the attention mechanism (Sukhbaatar et al., 2015; Bahdanau et al., 2014) and might be incorporated into that training in the future.

5 Conclusion

In this paper we proposed and evaluated a new approach for building character level statistical language models based on a program that parameterizes the model. We design a language `TChar` for character level language modeling and synthesize a program in this language. We show that our model works especially well for structured data and is significantly more precise that prior work. We also demonstrate competitive results in the less structured task of modeling English text.

Expressing the language model as a program results in several advantages including easier interpretability, debugging and extensibility with deep semantic domain knowledge by simply incorporating a new instruction in the DSL. We believe that this development is an interesting result in bridging synthesis and machine learning with much potential for future research.

ACKNOWLEDGMENTS

The research leading to these results was partially supported by an ERC Starting Grant 680358.

REFERENCES

Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. A convolutional attention network for extreme summarization of source code. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pp. 2091–2100, 2016.

Rajeev Alur, Rastislav Bodík, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pp. 1–8, 2013.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155, 2003.

Pavol Bielik, Veselin Raychev, and Martin T. Vechev. PHOG: probabilistic model for code. In *Proceedings of the 33th International Conference on Machine Learning*, ICML '16, pp. 2933–2942, 2016.

Stanley F. Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. Technical report, Harvard University, 1998.

Junyoung Chung, Sungjin Ahn, and Yoshua Bengio. Hierarchical multiscale recurrent neural networks. *ICLR*, 2017.

Yann N. Dauphin, Harm de Vries, and Yoshua Bengio. Equilibrated adaptive learning rates for non-convex optimization. In *Proceedings of the 28th International Conference on Neural Information Processing Systems*, NIPS'15, pp. 1504–1512. MIT Press, 2015.

Alex Graves. Generating sequences with recurrent neural networks. *CoRR*, abs/1308.0850, 2013.

Sumit Gulwani. Dimensions in program synthesis. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, PPDP '10, pp. 13–24. ACM, 2010.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, June 2016. doi: 10.1109/CVPR.2016.90.

Stefan Heule, Manu Sridharan, and Satish Chandra. Mimic: Computing models for opaque code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pp. 710–720, 2015.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997.

Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Netw.*, 4(2): 251–257, 1991. ISSN 0893-6080.

Marcus Hutter. The human knowledge compression contest, 2012. URL `http://prize.hutter1.net/`.

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pp. 215–224, New York, NY, USA, 2010. ACM.

Rafal Józefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *CoRR*, abs/1602.02410, 2016.

Andrej Karpathy, Justin Johnson, and Fei-Fei Li. Visualizing and understanding recurrent networks. *CoRR*, abs/1506.02078, 2015.

Yoon Kim, Yacine Jernite, David Sontag, and Alexander M. Rush. Character-aware neural language models. *CoRR*, abs/1508.06615, 2015.

Reinhard Kneser and Hermann Ney. Improved backing-off for m-gram language modeling. In *In Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume I, May 1995.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (eds.), *Advances in Neural Information Processing Systems 25*, pp. 1097–1105. Curran Associates, Inc., 2012.

R. Kuhn and R. De Mori. A cache-based natural language model for speech recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(6):570–583, 1990.

Jiwei Li, Xinlei Chen, Eduard H. Hovy, and Dan Jurafsky. Visualizing and understanding neural models in NLP. In *NAACL HLT 2016, The 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego California, USA, June 12-17, 2016*, pp. 681–691, 2016. URL `http://aclweb.org/anthology/N/N16/N16-1082.pdf`.

Tomas Mikolov, Ilya Sutskever, Anoop Deoras, Hai-Son Le, Stefan Kombrink, and Jan Cernocky. Subword language modeling with neural networks. Technical report, 2012. URL `www.fit.vutbr.cz/~imikolov/rnnlm/char.pdf`.

Kyoung-Su Oh and Keechul Jung. GPU implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, 2004.

Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pp. 419–428. ACM, 2014.

Veselin Raychev, Pavol Bielik, and Martin Vechev. Probabilistic model for code with decision trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pp. 731–747. ACM, 2016a.

Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pp. 761–774. ACM, 2016b.

Armando Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pp. 404–415, 2006.

Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. End-to-end memory networks. In *Advances in Neural Information Processing Systems 28*, pp. 2440–2448. Curran Associates, Inc., 2015.

Ilya Sutskever, James Martens, and Geoffrey Hinton. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning*, ICML '11, pp. 1017–1024. ACM, 2011.

Yee Whye Teh. A hierarchical bayesian language model based on pitman-yor processes. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th Annual Meeting of the Association for Computational Linguistics*, ACL-44, pp. 985–992, 2006.

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

Yuhuai Wu, Saizheng Zhang, Ying Zhang, Yoshua Bengio, and Ruslan Salakhutdinov. On multiplicative integration with recurrent neural networks. In D. D. Lee, U. V. Luxburg, Isabelle Guyon, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 29 (NIPS 2016)*, pp. 2856–2864. Curran Associates, Inc., 2016.

APPENDIX

We provide three appendices. In Appendix A, we describe how to obtain a probabilistic model from a `LMProgram`. In Appendix B, we provide details on how `TChar` programs are synthesized from data. Finally, in Appendix C, we provide detailed formal semantics of `TChar` programs.

A OBTAINING A PROBABILISTIC MODEL FROM `TChar`

We start by describing the program formulation used in the learning and then describe how a probability distribution is obtained from a program in `TChar`.

Dataset Given an input sentence $s = x_1 \cdot x_2 \cdot ... \cdot x_n$, the training dataset is defined as $\mathcal{D} = \{(x_t, x_{<t})\}_{t=1}^{n}$, where x_t is the character to be predicted at position t and $x_{<t}$ are all the preceding characters seen in the input ($x_{<1}$ is ϵ – the empty sequence).

Cost function Let $r\colon \wp(\textit{charpairs}) \times \texttt{TChar} \rightarrow \mathbb{R}^+$ be a cost function. Here, *charpairs* $= (char, char^*)$ is the set containing of all possible pairs of a character and a sequence of characters ($char$ can be any single character). Conceptually, every element from *charpairs* is a character and all the characters that precede it in a text. Then, given a dataset $\mathcal{D} \subseteq charpairs$ and a learned program $p \in \texttt{TChar}$, the function r returns the cost of p on $\mathcal{D}$ as a non-negative real number – in our case the average *bits-per-character* (BPC) loss:

$$r(\mathcal{D}, p) = \frac{1}{n} \sum_{t=1}^{n} -\log_2 P(x_t \mid x_{<t}, p) \tag{1}$$

Problem Statement The learning problem is the following:

$$\text{find a program } \; p_{best} = \arg\min_{p \in \texttt{TChar}} r(\mathcal{D}, p) + \lambda \cdot \Omega(p) \tag{2}$$

where $\Omega\colon \texttt{TChar} \rightarrow \mathbb{R}^+$ is a regularization term used to avoid over-fitting to the data by penalizing complex programs and $\lambda \in \mathbb{R}$ is a regularization constant. We instantiate $\Omega(p)$ to return the number of instruction in a program p. The objective of the learning is therefore to find a program that minimizes the cost achieved on the training dataset $\mathcal{D}$ such that the program is not overly complex (as controlled by the regularization).

Obtaining the Probability Distribution $P(x_t \mid x_{<t}, p)$. We use $P(x_t \mid x_{<t}, p)$ above to denote the probability of label x_t in a probability distribution specified by program p when executed on context $x_{<t}$ at position t. As described in Section 2 this probability is obtained in two steps:

- Execute program $p(t, x_{<t})$ to obtain a program $f \in \texttt{LMProgram}$.
- Calculate the probability $P_f(x_t \mid f(t, x_{<t}))$ using f.

The semantics of how programs p and f are executed are described in detail in Appendix C.1. In the remained of this section we describe how the probability distribution P_f is obtained. We note that $P(x_t \mid x_{<t}, p)$ is a valid probability distribution since $p(t, x_{<t})$ always computes a unique program f and the distributions $P_f(x_t \mid f(t, x_{<t}))$ are valid probability distributions as defined below.

Obtaining the Probability Distribution $P_f(x_t \mid f(t, x_{<t}))$. As discussed in Section 2, a function $f \in \texttt{LMProgram}$ is used to define a probability distribution. Let us show how we compute this distribution for both cases, first, when $f \in \texttt{SimpleProgram}$ and then, when f consists of several `SimplePrograms` connected via backoff.

Assume we are given a function $f \in \texttt{SimpleProgram}$ and a training sample $(x_t, x_{<t})$, where x_t is a character to be predicted at position t and $x_{<t}$ are all the preceding characters seen in the input ($x_{<1}$ is ϵ). The probability of label x_t is denoted as $P_f(x_t \mid f(t, x_{<t}))$ – that is, we condition the prediction on a context obtained by executing program f from position t in input $x_{<t}$ (the formal

semantics of executing such programs are defined in Section C.1). This probability is obtained using a maximum likelihood estimation (counting):

$$P_f(x_t \mid f(t, x_{<t})) = \frac{Count(f(t, x_{<t}) \cdot x_t)}{Count(f(t, x_{<t}))}$$

where $Count(f(t, x_{<t}) \cdot x_t)$ denotes the number of times label x_t is seen in the training dataset after the context produced by $f(t, x_{<t})$. Similarly, $f(t, x_{<t})$ denotes the number of times the context was seen regardless of the subsequent label. To deal with data sparseness we additionally adjust the obtained probability by using Witten-Bell interpolation smoothing Witten & Bell (1991).

Now lets us consider the case when f consists of multiple `SimplePrograms` with backoff, i.e., $f = f_1$ **`backoff`** d; f_2, where $f_1 \in$ `SimpleProgram` and $f_2 \in$ `LMProgram`.

Recall from Section 2 that we backoff to the next model if the probability of the most likely character according to the current model is less than a constant d. More formally:

$$P_f(x_t \mid f(t, x_{<t}))_{f=f_1 \texttt{ backoff } d;\ f_2} = \begin{cases} P_{f_1}(x_t \mid f_1(t, x_{<t})) & \text{if } d \geq \arg\max_{y \in V} P_{f_1}(y \mid f_1(t, x_{<t})) \\ P_{f_2}(x_t \mid f_2(t, x_{<t})) & \text{otherwise} \end{cases}$$

This means that if f_1 produces a probability distribution that has confidence greater than d about its best prediction, then f_1 is used, otherwise a probability distribution based on f_2 is used. When backoff is used, both probabilities are estimated from the same set of training samples (the dataset that would be used if there was no backoff).

From Programs to Probabilistic Models: an Illustrative Example We illustrate how programs in `TChar` are executed and a probability distribution is built using a simple example shown in Fig. 2. Consider the program $p \in$ `TChar` containing two `LMPrograms` f_1 and f_2 that define two probability distributions P_{f_1} and P_{f_2} respectively.

The program p determines which one to use by inspecting the previous character in the input. Let the input sequence of characters be "`a1a2b1b2`" where each character corresponds to one training sample. Given the training samples, we now execute p to determine which `LMProgram` to use as shown in Fig. 2 (left). In our case this splits the dataset into two parts – one for predicting letters and one for numbers.

For numbers, the model based on $f_1 =$ `LEFT LEFT WRITE_CHAR` conditions the prediction on the second to left character in the input. For letters, the model based on $f_2 =$ `empty` is simply an unconditional probability distribution since executing `empty` always produces empty context. As can be seen, the context obtained from f_1 already recognizes that the sequence of numbers is repeating. The context from f_2 can however be further improved, especially upon seeing more training samples.

B SYNTHESIZING `TChar` PROGRAMS

In this section we describe how programs in our `TChar` language are synthesized.

B.1 LEARNING SIMPLEPROGRAMS

We next describe the approach to synthesize good programs $f \in$ `SimpleProgram` such that they minimize the objective function:

$$f_{best} = \underset{f \in \texttt{SimpleProgram}}{\arg\min} \frac{1}{n} \sum_{t=1}^{n} -\log_2 P(x_t \mid f(t, x_t)) + \lambda \cdot \Omega(f)$$

We use a combination of three techniques to solve this optimization problem and find $f^{\approx}_{best}$ – an *exact* enumeration, *approximate* genetic programming search, and MCMC search.

Input:
"a1a2b1b2"

t	$x_{<t}$	x_t	$p(t, x_{<t})$
1		a	f_2
2	a	1	f_1
3	a1	a	f_2
4	a1a	2	f_1
5	a1a2	b	f_2
6	a1a2b	1	f_1
7	a1a2b1	b	f_2
8	a1a2b1b	2	f_1

t	$f_1(t, x_{<t})$	$P_{f_1}(x_t \mid f_1(t, x_{<t}))$
2	unk	$\frac{Count(\text{unk}\cdot 1)}{Count(\text{unk})} = \frac{1}{1} = 1.0$
4	1	$\frac{Count(1\cdot 2)}{Count(1)} = \frac{2}{2} = 1.0$
6	2	$\frac{Count(2\cdot 1)}{Count(2)} = \frac{1}{1} = 1.0$
8	1	$\frac{Count(1\cdot 2)}{Count(1)} = \frac{2}{2} = 1.0$

t	$f_2(t, x_{<t})$	$P_{f_2}(x_t \mid f_2(t, x_{<t}))$
1	ε	$\frac{Count(\varepsilon\cdot a)}{Count(\varepsilon)} = \frac{2}{4} = 0.5$
3	ε	$\frac{Count(\varepsilon\cdot a)}{Count(\varepsilon)} = \frac{2}{4} = 0.5$
5	ε	$\frac{Count(\varepsilon\cdot b)}{Count(\varepsilon)} = \frac{2}{4} = 0.5$
7	ε	$\frac{Count(\varepsilon\cdot b)}{Count(\varepsilon)} = \frac{2}{4} = 0.5$

```
p ::= switch LEFT WRITE_CHAR
        case 'a' or 'b' then LEFT LEFT WRITE_CHAR   (f1)
        default empty                               (f2)
```

Figure 2: Illustration of a probabilistic model built on an input sequence $'$a1a2b1b2$'$ using program p. (Left) The dataset of samples specified by the input and the result of executing program p on each sample. (Top Right) The result of executing program f_1 on a subset of samples selected by program p and the corresponding probability distribution P_{f_1}. (Bottom Right) The result of executing empty program f_2 and it's corresponding unconditioned probability distribution P_{f_2}. We use the notation ε to describe an empty sequence.

Enumerative Search We start with the simplest approach that enumerates all possible programs up to some instruction length. As the number of programs is exponential in the number of instructions, we enumerate only short programs with up to 5 instructions (not considering PREV_CHAR(c)) that contain single Write instruction. The resulting programs serve as a starting population for a follow-up genetic programming search.

The reason we omit the PREV_CHAR(c) instruction is that this instruction is parametrized by a character c that has a large range (of all possible characters in the training data). Considering all variations of this instruction would lead to a combinatorial explosion.

Genetic Programming Search The genetic programming search proceeds in several epochs, where each epoch generates a new set of candidate programs by randomly mutating the programs in the current generation. Each candidate program is generated using following procedure:

1. select a random program f from the current generation
2. select a random position i in f, and
3. apply mutation that either removes, inserts or replaces the instruction at position i with a randomly selected new instruction (not considering PREV_CHAR(c)).

These candidate programs are then scored with the objective function (2) and after each epoch a subset of them is retained for the next generation white the rest is discarded. The policy we use to discard programs is to randomly select two programs and discard the one with worse score. We keep discarding programs until the generation has less than 25 candidate programs. Finally, we do not apply a cross-over operation in the genetic search procedure.

Markov Chain Monte Carlo Search (MCMC) Once a set of candidate programs is generated using a combination of enumerative search and genetic programming, we apply MCMC search to further improve the programs. This procedure is the only one that considers the PREV_CHAR(c) instruction (which has 108 and 212 variations for the *Linux Kernel* and *Hutter Prize Wikipedia* datasets respectively).

The synthesized program is a combination of several basic building blocks consisting of a few instructions. To discover a set of good building blocks, at the beginning of the synthesis we first build a probability distribution I that determines how likely a building block will be the one with the best cost metric, as follows:

- consider all building blocks that consist of up to three `Move` and one `Write` instruction, $\mathcal{B} : \{\texttt{empty}, \texttt{Move}\}^3 \times \texttt{Write}$.
- score each building block $b \in \mathcal{B}$ on the full dataset $\mathcal{D}$ by calculating the bits-*per-character* (BPC) b_{bpc} as defined in (1) and the error rate b_{error_rate} (as usually defined) on dataset $\mathcal{D}$.
- accept the building block with probability $min(1.0, \texttt{empty}_{bpc}/b_{bpc})$ where $\texttt{empty}_{bpc}$ is the score for the unconditioned `empty` program. Note that for the BPC metric, lower is better. That is, if the program has better (lower) BPC than the `empty` program it is always accepted and accepted otherwise with probability $\texttt{empty}_{bpc}/b_{bpc}$.
- for an accepted building block b, set the score as $I'(b) = 1.0 - b_{error\_rate}$, that is, the score is proportional to the number of samples in the dataset $\mathcal{D}$ that are classified correctly using building block b.
- set the probability with which a building block $b \in \mathcal{B}$ will be sampled by normalizing the distribution I', that is, $I(b) = I'(b)/\sum_{b' \in \mathcal{B}} I'(b')$.

Given the probability distribution I, we now perform random modifications of a candidate program p by appending/removing such blocks according to the distribution I in a MCMC procedure that does a random walk over the set of possible candidate programs. That is, at each step we are given a candidate program, and we sample a random piece from the distriution I to either randomly add it to the candidate program or remove it (if present) from the candidate program. Then, we keep the updated program either if the score (2) of the modified candidate program improves, or sometimes we randomly keep it even if the score did not improve (in order to avoid local optimums).

Backoff Programs The synthesis procedure described above can be adapted to synthesize `SimplePrograms` with backoff using a simple greedy technique that synthesizes the backoff programs one by one. In our implementation we however synthesize only a single `SimpleProgram` and then backoff to *n*-grams as described in Section 3. We optimize the backoff threshold $d \in \langle 0, 1 \rangle$ using a grid search with a step size 0.02 on a validation dataset.

B.2 Learning SwitchPrograms

Recall from Fig. 1 that the `SwitchPrograms` have the following syntax:

```
SwitchProgram ::= switch SimpleProgram
                    case v_1 or ··· or v_k then TChar
                    ···
                    case v_j or ··· or v_n then TChar
                    default TChar
```

To synthesize a `SwitchProgram` we need to learn the following components: (i) the predicate in the form of a `SimpleProgram`, (ii) the branch targets $v_1, v_2, ..., v_n$ that match the values obtained by executing the predicate, and (iii) a program for each target branch as well as a program that matches the default branch. This is a challenging task as the search space of possible predicates, branch targets and branch programs is huge (even infinite).

To efficiently learn `SwitchProgram` we use a decision tree learning algorithm that we instantiate with the `ID3+` algorithm from Raychev et al. (2016a). The main idea of the algorithm is to synthesize the program in smaller pieces, thus making the synthesis tractable in practice by using the same idea as in the ID3 decision tree learning algorithm. This synthesis procedure runs in two steps – a top-down pass that synthesizes branch predicates and branch targets, followed by a bottom-up pass that synthesizes branch programs and prunes unnecessary branches.

The top-down pass considers all branch programs to be an unconditioned probability distribution (which can be expressed as an empty `SimpleProgram`) and searches for best predicates and branch targets that minimize our objective function (i.e., regularized loss of the program on the training dataset). For a given predicate we consider the 32 most common values obtained by executing the predicate on the training data as possible branch targets. We further restrict the generated program to avoid over-fitting to the training data by requiring that each synthesized branch contains either more than 250 training data samples or 10% of the samples in the training dataset. Using this approach we search for a good predicate using the same techniques as described in Appendix B.1 and then apply the same procedure recursively for each program in the branch. The synthesis stops when no predicate is found that improves the loss over a program without branches. For more details and analysis of this procedure, we refer the reader to the work of Raychev et al. (2016a).

The bottom-up pass then synthesizes approximately best programs for individual branches by invoking the synthesis as described in Appendix B.1. Additionally, we prune branches with higher loss compared to a single `SimpleProgram` trained on the same dataset.

C Formal Semantics of the TChar Language

This section provides the formal semantics of `TChar`.

C.1 Small-step Semantics of TChar.

We formally define `TChar` programs to operate on a state $\sigma = \langle x, i, ctx, counts \rangle \in States$ where the domain $States$ is defined as $States = char^* \times \mathbb{N} \times Context \times Counts$, where $x \in char^*$ is an input sequence of characters, $i \in \mathbb{N}$ is a position in the input sequence, $Context : (char \cup \mathbb{N})^*$ is a list of values accumulated by executing `Write` instructions and $Counts$: `StateSwitch` $\rightarrow \mathbb{N}$ is mapping that contains a value denoting current count for each `StateSwitch` program. Initially, the execution of a program $p \in lang$ starts with the empty context $[] \in Context$ and counts initialized to value 0 for every `StateSwitch` program, i.e., $counts_0 = \forall sp \in$ `StateSwitch` $. 0$. For a program $p \in$ `TChar`, an input string x, and a position i in x, we say that p computes a program $m \in$ `LMProgram`, denoted as $p(i, x) = m$, iff there exists a sequence of transitions from configuration $\langle p, x, i, [], counts_{i-1} \rangle$ to configuration $\langle m, x, i, [], counts_i \rangle$. As usual, a configuration is simply a program coupled with a state. The small-step semantics of executing a `TChar` program are shown in Fig. 3. These rules describe how to move from one configuration to another configuration by executing instructions from the program p. We next discuss these semantics in detail.

C.1.1 Semantics of Write Instructions

The semantics of the `Write` instructions are described by the [Write] rule in Fig. 3. Each write accumulates a value c to the context ctx according to the function wr:

$$wr : \texttt{Write} \times char^* \times \mathbb{N} \rightarrow char \cup \mathbb{N}$$

defined as follows:

- $wr(\texttt{WRITE\_CHAR}, x, i)$ returns character at position i in the input string x. If i is not within the bounds of x (i.e., $i < 1$ or $i \geq len(x)$) then a special unk character is returned.
- $wr(\texttt{WRITE\_HASH}, x, i)$ returns the hash of all characters seen between the current position i and the position of the latest `Write` instruction that was executed. More formally, let i_{prev} be the position of the previous write. Then $wr(\texttt{WRITE\_HASH}, x, i) = H(x, i, min(i + 16, i_{prev}))$, where $H : char^* \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ is a hashing function that hashes characters in the string from the given range of positions. The hashing function used in our implementation is defined as follows:

$$H(x, i, j) = \begin{cases} x_i & \text{if } i = j \\ H(x, i, j-1) * 137 + x_j & \text{if } i < j \\ \perp & \text{otherwise} \end{cases}$$

  where we use $\perp$ to denote that the hash function returns no value. This happens in case $i > j$ (i.e., $i > i_{prev}$).

$$x \in char^* \qquad i, n \in \mathbb{N} \qquad ctx \in Context \qquad s \in \texttt{TChar}$$
$$op \in \{\texttt{SwitchProgram}, \texttt{StateUpdate}, \texttt{StateSwitch}\} \qquad op.guard \in \texttt{SimpleProgram}$$
$$op.cases \in Context \hookrightarrow \texttt{TChar} \qquad op.default \in \texttt{TChar} \qquad counts : \texttt{StateSwitch} \to \mathbb{N}$$

$$\frac{w \in \texttt{Write} \qquad v = wr(w, x, i)}{\langle w :: s, x, i, ctx, counts\rangle \to \langle s, x, i, ctx \cdot v, counts\rangle} \text{ [WRITE]}$$

$$\frac{m \in \texttt{Move} \qquad i' = mv(m, x, i)}{\langle m :: s, x, i, ctx, counts\rangle \to \langle s, x, i', ctx, counts\rangle} \text{ [MOVE]}$$

$$\frac{op \in \texttt{SwitchProgram} \qquad \langle op.guard, x, i, []\rangle \to \langle \epsilon, x, i', ctx_{guard}\rangle \qquad ctx_{guard} \in dom(op.cases)}{\langle op, x, i, ctx, counts\rangle \to \langle op.cases(ctx_{guard}), x, i, [], counts\rangle} \text{ [SWITCH]}$$

$$\frac{op \in \texttt{SwitchProgram} \qquad \langle op.guard, x, i, []\rangle \to \langle \epsilon, x, i', ctx_{guard}\rangle \qquad ctx_{guard} \notin dom(op.cases)}{\langle op, x, i, ctx, counts\rangle \to \langle op.default, x, i, [], counts\rangle} \text{ [SWITCH-DEF]}$$

$$\frac{op \in \texttt{StateUpdate} \qquad \langle op.guard, x, i, []\rangle \to \langle \epsilon, x, i', ctx_{guard}\rangle \qquad ctx_{guard} \in dom(op.cases)}{\langle op :: s, x, i, ctx, counts\rangle \to \langle op.cases(ctx_{guard}) :: s, x, i, [], counts\rangle} \text{ [STATEUPDATE]}$$

$$\frac{op \in \texttt{StateUpdate} \qquad \langle op.guard, x, i, []\rangle \to \langle \epsilon, x, i', ctx_{guard}\rangle \qquad ctx_{guard} \notin dom(op.cases)}{\langle op :: s, x, i, ctx, counts\rangle \to \langle s, x, i, [], counts\rangle} \text{ [STATEUPDATE-DEF]}$$

$$\frac{op_1 \in \{\texttt{Inc}, \texttt{Dec}\} \qquad n = update(op_1, counts(op_2)) \qquad counts' = counts[op_2 \to n]}{\langle op_1 :: op_2, x, i, ctx, counts\rangle \to \langle op_2, x, i, [], counts'\rangle} \text{ [UPDATE]}$$

$$\frac{op \in \texttt{StateSwitch} \qquad counts[op] \in dom(op.cases)}{\langle op, x, i, ctx, counts\rangle \to \langle op.cases(counts(op)), x, i, [], counts\rangle} \text{ [STATESWITCH]}$$

$$\frac{op \in \texttt{StateSwitch} \qquad counts[op] \notin dom(op.cases)}{\langle op, x, i, ctx, counts\rangle \to \langle op.default, x, i, [], counts\rangle} \text{ [STATESWITCH-DEF]}$$

Figure 3: TChar language small-step semantics. Each rule is of the type: $\texttt{TChar} \times States \to \texttt{TChar} \times States$.

- $wr(\texttt{WRITE\_DIST}, x, i)$ returns a distance (i.e., the number of characters) between the current position and the position of latest Write instruction. In our implementation we limit the return value to be at most 16, i.e., $wr(\texttt{WRITE\_DIST}, x, i) = min(16, |i - i_{prev}|)$.

C.1.2 SEMANTICS OF Move INSTRUCTIONS

The semantics of Move instructions are described by the [MOVE] rule in Fig. 3. Each Move instruction changes the current position in the input by using the following function mv:

$$mv\colon \texttt{Move} \times char^* \times \mathbb{N} \to \mathbb{N}$$

defined as follows:

- $mv(\texttt{LEFT}, x, i) = max(1, i - 1)$, that is, we move to the position of the previous character in the input.
- $mv(\texttt{RIGHT}, x, i) = min(len(x) - 1, i + 1)$, that is, we move to the position of the next character in the input. We use $len(x)$ to denote the length of x.
- $mv(\texttt{PREV\_CHAR}, x, i) = i'$, where i' is the position of the most recent character with the same value as the character at position i, i.e., maximal i' such that $x_{i'} = x_i$ and $i' < i$. If no such character is found in x, a value -1 is returned.

- $mv(\texttt{PREV\_CHAR}(\texttt{c}), x, i) = i'$, where i' is the position of the most recent character with the same value as character c, i.e., maximal i' such that $x_{i'} = c$ and $i' < i$. If no such character c is found in x a value -1 is returned.
- $mv(\texttt{PREV\_POS}, x, i) = i'$, where i' is the position of the most recent character for which executing program p results in the same $f \in \texttt{LMProgram}$, i.e., maximal i' such that $p(i', x_{<i'}) = p(i, x_{<i})$ and $i' < i$. This means that the two positions fall into the same branch of the program p. If no such i' is found in x a value -1 is returned. We note that two `LMPrograms` are considered equal if they have the same identity and not only when they contain same sequence of instructions (e.g., programs f_1 and f_2 in the example from Section A are considered different even if they contain same sequence of instructions).

C.1.3 SEMANTICS OF `SwitchProgram`

The semantics of `SwitchProgram` are described by the [SWITCH] and [SWITCH-DEF] rules. In both cases the guard of the `SwitchProgram` denoted as $op.guard \in \texttt{SimpleProgram}$ is executed to obtain the context ctx_{guard}. The context is then matched against all the branch targets in $op.cases$ which is a partial function from contexts to `TChar` programs. If an exact match is found, that is the function is defined for a given context ctx_{guard} denoted as $ctx_{guard} \in dom(op.cases)$, then the corresponding program is selected for execution (rule [SWITCH]). If no match is found, the default program denotes as $op.default$ is selected for execution (rule [SWITCH-DEF]).

In both cases, the execution continues with the empty context and with the original position i with which `SwitchProgram` was called.

C.1.4 SEMANTICS OF `StateProgram`

The semantics of `StateProgram` are described by rules [STATEUPDATE], [UPDATE] and [STATESWITCH]. In our work the state is represented as a set of counters associated with each `SwitchProgram`. First, the rule [STATEUPDATE] is used to execute `StateUpdate` program which determines how the counters should be updated. The execution of `StateUpdate` is similar to `SwitchProgram` and results in selecting one of the update operations `INC`, `DEC` or `SKIP` to be executed next.

The goal of the $update$ instructions is to enable the program to count (e.g. opening, closing brackets and others). Their semantics are described by the [UPDATE] rule and each instruction computes value of the updated counter and is described by following function $update$:

$$update\colon \{\texttt{INC}, \texttt{DEC}, \texttt{SKIP}\} \times \mathbb{N} \to \mathbb{N}$$

defined as follows:

- $update(\texttt{INC}, n) = n + 1$ increments the value.
- $update(\texttt{DEC}, n) = max(0, n - 1)$ decrements the value. Bounded from below by value 0.
- $update(\texttt{SKIP}, n) = n$ keeps the value unchanged.

After the value of counter for given `SwitchProgram` is updated, then depending on its value next program to execute is selected by executing the `SwitchProgram` as described by rules [STATESWITCH] and [STATESWITCH-DEF].