Abstract

We consider the problem of building probabilistic models that jointly model short natural language utterances and source code snippets. The aim is to bring together recent work on statistical modelling of source code and work on bimodal models of images and natural language. The resulting models are useful for a variety of tasks that involve natural language and source code. We demonstrate their performance on two retrieval tasks: retrieving source code snippets given a natural language query, and retrieving natural language descriptions given a source code query (i.e., source code captioning). Experiments show there to be promise in this direction, and that modelling the structure of source code improves performance.

1. Introduction

Software plays a central role in society, touching billions of lives on a daily basis. Writing and maintaining software — in the form of source code — is a core activity of software engineers, who aim to provide reliable and functional software. However, writing and maintaining source code is a costly business; software developers need to constantly look at documentation and online resources, and they need to make sense of large existing code bases. Both of these can be challenging and slow down the development process.

This need motivates our work here, where we seek to build joint models of natural language and snippets of source code. Advances in joint models of these two modalities could lead to tools that make writing and understanding software significantly faster and easier. Our approach combines two lines of work. First, Hindle et al. (2012); Maddison & Tarlow (2014); Raychev et al. (2015); Tu et al. (2014) have built increasingly sophisticated statistical models of source code. Second, in machine learning and computer vision there have been rapid recent advances in bimodal models that map between images and natural language (Srivastava & Salakhutdinov, 2012; Kiros et al., 2013; Socher et al., 2014; Fang et al., 2014; Vinyals et al., 2014; Karpathy & Fei-Fei, 2014). Can we take inspiration from these recent works and build models that map from natural language to source code, and from source code to natural language?

We explore the problem in this work, building a model that allows mapping in both directions. We leverage data that has short natural language utterances paired with source code snippets, like titles of questions along with source code found in answers from StackOverflow.com. We make three main contributions: (1) combining ideas from the two lines of work mentioned above, showing that this direction has promise going forward; (2) describing modelling and learning challenges that arose in the process of building these models and giving solutions that allowed us to overcome the challenges; and (3) showing how the performance of the models are affected by increasingly difficult instances of the problem. Results on the retrieval task show that we can often discern the proper natural language description for previously unseen source code snippets from a reasonably sized set of candidate descriptions.

2. Preliminaries

Let \( \mathcal{L} \) be a sequence of words in natural language and \( \mathcal{C} \) be a source code snippet. The first high level choice is whether to formulate a model where code is conditional upon language (i.e., \( P(\mathcal{C} | \mathcal{L}) \)), language is conditional upon code (i.e., \( P(\mathcal{L} | \mathcal{C}) \)), or perhaps use an undirected model. While any of these would be possible, we decided to define the model in terms of \( P(\mathcal{C} | \mathcal{L}) \) because it leads to the more natural way of encoding known structure of source code.
if (x < 100) x = 0;

Figure 1. Source code Example

(1) IfStatement
   (2) IfToken -- (3) if
   (4) OpenParenToken -- (5) ( 
   (6) LessThanExpression
     (7) IdentifierName -- (8) IdentifierToken -- (9) x
     (10) <
     (11) NumericalLiteralExpression
     (12) NumericalLiteralToken -- (13) 100
   (14) CloseParenToken -- (15) )
   (16) ExpressionStatement...

Figure 2. Parse Tree corresponding to Figure 1. Shaded nodes are terminal nodes. Numbers in parentheses are the identity of the node, labeled according to a left-to-right depth first traversal.

The next high level decision is how to represent source code. In its most basic form, source code is simply a string. However, programming languages are designed such that two transformations can be done unambiguously: converting an unstructured string into a sequence of tokens (lexing), and converting the sequence of tokens into a parse tree (parsing). Moreover, these operations can be performed easily using modern compilers. We choose to take advantage of the parse tree structure and follow previous works like Maddison & Tarlow (2014) in formalizing the model over source code. From hereon, when we refer to a source code snippet \( C \), we mean its parse tree structure. As an example, a source code snippet and its associated parse tree are shown in Figure 1 and Figure 2 respectively. The leaf nodes (gray) of the tree are tokens (i.e., strings that appeared in the original source code text). The internal nodes are specific to the programming language and correspond to expressions, statements or other high level syntactic elements such as ForStatement and Expression. These node types are specified by the programming language designers, and parsers allow us to map from a raw string to these trees.

Notation. We let \( I \) be the set of internal nodetypes (nonterminals) and \( K \) be the set of tokens (terminals) that appear across all snippets in our dataset. A parse tree \( C = (N, ch, \text{val}) \) is a triple made up of nodes \( N = \{1, \ldots, N\} \), a children function \( ch : N \rightarrow N^* \) that maps parent nodes to tuples of children nodes, and a value function \( \text{val} : N \rightarrow I \cup K \) that maps nodes to an internal node type or a token. We use the convention that \( ch(n) = \emptyset \) means \( n \) is a leaf node. For convenience, we will also overload notation and define tuple operations \( ch(n_1, \ldots, n_K) = (ch(n_1), \ldots, ch(n_K)) \) and \( v = \text{val}((n_1, \ldots, n_K)) = (\text{val}(n_1), \ldots, \text{val}(n_K)) \). Nodes are indexed according to when they would be instantiated during a left-to-right depth first traversal of the tree. For example, if \( a \) is the root, \( ch(a) = (b, c) \), \( ch(b) = (d) \), and \( ch(c) = (e) \), then the nodes would be labeled as \( a = 1, b = 2, c = 3, d = 4, e = 5 \). Finally, we also define partial parse trees \( C_{\leq n} \) to be equal to \( C \) but with the nodes restricted to be \( N = \{1, \ldots, n\} \), and for any node \( n' \) such that \( ch(n') \) contains a node with index \( > n \), \( ch(n') \) is set to \( \emptyset \).

Model Overview. We model a parse tree with a directed model that sequentially generates a child tuple for node \( n \) conditioned upon the natural language input \( L \) and the partial tree \( C_{\leq n} \):

\[
P(C | L) = \prod_{n \in N : \text{ch}(n) \neq \emptyset} P(\text{val}(\text{ch}(n)) | L, C_{\leq n}).
\]

In a bit more detail, we define \( \text{supp}(i) = \{v : v = \text{val}(\text{ch}(n)) \land \text{val}(n) = i \} \) for some \( n \) in dataset (the “empirical support”). To define our models, we will construct scoring functions \( s_\theta(v, L, C_{\leq n}) \) that can be converted to probabilities by exponentiating and normalizing over the support of the parent node type:

\[
P(v | L, C_{\leq n}) = \frac{\exp s_\theta(v, L, C_{\leq n})}{\sum_{v' \in \text{supp}(\text{val}(n))} \exp s_\theta(v', L, C_{\leq n})},
\]

where \( \theta \) are parameters to be learned from training data. This formulation gives quite a bit of flexibility over the range of models depending on how exactly the generation of the next children tuple depends on the natural language and previous partial tree. For example, if we were to define \( s_\theta(v, L, C_{\leq n}) = \log \text{count}(v, \text{val}(n)) \), where \( \text{count} \) is the number of times that \( v \) has appeared as a child of a node of type \( \text{val}(n) \), then this is a probabilistic context free grammar (PCFG). Following previous work on modelling source code, we explore models with richer dependency structure.

To sample from these models at test time, we can incrementally construct a tree by fixing the root node, sampling a children tuple conditional upon the initial partial tree (just the root node), then recursing to the leftmost child \( n \) such that \( \text{val}(n) \in I \), updating the partial tree, and repeating until children tuples have been chosen for all nonterminals.

3. Joint Code and Natural Language Models

We now turn our attention to the modelling specifics. Within all models we adhere to the structure described in the previous section. The variation in models will come from three modelling choices: how to represent the natural language \( L \); how to represent the partial trees \( C_{\leq n} \); and how to combine the above representations. As is now common (e.g., Kiros et al. (2013)), we focus on learning fixed-length real-valued
vector representations for each component of the model. There are three classes of representation vector: natural language vector \( l \in \mathbb{R}^D \) computed from a given \( L \), partial tree vector \( c \in \mathbb{R}^D \) computed from a given \( C_{\leq n} \), and production vector \( r \in \mathbb{R}^D \) that is unique to each parent-child (\( i, v \)) pair. Finally, there are production-specific biases \( b_{i,v} \).

3.1. Combining Representations

We experimented with two ways of mapping from representation vectors to score functions. The first is an additive model where \( s(v, L, C_{\leq n}) = (I + c)^\top r + b \), and the second is a multiplicative model where \( s(v, L, C_{\leq n}) = (I \odot c)^\top r + b \), where \( \odot \) denotes elementwise multiplication. The multiplicative model is similar to the multiplicative compositional models of Mitchell & Lapata (2010) and a special case of the factored model of Kiros et al. (2013). We have found reason to prefer the multiplicative interactions in the natural language-to-source code domain. A detailed explanation appears in Sec. 4.2 when we discuss results on a simple synthetic data set.

3.2. Natural Language Representations

Here we focus on how to map \( L \) to \( l \). Our approach is to use a bag of words (BoW) assumption, learning a distributed representation for each word, then combining them as a simple average. More specifically, in the BoW model, we let each natural language word \( w \) have a \( D \)-dimensional representation vector \( l_w \) that is shared across all appearances of \( w \), then we let \( l \) be the average of the representation vectors for words in \( L \); i.e., \( l = \frac{1}{|L|} \sum_{w \in L} l_w \).

While this representation may appear simplistic, we believe it to be reasonable when natural language takes the form of short search query-like utterances, which is the case in much of our data. In future work, as we turn towards learning from longer natural language utterances, we would like to experiment with alternative models like LSTMs (Hochreiter & Schmidhuber, 1997; Sutskever et al., 2014).

3.3. Partial Tree Representations

The final component of the model is how to represent decisions that have been made so far in constructing the tree. This involves extracting features of the partial tree that we believe to be relevant to the prediction of the next children tuple. The two main features that we use are based on the 10 previous tokens that have been generated (note that due to the definition of \( C_{\leq n} \), tokens are generated in the order that they appear in the input source code text), and the 10 previous internal node types that are encountered by following the path from node \( n \) to the root. In both cases, padding symbols are added as necessary. Feature values (e.g., \texttt{int}, \texttt{i}, \texttt{=} ) are shared across different feature positions (e.g., previous token, two tokens back, three tokens back). For each feature value \( \phi \), there is a representation vector \( e_{\phi} \). The \( e_{\phi} \) vectors depend only on the feature value, so in order to preserve the (position, value) pairs and not just the set of values, a different context matrix \( H_j \) is needed for each feature position \( j \). We then modulate the feature vectors by a position-specific diagonal context matrix to get the \( c \) vector: \( c = \sum_{j=1}^{J} H_j e_{\phi_j} \).

3.4. Learning

At training time, we observe \( (L, C) \) pairs. Our goal is to learn parameters (representation vectors and context matrices) so as to maximize the probability \( P(C | L) \). All partial trees of \( C \) can be constructed easily, so training amounts to maximizing the sum of log probabilities of each production given the associated natural language and partial tree up to the current prediction point. We approximate the objective using noise contrastive estimation (NCE) (Gutmann & Hyvärinen, 2012; Mnih & Teh, 2012), which eliminates the need to compute expensive normalizing constants. Letting \( k \) be a parameter of the NCE training and \( \Delta s(v, L, C_{\leq n}) = s_\theta(v, L, C_{\leq n}) - \log(k P_{\text{noise}}(v | \text{val}(n))) \), the objective can be written as in Mnih & Kavukcuoglu (2013):

\[
\begin{align*}
E_{(L, C_{\leq n}, v) \sim D} [\log \Delta s(v, L, C_{\leq n})] \\
+ k E_{(L, C_{\leq n}, v') \sim \text{noise}} [\log(1 - \Delta s(v', L, C_{\leq n}))],
\end{align*}
\]

where \( D \) is the data distribution, and \( \text{noise} \) is the distribution where \( (L, C_{\leq n}) \) pairs are sampled from the data distribution then child tuple \( v' \) is drawn from the noise distribution, which can be conditional upon \( L \) and \( C_{\leq n} \). Our noise distribution \( P_{\text{noise}}(v | i) \) is the posterior PCFG of the training data with a simple Dirichlet prior (so it only depends on the partial tree \( C_{\leq n} \)). For optimization, we use AdaGrad (Duchi et al., 2011). We initialize the biases \( b_{i,v} \) to the noise PCFG distribution such that \( b_{i,v} = \log P_{\text{noise}}(v | i) \). The rest of the representations are initialized randomly around a central number with some small additive noise. \( l_i \) components are initialized with center 0, \( e_{\phi_i} \) components centered at 1 when using the multiplicative model or centered at 0 for the additive model and the diagonals of \( H \), at \( \frac{1}{2} \).

4. Evaluation

In this section, we evaluate the bimodal source code language model, using natural language descriptions and C# code. We use Roslyn (.NET Compiler Platform) to parse C# source code snippets into parse trees. For each of the evaluation datasets, we create three distinct sets: the trainset that contains 70% of the code snippets, the test1 set that contains the same snippets as the trainset but novel natural language queries (if any) and the test2 set that contains the remaining 30% of the snippets with their associated natural language queries. Each snippet is described by a constant number of
queries, uniformly sampled with replacement from all the
queries describing it. The rationale for this choice is to avoid
balancing training and evaluation against code snippets that
are expressed with disproportionately more natural language
queries than other snippets.

**Experimental Setup.** The goal of our current work is to
use the code language model to assist two retrieval tasks:
The **Snippet Retrieval Task** ($L \rightarrow C$) refers to the problem
of retrieving the most relevant snippet $C$, given a natural
language query $L$. The reverse problem, i.e. the **Query Re-
trieval Task** ($C \rightarrow L$) aims to retrieve a natural-language
query $L$, given a specific snippet $C$. To evaluate the perform-
ance of the models, we sample at uniform 100 retrieval
targets (i.e. snippets and queries pairs). Then, for each target
pair, we sample 49 distractor targets. We then compute the
probability of each of the query-snippet pairs and rank them
according to the per production cross-entropy. Based on this
ranking, we compute the mean reciprocal rank (MRR). As
a baseline, we include a natural language only (NL-only)
model that does not take into account the tree representation
c and thus has $s(v, L, C_{<n}) = I^\top r + b$. This model can be
interpreted as a PCFG conditioned on natural language.

### 4.1. Synthetic Data

We generate synthetic data on a limited domain to test the
ability of the model to learn in a controlled environment.
The **Text** dataset is concerned with simple text operations
that may be performed in strings. String operations include
splitting delimited strings (i.e. delimited with comma, tab,
space or new line), uppercasing or lowercasing the letters
of a word, counting the number of characters, retrieving
a substring of a string, getting the length of a string
and parsing a string to a double. An aggregation operation
may also be applied; such operations include concatenat-
ing strings, finding the minimum or maximum of some
(numeric) elements, summing, counting, finding distinct
elements, averaging numerical elements and retrieving the
first or last element of a list. We synthesize LINQ queries
(MSDN) that correspond to all the type-correct operations
and a large number of synthetic language queries for each
snippet. The resulting dataset contains 163 code snippets
with 27 natural language queries in each of the train, test1
and test2 datasets.

We then train ($D = 20$, 100 iterations) and evaluate the
logbilinear models on the synthetic data. Results are shown in
Table 1. The multiplicative model performs the best,
while the additive and the natural language models have
inferior performance. Since this is a synthetic dataset, in a
limited domain, we can achieve very high MRR. By using
the model to generate snippets give code, we observe that
it can correctly generate previously unseen snippets from
new natural language queries. For example, the test natural
language query “each element parse double separated by a
tab and get max”, returns the snippet

```csharp
var res = input_string.Split('\t').Select((string x) => Double.Parse(x)).Max();
```

The model was able to generate this snippet although it never
saw the snippet before. However, the model had learned from
the training snippets

```csharp
var res = input_string.Split(' ').Select((string x) => Double.Parse(x)).Max();
```

```csharp
var res = input_string.Split('\t').Select((string x) => Double.Parse(x)).Min();
```

the correct mapping between the natural language and
source code, generalizing successfully.

### 4.2. The Importance of Multiplicative Combination

Looking at Table 1 the multiplicative model has a clear
performance advantage. Indeed, the multiplicative and the
additive models have different representational capacities.
While the gating behavior of multiplicative models has been
discussed previously (Memisevic & Hinton, 2007; Taylor
& Hinton, 2009; Kiros et al., 2013), our aim here is to
explain the importance of these multiplicative interactions
in the context of source code modelling, and to point out a
concrete difference in their representational abilities.

Suppose we have two modalities with two possible values
each. The natural language $L$ specifies one of two possible
values for $I$, say whether the goal is to iterate over a matrix

<table>
<thead>
<tr>
<th>Model</th>
<th>Train</th>
<th>Test 1</th>
<th>Test 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiplicative</td>
<td>0.986</td>
<td>0.988</td>
<td>0.921</td>
</tr>
<tr>
<td>additive</td>
<td>0.890</td>
<td>0.805</td>
<td>0.919</td>
</tr>
<tr>
<td>NL-only</td>
<td>0.876</td>
<td>0.817</td>
<td>0.803</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model</th>
<th>Train</th>
<th>Test 1</th>
<th>Test 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>multiplicative</td>
<td>0.995</td>
<td>1.000</td>
<td>0.892</td>
</tr>
<tr>
<td>additive</td>
<td>0.860</td>
<td>0.883</td>
<td>0.892</td>
</tr>
<tr>
<td>NL-only</td>
<td>0.917</td>
<td>0.895</td>
<td>0.845</td>
</tr>
</tbody>
</table>

Table 1. Mean Reciprocal Rank for the **Text** synthetic dataset for the two retrieval problems.

<table>
<thead>
<tr>
<th>$L$</th>
<th>$C$</th>
<th>Required Relationships</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$a \rightarrow p$</td>
<td>$(l_1 \diamond c_a)\top r_p \gg (l_1 \diamond c_b)\top r_p$</td>
</tr>
<tr>
<td></td>
<td>$b \rightarrow q$</td>
<td>$(l_1 \diamond c_q)\top r_q \gg (l_1 \diamond c_q)\top r_q$</td>
</tr>
</tbody>
</table>

Table 2. Synthetic example: Modality $L (L \in \{1, 2\})$ modulates modality $C (C \in \{a, b\})$ for the target space $\{p, q\}$. The composition operation $\diamond$ can be either an addition or a multiplication.

Bimodal Modelling of Source Code and Natural Language
in row major order ($I_1$) or column major order ($I_2$). Similarly, the context representation $c$ has two possible values, denoting the context of being about to declare an identifier within an outer for loop ($c_a$) or an inner for loop ($c_b$). Finally, assume our data always has $i$ as the row index and $j$ as the column index, and our goal is to choose whether to use $i$ ($r_p$) or $j$ ($r_q$). In this example, the natural language modality $L$ needs to acts as a switch that inverts the meaning of the context. This can be done by the multiplicative model, but cannot be done by the additive model.

Table 2 formalizes this claim. The required relationships come from writing down the constraints implied by the above situation. When substituting the composition operation $\circ$ with $+$, the $L^j r$ terms cancel and we are left with ($L = 1$ case)

$$c_a^\top r_p \gg c_b^\top r_p \land c_b^\top r_q \gg c_a^\top r_q \implies c_a^\top r_p \gg c_a^\top r_q \quad (4)$$

and ($L = 2$ case)

$$c_a^\top r_q \gg c_b^\top r_q \land c_b^\top r_p \gg c_a^\top r_p \implies c_a^\top r_q \gg c_a^\top r_p \quad (5)$$

which are contradictory and thus impossible to satisfy. The inadequacy of the additive model resembles the inability of a single perceptron to learn a XOR relationship. In contrast, it is easy to show that the multiplicative model (i.e. substituting $\circ$ with $\odot$) is able to satisfy the inequalities in Table 2. In early development of these models, we encountered situations where this issue manifested itself, and we believe (perhaps softer versions of it) to be an important property of multimodal models of source code.

4.3. Real-World Datasets

We now create two real-world datasets to evaluate the model’s ability to achieve good performance in the retrieval tasks. We mine C# snippets from two sources: First, we use StackOverflow\(^1\), a question and answer site that is commonly used by developers. StackOverflow contains questions in natural language and answers that may contain snippets of code. StackOverflow data is freely available online though the StackExchange Data Explorer. We extract all questions and answers tagged with the C# tag and use the title of the question as the natural language query and the code snippets in the answers as the target source code. We filter out any questions that have less than 2 votes or answers that have less than 3 votes, or have been viewed less than 1000 times or have no code snippets or the code snippet cannot be parsed by Roslyn. We also remove snippets that contain more than 300 characters, assuming that longer snippets will be less informative. The goal of this filtering is to create a high-quality corpus with snippets that have been deemed useful by other developers. We use the abbreviation SO to refer to this dataset. Similarly, Dot Net Perls\(^2\) is a popular site with C# tutorials. We scraped the site for code snippets along with the natural language captions they are associated with. We refer to this dataset as perls-all.

For both datasets, to increase the natural language data, we additionally use data from a large online general-purpose search engine adding search engine queries that produced clicks that led to any of the StackOverflow or Dot Net Perls web pages in the original dataset. We remove any queries that mapped to more than 4 different snippets, since they are probably vague. For each of the two datasets we create the three different sets, train, test1 and test2 as explained in the beginning of this section. The size of each of the resulting datasets are shown in Table 3 and samples of the datasets are shown in Table 4 and in the supplemental materials of this paper. Specifically, for Dot Net Perls, we also create the perls-captions dataset that contains only one natural language description for each snippet (the caption), excluding the queries from the general-purpose search engine. perls-captions does not have a test1 set, since we have no alternative natural language descriptions.

We randomly sample five full datasets (train, test1, test2) from the original data and train our models. The evaluation results are reported in Table 5. The multiplicative model is achieving the highest MRR for both retrieval problems and overall the performance on the $C \rightarrow L$ task is significantly better than the performance in $L \rightarrow C$. Samples of retrieved queries ($C \rightarrow L$) are also shown in Table 4. Figure 3 shows how the performance of the models change for different values of $D$. As expected, when $D$ increases, MRR improves with diminishing returns. Also, the perls-all and perls-captions dataset which are smaller and more sparse have minor improvements for $D$ larger than 50 and are more prone to overfitting. Finally, the additive model fails to improve significantly as $D$ increases.

**Table 3. Size of the real-world datasets**

```
<table>
<thead>
<tr>
<th></th>
<th>SO</th>
<th>perls-captions</th>
<th>perls-all</th>
</tr>
</thead>
<tbody>
<tr>
<td>Snippets</td>
<td>Train</td>
<td>24812</td>
<td>1467</td>
</tr>
<tr>
<td></td>
<td>Test1</td>
<td>17462</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Test2</td>
<td>11469</td>
<td>328</td>
</tr>
<tr>
<td>Queries per Snippet</td>
<td>Train</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Test1</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Test2</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>
```

**Qualitative Analysis.** To qualitatively analyse the performance of the models, we use the trained models as conditional generative models of snippets given an input test query. We do not expect the model to generate perfect snippets that match exactly the target snippet, but we hope to

\(^1\)http://stackoverflow.com

\(^2\)http://dotnetperls.com
while (number >= 10)  
number /= 10;

string SearchText = "7,true,NA,\false;67,\false,NA,\false;5";  
string Regex = @"\btrue\b";  
int NumberOfTrues = Regex.Matches(SearchText, Regex).Count;

using (var cc = new ConsoleCopy("mylogfile.txt")) {  
Console.WriteLine("testing 1-2");  
Console.WriteLine("testing 3-4");  
Console.ReadKey();  
}

Table 4. Examples from datasets and natural language query retrieval examples. To retrieve the queries we restrict the method by removing all natural language descriptions that are assigned to the target snippet, except for one. All samples from test2 datasets.
see the correlations that have been learned from the data. Two random examples from the datasets follow.

**perls-all Query:** dictionary lookup

```csharp
using System;
using Generic;

class Generic {
    static void dictionary() {
        dictionary.ContainsKey(ContainsKey);
    }
}
```

**SO Query:** comma delimited list trailing comma

```csharp
foreach (string Split in Join) {
    return string.Join(',,,
```

While the sampled code is far from the ground truth snippets, the model has learned interesting connections between source code and natural language features. For example, for the perls-all query “dictionary lookup” the model has learned that identifiers of variables such as dictionary or method names like ContainsKey are relevant to the query. Similarly, for the SO query “comma delimited list trailing comma” that string operations are relevant, as well as the comma character. It can also be seen that the target snippets are very noisy, containing literals and code that do not necessarily correlate with the target query. It can also be seen that the SO dataset usually contains shorter (but sometimes more noisy) snippets. Finally, it is important to note that while our source code only produces parsable snippets under the assumption that input source code is parsable, this assumption is sometimes violated with SO data, which causes Roslyn to return erroneous or incomplete partial parses. This noise in the StackOverflow data introduces some noise in the code generation, but as seen from Table 5 the model can score reasonably snippet-query pairs in the retrieval tasks.

**Ranking Confidence.** During evaluation, we noticed that the model made errors when it did not have enough information to discriminate between the target and the distractors. Figure 4 (a) and Figure 4 (b) plot the average MRR depending as we vary the minimum confidence threshold for making a suggestion. We define the confidence metric for a single ranking as the difference between the maximum and the model made errors when it did not have enough information to discriminate between the target and the distractors. We define the confidence metric for a single ranking as the difference between the maximum and the minimum MRR. While the sampled code is far from the ground truth snippets, the model has learned interesting connections between source code and natural language features. For example, for the perls-all query “dictionary lookup” the model has learned that identifiers of variables such as dictionary or method names like ContainsKey are relevant to the query. Similarly, for the SO query “comma delimited list trailing comma” that string operations are relevant, as well as the comma character. It can also be seen that the target snippets are very noisy, containing literals and code that do not necessarily correlate with the target query. It can also be seen that the SO dataset usually contains shorter (but sometimes more noisy) snippets. Finally, it is important to note that while our source code only produces parsable snippets under the assumption that input source code is parsable, this assumption is sometimes violated with SO data, which causes Roslyn to return erroneous or incomplete partial parses. This noise in the StackOverflow data introduces some noise in the code generation, but as seen from Table 5 the model can score reasonably snippet-query pairs in the retrieval tasks.

**Ranking Confidence.** During evaluation, we noticed that the model made errors when it did not have enough information to discriminate between the target and the distractors. Figure 4 (a) and Figure 4 (b) plot the average MRR depending as we vary the minimum confidence threshold for making a suggestion. We define the confidence metric for a single ranking as the difference between the maximum and the minimum MRR. While the sampled code is far from the ground truth snippets, the model has learned interesting connections between source code and natural language features. For example, for the perls-all query “dictionary lookup” the model has learned that identifiers of variables such as dictionary or method names like ContainsKey are relevant to the query. Similarly, for the SO query “comma delimited list trailing comma” that string operations are relevant, as well as the comma character. It can also be seen that the target snippets are very noisy, containing literals and code that do not necessarily correlate with the target query. It can also be seen that the SO dataset usually contains shorter (but sometimes more noisy) snippets. Finally, it is important to note that while our source code only produces parsable snippets under the assumption that input source code is parsable, this assumption is sometimes violated with SO data, which causes Roslyn to return erroneous or incomplete partial parses. This noise in the StackOverflow data introduces some noise in the code generation, but as seen from Table 5 the model can score reasonably snippet-query pairs in the retrieval tasks.

**Ranking Confidence.** During evaluation, we noticed that the model made errors when it did not have enough information to discriminate between the target and the distractors. Figure 4 (a) and Figure 4 (b) plot the average MRR depending as we vary the minimum confidence threshold for making a suggestion. We define the confidence metric for a single ranking as the difference between the maximum and the minimum MRR. While the sampled code is far from the ground truth snippets, the model has learned interesting connections between source code and natural language features. For example, for the perls-all query “dictionary lookup” the model has learned that identifiers of variables such as dictionary or method names like ContainsKey are relevant to the query. Similarly, for the SO query “comma delimited list trailing comma” that string operations are relevant, as well as the comma character. It can also be seen that the target snippets are very noisy, containing literals and code that do not necessarily correlate with the target query. It can also be seen that the SO dataset usually contains shorter (but sometimes more noisy) snippets. Finally, it is important to note that while our source code only produces parsable snippets under the assumption that input source code is parsable, this assumption is sometimes violated with SO data, which causes Roslyn to return erroneous or incomplete partial parses. This noise in the StackOverflow data introduces some noise in the code generation, but as seen from Table 5 the model can score reasonably snippet-query pairs in the retrieval tasks.

**Ranking Confidence.** During evaluation, we noticed that the model made errors when it did not have enough information to discriminate between the target and the distractors. Figure 4 (a) and Figure 4 (b) plot the average MRR depending as we vary the minimum confidence threshold for making a suggestion. We define the confidence metric for a single ranking as the difference between the maximum and the minimum MRR. While the sampled code is far from the ground truth snippets, the model has learned interesting connections between source code and natural language features. For example, for the perls-all query “dictionary lookup” the model has learned that identifiers of variables such as dictionary or method names like ContainsKey are relevant to the query. Similarly, for the SO query “comma delimited list trailing comma” that string operations are relevant, as well as the comma character. It can also be seen that the target snippets are very noisy, containing literals and code that do not necessarily correlate with the target query. It can also be seen that the SO dataset usually contains shorter (but sometimes more noisy) snippets. Finally, it is important to note that while our source code only produces parsable snippets under the assumption that input source code is parsable, this assumption is sometimes violated with SO data, which causes Roslyn to return erroneous or incomplete partial parses. This noise in the StackOverflow data introduces some noise in the code generation, but as seen from Table 5 the model can score reasonably snippet-query pairs in the retrieval tasks.
Another related area is that of semantic parsing, where the goal is to map a natural language utterance to a logical description of its meaning (Zelle & Mooney, 1996; Zettlemoyer & Collins, 2005; Liang et al., 2013). The difference between our setting and these is that the connection between natural language and target code is much looser. We do not expect the natural language to describe step-by-step how to construct the code; a semantic parse of the natural language would bear little resemblance to the target source code. These systems also often require additional hand-specified knowledge about the mapping between natural language and the logical forms; for example, to create a lexicon. We note Kushner & Barzilay (2013) weaken the assumption that language and target are well-aligned in a natural language to regular expression application, but the method is specific to regular expressions, and the mapping is still more aligned than in our setting.

6. Discussion

While the task considered in this work is very hard, the results are reasonably good. On the perls-captions data, we are able to rank the true caption for a previously unseen piece of code amongst the top few when presented against the distractors captions. As we move to noisier natural language (perls-captions) and noisier code snippets (SO), performance degrades, but only moderately. Interestingly, it appears easier to pick the proper natural language for a given piece of code than it is to pick the proper code for a given piece of natural language. We think this is due to the fact that there is less variability in the code, so picking apart subtle differences is more difficult. Another interesting note is how the models that incorporate structure of the code consistently and soundly outperform the models that do not. Our interpretation is that the models that ignore code structure force the model to account for correlations in the source code via the natural language, which makes the task harder while also increasing the risk of overfitting.

While the dataset sizes we have used are moderate, we think a promising path forward is to find larger datasets. Some ideas for where to find these include generalizing the model to handle a larger set of programming languages, and/or learning from longer text snippets, as would be found in a programming language specification document. This will likely require more sophisticated representations in the natural language component.

Finally, we think there is value in establishing the analog between multimodal source code and natural language models, and multimodal image and natural language models. As these models improve, more applications will become possible, and we are particularly excited by the potential for cross-fertilization between these two application areas.
References


