Bimodal Modelling of Source Code and Natural Language

Miltiadis Allamanis[†] M.ALLAMANIS@ED.AC.UK
School of Informatics University of Edinburgh, Edinburgh, EH8 9AB, United Kingdom

Daniel Tarlow DTARLOW@MICROSOFT.COM
Andrew D. Gordon ADG@MICROSOFT.COM
Yi Wei YIWE@MICROSOFT.COM
Microsoft Research, 21 Station Road, Cambridge, CB1 2FB, United Kingdom

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., $P(\mathcal{C} \mid \mathcal{L})$), language is conditional upon code (i.e., $P(\mathcal{L} \mid \mathcal{C})$), or perhaps use an undirected model. While any of these would be possible, we decided to define the model in terms of $P(\mathcal{C} \mid \mathcal{L})$ because it leads to the more natural way of encoding known structure of source code.

[†]Work done primarily while author was an intern at Microsoft Research.

Proceedings of the 32^{nd} International Conference on Machine Learning, Lille, France, 2015. JMLR: W&CP volume 37.

```
if (x < 100) x = 0;
```

Figure 1. Source code Example

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

Model Overview. We model a parse tree with a directed model that sequentially generates a child tuple for node n conditional upon the natural language input $\mathcal{L}$ and the partial tree $\mathcal{C}_{\leq n}$:

$$P(\mathcal{C} \mid \mathcal{L}) = \prod_{n \in \mathcal{N} : \text{ch}(n) \neq \emptyset}^{N} P(\text{val}(\text{ch}(n)) \mid \mathcal{L}, \mathcal{C}_{\leq n}). \quad (1)$$

In a bit more detail, we define $\text{supp}(i) = \{\boldsymbol{v} : \boldsymbol{v} = \text{val}(\text{ch}(n)) \wedge \text{val}(n) = i \text{ for some } n \text{ in dataset}\}$ to be the set of all children tuples that appear as the children of a node of type i in our dataset (the "empirical support"). To define our models, we will construct scoring functions $s_\theta(\boldsymbol{v}, \mathcal{L}, \mathcal{C}_{\leq n})$ that can be converted to probabilities by exponentiating and normalizing over the support of the parent node type:

$$P(\boldsymbol{v} \mid \mathcal{L}, \mathcal{C}_{\leq n}) = \frac{\exp s_\theta(\boldsymbol{v}, \mathcal{L}, \mathcal{C}_{\leq n})}{\sum_{\boldsymbol{v}' \in \text{supp}(\text{val}(n))} \exp s_\theta(\boldsymbol{v}, \mathcal{L}, \mathcal{C}_{\leq n})}, \quad (2)$$

where θ 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 $s_\theta(\boldsymbol{v}, \mathcal{L}, \mathcal{C}_{\leq n}) = \log \text{count}(\boldsymbol{v}, \text{val}(n))$, where count is the number of times that $\boldsymbol{v}$ 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 left-most child n such that $\text{val}(n) \in \mathcal{I}$, 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 $\mathcal{L}$; how to represent the partial trees $\mathcal{C}_{\leq n}$; 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 $\boldsymbol{l} \in \mathbb{R}^D$ computed from a given $\mathcal{L}$, partial tree vector $\boldsymbol{c} \in \mathbb{R}^D$ computed from a given $\mathcal{C}_{<n}$, and production vector $\boldsymbol{r} \in \mathbb{R}^D$ that is unique to each parent-children $(i, \boldsymbol{v})$ pair. Finally, there are production-specific biases $b_{i,\boldsymbol{v}}$.

3.1. Combining Representations

We experimented with two ways of mapping from representation vectors to score functions. The first is an *additive* model where $s(\boldsymbol{v}, \mathcal{L}, \mathcal{C}_{<n}) = (\boldsymbol{l} + \boldsymbol{c})^\top \boldsymbol{r} + b$, and the second is a *multiplicative* model where $s(\boldsymbol{v}, \mathcal{L}, \mathcal{C}_{<n}) = (\boldsymbol{l} \odot \boldsymbol{c})^\top \boldsymbol{r} + b$, 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 $\mathcal{L}$ to $\boldsymbol{l}$. 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 $\boldsymbol{l}_w$ that is shared across all appearances of w, then we let $\boldsymbol{l}$ be the average of the representation vectors for words in $\mathcal{L}$; i.e., $\boldsymbol{l} = \frac{1}{|\mathcal{L}|} \sum_{w \in \mathcal{L}} \boldsymbol{l}_w$.

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 $\mathcal{C}_{<n}$, 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., `int`, `i`, `=`) are shared across different feature positions (e.g., previous token, two tokens back, three tokens back). For each feature value ϕ, there is a representation vector $\boldsymbol{c}_\phi$. The $\boldsymbol{c}_\phi$ 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 $\mathbf{H}_j$ is needed for each feature position j. We then modulate the feature vectors by a position-specific diagonal context matrix to get the $\boldsymbol{c}$ vector: $\boldsymbol{c} = \sum_{j=1}^{J} \mathbf{H}_j \boldsymbol{c}_{\phi_j}$.

3.4. Learning

At training time, we observe $(\mathcal{L}, \mathcal{C})$ pairs. Our goal is to learn parameters (representation vectors and context matrices) so as to maximize the probability $P(\mathcal{C} \mid \mathcal{L})$. All partial trees of $\mathcal{C}$ 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 $\Delta s(\boldsymbol{v}, \mathcal{L}, \mathcal{C}_{<n}) = s_\theta(\boldsymbol{v}, \mathcal{L}, \mathcal{C}_{<n}) - \log(k P_{noise}(\boldsymbol{v} \mid \text{val}(n)))$, the objective can be written as in Mnih & Kavukcuoglu (2013):

$$\mathbb{E}_{(\mathcal{L}, \mathcal{C}_{<n}, \boldsymbol{v}) \sim \mathcal{D}} \left[\log \Delta s(\boldsymbol{v}, \mathcal{L}, \mathcal{C}_{<n})\right] + k \mathbb{E}_{(\mathcal{L}, \mathcal{C}_{<n}, \boldsymbol{v}') \sim noise} \left[\log(1 - \Delta s(\boldsymbol{v}', \mathcal{L}, \mathcal{C}_{<n}))\right], \quad (3)$$

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

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 disproportionally 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* ($\mathcal{L} \rightarrow \mathcal{C}$) refers to the problem of retrieving the most relevant snippet $\mathcal{C}$, given a natural language query $\mathcal{L}$. The reverse problem, *i.e.* the *Query Retrieval Task* ($\mathcal{C} \rightarrow \mathcal{L}$) aims to retrieve a natural-language query $\mathcal{L}$, given a specific snippet $\mathcal{C}$. To evaluate the performance 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 $\boldsymbol{c}$ and thus has $s(\boldsymbol{v}, \mathcal{L}, \mathcal{C}_{\leq n}) = \boldsymbol{l}^\top \boldsymbol{r} + b$. 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 the 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 concatenating 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

	Model	Train	Test 1	Test 2
$\mathcal{L} \rightarrow \mathcal{C}$	multiplicative	**0.986**	**0.988**	**0.921**
	additive	0.890	0.805	0.919
	NL-only	0.876	0.817	0.803
$\mathcal{C} \rightarrow \mathcal{L}$	multiplicative	**0.995**	**0.995**	**1.000**
	additive	0.860	0.883	0.892
	NL-only	0.917	0.895	0.845

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

$\mathcal{L}$	$\mathcal{C}$	Required Relationships
1	$a \rightarrow p$	$(\boldsymbol{l}_1 \diamond \boldsymbol{c}_a)^\top \mathbf{r}_p \gg (\boldsymbol{l}_1 \diamond \boldsymbol{c}_b)^\top \mathbf{r}_p$
	$b \rightarrow q$	$(\boldsymbol{l}_1 \diamond \boldsymbol{c}_b)^\top \mathbf{r}_q \gg (\boldsymbol{l}_1 \diamond \boldsymbol{c}_a)^\top \mathbf{r}_q$
2	$a \rightarrow q$	$(\boldsymbol{l}_2 \diamond \boldsymbol{c}_a)^\top \mathbf{r}_q \gg (\boldsymbol{l}_2 \diamond \boldsymbol{c}_b)^\top \mathbf{r}_q$
	$b \rightarrow p$	$(\boldsymbol{l}_2 \diamond \boldsymbol{c}_b)^\top \mathbf{r}_p \gg (\boldsymbol{l}_2 \diamond \boldsymbol{c}_a)^\top \mathbf{r}_p$

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

language query *"each element parse double separated by a tab and get max"*, returns the snippet

```
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

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

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 $\mathcal{L}$ specifies one of two possible values for $\boldsymbol{l}$, say whether the goal is to iterate over a matrix

in row major order ($\boldsymbol{l}_1$) or column major order ($\boldsymbol{l}_2$). Similarly, the context representation $\boldsymbol{c}$ has two possible values, denoting the context of being about to declare an identifier within an outer `for` loop ($\boldsymbol{c}_a$) or an inner `for` loop ($\boldsymbol{c}_b$). 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` ($\boldsymbol{r}_p$) or `j` ($\boldsymbol{r}_q$). In this example, the natural language modality $\mathcal{L}$ 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 $\diamond$ with $+$, the $\boldsymbol{l}^\top \boldsymbol{r}$ terms cancel and we are left with ($\mathcal{L} = 1$ case)

$$\boldsymbol{c}_a^\top \mathbf{r}_p \gg \boldsymbol{c}_b^\top \mathbf{r}_p \wedge \boldsymbol{c}_b^\top \mathbf{r}_q \gg \boldsymbol{c}_a^\top \mathbf{r}_q \implies \boldsymbol{c}_a^\top \mathbf{r}_p \gg \boldsymbol{c}_a^\top \mathbf{r}_q \quad (4)$$

and ($\mathcal{L} = 2$ case)

$$\boldsymbol{c}_a^\top \mathbf{r}_q \gg \boldsymbol{c}_b^\top \mathbf{r}_q \wedge \boldsymbol{c}_b^\top \mathbf{r}_p \gg \boldsymbol{c}_a^\top \mathbf{r}_p \implies \boldsymbol{c}_a^\top \mathbf{r}_q \gg \boldsymbol{c}_a^\top \mathbf{r}_p, \quad (5)$$

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 $\diamond$ 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

[1] http://stackoverflow.com

		SO	perls-captions	perls-all
Snippets	Train	24812	1467	1467
	Test1	17462	-	1467
	Test2	11469	328	328
Queries per Snippet	Train	10	1	15
	Test1	10	-	15
	Test2	10	1	15

Table 3. Size of the real-world datasets

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 $\mathcal{C} \rightarrow \mathcal{L}$ task is significantly better than the performance in $\mathcal{L} \rightarrow \mathcal{C}$. Samples of retrieved queries ($\mathcal{C} \rightarrow \mathcal{L}$) 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.

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

[2] http://dotnetperls.com

	$\mathcal{C}$	$\mathcal{L}$	Retrieval Results
SO	`while (number >= 10)` `  number /= 10;`	how to get ones digit, first digit of int, get first two digits of a number, get ones place of int, get specific digit, how to get the first number in a integer, get specific digit, how to get a digit in int, get the first 3 digits of a number	1. **how to get ones digit** 2. string generate with number of spaces 3. check digit in string 4. number within certain range of another 5. integer between 3 and 4

	`string SearchText = "7,true,NA,\`
	`  false:67,false,NA,false:5";`
	`string Regex = @"\btrue\b";`
	`int NumberOfTrues = Regex.Matches(`
	`  SearchText, Regex).Count;`
		count how many times same string appears, how to search a character maximum no in a file, how to count the number of times a string appears in a list, determine how many times a character appears in a string, how to search a character maximum no in a file, count how many times a string appears in another string	1. string []
			2. setvalue letters
			3. truncate string to a length
			4. replace multiple groups
			5. efficient way to count zeros in byte array

	`using (var cc = new ConsoleCopy(`
	`  "mylogfile.txt")) {`
	`  Console.WriteLine("testing 1-2");`
	`  Console.WriteLine("testing 3-4");`
	`  Console.ReadKey();`
	`}`
		write to file or console, copy all console output to file, console output to a file, console app output log, write console, send output to console window and a file, add log file to console app,	1. do not overwrite file
			2. **copy all console output**
			3. open file path starting with
			4. copy file in addition to file
			5. hashing table

perls-all	`path=Path.GetFullPathInternal(path);`
	`new FileIOPermission(`
	`  FileIOPermissionAccess.Read,`
	`  new string[] { path },`
	`  false, false).Demand();`
	`flag = InternalExists(path);`
		check for file extension, how to tell if a directory exists, exist, determine a file exist on shared folder, check if list of files exists, how to tell if a directory exists createifexist file, excel file does not exist	1. wpf get directory name from path
			2. **determine a file exist on shared folder**
			3. open file dialog class
			4. create directory pathname
			5. load binary file to variable

	`using System;`
	`class Program {`
	`  static void Main() {`
	`  string input = "Dot Net Perls";`
	
	`  char[] array = input`
	`  .ToCharArray();`
	`  for (int i = 0;`
	`  i < array.Length; i++) {`
	`  char let = array[i];`
	`  if (char.IsUpper(let))`
	`  array[i] = char`
	`  .ToLower(let);`
	`  else if (let == ' ')`
	`  array[i] = '-';`
	`  else if (let == 'e')`
	`  array[i] = 'u';`
	`  }`
	`  string result = new string(array);`
	`  Console.WriteLine(result);`
	`  }`
	`}`
		how do i replace asingle string character, single character in array, modify string at, char to caps, single character in array, change string to, replace a string of characters, replace character in string position, change one char in a string, how to replace a character in a string at a postion, how do i replace asingle string character, how to modifiy a char in string, replace at position string	1. get number of character in a string
			2. remove a value from a list
			3. check if selected tab is null or not
			4. **convert string to**
			5. modify string at

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.

	Model	SO Test 1		SO Test 2		perls-captions Test2		perls-all Test 1		perls-all Test 2	
$\mathcal{C}$ ↑ $\mathcal{L}$	multiplicative	**0.182**	±.009	**0.170**	±.012	**0.254**	±.017	**0.441**	±.036	**0.270**	±.016
	additive	0.099	±.008	0.105	±.005	0.152	±.057	0.078	±.003	0.106	±.010
	NL-only	0.120	±.008	0.125	±.005	0.090	±.002	0.239	±.024	0.205	±.013
$\mathcal{L}$ ↑ $\mathcal{C}$	multiplicative	**0.434**	±.003	**0.413**	±.018	**0.650**	±.012	**0.716**	±.007	**0.517**	±.012
	additive	0.218	±.011	0.211	±.013	0.356	±.017	0.426	±.041	0.309	±.011
	NL-only	0.248	±.008	0.261	±.008	0.145	±.013	0.599	±.018	0.453	±.015

Table 5. Macro-averaged Mean Reciprocal Rank (MRR) and Standard Error for 5 random splits of SO and for the two retrieval problems. $D = 50$ for all models.

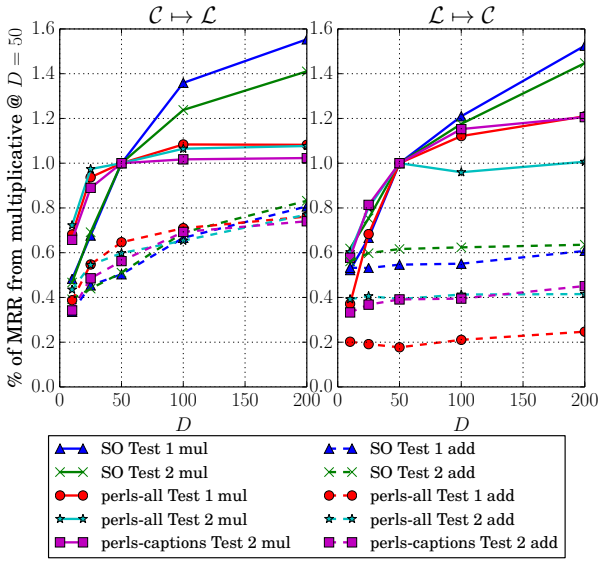


Figure 3. Performance on the datasets for different values of D. Graph shows the relative performance compared to the multiplicative model at $D = 50$ as shown in Table 5.

see the correlations that have been learned from the data. Two random examples from the datasets follow.

perls-all Query: dictionary lookup

Generated

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

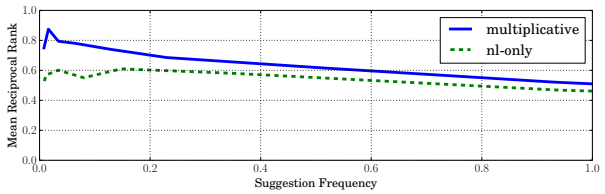
SO Query: comma delimited list trailing comma

Generated

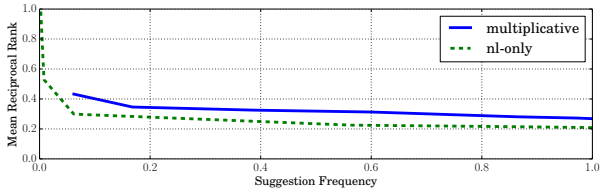
```
foreach (string Split in Join) {
  return string.Join(',',
    this string s => string.ToString(','));
}
```

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 mean cross-entropy of the retrieved snippets. It can be seen that as the confidence level increases (and thus the suggestion frequency decreases), the MRR also increases. For the query retrieval problem, the two quantities correlate with spearman correlation $\rho_{full} = 0.315$ for the multiplicative model and $\rho_{nl} = 0.268$ for the NL-only model. Similarly, for the snippet retrieval problem, $\rho_{full} = 0.218$ while $\rho_{nl} = 0.132$. All ρ are statistically significant ($p \ll 10^{-5}$) except from ρ_{nl} of the snippet retrieval task. The difference between the maximum and average cross-entropies seems to be a reasonable suggestion confidence metric that allows us



(a) $\mathcal{C} \rightarrow \mathcal{L}$



(b) $\mathcal{L} \rightarrow \mathcal{C}$

Figure 4. Suggestion Frequency vs MRR for the (a) query and (b) snippet retrieval tasks for perls-all. As the minimum confidence level increases (fewer suggestions are made), MRR increases.

to measure how much the retrieved modality can change the probability of the data under the model. If the confidence is high, the model has more evidence on how the retrieved modality is correlated with the base modality. Using the standard deviation of cross-entropies as a confidence metric returns similar but slightly inferior results. For example, for the query retrieval task, we can increase the MRR to 0.698 (from 0.517, a 35% improvement) by making suggestions on 16% of the snippets.

5. Related Work

In recent years, the use of probabilistic models for software engineering applications has grown. Hindle et al. (2012); Nguyen et al. (2013); Allamanis & Sutton (2013); Tu et al. (2014) have argued that even simple n-gram-based models can improve over traditional code autocompletion systems. Maddison & Tarlow (2014) built a more sophisticated generative model of source code that is closely related to the source code model used in this work. Other applications include extracting code idioms (Allamanis & Sutton, 2014), code migration (Karaivanov et al., 2014), inferring coding conventions (Allamanis et al., 2014) and type inference (Raychev et al., 2015). Movshovitz-Attias et al. (2013) use simple statistical models like n-grams for predicting comments given a piece of source code. It is worth noting that comments are often focused on "why?" rather than "what?" which makes the task rather different. Gulwani & Marron (2014) synthesize a restricted domain-specific language for spreadsheets given a natural language query, using translation-inspired algorithms. Searching source code is still an active area, but most work (Bajracharya et al., 2014; Keivanloo et al., 2014) has focused on retrieving snippets given code tokens, or by retrieving snippets of code via text available in some surrounding context.

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 Kushman & 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

Allamanis, Miltiadis and Sutton, Charles. Mining source code repositories at massive scale using language modeling. In *Working Conference on Mining Software Repositories (MSR)*, 2013.

Allamanis, Miltiadis and Sutton, Charles. Mining Idioms from Source Code. In *Symposium on the Foundations of Software Engineering (FSE)*, 2014.

Allamanis, Miltiadis, Barr, Earl T, Bird, Christian, and Sutton, Charles. Learning natural coding conventions. In *Symposium on the Foundations of Software Engineering (FSE)*, 2014.

Bajracharya, Sushil, Ossher, Joel, and Lopes, Cristina. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. *Science of Computer Programming*, 79:241–259, 2014.

Duchi, John, Hazan, Elad, and Singer, Yoram. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

Fang, Hao, Gupta, Saurabh, Iandola, Forrest, Srivastava, Rupesh, Deng, Li, Dollár, Piotr, Gao, Jianfeng, He, Xiaodong, Mitchell, Margaret, Platt, John, et al. From captions to visual concepts and back. *arXiv preprint arXiv:1411.4952*, 2014.

Gulwani, Sumit and Marron, Mark. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. SIGMOD, 2014.

Gutmann, Michael U and Hyvärinen, Aapo. Noise-contrastive estimation of unnormalized statistical models, with applications to natural image statistics. *The Journal of Machine Learning Research*, 13(1):307–361, 2012.

Hindle, Abram, Barr, Earl T, Su, Zhendong, Gabel, Mark, and Devanbu, Premkumar. On the naturalness of software. In *International Conference on Software Engineering (ICSE)*, 2012.

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

Karaivanov, Svetoslav, Raychev, Veselin, and Vechev, Martin. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pp. 173–184. ACM, 2014.

Karpathy, Andrej and Fei-Fei, Li. Deep visual-semantic alignments for generating image descriptions. *arXiv preprint arXiv:1412.2306*, 2014.

Keivanloo, Iman, Rilling, Juergen, and Zou, Ying. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*, pp. 664–675. ACM, 2014.

Kiros, Ryan, Zemel, R, and Salakhutdinov, Ruslan. Multimodal neural language models. In *Proc. NIPS Deep Learning Workshop*, 2013.

Kushman, Nate and Barzilay, Regina. Using semantic unification to generate regular expressions from natural language. North American Chapter of the Association for Computational Linguistics (NAACL), 2013.

Liang, Percy, Jordan, Michael I, and Klein, Dan. Learning dependency-based compositional semantics. *Computational Linguistics*, 39(2):389–446, 2013.

Maddison, Chris and Tarlow, Daniel. Structured generative models of natural source code. In *Proceedings of The 31st International Conference on Machine Learning*, pp. 649–657, 2014.

Memisevic, Roland and Hinton, Geoffrey. Unsupervised learning of image transformations. In *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*, pp. 1–8. IEEE, 2007.

Mitchell, Jeff and Lapata, Mirella. Composition in distributional models of semantics. *Cognitive Science*, 34(8): 1388–1439, 2010.

Mnih, Andriy and Kavukcuoglu, Koray. Learning word embeddings efficiently with noise-contrastive estimation. In *Advances in Neural Information Processing Systems*, pp. 2265–2273, 2013.

Mnih, Andriy and Teh, Yee Whye. A fast and simple algorithm for training neural probabilistic language models. In *ICML*, 2012.

Movshovitz-Attias, Dana, Cohen, WW, and W. Cohen, William. Natural Language Models for Predicting Programming Comments. In *Proc of the ACL*, 2013.

MSDN. URL https://MSDN.microsoft.com/en-us/library/bb397906.aspx.

.NET Compiler Platform. URL https://github.com/dotnet/roslyn.

Nguyen, Tung Thanh, Nguyen, Anh Tuan, Nguyen, Hoan Anh, and Nguyen, Tien N. A statistical semantic language model for source code. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2013.

Raychev, Veselin, Vechev, Martin, and Krause, Andreas. Predicting program properties from big code. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 111–124. ACM, 2015.

Socher, Richard, Karpathy, Andrej, Le, Quoc V, Manning, Christopher D, and Ng, Andrew Y. Grounded compositional semantics for finding and describing images with sentences. *Transactions of the Association for Computational Linguistics*, 2:207–218, 2014.

Srivastava, Nitish and Salakhutdinov, Ruslan R. Multimodal learning with deep boltzmann machines. In *Advances in neural information processing systems*, pp. 2222–2230, 2012.

Sutskever, Ilya, Vinyals, Oriol, and Le, Quoc VV. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pp. 3104–3112, 2014.

Taylor, Graham W and Hinton, Geoffrey E. Factored conditional restricted boltzmann machines for modeling motion style. In *Proceedings of the 26th annual international conference on machine learning*, pp. 1025–1032. ACM, 2009.

Tu, Zhaopeng, Su, Zhendong, and Devanbu, Premkumar. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 269–280. ACM, 2014.

Vinyals, Oriol, Toshev, Alexander, Bengio, Samy, and Erhan, Dumitru. Show and tell: A neural image caption generator. *arXiv preprint arXiv:1411.4555*, 2014.

Zelle, John M. and Mooney, Raymond J. Learning to parse database queries using inductive logic programming. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1996.

Zettlemoyer, Luke S and Collins, Michael. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In *Proceedings of Uncertainty in Artificial Intelligence (UAI)*, 2005.