Code Prediction by Feeding Trees to Transformers

Seohyun Kim[†]
Facebook Inc.
U.S.A.
skim131@fb.com

Jinman Zhao[†]
University of Wisconsin-Madison
U.S.A.
jz@cs.wisc.edu

Yuchi Tian
Columbia University
U.S.A.
yuchi.tian@columbia.edu

Satish Chandra
Facebook Inc.
U.S.A.
schandra@acm.org

***Abstract*—Code prediction, more specifically autocomplete, has become an essential feature in modern IDEs. Autocomplete is more effective when the desired next token is at (or close to) the top of the list of potential completions offered by the IDE at cursor position. This is where the strength of the underlying machine learning system that produces a ranked order of potential completions comes into play.**

We advance the state-of-the-art in the accuracy of code prediction (next token prediction) used in autocomplete systems. Our work uses Transformers as the base neural architecture. We show that by making the Transformer architecture aware of the syntactic structure of code, we increase the margin by which a Transformer-based system outperforms previous systems. With this, it outperforms the accuracy of several state-of-the-art next token prediction systems by margins ranging from 14% to 18%.

We present in the paper several ways of communicating the code structure to the Transformer, which is fundamentally built for processing sequence data. We provide a comprehensive experimental evaluation of our proposal, along with alternative design choices, on a standard Python dataset, as well as on Facebook internal Python corpus. Our code and data preparation pipeline will be available in open source.

***Index Terms*—code embedding, code prediction, autocomplete**

I. Introduction

A. Code Prediction

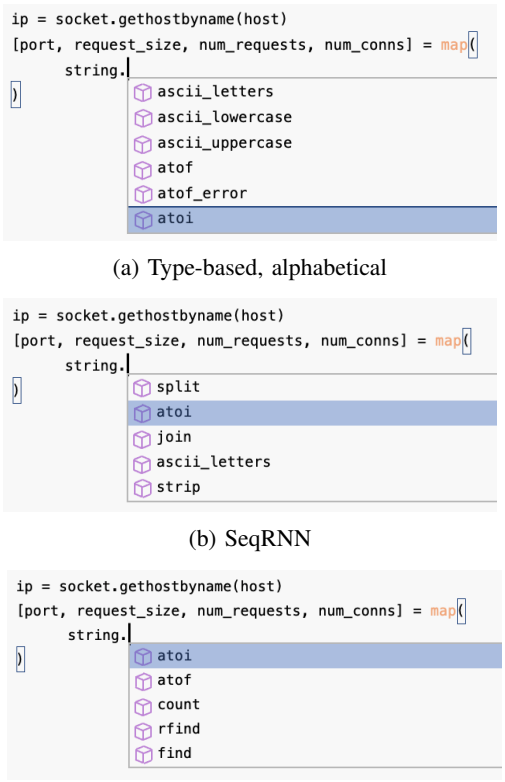
The idea of code prediction in general is to predict some code element, given code surrounding the point of prediction. Code prediction is commonly used in an IDE for *autocomplete*, where based on the code already written up to the developer's cursor position, the IDE offers the most likely next tokens, perhaps as a drop down list to choose from as shown in IDE views in Fig 1. Other forms of code prediction could predict missing tokens at arbitrary code locations or predict larger units of code; in this paper we will concern ourselves with prediction of the immediate next token at the cursor position.

Consider the Python code fragment shown in Fig 1. Suppose a developer has written code up to `string` following by a dot. At this point, it will be helpful for the IDE to prompt the developer with attribute names that are *likely* to follow, preferably, with `atoi` ranked at the top because in this case that is the correct next token.

Developers have come to rely on autocomplete in their IDEs for multiple reasons. First, and most obviously, it saves the effort of typing in the next token(s) in the IDE. For this reason alone, most modern IDEs come with at least some autocomplete facility for the languages they support. Notice that a top-ranked suggestion is often selectable by hitting a tab—as would be the case in Fig 1(c)—whereas the lower ranked suggestions have to be selected by scrolling (Fig 1(a,b)), which is more effort. Thus, providing the right completion at the top of the list, or if not, among the top few, is important.

Second, autocomplete is also a powerful code discovery mechanism. For instance, a developer might not know the name of an API call they need off the top of their head, but is

[†]Both authors contributed equally to this research.



(a) Type-based, alphabetical

(b) SeqRNN

(c) TravTrans

Fig. 1: Screenshots of ranked autocomplete predictions from three different models–Type-based alphabetical, SeqRNN, and TravTrans, as *would* appear in an IDE. A type-based autocomplete tool such as Jedi that sorts choices alphabetically ranks "atoi" low. SeqRNN, a RNN-based model predicts it as the second result. TravTrans, a Transformer-based model predicts it as the first result. Fewer keystrokes are needed to choose the correct answer as we go from left to right.

able to choose among the choices shown by an autocomplete tool. Without assistance from IDE, the developer might need to change their mental context, go to Stack Overflow or some other web site, and come back to the IDE. However, the code discovery assistance from autocomplete works only when a contextually appropriate code suggestion is offered among the *top* choices in the list, because developers do not have the time to go through a comprehensive list of completions.

B. Machine Learning for Code Prediction

It is clear that effective autocomplete requires the intended next token to be predicted at the top of the list, or as close to the top as possible. Type-based autocomplete (e.g. Eclipse[1] and Jedi[2]) returns a list of type-compatible names, but does naive ranking: alphabetically or with simple count-based statistics. For instance, the autocomplete model used in the IDE (Jedi, which ranks type-compatible suggestions alphabetically) shown in Figure 1(a) ranks `atoi` fairly low. The example shows why one of the earliest attempts of code prediction, using type-based methods, is not very effective. Additionally, for dynamic languages, it is extremely difficult to gather an accurate type-compatible list of tokens that could occur in a context.

These limitations have motivated the use of machine learning for code prediction, as machine learning methods are able to base their predictions on the *naturalness* [1] of code. Early approaches adapted n-gram language models on linearized source code tokens [2], [3]. More recently, deep neural networks have been applied to code prediction, surpassing n-gram models. The most common neural technique for code prediction is Recurrent Neural Networks (RNNs) [4] and their variants [5]–[10], where the code represented as a linear sequence is fed as input to the model. Fig 1(b) shows how an RNN model does better than a non-ML alphabetical ranking Fig 1(a) in showing the *expected* item closer to the top.

Researchers have also investigated using the *syntactic* structure of code for prediction, as opposed to seeing code as text: both using probabilistic graphical models (probabilistic context-free grammars [11] and probabilistic higher-order grammars [12]–[14]), as well as using deep learning [15].

C. Is this a solved problem?

Although predictive models cannot be expected to be perfect, the accuracy of the current state-of-the-art methods leaves substantial margin to be improved. For instance, a typical RNN-based method provides less than 37% mean reciprocal rank (this equates to the correct answer being in the top $(37\%)^{-1} \approx 2.7$ results, Sec IV-C) on the py150 benchmark. **Improving this metric is exactly the goal of this paper.** We report that our techniques are able to suggest the correct next tokens at ranks better — showing correct result to the developer by 0.5 to 1 ranks higher — than those achieved by previous methods (MRR increase of 14% to 18%.)

[1]https://www.eclipse.org/pdt/help/html/working_with_code_assist.htm

[2]https://github.com/davidhalter/jedi

```
...
ip = socket.gethostbyname (host)
[port, request_size, num_requests, num_conns] = map (
                string.atoi, sys.argv[2:]
)
chain = build_request_chain(num_requests, host, request_size)
...
```

Fig. 2: Running example of Python code. The code snippet [3]is from the py150 dataset [16].

As a concrete example: Table I shows the ranks of the various non-punctuation tokens to be predicted for the code in Figure 2, using various recent methods, as well as for our work. Specifically, the rank of `atoi` is predicted at rank 1 *only* by the new methods we propose in this paper.

D. Feeding Trees to Transformers

To improve the accuracy of next token predicted, a number of alternatives come to mind. First, we could strengthen the neural architecture alone. For instance, researchers have suggested adding *attention* to RNNs [5], [17] to compensate for loss of signal on long range dependence. Without long-range dependence a model would be making a (potentially sub-optimal) decision only on the most recent tokens. Transformers handle long-range dependencies better. In the NLP community, *Transformers* have achieved state-of-the-art results [18]–[20], outperforming RNNs, for a variety of NLP tasks such as language modeling, question answering, and sentence entailment. For us, since training Transformers did not take much more resources that training RNNs (Table IV), we decided to proceed with Transformers.

An orthogonal way to improve the accuracy is to enable the machine learning system to "see" more code structure. Raychev et al. [14] had found that—for the code prediction problem—a non-neural but AST-aware engine could outperform RNNs. In the same spirit, Alon et al. [21] had found—for code summarization problem (though *not* for code prediction in their paper)—that embedding the AST structure of code vastly outperformed purely sequence-based methods.

Inspired by the above, we explore how to leverage code structure while using Transformers on code. This is not obvious; we cannot simply jam an AST into the Transformer, which is a sequence processing model. We explore two models that represent two ways to capture the (partial) structure of an AST: one, based on decomposing the tree into paths (PathTrans), and the other, based on a tree traversal order (TravTrans). We also investigated a variant of TravTrans that takes into account even more tree structure.

E. Key Results

Tab II compares the new models we introduce in this paper (in bold), with three state-of-the-art models from previous work (Sec II-A discusses these models.) We report results based on training and evaluating on the py150 [16] dataset, and

[3]data/JeremyGrosser/supervisor/src/supervisor/medusa/test/test_11.py

	Token value	ip	socket	get*Name	host	map	string	atoi	sys	argv	2	chain
Previous work	SeqRNN	>10	>10	3	2	7	>10	2	**1**	**1**	3	>10
	Deep3	>10	9	4	>10	>10	>10	>10	6	**1**	7	>10
	Code2Seq	>10	2	**1**	3	>10	>10	8	**1**	**1**	3	>10
Our work	SeqTrans	>10	**1**	**1**	6	>10	>10	**1**	10	**1**	**1**	>10
	PathTrans	10	2	**1**	**1**	8	**1**	2	**1**	**1**	**1**	>10
	TravTrans	>10	**1**	5	**1**	4	**1**	**1**	**1**	**1**	**1**	>10

TABLE I: Ranks for the predictions for the leaf nodes listed in Fig 3. >10 means the model did not get the right answer in the top 10 results.

Model	Sequence	AST
Transformer	**SeqTrans (54.9%)**	**TravTrans (58.0%)** **PathTrans (55.1%)**
Attention		Code2Seq [15] (43.7%)
RNN	SeqRNN [22] (36.6%)	
Decision tree		Deep3 [14] (43.9%)

TABLE II: Overview of the models considered in this paper. Models in bold font are models from this work; SeqTrans feeds source tokens in linear order to the Transformer. The numbers in parenthesis denote accuracy (see Sec V.) It is clear that the accuracy increases as models uses information from the AST (columns), and as more sophisticated neural architectures are used (rows).

for each, we report accuracy in mean reciprocal rank (MRR) as a percentage (see Sec IV).

Our best model TravTrans, which communicates the tree structure to the Transformer, **significantly outperforms all previous models for code prediction,** with improvements in reciprocal ranks:

- from 43.9% to 58.0% when comparing a non-neural tree based model Deep3 [14] vs. TravTrans;
- from 43.6% to 58.0% when comparing Code2Seq [15] vs. TravTrans;
- from 36.6% to 54.9% when comparing an RNN implementation SeqRNN vs. TravTrans[4];

We also evaluated our trained model on a dataset selected from a Python code repository *internal* to a Facebook, and found the relative benefits of the Transformer models to be similar to those on the py150 dataset. This indicates that the relative advantage of Transformer models carries over to other datasets. (Sec V, Table VIII)

Deep learning models can be rather opaque as to their working. To better understand whether TravTrans is taking advantage of the tree structure, we employed saliency maps [23] to examine where the model focuses its attention when making a prediction. We found that indeed, the model tends to focus on the most relevant parts of the tree, starting from the parent node (Sec VI).

F. Contributions

We move the state-of-the-art forward in accurate next token prediction, a capability increasingly expected in modern IDEs.

- We describe ways of using Transformers for the task of next token prediction, especially ways that *profitably* communicate the syntactic structure of code. (Sec II). Although there have been previous work in applying Transformers in the context of code (code summarization [24], code correction [25], and code translation [26]), this paper is the among the first to explore and evaluate Transformers for code (next token) prediction.
- We present a systematic comparison of our proposed models with the most effective models from prior work[5] that are applicable to next token prediction, on a widely available Python dataset *py150*. The findings clearly indicate a 14% to 18% gain in accuracy with our best model relative to prior state-of-the-art.
- We provide a preliminary attribution study in an attempt to understand the prediction given by our best performing model, TravTrans. This study (Sec VI) indicates that TravTrans indeed conditions its predictions on the most pertinent tokens in the context. To our knowledge, this kind of model interpretability analysis for autocomplete is also a first.

Our overall conclusion is that Transformer based models over ASTs provide the best prediction power for autocomplete.

G. Outline

Sec II provides background on the previous models that we use in our evaluation, along with a primer on Transformers. Sec III explains the Transformer-based models of our own creation. Sec IV describes our datasets and implementation. Sec V presents our quantitative results. Sec VI takes a closer look into why our models worked well (or did not). Sec VII lists some threats to validity. Sec VIII discusses prior related work in the field of code prediction and Transformers. We conclude the paper with our future work in Sec IX.

II. Background

In this section, we define the code prediction task we examine in this work, followed by details of previous state-of-the-art methods of code prediction we use for comparison. We end the section with a brief introduction to the original Transformer model. We will refer to the nodes of the AST for Fig 2, as shown in Fig 3.

[4]comparing TravTrans against SeqRNN is slightly different than comparing it against Deep3 or Code2Seq, hence the difference in MRR. We discuss in detail in Sec V.

[5]Incidentally, this paper is also the first to compare the three prior works on a common dataset.

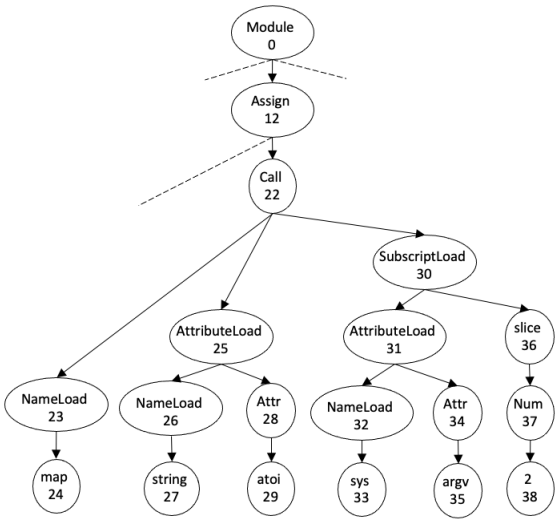


Fig. 3: Part of the AST for the example in Fig 2. The leaf (terminal) nodes have values and the interior (non-terminal) nodes have types.

A. Code Prediction Task

Code prediction task studied in this work is to predict the next code unit given the partial program up to the point of prediction. Let $p^*(unit \mid ctx)$ be the empirical distribution of code unit given the partial program context ctx. Our task is to learn to approximate p^* using a machine learning model M. In our proposals, M will be some Transformer $Trans$. The learned distribution can be viewed as

$$p(unit \mid ctx) = M(ctx; \boldsymbol{\theta}),$$

where $\boldsymbol{\theta}$ represents the trainable parameters of the model. We train the models by minimizing the KL-divergence between p and p^*, or equivalently, minimizing the cross-entropy loss l over all code prediction locations.

We explore several ways of representing partial program and predicting different kinds of code units. When using source code token as code unit and representing partial program as sequence of source code tokens (SeqTrans), the problem aligns with the traditional notion of language modeling – predicting the next token in a sequence given all previous tokens: $p(t_i \mid t_1, \ldots, t_{i-1})$ where t_i is the i-th token.

More interestingly, we explore various representations of a partial program to better utilize its AST information. The intuition is that the more we can utilize the syntactic information provided by the AST, the better we can predict the next token. The next section discusses three models used in previous work.

B. SeqRNN

For next token prediction, a popular method is to feed the source sequence tokens into an RNN (or LSTM) [8]–[10], [22]. An RNN embeds the input tokens into a vector: $\boldsymbol{x_t} = emb(w_t)$, where w_t is the source token seen at the t'th time step. The hidden state $\boldsymbol{h}_{t+1}$ at the $(t+1)$-th time step is computed as $\boldsymbol{h}_{t+1} = rnn(\boldsymbol{x}_t, \boldsymbol{h}_t)$, where rnn is a trainable RNN unit. The last hidden state is then fed through a classification layer.

The pertinent point to note is that the hidden state $\boldsymbol{h_t}$ encodes the knowledge of not just the current token, but of last several (and theoretically all) previous tokens via the propagation of information in previous hidden states.

In our experiments, we feed the source code tokens into an LSTM and call this model SeqRNN.

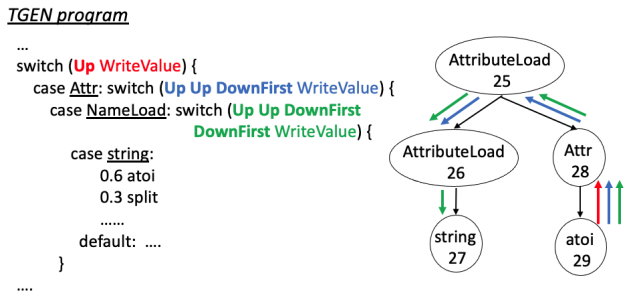


Fig. 4: Fragment of a TGEN program encoding a decision tree on the left (bold words are the steps that comprise a path), with the corresponding paths shown on the AST on the right.

C. Deep3

Raychev et al. [14] presented a system, Deep3, based on a learned decision tree combined with count-based probabilities at the leaves of the decision tree.

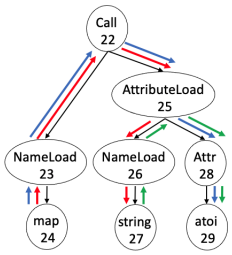
Fig 4 shows part of a learned decision tree, written in the form of program in a specialized language called TGEN. Given an AST t and a starting node n, a TGEN program walks certain paths in t starting from n. For example, `Up WriteValue` (line 1) goes to the parent of n and records the label. At the end of a TGEN program is a probability distribution for the possible values of the starting node. For example, starting with node 29, the TGEN program predicts "atoi" with 60%, "split" with 30%, etc.

A TGEN program is learned—on a specific corpus—by a genetic search procedure that simultaneously selects paths and grows the decision tree from the training data, with an entropy minimization objective. In this paper, we use their pretrained model [27] as well as their Python dataset [16] for our experiments.

D. Code2Seq

Code2Seq is a model by Alon et al. [15] that embeds code snippets by embedding AST paths in a neural network.

At a high-level, given an AST, Code2Seq creates path representations for all leaf-to-leaf paths. For example, Fig 5 shows three leaf-to-leaf paths for nodes 22-29 from the full AST (Fig 3). For each path, a path representation is created with: 1. the starting leaf value, tokenized by snake and camel case, 2. the path itself, and 3. the ending leaf value, also tokenized. 1 and 3 are embedded using LSTMs, and 2 is embedded using bi-directional LSTMs. These three embeddings



	start	Path	end
Path 1	map	map, NameLoad, Call, AttributeLoad, NameLoad, string	string
Path 2	map	map, NameLoad, Call, AttributeLoad, Attr, <placeholder>	<placeholder>
Path 3	string	string, NameLoad, AttributeLoad, Attr, <placeholder>	<placeholder>

Fig. 5: Example of an input for Code2Seq, which consists of leaf-to-leaf path representations given a partial AST. A path representation is made of tokenized starting tokens, path, and tokenized ending tokens. If the path ends with the target node (in this example, `atoi`), the value is replaced by `<placehholder>`.

are concatenated and then fed through a feed forward network. Finally, all of the path represention embeddings in the AST are combined using a simple attention mechanism.

In Code2Seq, a decoder is then used to solve a code summarization task: given a method body, how well can Code2Seq generate the correct method name? The training proposed in [15] is not well suited for next token prediction. In code summarization, a set of leaf-to-leaf paths needs to be created one time for a method. By contrast, in code prediction, a new set of leaf-to-leaf paths has to be created for each point of prediction.

For example, to predict `atoi` (node 29) in Fig 3, we must first create an representative embedding for the partially completed AST up to node 29, using all leaf-to-leaf paths available up to node 29. Paths that end in `atoi` are also used, with `atoi` replaced with a *placeholder* token to prevent information leak (e.g. Paths 2 and 3 in Fig 5). The representative embedding is then fed through a classification layer to generate predictions.[6]

By treating each point of prediction as a separate data point (compared to a language model, where one sequence is considered one data point), the number of training data points, along with the effort to create them makes Code2Seq computationally very expensive.

E. A Primer on Transformers

Here we present a brief introduction of Transformers. Readers familiar with Transformers can skip ahead to Section III.

Transformers belong to a class of deep neural networks that are designed for sequence processing. In Transformers, information from any previous location of the sequence can directly affect the encoding of the next token, through a mechanism called *self-attention*, which helps greatly improve the connectivity in long sequences.

To be precise, a Transformer is a stack of Attention blocks (AttnBlk) preceded by an input embedding layer (Emb) and

[6]Note that this is different than the generative decoder that Code2Seq uses.

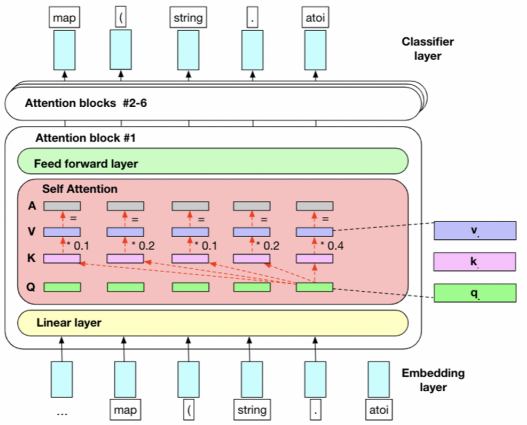


Fig. 6: Schematic of a GPT2 Transformer. The self-attention layer is able to consider all tokens in the input up to the point of prediction. Here the self-attention box depicts the information flow when predicting next token after the "."; see Table III for where the numbers come from.

followed by a classification layer (Clsfr), where AttnBlk is repeated n_{block} times.

$$Trans(ctx) = \text{Clsfr}(\text{AttnBlk}(\dots(\text{AttnBlk}(\text{Emb}(ctx)))\dots))$$

See Fig 6 for a schematic of a Transformer with embedding layer, a stack of (here 6) attention blocks, and finally a classification layer.

The self-attention layer—which constitutes the main part of an attention block—is the crux of the model. The intuition here is to attend to the elements in the input sequence in proportion of their relevance to the location being predicted. For example, take an example input token sequence ["map", "(", "string", "."], and the target next token being "atoi." It is first fed through the initial embedding layer to give: $\boldsymbol{E} = [e_{\text{map}}, e_{(}, e_{\text{string}}, e_{.}]$. Then, we feed $\boldsymbol{E}$ to three fully-connected networks $(\boldsymbol{W_q}, \boldsymbol{W_k}, \boldsymbol{W_v})$ to create query, key, and value embeddings:

$$\boldsymbol{Q} = \boldsymbol{E}\boldsymbol{W_q}, \boldsymbol{K} = \boldsymbol{E}\boldsymbol{W_k}, \boldsymbol{V} = \boldsymbol{E}\boldsymbol{W_v},$$

Fig 6 depicts the vectors $\boldsymbol{Q}$ comprised of its elements $q_{\text{map}}, q_{(}, q_{\text{string}}, q_{.}$ (in green), and likewise for $\boldsymbol{K}$ and $\boldsymbol{V}$.

The self-attention then works by querying keys k using queries q and then using the result to summarize values v through the attention function:

$$\text{Attn}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d_k}}\right)\boldsymbol{V}$$

where d_k is the dimension of key vectors. Here, $\boldsymbol{Q}\boldsymbol{K}^\top$ is used to determine which token relationships are the most important, resulting in a matrix of size $n \times n$, where n is the length of the input sequence. Each row is then normalized and passed through a softmax layer. Table III shows an example of the self-attention weights. Looking at the last row, we can see that most of the attention is given to ".", meaning it has a greater

...	map	(	string	.
map	0.9			
(	0.6	0.1		
string	0.1	0.1	0.7	
.	0.2	0.1	0.2	0.4

TABLE III: Example matrix for the numerical self-attention scores after taking the softmax over the normalized values of $\boldsymbol{QK}^\intercal$. For example, the entry against $(\text{string}, \text{map})$ is obtained by multiplying q_{string} with k_{map}, after softmax and normalization. Note that the rows listed here do not sum up to exactly 1 since there are previous tokens in the input sequence that are not shown in this matrix.

factor in predicting the next token "atoi". Note how the matrix is a lower triangular matrix: this is because self-attention cannot be applied to tokens that have not been seen before. After multiplying this matrix with the value vector, in our example, $\text{Attn}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = [0.2 * v_{\text{map}}, 0.1 * v_{(}, 0.2 * v_{\text{string}}, 0.4 * v_{.}]$.

Transformers also uses multiple heads of these self-attention blocks, called multi-headed attention, which enables the model to simultaneously consider different ways of attending to previous information within one block and also across other blocks. In our implementation, we omit positional encoding (see Sec IV-B.) For more details, please refer to [28] and [20].

The next sections discuss various ways of feeding code fragments into this Transformer architecture.

III. Our work: Transformer-based Models

The question that interests us is: can Transformer-based models also benefit from syntactic structure, and if so, how can we communicate the syntactic structure to Transformer?

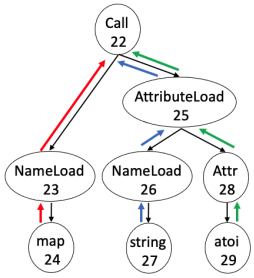
In this section, we first begin with a model SeqTrans that uses a Transformer to take source code tokens as input. Then we introduce two other models, PathTrans and TravTrans, that use more syntactic information obtained from the AST.

1) ***SeqTrans****:* Our first model is to apply a Transformer over source token sequences, which can be easily obtained by applying a tokenizer. Here the input is the partial program represented as source token sequences and the output is a source code token. This is a straightforward application of the original Transformer design, and functions as a baseline for our later attempts that take on more AST information. The model can be written as

$$\boldsymbol{o} = Trans\left((\boldsymbol{e}_t)_{t \in src\_seq}\right)$$

where $\boldsymbol{o}$ is a distribution over all possible tokens, and $\boldsymbol{e}_t$ is the embedding for source token t for every t in the source token sequence. As we show in the experiments, SeqTrans turns out to be an already strong model as a direct comparison to the baseline RNN model SeqRNN.

Since Transformers are originally designed as a sequential model, the challenge becomes finding ways to convey AST information to Transformers. In the next subsection, we will vary the inputs and the outputs to the Transformer, but the principles of operation will remain the same as in SeqTrans.



	leaf token	*root path*
Path 1	map	NameLoad, Call
Path 2	string	NameLoad, AttributeLoad, Call
Path 3	atoi	Attr, AttributeLoad, Call

Fig. 7: Example of an input to the PathTrans model. It takes all leaf nodes, along with its path to root, to make the next leaf token prediction. The root paths are embedded using an LSTM, and the leaf tokens are embedded using an embedding layer. These two embeddings are added together to create a path representation embedding, which is then fed to the Transformer as shown in Fig 6. The classification layer of the Transformer outputs leaf tokens.

2) ***PathTrans****:* PathTrans enhances SeqTrans by exposing tree structure to the Transformer via root-paths. A root-path is the path from the leaf node t to the root of the AST by traversing up its ancestors, recording all the nodes along with it, and thus a sequence of internal AST nodes. Fig 7 shows an example of an input datapoint for predicting node 29 of Fig 3.

The root-paths are first fed into a an LSTM[7] added with the embedding of the leaf node, and is fed through a Transformer:

$$\boldsymbol{o} = Trans\left((\boldsymbol{e}_t + \boldsymbol{p}_t)_{t \in leaf\_seq}\right)$$

where $\boldsymbol{o}$ is a distribution over all possible leaf nodes, and $\boldsymbol{e}_t$ is the embedding for AST node t and $\boldsymbol{p}_t = \text{LSTM}\left((\boldsymbol{e}_u)_{u \in Rootpath(t)}\right)$ is the summarized representation of root-path from t for every leaf node t in the leaf sequence $leaf\_seq$. [8] The hope here is that the root-paths captures the local syntactical information and thus can help the prediction.

Since the points of prediction are the leaf nodes of the AST, the loss is taken over only the leaf AST nodes, and it predicts only leaf tokens.

3) ***TravTrans****:* As a Transformer naturally only takes a sequence as input, we provide the AST nodes as a sequence in pre-order traversal, or a depth-first-search (DFS) order. For Fig 3, for node 29, the previous nodes in DFS order would be: [..., "Call", "NameLoad", "map", "AttributeLoad", "NameLoad", "string", "Attr"].

The TravTrans model can be written as:

$$\boldsymbol{o} = Trans\left((\boldsymbol{e}_t)_{t \in AST\_seq}\right)$$

[7]We could have used a Transformer to embed the path sequences in lieu of an LSTM, but since the path sequences are short (capped at 13 tokens) enough for LSTMs to perform adequately well, we decided to use an LSTM. See Sec IV-B for details.

[8]$+$ is used here following the convention of Transformer computations and to keep the embedding dimension the same for every component.

where $\boldsymbol{o}$ is a distribution over all possible tokens, and $\boldsymbol{e}_t$ is the embedding for AST token t for every t in the partial program represented as a AST token sequence AST_seq in DFS order.

4) Capturing even more AST structure?: TravTrans presents the tree nodes in a pre-determined order, but still does not retain detailed structural relationship between nodes. For example, consider the sequence of nodes 26 - 28 in Fig 3. This would be represented as ["NameLoad", "string", "attr"], the three nodes appearing consecutively in DFS order. Looking at the AST, we can see that the relations between ("NameLoad" & "string", and "string" & "attr") are actually quite different: "NameLoad" is one node up from "string", while "string" is two nodes up and one node down from "attr".

We create a new model, TravTrans+ that enhances TravTrans by capturing these richer path-based relations. Similarly to Hellendoorn et al. [29], we enhance the self attention block of the Transformer with a matrix $\boldsymbol{R}$ that captures the (unique) path needed to reach from a to b. This path is represented abstractly only in terms of up and down moves:

$$UDpath(a, b) = U^i D^j$$

where i, and j are the number of up and down nodes, respectively, node a has to travel to reach node b. For example, *UDpath*(29, 27) = U^2D^2 for node 29 and 27 in Fig 3. $\boldsymbol{R}$ is introduced to the Transformer by replacing the Attn function with the following $\text{Attn}_{\text{TreeRel}}$ function.

$$\text{Attn}_{\text{TreeRel}}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}, \boldsymbol{R}) = \text{softmax}\left(\frac{\boldsymbol{R} \odot (\boldsymbol{Q}\boldsymbol{K}^\top)}{\sqrt{d_k}}\right)\boldsymbol{V}$$

where $\odot$ is element-wise product. This provides a way for the self attention to consider the previous tokens, taking into account the AST relationship between pairs of nodes as well.

IV. Implementation and Datasets

A. Dataset

We train our models using the py150 dataset used in [14]. The dataset consists of 150k Python 2 source code files from GitHub repositories, along with their parsed ASTs, split into 100k for training and 50k for evaluation. In this work, we slightly modify the AST to ensure that the internal nodes only carry syntactic types and the leaf nodes only carry token values. To incorporate large trees (greater than 1000 nodes, which is the limit we chose for transformer window), we deploy a technique adopted by [30], which slices a large tree into shorter segments with a sliding window to maintain part of the previous context.

We evaluate our models on two evaluation datasets:

- **py150**: We use the evaluation dataset used in [14], which consists of 50k Python ASTs. After the modifications mentioned above, there are 16,003,628 leaf nodes.
- **internal**: We also created an evaluation dataset consisting of 5000 Python files from a code repository internal to Facebook. With this dataset, we can evaluate how our trained model can generalize to a different dataset, even if the code comes from disjoint projects. After the modifications, there are 1,669,085 leaf nodes.

B. Implementation

a) Transformers: For the models that use Transformers (SeqTrans, PathTrans, TravTrans, TravTrans+), we adapt the Pytorch implementation [9] of GPT-2 small [20]. We use six Transformer blocks, six heads in each block, $n\_ctx = 1000$, and $embedding\_dim = 300$. We borrow other hyperparameters from [20]. We limit the token vocabulary size to 100k, which covers over 90% of the tokens used in the training dataset. For PathTrans, we limit the maximum length of the path from leaf node to root to be 13, which covers over 90% of the nodes. For any path longer than 13, we keep the nodes closest to the leaf, and truncate the nodes near the root.

In our implementation, we did not use positional encoding [28] or positional embedding [20] to provide extra positional information over elements since our early trials with LeafSeq suggested positional embedding is rather hurting than helping. This is also supported by the claims in [31] that positional encoding does not help for language modeling. Recently, [26] tried to introduce tree structures to Transformer models via positional encoding. However, their relative improvement is small compared to what we see with tree-relational prior in Section V.

b) RNN: For SeqRNN, we adapt the PyTorch LSTM example implementation [10]. We use embedding dimension $d_{\text{model}} = 300$, with $dropout = 0.5$ and $n\_layers = 1$. We maintain the same vocabulary size at 100k.

c) Code2Seq: For Code2Seq, we used a PyTorch adaptation of the publicly released model[11], using the same hyperparameters, except changing the vocab size to 100k. For selecting 200 (max number of paths) paths per AST, we first picked paths that ended with the target (to maximize the amount of local context). Since for each prediction point in the AST, a new set of leaf-to-leaf paths have to be generated, the data processing for Code2Seq takes a substantial amount of time (magnitude of days worth of time).

We trained all models on Nvidia Tesla V100 (using 4 GPUs at a time) until the loss converged, with all of the parameters randomly initialized. We used the Adam optimizer with the learning rate set to 1e-3. Implementation details regarding number of epochs until convergence, training time (minutes per epoch), inference time (to evaluate over the py150 dataset), and model size, are listed in Table IV.

d) Deep3: For the Deep3 model, since the authors have shared only the model and not the training algorithm, we used the model pretrained on py150.

C. Evaluation Metric

We evaluate the models on next token prediction for the leaf tokens. We report numbers for all leaf token predictions, as well as breaking down into more interesting categories: attribute access, numeric constant, name (variable, module), and function parameter name.

[9] https://github.com/graykode/gpt-2-Pytorch.
[10] https://github.com/pytorch/examples/tree/master/word_language_model
[11] https://github.com/tech-srl/code2seq

	Prior work			Our work		
	SeqRNN	Deep3	Code2Seq	SeqTrans	PathTrans	TravTrans
Num epochs	9	n/a	10	9	16	11
Training time (min / epoch)	45	n/a	210	45	45	60
Inference time (min)	40	75	45	40	20	50
Model size (MB)	233	n/a	149	163	280	163

TABLE IV: Implementation details for all the models - number of epochs until convergence, training time (minutes per epoch), inference time (to evaluate over the py150 dataset), and model size (state dict of PyTorch - the learnable parameters of the model). Note that some information about Deep3 is not available since the authors have shared only the model.

To measure performance on these tasks, we use mean reciprocal rank (MRR). The rank is defined as

$$MRR = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{rank_i} \quad (1)$$

where n is the number of predicting locations and $rank_i$ is the rank of the correct label given by the model for the i^{th} data point. We present MRR as a percentage, in keeping with prior work [7], [32].

While Acc@1 only gives score when the correct label is ranked at the top, MRR also give scores when the true label is not ranked as the top, but among top few prediction. Comparing to the hit-or-miss style metric (Acc@1), this is closer to the realistic scenario when completion suggestions are presented to developers. With this practical perspective and for ease of computation, we only consider $rank_i \leq 10$ for each location i (all $rank_i > 10$ will have a score of 0). We share our data processing scripts and model implementations at https://github.com/facebookresearch/code-prediction-transformer.

V. Evaluation

RQ1: Given a source token sequence, does a Transformer work better than an RNN, as in previous work? Comparing SeqRNN and TravTrans, we find that a Transformer does work better than an RNN. For the py150 dataset, we can see a significant improvement in MRR for predicting all leaf tokens in Table V, from 36.6% to 50.1% for the SeqRNN and SeqTrans models, respectively. The same holds for comparing on the internal dataset, as shown in Table VI: 23.8% vs 36.5%. Consistent improvements can be seen for specific types of leaf tokens.

	Prior work	Our work	
Applications	SeqRNN	SeqTrans	TravTrans (type + value)
Attribute access	39.3%	55.9%	**60.4%**
Numeric constant	40.6%	55.9%	**57.0%**
Name (variable, module)	38.2%	54.1%	**62.7%**
Function parameter name	57.7%	**66.2%**	64.8%
All leaf tokens	36.6%	50.1%	**54.9%**

TABLE V: MRR of various types of next token predictions for py150. To fairly compare these models, TravTrans makes two predictions - one for the leaf node and then one for its parent internal node (see RQ3 for details).

RQ2: Do the Transformer models on tree outperform previous work on AST-based prediction?

	Prior work	Our work	
Applications	SeqRNN	SeqTrans	TravTrans (type + value)
Attribute access	26.4%	41.0%	**44.5%**
Numeric constant	32.2%	51.7%	**53.0%**
Name (variable, module)	25.0%	39.3%	**47.2%**
Function parameter name	45.5%	**54.3%**	51.4%
All leaf tokens	23.8%	36.5%	**40.7%**

TABLE VI: MRR of various types of next token predictions for internal dataset. TravTrans makes two predictions - one for the leaf node and one for its parent internal node (see RQ3 for details).

We compare Deep3 and Code2Seq against PathTrans, and TravTrans (Table VII). Overall, we found that both transformer-based models, achieve better scores than both Deep3 and Code2Seq for all leaf tokens as well as for specific types of leaf tokens. Our best performing model, TravTrans, improves Deep3's MRR by 14.1% (from 43.9% to 58.0%), and Code2Seq's MRR by 14.4% (from 43.7% to 58.0%). Similar results can be seen for the internal dataset (Table VIII).

We also compared the accuracy on AST internal predictions, comparison Deep3 and TravTrans, as they are the only models with this capability. In Table IX we see that TravTrans improves accuracy over Deep3 across the board. Table IX show non-terminal value prediction for both py150 and internal dataset, respectively. We see that TravTrans is outperforms Deep3 for all internal node types as well. [12]

	Prior work		Our work	
Applications	Deep3	Code2Seq	PathTrans	TravTrans
Attribute access	45.3%	39.3%	57.2%	**60.5%**
Numeric constant	53.2%	49.5%	59.1%	**63.5%**
Name (variable, module)	48.9%	45.8%	63.5%	**66.6%**
Function parameter name	58.1%	56.8%	65.8%	**67.2%**
All leaf nodes	43.9%	43.7%	55.1%	**58.0%**

TABLE VII: MRR of various types of next token predictions for py150.

RQ3: Does adding syntactic structure help? Comparing SeqTrans and TravTrans, we confirm that adding syntactic structure does help.

Comparing SeqTrans against TravTrans on leaf nodes fairly is a bit tricky. A source code token we are looking at here

[12]We do not include Code2Seq comparison for non-terminal node predictions due to the overhead required to prepare and process the dataset. Since the main part of the paper was on leaf token prediction, and we have shown that TravTrans performs significantly better than Code2Seq, we did not deem it essential to include the results on non-terminal value predictions.

	Prior work		Our work	
Applications	Deep3	Code2Seq	PathTrans	TravTrans
Attribute access	38.5%	26.4%	41.5%	**44.7%**
Numeric constant	46.5%	45.4%	56.1%	**61.5%**
Name (variable, module)	41.0%	31.2%	48.0%	**50.7%**
Function parameter name	50.6%	39.3%	52.1%	**53.3%**
All leaf nodes	31.6%	31.0%	40.8%	**43.9%**

TABLE VIII: MRR of various types of next token value prediction for internal dataset.

Dataset	**py150**		**internal**	
Applications	Deep3	TravTrans	Deep3	TravTrans
Function call	81.6%	**88.5%**	78.2%	**86.0%**
Assignment	76.5%	**78.9%**	78.5%	**79.7%**
Return	52.8%	**67.8%**	59.9%	**72.2%**
List	59.4%	**76.0%**	40.8%	**63.1%**
Dictionary	66.3%	**15.0%**	39.8%	**23.5%**
Raise	35.0%	**63.3%**	33.5%	**59.3%**
All types	81.9%	**87.3%**	79.9%	**87.7%**

TABLE IX: MRR of various type predictions for py150 and internal dataset.

contains both syntactic type and lexical value, which translates to two nodes in the AST. For example, the source code token "2" is equivalent to an internal node with value "Num" and its child leaf node "2". Thus, for a fair comparison, TravTrans has to make two predictions – one for the leaf node and one for its parent internal node. For example, given the sequence "y =", and the next prediction should be "2", TravTrans must predict both the internal "Num" node as well as the leaf "2" node. For evaluation, we implemented a local beam search to choose the pair with the maximum joint probability. Table V shows that TravTrans still performs better than SeqRNN and SeqTrans (except for predicting function parameter name) for predicting the leaf tokens.

It is important to note that even with this correction, it is tricky to completely fairly compare the accuracy of SeqTrans against TravTrans model on leaf nodes. The main issue here is that the contexts used to predict a leaf value is different in the two models, because of different order of seeing information. Consider the case of a code fragment `x = y + z`, and let us say we need to predict what comes after the "=". In the case of source token based prediction, the predictor has seen "x =" and would be expected to produce the next token ("y"). In an AST, one would first construct an "Assign" internal node, with the right child to be filled next, and the next prediction would be actually an interior node "BinOpPlus". The "y" would be predicted as the left child of BinOpPlus. In a context in which the AST has been augmented with the BinOpPlus node, the prediction of "y" has more information compared to what a source token model did immediately after the "=". This makes a direct comparison of token predictions difficult. One way to achieve better parity between the models would be to use an in-order traversal instead of a pre-order traversal; we chose the latter because we want to do top-down generation, which will be useful in future AST generative models.

Applications	TravTrans	TravTrans+
Attribute access.	60.5%	**61.2%**
Numeric constant	**63.5%**	**63.5%**
Name (variable, module)	66.6%	**67.7%**
Function parameter name	**67.2%**	67.0%
All leaf nodes	58.0%	**58.8%**
All internal nodes	87.3%	**91.9%**

TABLE X: MRR TravTrans compared against its variant TravTrans+ that incorporates more tree structure.

We also compare TravTrans with its variant TravTrans+ that extends the model to incorporate even more tree structure, as mentioned in Sec III. Table X shows a slight increase in MRR: 58.0% vs 58.8% for all leaf nodes, and 87.3% to 91.9% for internal nodes. Due to mixed benefit on leaf node predictions, we continue to treat TravTrans as our flagship model. But this does hint at the existence of more powerful models, if we incorporate more tree structure.

VI. Model Inspection

Despite their wide adoption in many areas of computing, deep learning models have been repeatedly shown susceptible to adversarial examples [33]–[35]. Interpretability studies [36]–[39] that provide human-understandable justifications, have thus become an important property in building safe and trustworthy AI systems. With the growing adoption of neural models for software engineering tasks, the same concern raises. Recent discoveries of unexpected behaviours of deep learning models of code [40]–[42] calls attention for such studies. Besides of helping understand the behaviour of deep learning models, interpretability methods have also been used to directly improve the process of software engineering. For example, [43], [44] used LIME [45], a model-agnostic explainability method, to pinpoint the defective lines of code from defect detection models.

As a crucial first step towards interpretability, in this section, we reveal what TravTrans has learned that leads to its good predicative power. We found that TravTrans has learned to attribute the prediction to relevant previous tokens. While this is not a principled study, we hope it sheds light on a possible approach to perform a complete model inspection study.

Although our Transformer-based models heavily relies on attentions, direct visualizations of weights in individual attention heads did not yield clear insights as the attentions are stacked across layers and multiple attention heads are in effect in each layer (see Sec II-E). We thus turn to gradient-based saliency maps [23], [46], [47], for a more comprehensive account of the influence of each input token. Following [23], the influence of each input token is computed by taking the partial derivative of the loss function with respect to the embedding vector of that token. Fig 8 is the saliency map we got for TravTrans, which visualizes the magnitudes of the gradients fall at each input token (x-axis) when the model predicts a particular output (y-axis). Intuitively, the larger the value for a particular token, the more sensitive the output is to the variations at that input token.

We first observe that the parent AST node (the internal node right above the leaf) is generally important in pre-

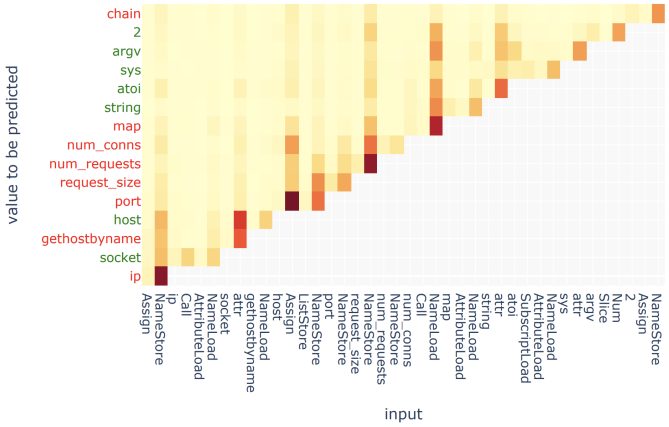


Fig. 8: Influence of previous nodes in value prediction of the example in Fig 3 by TravTrans. x-axis is labeled with the input values. y-axis is labeled with the values to be predicted. Color indicates the model's prediction is correct or wrong.

dicting the next leaf node value: the last token usually has a high influence. More generally, we found that TravTrans tends to attribute more towards the internal nodes that are directly followed by relevant leaf nodes. For example, we can see the prediction of `atoi` is influenced by (the locations previous to) `string`, `map` and `num_requests`. It is not shown in the figure but the predictions of `sys` and `argv` before `2` are influenced by the previous occurrence of the same values followed by `0` and `1`. The prediction of `host`, `gethostbyname` are influenced by (the location previous to) `ip`. For the position of `gethostbyname`, TravTrans predicts `socket` with probability 0.31, with the correct answer ranked second with slightly lower probability 0.24. All above suggest that TravTrans is capable of picking up information from relevant previous code locations. The reason TravTrans tends to focus more on the parent nodes may be that all relevant information for the next token are already summarized in the hidden representation at the location previous to the token (aka parent nodes for the leaf nodes).

On an orthogonal note, we also observe that for many predicting locations, the magnitude of gradients are very small, suggesting the robustness of the model in the sense that it is less sensitive to minor perturbations of the input sequence.

VII. Threats to Validity

a) Out-of-vocabulary (OOV) handling: One of the challenges in next-token prediction is predicting tokens that are rare or unseen in the training corpus.

When trained on py150, 21% and 35% of prediction locations observe OOV tokens for the py150 and the internal test set, respectively. None of the models in our main evaluation handle OOV predictions and thus are unable to correctly predict for such locations.

Handling of OOV tokens, while important, is orthogonal to core model design, and is outside the scope of our current work. Adding OOV handling to a model should further improve its performance, as suggested by previous works [5], [7], [48]. Moreover, a quick comparison (see Sec VIII) against PointerMixture from [5] found that TravTrans already has an edge even without OOV handling for the py150 dataset. We thus do not expect OOV handling to reverse our findings.

b) Training Corpus: While larger Python corpora have appeared, py150 is still sizable at 500MB; we do not expect the larger corpora to reverse our findings.

c) Python specificity: We have only carried out evaluations on Python, and have not demonstrated that our results would carry over (in trends) to other languages. The Deep3 paper did find their results (in trends) to roughly carry over from Python to JavaScript.

d) Model comparison: To bound the scope of this paper, we limit our investigation to techniques that *do not* require any compiler support beyond constructing the AST; thus, we exclude ways to communicate program analysis results such as def-use information, as in [49]. We also limit our study to the Transformer architecture, and purposely exclude the graph neural networks (GNN), because there is already an active debate between GNN and Transformer for NLP applications that transcends code prediction.

VIII. Related Work

We compared our models extensively with three baselines: SeqRNN, Deep3 and Code2Seq, both qualitatively in Sec II and quantitatively in Sec V. Before proceeding to a broader discussion, we briefly compare our work with the attentive pointer mixture model (PointerMixture) from Li at al. [5] as another plausible baseline. We are particularly interested in this model, because it (1) works with serialized AST node sequences, (2) uses attention mechanism on top of LSTM, and (3) has reportedly achieved better accuracy than Deep3's results.

The similarities and differences in relation to our work are as follows. For (1), the preorder traversal of the AST they used is similar to ours. They did not separate types and values into different nodes as we do, which makes their sequences more compact than ours. For (2), their attention is applied within a fixed-sized window on top of LSTM outputs, whereas our Transformer-based models use stacked layers of attentions as the key mechanism for computing throughout the network. [5] also used "parent attention", which is based on the parent-child relation in the AST. In comparison, our PathTrans and TravTrans+ extend far beyond the direct parent-child relation by making use of paths.

For (3), the accuracy reported in [5] included the predictions of null values for internal nodes, whereas our numbers only consider leaf nodes with non-empty values. To fairly compare the predicative power between our proposal and theirs, we implemented PointerMixture in our setting.[13] Note that PointerMixture uses a *pointer network* [50] which decides for each

[13]To maintain consistency, the hidden size and the embedding sizes were set to 300, and the vocab size was increased to 100k. The model was trained for 8 epochs.

Dataset	py150			internal		
Applications	PointerMixture	TravTrans	OOV Rate (%)	PointerMixture	TravTrans	OOV Rate (%)
Attribute access	54.2%	**60.5%**	19.3%	**50.4%**	44.7%	33.8%
Numeric constant	50.6%	**63.5%**	10.3%	40.4%	**61.5%**	6.3%
Name (variable, module)	49.5%	**66.6%**	15.5%	42.1%	**50.7%**	31.4%
Function parameter name	59.7%	**67.2%**	7.7%	**55.2%**	53.3%	18.7%
All leaf nodes	51.9%	**58.0%**	21.1%	**46.3%**	43.9%	34.8%

TABLE XI: MRR of PointerMixture compared against TravTrans for various types of next token prediction for py150 and internal dataset. The out-of-vocabulary (OOV) rates for internal dataset is much higher compared to py150 dataset.

point of prediction whether to use the LSTM output or to copy from a previous location, whereas our TravTrans has no means for handling out-of-vocabulary (OOV) tokens (i.e. all predictions requiring an OOV token are incorrect.)

We found on the basis of py150 dataset that PointerMixture outperforms SeqRNN, as well as the other baselines of Deep3 and Code2Seq. However, TravTrans *outperforms* PointerMixture, even though all the OOV predictions count as wrong for TravTrans! For the internal dataset, PointerMixture model does better than TravTrans for some prediction types. This is expected: we trained the model for illustration on a different dataset (py150) than the one for evaluation (internal), causing a significantly higher percentage of OOV predictions. In actual usage, we would retrain our models on the internal dataset. The results are shown in Table XI, focusing on PointerMixture vs. TravTrans.

Since we have already introduced a brief history of autocomplete in the paper, this section will focus on other explorations in the Transformer and code prediction space.

a) Code Prediction: In this paper, we viewed the context of prediction as code that appears strictly before the cursor [4]–[6], [11]–[14]. Among other flavors of code prediction are the ones where code after the prediction location, when available, is taken into account [9], [49], [51], [52], e.g. when completing a "hole" in a program, or correcting a misused local variable instance. Another dimension of work considers prediction at varying granularities of predictions, e.g. from characters [53] to subtokens [7]) to AST fragments (e.g. sub-ASTs [52]). In the context of autocomplete, we believe that subtokens or BPE are indeed a promising future direction, as we mentioned in Sec IX.

There have also been work discussing the practical implications of applying a code prediction tool into production. [54], [55] state that synthetic benchmarks are not representative of real-world data, and accuracy of the models drops when evaluated on real-world data. [56] discusses approaches to make models more lightweight to allow for faster computations and less memory usage in an IDE. A user evaluation of an autocomplete tool with stronger ML models is outside the scope of this paper. We would also like to point out that advantages in idealistic settings often transfer to advantages in more practical settings, as confirmed by the results reported in the above two works (Table II and III in [54], and Table 2 in [56]).

b) Transformers: Other than next token prediction, Transformers have been used recently for code summarization [24]. Furthermore, there has been a surge of interest since 2019 in extending Transformer models to handle beyond sequential structures for NLP [57]–[59]. It has been shown that taking tree structure into account helped code correction [25] and code translation [26].

There is practical interest, outside of academic literature, in the topic of code prediction using Transformers. Galois [60] is an open source project that uses GPT-2 [20] for code prediction. TabNine™ published a blog post [61] in July 2019 mentioning the use of GPT-2 in their code prediction but revealed no technical detail. Recently a team from Microsoft also published their ongoing efforts [62] on applying Transformers for code autocomplete. We believe we are among the first to systematically evaluate Transformers for code prediction and compare them to previous models.

c) Deep learning techniques over Code, beyond Code Prediction: There are many other uses of deep learning techniques for code, beyond code prediction. These include techniques for code summarization [21], bug finding [49], repair [63] and many others. An interesting aspect of this body of work is in the different ways in which they represent a program as an input to a neural architecture. These representations have ranged from linear token sequence (as for code prediction [5], [7], [32]), to paths in an AST [15], [21], [52], and sometimes even ways to convey static analysis information to the neural network [24], [49], [51], [64].

IX. Conclusion and Future Work

In this paper, we presented ways to using the Transformer for code prediction. We showed that the Transformer outperforms existing models for code prediction, and when supplied with code's structural information, we are able to get even better predictive power. Attribution study show that our best model tends to focus on relevant code locations for prediction.

In the future, one avenue we wish to continue working on is handling out-of-vocabulary words better. Source code presents a difficulty shared with NLP in handling large vocabularies and rare words. The token/word to be predicted in test data may not appear in the training data. This is even more challenging when predicting identifiers, such as method names, variable names, as developers can come up with arbitrary identifier names. Possible mitigation includes copying mechanism [48], [51], [65] and open-vocabulary models [7], [66].

This paper focused on predicting the next token, as it is already a challenging task. In future, we also want to explore predicting multiple tokens at a time, i.e. autocompleting entire expressions.

References

[1] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[2] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[3] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 532–542. [Online]. Available: https://doi.org/10.1145/2491411.2491458

[4] C. Liu, X. Wang, R. Shin, J. E. Gonzalez, and D. Song, "Neural code completion," 2016. [Online]. Available: https://openreview.net/forum?id=rJbPBt9lg

[5] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, ser. IJCAI'18. AAAI Press, 2018, p. 4159–25.

[6] F. Liu, L. Zhang, and Z. Jin, "Modeling programs hierarchically with stack-augmented LSTM," *Journal of Systems and Software*, p. 110547, 2020. [Online]. Available: https://doi.org/10.1016/j.jss.2020.110547

[7] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code != big vocabulary: Open-vocabulary models for source code," in *International Conference on Software Engineering (ICSE)*, 2020.

[8] A. Svyatkovskiy, Y. Zhao, S. Fu, and N. Sundaresan, "Pythia: AI-assisted code completion system," *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Jul 2019. [Online]. Available: http://dx.doi.org/10.1145/3292500.3330699

[9] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 419–428. [Online]. Available: https://doi.org/10.1145/2594291.2594321

[10] G. A. Aye and G. E. Kaiser, "Sequence model design for code completion in the modern IDE," *arXiv preprint arXiv:2004.05249*, 2020.

[11] M. Allamanis and C. Sutton, "Mining idioms from source code," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 472–483.

[12] P. Bielik, V. Raychev, and M. Vechev, "PHOG: probabilistic model for code," in *International Conference on Machine Learning*, 2016, pp. 2933–2942.

[13] V. Raychev, P. Bielik, M. Vechev, and A. Krause, "Learning programs from noisy data," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 761–774. [Online]. Available: https://doi.org/10.1145/2837614.2837671

[14] V. Raychev, P. Bielik, and M. 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*, ser. OOPSLA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 731–747. [Online]. Available: https://doi.org/10.1145/2983990.2984041

[15] U. Alon, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX

[16] "150k python dataset," 2016. [Online]. Available: https://eth-sri.github.io/py150

[17] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 2073–2083. [Online]. Available: https://www.aclweb.org/anthology/P16-1195

[18] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[19] L. Dong, N. Yang, W. Wang, F. Wei, X. Liu, Y. Wang, J. Gao, M. Zhou, and H.-W. Hon, "Unified language model pre-training for natural language understanding and generation," in *Advances in Neural Information Processing Systems*, 2019, pp. 13 042–13 054. [Online]. Available: https://papers.nips.cc/paper/9464-unified-language-model-pre-training-for-natural-language-understanding-and-generation

[20] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," in *OpenAI Blog*, 2019. [Online]. Available: https://openai.com/blog/better-language-models/

[21] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019. [Online]. Available: https://doi.org/10.1145/3290353

[22] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 763–773.

[23] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," *arXiv preprint arXiv:1312.6034*, 2013.

[24] V. J. Hellendoorn, C. Sutton, R. Singh, and P. Maniatis, "Global relational models of source code," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=B1lnbRNtwr

[25] J. Harer, C. Reale, and P. Chin, "Tree-Transformer: A transformer-based method for correction of tree-structured data," *arXiv preprint arXiv:1908.00449*, 2019.

[26] V. Shiv and C. Quirk, "Novel positional encodings to enable tree-based transformers," in *Advances in Neural Information Processing Systems*, 2019, pp. 12 058–12 068. [Online]. Available: https://papers.nips.cc/paper/9376-novel-positional-encodings-to-enable-tree-based-transformers

[27] "Pretrained probabilistic models for code," 2017. [Online]. Available: https://github.com/eth-sri/ModelsPHOG

[28] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[29] V. J. Hellendoorn, C. Sutton, R. Singh, P. Maniatis, and D. Bieber, "Global relational models of source code," in *International conference on learning representations*, 2019. [Online]. Available: https://openreview.net/forum?id=B1lnbRNtwr

[30] R. Al-Rfou, D. Choe, N. Constant, M. Guo, and L. Jones, "Character-level language modeling with deeper self-attention," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 3159–3166.

[31] K. Irie, A. Zeyer, R. Schlüter, and H. Ney, "Language modeling with deep transformers," *Interspeech 2019*, Sep 2019. [Online]. Available: http://dx.doi.org/10.21437/Interspeech.2019-2225

[32] V. J. Hellendoorn and P. T. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[33] N. Akhtar and A. Mian, "Threat of adversarial attacks on deep learning in computer vision: A survey," *IEEE Access*, vol. 6, pp. 14 410–14 430, 2018.

[34] A. Chakraborty, M. Alam, V. Dey, A. Chattopadhyay, and D. Mukhopadhyay, "Adversarial attacks and defences: A survey," *arXiv preprint arXiv:1810.00069*, 2018.

[35] W. E. Zhang, Q. Z. Sheng, A. Alhazmi, and C. Li, "Adversarial attacks on deep-learning models in natural language processing: A survey," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 11, no. 3, pp. 1–41, 2020.

[36] S. Chakraborty, R. Tomsett, R. Raghavendra, D. Harborne, M. Alzantot, F. Cerutti, M. Srivastava, A. Preece, S. Julier, R. M. Rao *et al.*, "Interpretability of deep learning models: a survey of results," in *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. IEEE, 2017, pp. 1–6.

[37] Q.-S. Zhang and S.-C. Zhu, "Visual interpretability for deep learning: a survey," *Frontiers of Information Technology & Electronic Engineering*, vol. 19, no. 1, pp. 27–39, 2018.

[38] D. V. Carvalho, E. M. Pereira, and J. S. Cardoso, "Machine learning interpretability: A survey on methods and metrics," *Electronics*, vol. 8, no. 8, p. 832, 2019.

[39] X. Huang, D. Kroening, W. Ruan, J. Sharp, Y. Sun, E. Thamo, M. Wu, and X. Yi, "A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability," *Computer Science Review*, vol. 37, p. 100270, 2020.

[40] K. Wang and M. Christodorescu, "Coset: A benchmark for evaluating neural program embeddings," *arXiv preprint arXiv:1905.11445*, 2019.

[41] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, Nov. 2020. [Online]. Available: https://doi.org/10.1145/3428230

[42] G. Ramakrishnan, J. Henkel, Z. Wang, A. Albarghouthi, S. Jha, and T. Reps, "Semantic robustness of models of source code," *arXiv preprint arXiv:2002.03043*, 2020.

[43] J. Jiarpakdee, C. Tantithamthavorn, H. K. Dam, and J. Grundy, "An empirical study of model-agnostic techniques for defect prediction models," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[44] S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto, "Predicting defective lines using a model-agnostic technique," *IEEE Transactions on Software Engineering*, no. 01, pp. 1–1, sep 5555.

[45] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should i trust you?": Explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1135–1144. [Online]. Available: https://doi.org/10.1145/2939672.2939778

[46] D. Smilkov, N. Thorat, B. Kim, F. Viégas, and M. Wattenberg, "Smoothgrad: removing noise by adding noise," *arXiv preprint arXiv:1706.03825*, 2017.

[47] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17. JMLR.org, 2017, p. 3319–3328.

[48] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 2091–2100. [Online]. Available: http://proceedings.mlr.press/v48/allamanis16.html

[49] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=BJOFETxR-

[50] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," *arXiv preprint arXiv:1506.03134*, 2015.

[51] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O. Polozov, "Generative code modeling with graphs," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=Bke4KsA5FX

[52] U. Alon, R. Sadaka, O. Levy, and E. Yahav, "Structural language models of code," in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, H. D. III and A. Singh, Eds., vol. 119. PMLR, 13–18 Jul 2020, pp. 245–256. [Online]. Available: http://proceedings.mlr.press/v119/alon20a.html

[53] P. Bielik, V. Raychev, and M. Vechev, "Program synthesis for character level language modeling," in *International Conference on Learning Representations*, 2016. [Online]. Available: https://openreview.net/forum?id=ry_sjFqgx

[54] V. J. Hellendoorn, S. Proksch, H. C. Gall, and A. Bacchelli, "When code completion fails: A case study on real-world completions," in *Proceedings of the 41st International Conference on Software Engineering*, 2019, p. 960–970.

[55] G. A. Aye, S. Kim, and H. Li, "Learning autocompletion from real-world datasets," in *Proceedings of the ACM/IEEE 43rd International Conference on Software Engineering: Software Engineering in Practice*. [Online]. Available: https://arxiv.org/abs/2011.04542

[56] A. Svyatkovskiy, S. Lee, A. Hadjitofi, M. Riechert, J. Franco, and M. Allamanis, "Fast and memory-efficient neural code completion," 2020.

[57] Y. Wang, H.-Y. Lee, and Y.-N. Chen, "Tree transformer: Integrating tree structures into self-attention," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019, pp. 1060–1070. [Online]. Available: https://www.aclweb.org/anthology/D19-1098/

[58] M. Ahmed, M. R. Samee, and R. E. Mercer, "You only need attention to traverse trees," in *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019, pp. 316–322. [Online]. Available: https://www.aclweb.org/anthology/P19-1030/

[59] X.-P. Nguyen, S. Joty, S. Hoi, and R. Socher, "Tree-structured attention with hierarchical accumulation," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=HJxK5pEYvr

[60] "Galois autocompleter," https://github.com/iedmrc/galois-autocompleter.

[61] TabNine, "Autocompletion with deep learning," *TabNine Blog*, Jul 2019. [Online]. Available: https://tabnine.com/blog/deep

[62] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, *IntelliCode Compose: Code Generation Using Transformer*. New York, NY, USA: Association for Computing Machinery, 2020, p. 1433–1443.

[63] M. Vasic, A. Kanade, P. Maniatis, D. Bieber, and R. Singh, "Neural program repair by jointly learning to localize and repair," *arXiv preprint arXiv:1904.01720*, 2019.

[64] Y. Yang and C. Xiang, "Improve language modelling for code completion through learning general token repetition of source code," in *31st International Conference Software Engineering and Knowledge Engineering*, 2019, pp. 667–777.

[65] P. Fernandes, M. Allamanis, and M. Brockschmidt, "Structured neural summarization," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=H1ersoRqtm

[66] M. Cvitkovic, B. Singh, and A. Anandkumar, "Open vocabulary learning on source code with a graph-structured cache," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. Long Beach, California, USA: PMLR, 09–15 Jun 2019, pp. 1475–1485. [Online]. Available: http://proceedings.mlr.press/v97/cvitkovic19b.html