CODE2SEQ: GENERATING SEQUENCES FROM STRUCTURED REPRESENTATIONS OF CODE

Anonymous authors
Paper under double-blind review

ABSTRACT

The ability to generate natural language sequences from source code snippets has a variety of applications such as code summarization, documentation, and retrieval. Sequence-to-sequence (seq2seq) models, adopted from neural machine translation (NMT), have achieved state-of-the-art performance on these tasks by treating source code as a sequence of tokens. We present CODE2SEQ: an alternative approach that leverages the syntactic structure of programming languages to better encode source code. Our model represents a code snippet as the set of compositional paths in its abstract syntax tree (AST) and uses attention to select the relevant paths while decoding. We demonstrate the effectiveness of our approach for two tasks, two programming languages, and four datasets of up to 16M examples. Our model significantly outperforms previous models that were specifically designed for programming languages, as well as state-of-the-art NMT models.

1 INTRODUCTION

Modeling the relation between source code and natural language can be used for automatic code summarization (Allamanis et al., 2016), documentation (Iyer et al., 2016), retrieval (Allamanis et al., 2015b), and even generation (Balog et al., 2016; Rabinovich et al., 2017; Yin and Neubig, 2017; Devlin et al., 2017; Murali et al., 2017). In this work, we consider the general problem of generating a natural language sequence from a given snippet of source code.

A direct approach is to frame the problem as a machine translation problem, where the source sentence is the sequence of tokens in the code and the target sentence is a corresponding natural language sequence. This approach allows one to apply state-of-the-art neural machine translation (NMT) models from the sequence-to-sequence (seq2seq) paradigm (Sutskever et al., 2014; Cho et al., 2014; Bahdanau et al., 2014; Luong et al., 2015; Vaswani et al., 2017), yielding state-of-the-art performance on various code captioning and documentation benchmarks (Iyer et al., 2016; Allamanis et al., 2016; Loyola et al., 2017) despite having extremely long source sequences.

We present an alternative approach for encoding source code that leverages the syntactic structure of programming languages: CODE2SEQ. Specifically, we represent a given code snippet as a set of compositional paths over its abstract syntax tree (AST), where each path is compressed to a fixed-length vector using LSTMs (Hochreiter and Schmidhuber, 1997). During decoding, CODE2SEQ attends over a different weighted sum of the path-vectors to produce each output token, much like NMT models attend over token representations in the source sentence.

We show the effectiveness of our code2seq model on two tasks: (1) code summarization (Figure 1a), where we predict a Java method’s name given its body, and (2) code captioning (Figure 1b), where we predict a natural language sentence that describes a given C# snippet. On both tasks, our CODE2SEQ model outperforms models that were explicitly designed for code, such as the model of Allamanis et al. (2016) and CodeNN (Iyer et al., 2016), as well as state-of-the-art NMT models (Luong et al., 2015; Vaswani et al., 2017). To examine the importance of each component of the model, we conduct a thorough ablation study. In particular, we show the importance of structural encoding of code, by showing how our model yields a significant improvement over an ablation that uses only token-level information without syntactic paths. To the best of our knowledge, this is the first work to leverage the syntactic structure of code for end-to-end generation of sequences.
2 Representing Code as AST Paths

An Abstract Syntax Tree (AST) is a tree which uniquely represents a source code snippet in a given language and grammar. The leaves of the tree are called terminals, and usually refer to user-defined values which represent identifiers and names from the code. The non-leaf nodes are called nonterminals and represent a restricted set of structures in the language, e.g., loops, expressions, and variable declarations. For example, Figure 2 shows a partial AST for the code snippet of Figure 2. Names such as num and types such as int are represented as values of terminals; syntactic structures such as variable declaration (VarDec) and a do-while loop (DoStmt) are represented as nonterminals.

Given the AST of a code snippet, we consider all pairwise paths between terminals, and represent them as sequences of terminal and nonterminal nodes. We then use these paths with their terminals’ values to represent the code snippet itself. For example, consider the two Java methods of Figure 2. Both of these methods count occurrences of a character in a string. They have exactly the same functionality, although a different implementation, and therefore different surface forms. Encoding these snippets of code as sequences of tokens might overlook the recurring patterns that suggest the common method name. However, a structural observation reveals syntactic paths that are common to both methods, and differ only in a single node of a Do-while statement versus a For statement. This example shows the effectiveness of a syntactic encoding of code. Such an encoder can generalize much better to unseen examples because the AST normalizes a lot of the surface form variance. Since our encoding is compositional, the encoder can generalize even if the paths are not identical (e.g., a For node in one path and a While in the other).

Since a code snippet can contain an arbitrary number of such paths, we sample k paths as the representation of the code snippet. To avoid bias, k new paths are sampled afresh in every training iteration. In Section 5 we show that this runtime-sampling provides regularization and improves results compared to sampling the same k paths for each example in advance.

Formally, we use $C$ to denote a given snippet of code. Every training iteration $k$ pairs of terminals are uniformly sampled from within the AST of $C$. Each pair of terminals $(v^i_1, v^i_l)$ implies a single path between them: $v^i_1 v^i_2 ... v^i_{l_i}$. Finally, the input code example is represented as a set of these $k$ random AST paths: $\{ (v^1_1 v^1_2 ... v^1_{l_1}), ..., (v^k_1 v^k_2 ... v^k_{l_k}) \}$, where $l_j$ is the length of the $j$th path.

3 Model Architecture

Our model follows the standard encoder-decoder architecture for NMT (Section 3.1), with the significant difference that the encoder does not read the input as a flat sequence of tokens. Instead, the encoder creates a vector representation for each AST path separately (Section 3.2). The decoder
```java
int countOccurrences(String str, char ch) {
    int num = 0;
    int index = -1;
    do {
        index = str.indexOf(ch, index + 1);
        if (index >= 0) {
            num++;
        }
    } while (index >= 0);
    return num;
}

(a)
```

```java
int countOccurrences(String source, char value) {
    int count = 0;
    for (int i = 0; i < source.length(); i++) {
        if (source.charAt(i) == value) {
            count++;
        }
    }
    return count;
}

(b)
```

Figure 2: An example of two Java methods that have exactly the same functionality. Although having a different sequential (token-based) representation, considering syntactic patterns reveals recurring paths, which might differ only in a single node (a ForStmt node instead of a Do-while node).

then attends over the encoded AST paths (rather than the encoded tokens) while generating the target sequence. An illustration of our model is shown in Figure 3.

3.1 ENCODER-DECODER FRAMEWORK

Contemporary NMT models are largely based on an encoder-decoder architecture (Cho et al., 2014; Sutskever et al., 2014; Luong et al., 2015; Bahdanau et al., 2014), where the encoder maps an input sequence of tokens $x = (x_1, ..., x_n)$ to a sequence of continuous representations $z = (z_1, ..., z_n)$. Given $z$, the decoder then generates a sequence of output tokens $y = (y_1, ..., y_m)$ one token at a time, hence modeling the conditional probability: $p(y_1, ..., y_m|x_1, ..., x_n)$.

At each decoding step, the probability of the next target token depends on the previously generated token, and can therefore be factorized as:

$$p(y_1, ..., y_m|x_1, ..., x_n) = \prod_{j=1}^{m} p(y_j|y_{<j}, z_1, ..., z_n)$$

(1)

In attention-based models, at each time step $t$ in the decoding phase, a context vector $c_t$ is computed by attending over the elements in $z$ using the decoding state $h_t$, typically computed by an LSTM.

$$\alpha_t^j = softmax(h_t W_a z) \quad c_t = \sum_{i} \alpha_t^i z_i$$

(2)

The context vector $c_t$ and the decoding state $h_t$ are then combined to predict the current target token $y_t$. Previous work differs in the way the context vector is computed and in the way it is combined with the current decoding state. A standard approach (Luong et al., 2015) is to pass $c_t$ and $h_t$ through a multi-layer perceptron (MLP) and then predict the probability of the next token using softmax:

$$p(y_t|y_{<t}, z_1, ..., z_n) = softmax(W_s tanh(W_c [c_t; h_t]))$$

(3)
3.2 AST ENCODER

Given a set of AST paths \((x_1, ..., x_k)\), our goal is to create a vector representation \(z_i\) for each path \(x_i = v_1^{i} \ldots v_l^{i}\). We represent each path separately using a bi-directional LSTM to encode the path, and sub-token embeddings to capture the compositional nature of the terminals’ values (the tokens).

Path Representation Each AST path is composed of nodes and their child indices from a limited vocabulary of up to 364 symbols. We represent each node using a learned embedding matrix \(E_{\text{nodes}}\) and then encode the entire sequence using the final states of a bi-directional LSTM:

\[
h_1, ..., h_l = \text{LSTM}(E_{\text{nodes}} v_1, ..., E_{\text{nodes}} v_l) \tag{4}
\]

\[
\text{encode}\_\text{path}(v_1 \ldots v_l) = [h_1^a; h_1^r] \tag{5}
\]

Token Representation The first and last node of an AST path are terminals whose values are tokens in the code. Following [Allamanis et al. (2015a, 2016)], we split code tokens into subtokens; for example, a token with the value Array\_List, will be decomposed into Array and List. This is somewhat analogous to byte-pair encoding in NMT [Sennrich et al., 2016], although in the case of programming languages, coding conventions such as camel notation provide us with an explicit partition of each token. Specifically, we use a learned embedding matrix \(E_{\text{subtokens}}\) to represent each subtoken, and then sum the subtoken vectors to represent the full token:

\[
\text{encode}\_\text{token}(w) = \sum_{s \in \text{split}(w)} E_{s\text{subtokens}} \tag{6}
\]

The LSTM decoder may also predict subtokens at each step (e.g. when generating method names), although the decoder’s subtoken embedding matrix will be different.

Combined Representation To represent the entire path \(x = v_1 \ldots v_l\), we concatenate the path’s representation with each of the token representation of each terminal node, and apply a fully-connected layer:

\[
z = \tanh \left( W_{\text{in}} [\text{encode}\_\text{path}(v_1 \ldots v_l); \text{encode}\_\text{token}(\text{value}(v_1)); \text{encode}\_\text{token}(\text{value}(v_l))]) \right) \tag{7}
\]

where value is the mapping of a terminal node to its associated value, and \(W_{\text{in}}\) is a \((2d_{\text{path}} + 2d_{\text{token}}) \times d_{\text{hidden}}\) matrix.

Decoder Start State To provide the decoder with an initial state, we average the combined representations of all the paths:

\[
h_0 = \frac{1}{k} \sum_{i=1}^{k} z_i \tag{8}
\]

Unlike typical encoder-decoder models, the order of the input random paths is not taken into account. Each path is encoded separately and the combined representations are aggregated with mean pooling to initialize the decoder’s state. This represents the given source code as a set of random paths.
Attention  Finally, the decoder generates the output sequence while attending over the combined representations \( z_i, ..., z_k \), similarly to the way that seq2seq models attend over the source symbols.

4 Experiments

We evaluate our model on two code-to-sequence tasks: summarization (Section 4.1), in which we predict Java methods’ names from their bodies, and captioning (Section 4.2), where we generate natural language descriptions of C# code snippets. We thus demonstrate that our approach can produce both method names and natural language outputs, and can encode a code snippet in any language for which an AST can be constructed (i.e., a parser exists).

Setup The values of all of the parameters are initialized using the initialization heuristic of Glorot and Bengio (2010). We optimize the cross-entropy loss (Rubinstein, 1999, 2001) with a Nesterov momentum (Nesterov, 1983) of 0.95 and an initial learning rate of 0.01, decayed by a factor of 0.95 every epoch. We apply dropout (Srivastava et al., 2014) of 0.25 on the input vectors \( x_j \), and a recurrent dropout of 0.5 on the LSTM that encodes the AST paths. We used \( d_{tokens} = d_{nodes} = d_{hidden} = d_{target} = 128 \). Each LSTM that encodes the AST paths had 128 cells and the decoder LSTM had 320 cells. We used \( k = 200 \) as the number of random paths on each example.

4.1 Code Summarization

In this task, we predict a Java method’s name given its body. As was previously observed (Allamanis et al., 2016; Alon et al., 2018a), this is a good benchmark because a method name in open-source Java projects tends to be succinct and precise, and a method body is often a complete logical unit. We predict the target method name as a sequence of sub-tokens, e.g., `setMaxConnectionsPerServer` is predicted as the sequence “set max connections per server”. The target sequence length is about 3 on average. We adopt the measure used by Allamanis et al. (2016) and Alon et al. (2018a), who measured precision, recall, and F1 score over the target sequence, case insensitive.

Data We experiment with this task across three datasets:

Java-small – Contains 11 relatively large Java projects, which were originally used for 11 distinct models for training and predicting within the scope of the same project (Allamanis et al., 2016). We use the same data, but train and predict across projects: we took 9 projects for training, 1 project for validation and 1 project as our test set. This dataset contains about 700K examples.

Java-med – A new dataset of the 1000 top-starred Java projects from GitHub. We randomly select 800 projects for training, 100 for validation and 100 for testing. This dataset contains about 4M examples and we make it publicly available.

Java-large – A new dataset of the 9500 top-starred Java projects from GitHub that were created since January 2007. We randomly select 9000 projects for training, 200 for validation and 300 for testing. This dataset contains about 16M examples and we make it publicly available.

Baselines We re-trained all of the baselines on all of the datasets of this task using the original implementations of the authors. We compare CODE2SEQ to the following baselines: Allamanis et al. (2016) who used a convolutional attention network to predict method names, syntactic paths with Conditional Random Fields (CRFs) (Alon et al., 2018b), and code2vec (Alon et al., 2018a). In addition, we compared to three NMT baselines that read the input source code as a stream of tokens: a 2-layer bidirectional encoder-decoder LSTMs (split tokens and full tokens) with global attention (Luong et al., 2015), and the Transformer (Vaswani et al., 2017) which achieved state-of-the-art results for translation tasks.

Our model is incomparable to the model of Allamanis et al. (2018) because they targeted a different task of predicting variable names, and are unable to generate target sequences. We could not compare to the work of Liang and Zhu (2018) due to replicability issues.

We put significant effort to strengthen the NMT baselines in order to provide a fair comparison: (1) we split tokens to subtokens, as in our model (e.g., `HashSet → Hash Set`) - this was shown to
Table 1: Our model significantly outperforms previous PL-oriented and NMT models. Another visualization can be found in Appendix D.

<table>
<thead>
<tr>
<th>Model</th>
<th>Java-small</th>
<th>Java-med</th>
<th>Java-large</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Prec</td>
<td>Rec</td>
<td>F1</td>
</tr>
<tr>
<td>ConvAttention [Allamanis et al., 2016]</td>
<td>50.25</td>
<td>24.62</td>
<td>33.05</td>
</tr>
<tr>
<td>Paths+CRFs [Alon et al., 2018b]</td>
<td>8.39</td>
<td>5.63</td>
<td>6.74</td>
</tr>
<tr>
<td>code2vec [Alon et al., 2018a]</td>
<td>18.51</td>
<td>18.74</td>
<td>18.62</td>
</tr>
<tr>
<td>2-layer BiLSTM (no token splitting)</td>
<td>42.63</td>
<td>29.97</td>
<td>35.20</td>
</tr>
<tr>
<td>Transformer [Vaswani et al., 2017]</td>
<td>38.13</td>
<td>37.40</td>
<td>43.02</td>
</tr>
<tr>
<td>code2seq</td>
<td>50.64</td>
<td>37.40</td>
<td>43.02</td>
</tr>
</tbody>
</table>

Absolute Gain over BiLSTM       +8.01       +7.43     +7.82      +6.09      +5.32     +5.71      +0.50       +6.25      +4.01

improve the results by about 10 F1 points (Table 1), (2) we deliberately kept the original casing of the source tokens since we found it to improve their results, and (3) during inference, we replaced generated UNK tokens with the source tokens that were given the highest attention. For the 2-layer BiLSTM we used embeddings of size 512, each of the encoder and decoder had 512 cells, and the default hyperparameters of OpenNMT [Klein et al., 2017]. For the Transformer, we used their original hyperparameters [Vaswani et al., 2017]. This resulted in a Transformer model with 169M parameters, a BiLSTM model with 134M parameters, while our code2seq model had only 37M.

Performance Table 1 shows the results for the code summarization task. Our model significantly outperforms the baselines in both precision and recall across all the three datasets, demonstrating that there is added value in leveraging ASTs to encode source code. Our model improves over the best baselines, BiLSTM with split tokens, by between 4 to 8 F1 points on all benchmarks. The BiLSTM with split tokens consistently achieved about 10 F1 score more than BiLSTM with full tokens, and for this reason we included only a split token Transformer baseline. Our model outperforms ConvAttention [Allamanis et al., 2016], which was designed specifically for this task, and outperforms Paths+CRFs [Alon et al., 2018b] which used syntactic features. Examples for predictions made by our model and each of the baselines can be found in Appendix B.

Data Efficiency ConvAttention [Allamanis et al., 2016] performed even better than the Transformer on the Java-small dataset, but could not scale and leverage the larger datasets. Paths+CRFs showed very low results on the Java-small dataset, which is expected due to the sparse nature of their paths and the CRF model. When comparing our model with the best among the baselines (BiLSTM with split tokens), we see that our model achieves a relative improvement of 7.3% on Java-large, but as the dataset goes smaller – the larger the relative difference becomes: 13% on Java-med and 22% on Java-small; when comparing to the Transformer: a relative improvement of 23% on Java-large and 37% on Java-small. These results show the data efficiency of our architecture: while the data-hungry NMT baselines require large datasets, our model can leverage both small and large datasets.

4.2 Code Captioning

For this task we consider predicting a full natural language sentence given a short C# code snippet. We used the dataset of CodeNN [Iyer et al., 2016], which consists of 66,015 pairs of questions and answers from StackOverflow. They used a semi-supervised classifier to filter irrelevant examples and asked human annotators to provide two additional titles for the examples in the test set, making a total of three reference titles for each code snippet. The target sequence length in this task is about 10 on average. This dataset is especially challenging as it is orders of magnitude smaller than the code summarization datasets. Additionally, StackOverflow code snippets are typically short, incomplete at times, and aim to provide an answer to a very specific question. We evaluated using BLEU score with smoothing, using the same evaluation scripts as Iyer et al. [2016].

Baselines We present results compared to CodeNN, 2-layer bidirectional LSTMs with attention, and the Transformer. As before, we provide a fair comparison by splitting tokens to subtokens, and replacing UNK during inference. We also include numbers from baselines used by Iyer et al. [2016].

We also trained versions of the NMT baselines in which we down-matched the sizes and number of parameters to our model. These baselines seemed to benefit from more parameters, so the results reported here are for the versions that had many more parameters than our model.
Table 2: Our model outperforms previous work in the code captioning task. \(^1\)Results previously reported by Iyer et al. (2016), and verified by us. Another visualization can be found in Appendix C.

<table>
<thead>
<tr>
<th>Model</th>
<th>BLEU</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOSES(^1) (Koehn et al., 2007)</td>
<td>11.57</td>
</tr>
<tr>
<td>IR(^1)</td>
<td>13.66</td>
</tr>
<tr>
<td>SUM-NN(^1) (Rush et al., 2015)</td>
<td>19.31</td>
</tr>
<tr>
<td>2-layer BiLSTM</td>
<td>19.78</td>
</tr>
<tr>
<td>Transformer (Vaswani et al., 2017)</td>
<td>19.68</td>
</tr>
<tr>
<td>CodeNN(^1) (Iyer et al., 2016)</td>
<td>20.53</td>
</tr>
<tr>
<td>code2seq</td>
<td>23.04</td>
</tr>
</tbody>
</table>

Results: Table 2 summarizes the results for the code captioning task. Our model achieves a BLEU score of 23.04, which improves by 2.51 points (12.2% relative) over CodeNN, who introduced this dataset, and over all the other baselines including BiLSTMs and the Transformer, which achieved slightly lower results than CodeNN. Examples for predictions made by our model and each of the baselines can be found in Appendix E. These results show that when the training examples are short and incomplete code snippets, our model generalizes better to unseen examples than a shallow textual token-level approach, thanks to its syntactic observation of the data.

5 Ablation Study

To better understand the importance of different components of our model, we conducted an extensive ablation study. We vary our model in different ways and measure the change in performance. These experiments were performed for the code summarization task, on the validation set of the Java-med dataset. We examine several alternative designs:

1. No AST nodes – instead of encoding an AST path using an LSTM, take only the first and last terminal values for constructing an input vector
2. No decoder – no sequential decoding; instead, predict the target sequence as a single symbol using a single softmax layer.
3. No token splitting – no subtoken encoding; instead, embed the full token.
4. No tokens – use only the AST nodes without using the values associated with the terminals.
5. No attention – decode the target sequence given the initial decoder state, without attention.
6. No random – no re-sampling of \(k\) paths in each iteration; instead, sample in advance and use the same \(k\) paths for each example throughout the whole training process.

Table 3 shows the results of these alternatives. As seen, not encoding AST nodes resulted in a degradation especially in the precision: a decrease of 5.42 compared to 2.66 for the recall. Using a single prediction with no decoder reduces recall by more than a third. This shows that the method name prediction task should be addressed as a sequential prediction, despite the methods’ relatively short names. Using no token splitting or no tokens at all drastically hurt the results, showing the significance of encoding both subtokens and syntactic paths. Despite the low results of no tokens, it is still surprising that the model can achieve around half the score of the full model, as using no tokens is equivalent to reasoning about code which has no identifier names, types, APIs, and constant values and can be very difficult even for a human. The no attention experiment shows the contribution of attention in our model, which is very close in its relative value to the contribution of attention in seq2seq models (Luong et al., 2015; Bahdanau et al., 2014). The no random experiment shows the positive contribution of sampling \(k\) different paths at once on every training iteration, instead of using the same sample of paths from each example during whole training. This approach provides data-level regularization that gives additional improvement to a model that is very powerful already. Another visualization can be found in Appendix E.
Table 3: Variations on the code2seq model, performed on the dev set of Java-med.

<table>
<thead>
<tr>
<th>Model</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
<th>ΔF1</th>
</tr>
</thead>
<tbody>
<tr>
<td>code2seq (original model)</td>
<td>60.93</td>
<td>45.77</td>
<td>52.27</td>
<td></td>
</tr>
<tr>
<td>No AST nodes (only tokens)</td>
<td>55.51</td>
<td>43.11</td>
<td>48.53</td>
<td>-3.74</td>
</tr>
<tr>
<td>No decoder</td>
<td>47.99</td>
<td>28.96</td>
<td>36.12</td>
<td>-16.15</td>
</tr>
<tr>
<td>No token splitting</td>
<td>48.53</td>
<td>34.80</td>
<td>40.53</td>
<td>-11.74</td>
</tr>
<tr>
<td>No tokens (only AST nodes)</td>
<td>33.78</td>
<td>21.23</td>
<td>26.07</td>
<td>-26.20</td>
</tr>
<tr>
<td>No attention</td>
<td>57.00</td>
<td>41.89</td>
<td>48.29</td>
<td>-3.98</td>
</tr>
<tr>
<td>No random (sample k paths in advance)</td>
<td>59.08</td>
<td>44.07</td>
<td>50.49</td>
<td>-1.78</td>
</tr>
</tbody>
</table>

6 RELATED WORK

The growing availability of open source repositories creates new opportunities for using machine learning to process source code en masse. Several papers model code as a sequence of tokens (Iyer et al., 2016; Allamanis et al., 2016; Loyola et al., 2017), characters (Bielik et al., 2017), and API calls (Raychev et al., 2014). While sometimes obtaining satisfying results, these models treat code as a sequence rather than a tree. This forces these techniques to implicitly re-learn the (predefined) syntax of the programming language, wasting resources and reducing accuracy.

Code representation models that use syntactic information have usually been evaluated on relatively easier tasks, which mainly focus on “filling the blanks” in a given program (Alon et al., 2018b; Bielik et al., 2016; Raychev et al., 2016, 2015; Allamanis et al., 2018) or semantic classification of code snippets (Alon et al., 2018a). Moreover, none of the models that use syntactic relations are compositional, and therefore the number of possible syntactic relations is fixed either before or after training, and often consumes a lot of memory. The syntactic paths of Alon et al. (2018b,a) are represented monolithically, and are therefore limited to only a subset of the paths that were observed enough times during training. As a result, they cannot represent unseen relations. In contrast, by representing AST paths node-by-node using LSTMs, our model can represent and use any syntactic path in any unseen example. Further, our model decodes the output sequence step-by-step while attending over the input paths, and can thus generate unseen sequences, compared to code2vec (Alon et al., 2018a) which has a closed vocabulary.

Allamanis et al. (2018) represent code with Gated Graph Neural Networks. Nodes in the graph represent identifiers, and edges represent syntactic and semantic relations in the code such as “ComputedFrom” and “LastWrite”. The kinds of edges (features) are designed for the semantics of a specific programming language, for a specific task, and require an expert to think of and implement. In contrast, our model has minimal assumptions on the input language and is general enough not to require neither expert semantic knowledge nor the manual design of features. Our model can therefore be easily implemented for various input languages. Liang and Zhu (2018) presented a Tree-RNN model for learning code, but its evaluation contains many irregularities (for example, the seq2seq baselines were deprived of non-alphanumeric tokens, which are critical for capturing assignments, method calls, and other basic operations).

7 CONCLUSION

We presented a novel code-to-sequence model which considers the unique syntactic structure of source code with a sequential modeling of natural language. The core idea is to sample paths in the Abstract Syntax Tree of a code snippet, encode those paths with an LSTM and attend to them while generating the target sequence.

We demonstrate our approach by using it to predict method names across three datasets of varying sizes, predict natural language captions given partial and short code snippets, in two programming languages. Our model performs significantly better than previous programming-language-oriented works and state of the art NMT models applied in our settings.

We believe that the principles presented in this paper can serve as a basis for a wide range of tasks which involve source code and natural language, and can be extended to other kinds of generated outputs. To this end, we make all our code, datasets, and trained models publicly available.
REFERENCES


A  CODE CAPTIONING EXAMPLES

Figure 5 contains examples from our test set for the code captioning task in C#, along with the prediction of our model and each of the baselines.

Figure 4 shows a timestep-by-timestep example, with the symbol decoded at each timestep and the top-attended path at that step. The width of the path is proportional to the attention it was given by the model (because of space limitations we included only the top-attended path for each decoding step, but hundreds of paths are attended at each step).

B  CODE SUMMARIZATION

Figure 7 contains examples from our test set for the code summarization task in C#, along with the prediction of our model and each of the baselines. The presented predictions are made by models that were trained on the same Java-large dataset.

C  CODE CAPTIONING RESULTS

Figure 8 shows a bar chart of the BLEU score of our model and the baselines, in the code captioning task (predicting natural language descriptions for C# code snippets). The numbers are the same as in Table 2.

D  CODE SUMMARIZATION RESULTS

Figure 9 shows a bar chart of the F1 score of our model and the baselines, in the code summarization task (predicting method names in Java). The numbers are the F1 columns from Table 1.

E  ABLATION STUDY RESULTS

Figure 10 shows a bar chart of the relative decrease in precision and recall for each of the ablations described in Section 5 and presented in Table 3.

Sensitivity to $k$: we experimented with different values of $k$, the number of sampled paths from each example (which we set to 200 in the final model). Lower values than $k = 100$ showed worse results. Decreasing to $k = 100$ showed a minor degradation, and increasing to $k = 300$ did not seem to consistently improve.
Figure 4: Example of code captioning for a C# code snippet from our test set. The text boxes at the bottom of each figure are the predictions produced by our model at each decoding step. The highlighted paths in each figure are the top-attended paths in each decoding step, and their widths are proportional to their attention weight (because of space limitations we included only the top-attended path for each decoding step, but hundreds of paths are attended at each step).
```csharp
TreeView myTreeView = new TreeView();
myTreeView.Nodes.Clear();
foreach (string parentText in xml.parent)
{
    TreeNode parent = new TreeNode();
    parent.Text = parentText;
    myTreeView.Nodes.Add(treeNodeDivisions);
    foreach (string childText in xml.child)
    {
        TreeNode child = new TreeNode();
        child.Text = childText;
        parent.Nodes.Add(child);
    }
}
```

<table>
<thead>
<tr>
<th>Model</th>
<th>Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MOSES</strong>†</td>
<td>How can TreeView TreeView a TreeView nodes from XML parentText string to a</td>
</tr>
<tr>
<td></td>
<td>treeview node from a TreeView parentText of a tree treeNodeDivisions from to</td>
</tr>
<tr>
<td></td>
<td>childText XML node of MDI childText created in a tree nodes in</td>
</tr>
<tr>
<td><strong>IR</strong>†</td>
<td>How to set the name of a tabPage programmatically</td>
</tr>
<tr>
<td><strong>SUM-NN</strong>†</td>
<td>how to get data from xml file in c#</td>
</tr>
<tr>
<td>2-layer BiLSTM (split tokens)</td>
<td>how to add child nodes to treeview</td>
</tr>
<tr>
<td>Transformer (split tokens)</td>
<td>how to add child node in treeview in c #</td>
</tr>
<tr>
<td><strong>CodeNN</strong>†</td>
<td>How to get all child nodes in TreeView ?</td>
</tr>
<tr>
<td>code2seq (this work)</td>
<td>add a child node to a treeview in c #</td>
</tr>
</tbody>
</table>

```csharp
var excel = new ExcelQueryFactory("excelFileName");
var firstRow = excel.Worksheet().First();
var companyName = firstRow["CompanyName"];  
```

<table>
<thead>
<tr>
<th>Model</th>
<th>Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MOSES</strong>†</td>
<td>How into string based on an firstRow a companyName firstRow ? How to</td>
</tr>
<tr>
<td><strong>IR</strong>†</td>
<td>Facebook C# SDK Get Current User</td>
</tr>
<tr>
<td><strong>SUM-NN</strong>†</td>
<td>how can i get the value of a string?</td>
</tr>
<tr>
<td>2-layer BiLSTM (split tokens)</td>
<td>how to get the value of a cell in excel using c #</td>
</tr>
<tr>
<td>Transformer (split tokens)</td>
<td>getting the first row in excel</td>
</tr>
<tr>
<td><strong>CodeNN</strong>†</td>
<td>how do I get the value of an xml file in c # ?</td>
</tr>
<tr>
<td>code2seq (this work)</td>
<td>get the value of a column in excel using c #</td>
</tr>
</tbody>
</table>
static void Main(string[] args)
{
    // Create an instance of Bytescout.PDFRenderer.
    // RasterRenderer object and register it.
    RasterRenderer renderer = new RasterRenderer();
    renderer.RegistrationName = "demo";
    renderer.RegistrationKey = "demo";
    // Load PDF document.
    renderer.LoadDocumentFromFile("multipage.pdf");
    for (int i = 0; i < renderer.GetPageCount(); i++)
    {
        // Render first page of the document to BMP image file
        renderer.RenderPageToFile(i, RasterOutputFormat.BMP, "image" + i + ".bmp");
    }
    // Open the first output file in default image viewer.
    System.Diagnostics.Process.Start("image0.bmp");
}

/*
 * Figure 5: C# examples from our test set for the code captioning task, along with the prediction of our model and each of the baselines.
*/
```java
void ______(Counter childCounter, String request, String requestId, long duration, boolean systemError, int responseSize) {
    // si je suis le counter fils du counter du contexte parent
    // comme sql pour http alors on ajoute la requête fille
    if (parentContext != null && parentCounter.getName().equals(parentContext.getParentCounter().getChildCounterName())) {
        childHits++;
        childDurationsSum += (int) duration;
    }

    // pour drill-down on conserve pour chaque requête mère, les requêtes filles appelées et le nombre d'exécutions pour chacune
    if (parentContext == null) {
        addChildRequestForDrillDown(requestId);
    } else {
        parentContext.addChildRequestForDrillDown(requestId);
    }
}
```

### Table: Model Prediction

<table>
<thead>
<tr>
<th>Model</th>
<th>Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConvAttention (Allamanis et al., 2016)</td>
<td>add</td>
</tr>
<tr>
<td>Paths+CRFs (Alon et al., 2018b)</td>
<td>call</td>
</tr>
<tr>
<td>code2vec (Alon et al., 2018a)</td>
<td>log response</td>
</tr>
<tr>
<td>2-layer BiLSTM (full tokens)</td>
<td>handle request</td>
</tr>
<tr>
<td>2-layer BiLSTM (split tokens)</td>
<td>report child request</td>
</tr>
<tr>
<td>Transformer (split tokens)</td>
<td>add child</td>
</tr>
<tr>
<td>Gold:</td>
<td>add child request</td>
</tr>
<tr>
<td>code2seq (this work)</td>
<td>add child request</td>
</tr>
</tbody>
</table>

```java
public static int ______(int value) {
    return value <= 0 ? 1 :
            value >= 0x40000000 ? 0x40000000 :
            1 << (32 - Integer.numberOfLeadingZeros(value - 1));
}
```

### Table: Model Prediction

<table>
<thead>
<tr>
<th>Model</th>
<th>Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConvAttention (Allamanis et al., 2016)</td>
<td>get</td>
</tr>
<tr>
<td>Paths+CRFs (Alon et al., 2018b)</td>
<td>test bit inolz</td>
</tr>
<tr>
<td>code2vec (Alon et al., 2018a)</td>
<td>multiply</td>
</tr>
<tr>
<td>2-layer BiLSTM (full tokens)</td>
<td>next power of two</td>
</tr>
<tr>
<td>2-layer BiLSTM (split tokens)</td>
<td>(replaced UNK)</td>
</tr>
<tr>
<td>Transformer (split tokens)</td>
<td>get bit length</td>
</tr>
<tr>
<td>Gold:</td>
<td>find next positive power of two</td>
</tr>
<tr>
<td>code2seq (this work)</td>
<td>get power of two</td>
</tr>
</tbody>
</table>
BigInteger \( \text{BigInteger} \_\_\_\_\_\_\_\~(\text{int} \text{ bitlength, BigInteger e, BigInteger sqrdBound}) \)
{
for \( \text{int} \ i = 0; i != 5 \times \text{bitlength}; i++ \)
{
BigInteger \( p = \text{new BigInteger(bitlength, 1, param.getRandom());} \)
if \( (p.\text{mod(e)}).\text{equals(ONE))} \)
{ 
 continue;
}
if \( (p.\text{multiply(p)\.\text{compareTo(sqrdBound) < 0}} \)
{ 
 continue;
}
if \( (!\text{isProbablePrime(p)})} \)
{ 
 continue;
}
if \( (!e.\text{gcd(p.subtract(ONE)).equals(ONE)})\}
{ 
 continue;
}
return \( p \);
}
throw new \text{IllegalStateException("unable to generate prime number.....for RSA key")};
}

<table>
<thead>
<tr>
<th>Model</th>
<th>Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConvAttention (Allamanis et al., 2016)</td>
<td>test</td>
</tr>
<tr>
<td>Paths+CRFs (Alon et al., 2018b)</td>
<td>i</td>
</tr>
<tr>
<td>code2vec (Alon et al., 2018a)</td>
<td>to big integer</td>
</tr>
<tr>
<td>2-layer BiLSTM (full tokens)</td>
<td>generate prime</td>
</tr>
<tr>
<td>2-layer BiLSTM (split tokens)</td>
<td>generate prime number</td>
</tr>
<tr>
<td>Transformer (split tokens)</td>
<td>generate</td>
</tr>
<tr>
<td>Gold:</td>
<td>choose random prime</td>
</tr>
<tr>
<td>code2seq (this work)</td>
<td>generate prime number</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model</th>
<th>Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConvAttention (Allamanis et al., 2016)</td>
<td>is</td>
</tr>
<tr>
<td>Paths+CRFs (Alon et al., 2018b)</td>
<td>equals</td>
</tr>
<tr>
<td>code2vec (Alon et al., 2018a)</td>
<td>contains ignore case</td>
</tr>
<tr>
<td>2-layer BiLSTM (full tokens)</td>
<td>contains ignore case</td>
</tr>
<tr>
<td>2-layer BiLSTM (split tokens)</td>
<td>contains</td>
</tr>
<tr>
<td>Transformer (split tokens)</td>
<td>contains</td>
</tr>
<tr>
<td>Gold:</td>
<td>contains ignore case</td>
</tr>
<tr>
<td>code2seq (this work)</td>
<td>contains ignore case</td>
</tr>
</tbody>
</table>

Figure 7: Java examples from our test set for the code summarization task, along with the prediction of our model and each of the baselines.
Figure 8: Visualization of the BLEU score of our model compared to the baselines, for the code captioning task. The values are the same as in Table 2. Our model achieves significantly higher results than the baselines.

Figure 9: Visualization of the F1 score of our model compared to the baselines, for the code summarization task, across datasets. The values are the F1 columns from Table 1. Our model achieves significantly higher results than the baselines.
Figure 10: The relative decrease in precision and recall in each of the ablations, compared to the full model.