Summarizing Source Code using a Neural Attention Model

Srinivasan Iyer **Ioannis Konstas** **Alvin Cheung** **Luke Zettlemoyer**
Computer Science & Engineering
University of Washington
Seattle, WA 98195
{sviyer,ikonstas,akcheung,lsz}@cs.washington.edu

Abstract

High quality source code is often paired with high level summaries of the computation it performs, for example in code documentation or in descriptions posted in online forums. Such summaries are extremely useful for applications such as code search but are expensive to manually author, hence only done for a small fraction of all code that is produced. In this paper, we present the first completely data-driven approach for generating high level summaries of source code. Our model, CODE-NN , uses Long Short Term Memory (LSTM) networks with attention to produce sentences that describe C# code snippets and SQL queries. CODE-NN is trained on a new corpus that is automatically collected from StackOverflow, which we release. Experiments demonstrate strong performance on two tasks: (1) code summarization, where we establish the first end-to-end learning results and outperform strong baselines, and (2) code retrieval, where our learned model improves the state of the art on a recently introduced C# benchmark by a large margin.

1 Introduction

Billions of lines of source code reside in online repositories (Dyer et al., 2013), and high quality code is often coupled with natural language (NL) in the form of instructions, comments, and documentation. Short summaries of the overall computation the code performs provide a particularly useful form of documentation for a range of applications, such as code search or tutorials. However, such summaries are expensive to manually author.

1. Source Code (C#):

```
public int TextWidth(string text) {
  TextBlock t = new TextBlock();
  t.Text = text;
  return
    (int)Math.Ceiling(t.ActualWidth);
}
```

Descriptions:
a. Get rendered width of string rounded up to the nearest integer
b. Compute the actual textwidth inside a textblock

2. Source Code (C#):

```
  var input = "Hello";
  var regEx = new Regex("World");
  return !regEx.IsMatch(input);
```

Descriptions:
a. Return if the input doesn't contain a particular word in it
b. Lookup a substring in a string using regex

3. Source Code (SQL):

```
SELECT Max(marks) FROM stud_records
  WHERE marks <
  (SELECT Max(marks) FROM stud_records);
```

Descriptions:
a. Get the second largest value of a column
b. Retrieve the next max record in a table

Figure 1: Code snippets in C# and SQL and their summaries in NL, from StackOverflow. Our goal is to automatically generate summaries from code snippets.

As a result, this laborious process is only done for a small fraction of all code that is produced.

In this paper, we present the first completely data-driven approach for generating short high-level summaries of source code snippets in natural language. We focus on C#, a general-purpose imperative language, and SQL, a declarative language for querying databases. Figure 1 shows example code snippets with descriptions that summarize the overall function of the code, with the goal to generate high level descriptions, such as

Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, pages 2073–2083,
Berlin, Germany, August 7-12, 2016.

lookup a substring in a string. Generating such a summary is often challenging because the text can include complex, non-local aspects of the code (e.g., consider the phrase 'second largest' in Example 3 in Figure 1). In addition to being directly useful for interpreting uncommented code, high-quality generation models can also be used for code retrieval, and in turn, for natural language programming by applying nearest neighbor techniques to a large corpus of automatically summarized code.

Natural language generation has traditionally been addressed as a pipeline of modules that decide 'what to say' (content selection) and 'how to say it' (realization) separately (Reiter and Dale, 2000; Wong and Mooney, 2007; Chen et al., 2010; Lu and Ng, 2011). Such approaches require supervision at each stage and do not scale well to large domains. We instead propose an end-to-end neural network called CODE-NN that jointly performs content selection using an attention mechanism, and surface realization using Long Short Term Memory (LSTM) networks. The system generates a summary one word at a time, guided by an attention mechanism over embeddings of the source code, and by context from previously generated words provided by a LSTM network (Hochreiter and Schmidhuber, 1997). The simplicity of the model allows it to be learned from the training data without the burden of feature engineering (Angeli et al., 2010) or the use of an expensive approximate decoding algorithm (Konstas and Lapata, 2013).

Our model is trained on a new dataset of code snippets with short descriptions, created using data gathered from Stackoverflow,[1] a popular programming help website. Since access is open and unrestricted, the content is inherently noisy (ungrammatical, non-parsable, lacking content), but as we will see, it still provides strong signal for learning. To reliably evaluate our model, we also collect a clean, human-annotated test set.[2]

We evaluate CODE-NN on two tasks: code summarization and code retrieval (Section 2). For summarization, we evaluate using automatic metrics such as METEOR and BLEU-4, together with a human study for naturalness and informativeness of the output. The results show that CODE-NN outperforms a number of strong baselines and, to the best of our knowledge, CODE-NN is the first approach that learns to generate summaries of source code from easily gathered online data. We further use CODE-NN for code retrieval for programming related questions on a recent C# benchmark, and results show that CODE-NN improves the state of the art (Allamanis et al. (2015b)) for mean reciprocal rank (MRR) by a wide margin.

[1] `http://stackoverflow.com`

[2] Data and code are available at `https://github.com/sriniiyer/codenn`.

2 Tasks

CODE-NN generates a NL summary of source code snippets (GEN task). We have also used CODE-NN on the inverse task to retrieve source code given a question in NL (RET task).

Formally, let U_C be the set of all code snippets and U_N be the set of all summaries in NL. For a training corpus with J code snippet and summary pairs $(c_j, n_j), 1 \leq j \leq J, c_j \in U_C, n_j \in U_N$, we define the following two tasks:

GEN For a given code snippet $c \in U_C$, the goal is to produce a NL sentence $n^* \in U_N$ that maximizes some scoring function $s \in (U_C \times U_N \rightarrow \mathbb{R})$:

$$n^* = \underset{n}{\operatorname{argmax}}\ s(c, n) \tag{1}$$

RET We also use the scoring function s to retrieve the highest scoring code snippet c_j^* from our training corpus, given a NL question $n \in U_N$:

$$c_j^* = \underset{c_j}{\operatorname{argmax}}\ s(c_j, n), 1 \leq j \leq J \tag{2}$$

In this work, s is computed using an LSTM neural attention model, to be described in Section 5.

3 Related Work

Although we focus on generating high-level summaries of source code snippets, there has been work on producing code descriptions at other levels of abstraction. Movshovitz-Attias and Cohen (2013) study the task of predicting class-level comments by learning n-gram and topic models from open source Java projects and testing it using a character-saving metric on existing comments. Allamanis et al. (2015a) create models for suggesting method and class names by embedding them in a high dimensional continuous space. Sridhara et al. (2010) present a pipeline that generates summaries of Java methods by selecting relevant content and generating phrases using templates to describe them. There is also work on improving program comprehension (Haiduc et al.,

2010), identifying cross-cutting source code concerns (Rastkar et al., 2011), and summarizing software bug reports (Rastkar et al., 2010). To the best of our knowledge, we are the first to use learning techniques to construct completely new sentences from arbitrary code snippets.

Source code summarization is also related to generation from formal meaning representations. Wong and Mooney (2007) present a system that learns to generate sentences from lambda calculus expressions by inverting a semantic parser. Mei et al. (2016), Konstas and Lapata (2013), and Angeli et al. (2010) create learning algorithms for text generation from database records, again assuming data that pairs sentences with formal meaning representations. In contrast, we present algorithms for learning from easily gathered web data.

In the database community, Simitsis and Ioannidis (2009) recognize the need for SQL database systems to talk back to users. Koutrika et al. (2010) built an interactive system (LOGOS) that translates SQL queries to text using NL templates and database schemas. Similarly there has been work on translating SPARQL queries to natural language using rules to create dependency trees for each section of the query, followed by a transformation step to make the output more natural (Ngonga Ngomo et al., 2013). These approaches are not learning based, and require significant manual template-engineering efforts.

We use recurrent neural networks (RNN) based on LSTMs and neural attention to jointly model source code and NL. Recently, RNN-based approaches have gained popularity for text generation and have been used in machine translation (Sutskever et al., 2011), image and video description (Karpathy and Li, 2015; Venugopalan et al., 2015; Devlin et al., 2015), sentence summarization (Rush et al., 2015), and Chinese poetry generation (Zhang and Lapata, 2014). Perhaps most closely related, Wen et al. (2015) generate text for spoken dialogue systems with a two-stage approach, comprising an LSTM decoder semantically conditioned on the logical representation of speech acts, and a reranker to generate the final output. In contrast, we design an end-to-end attention-based model for source code.

For code retrieval, Allamanis et al. (2015b) proposed a system that uses Stackoverflow data and web search logs to create models for retrieving C# code snippets given NL questions and vice versa. They construct distributional representations of code structure and language and combine them using additive and multiplicative models to score (code, language) pairs, an approach that could work well for retrieval but cannot be used for generation. We learn a neural generation model without using search logs and show that it can also be used to score code for retrieval, with much higher accuracy.

Synthesizing code from language is an alternative to code retrieval and has been studied in both the Systems and NLP research communities. Giordani and Moschitti (2012), Li and Jagadish (2014), and Gulwani and Marron (2014) synthesize source code from NL queries for database and spreadsheet applications. Similarly, Lei et al. (2013) interpret NL instructions to machine-executable code, and Kushman and Barzilay (2013) convert language to regular expressions. Unlike most synthesis methods, CODE-NN is domain agnostic, as we demonstrate its applications on both C# and SQL.

4 Dataset

We collected data from StackOverflow (SO), a popular website for posting programming-related questions. Anonymized versions of all the posts can be freely downloaded.[3] Each post can have multiple tags. Using the *C#* tag for C# and the *sql*, *database* and *oracle* tags for SQL, we were able to collect 934,464 and 977,623 posts respectively.[4] Each post comprises a short title, a detailed question, and one or more responses, of which one can be marked as accepted. We found that the text in the question and responses is domain-specific and verbose, mixed with details that are irrelevant for our tasks. Also, code snippets in responses that were not accepted were frequently incorrect or tangential to the question asked. Thus, we extracted only the title from the post and use the code snippet from those accepted answers that contain exactly one code snippet (using <code> tags). We add the resulting (title, query) pairs to our corpus, resulting in a total of 145,841 pairs for C# and 41,340 pairs for SQL.

Cleaning We train a semi-supervised classifier to filter titles like *'Difficult C# if then logic'* or *'How can I make this query easier to write?'* that bear no relation to the corresponding code snippet.

[3] http://archive.org/details/stackexchange

[4] The data was downloaded in Dec 2014.

To do so, we annotate 100 titles as being *clean* or *not clean* for each language and use them to bootstrap the algorithm. We then use the remaining titles in our training set as an unsupervised signal, and obtain a classification accuracy of over 73% on a manually labeled test set for both languages. For the final dataset, we retain 66,015 C# (title, query) pairs and 32,337 SQL pairs that are classified as clean, and use 80% of these datasets for training, 10% for validation and 10% for testing.

Parsing Given the informal nature of StackOverflow, the code snippets are approximate answers that are usually incomplete. For example, we observe that only 12% of the SQL queries parse without any syntactic errors (using zql[5]). We therefore aim to perform a best-effort parse of the code snippet, using modified versions of an ANTLR parser for C# (Parr, 2013) and *python-sqlparse* (Albrecht, 2015) for SQL. We strip out all comments and to avoid being context specific, we replace literals with tokens denoting their types. In addition, for SQL, we replace table and column names with numbered placeholder tokens while preserving any dependencies in the query. For example, the SQL query in Figure 1 is represented as `SELECT MAX(col0) FROM tab0 WHERE col0 < (SELECT MAX(col0) FROM tab0)`.

Data Statistics The structural complexity and size of the code snippets in our dataset makes our tasks challenging. More than 40% of our C# corpus comprises snippets with three or more statements and functions, and 20% contains loops and conditionals. Also, over a third of our SQL queries contain one or more subqueries and multiple tables, columns and functions (like `MIN`, `MAX`, `SUM`). On average, our C# snippets are 38 tokens long and the queries in our corpus are 46 tokens long, while titles are 9-12 words long. Table 2 shows the complete data statistics.

Human Annotation For the GEN task, we use n-gram based metrics (see Section 6.1.2) of the summary generated by our model with respect to the actual title in our corpus. Titles can be short, and a given code snippet can be described in many different ways with little overlapping content between them. For example, the descriptions for the second code snippet in Figure 1 share very few words with each other. To address these limitations, we extend our test set by asking human annotators to provide two additional titles for 200 snippets chosen at random from the test set, making a total of three reference titles for each code snippet. To collect this data, annotators were shown only the code snippets and were asked to write a short summary after looking at a few example summaries. They were also asked to "think of a question that they could ask on a programming help website, to get the code snippet as a response." This encouraged them to briefly describe the key feature that the code is trying to demonstrate. We use half of this test set for model tuning (DEV, see Section 5) and the rest for evaluation (EVAL).

[5] `http://zql.sourceforge.net`

C#	**# Statements**		**# Functions**	
	≥ 3	23,611 (44.7%)	≥ 3	26,541 (51.0%)
	≥ 4	17,822 (33.7%)	≥ 4	20,221 (38.2%)
	# Loops		**# Conditionals**	
	≥ 1	10,676 (20.0%)	≥ 1	11,819 (22.3%)
SQL	**# Subqueries**		**# Tables**	
	≥ 1	11,418 (35%)	≥ 3	14,695 (44%)
	≥ 2	3,625 (11%)	≥ 4	10,377 (31%)
	# Columns		**# Functions**	
	≥ 5	12,366 (37%)	≥ 3	6,290 (19%)
	≥ 6	9,050 (27%)	≥ 4	3,973 (12%)

Table 1: Statistics for code snippets in our dataset.

C#	**Avg. code length**	38 tokens	**# tokens**	91,156
	Avg. title length	12 words	**# words**	24,857
SQL	**Avg. query length**	46 tokens	**# tokens**	1,287
	Avg. title length	9 words	**# words**	10,086

Table 2: Average code and title lengths together with vocabulary sizes for C# and SQL after post-processing.

5 The CODE-NN Model

Description We present an end-to-end generation system that performs content selection and surface realization jointly. Our approach uses an attention-based neural network to model the conditional distribution of a NL summary n given a code snippet c. Specifically, we use an LSTM model that is guided by attention on the source code snippet to generate a summary one word at a time, as shown in Figure 2.[6]

Formally, we represent a NL summary $n = n_1, \ldots, n_l$ as a sequence of 1-hot vectors

[6] We experimented with other sequence (Sutskever et al., 2014) and tree based architectures (Tai et al., 2015) as well. None of these models significantly improved performance, however, this is an important area for future work.

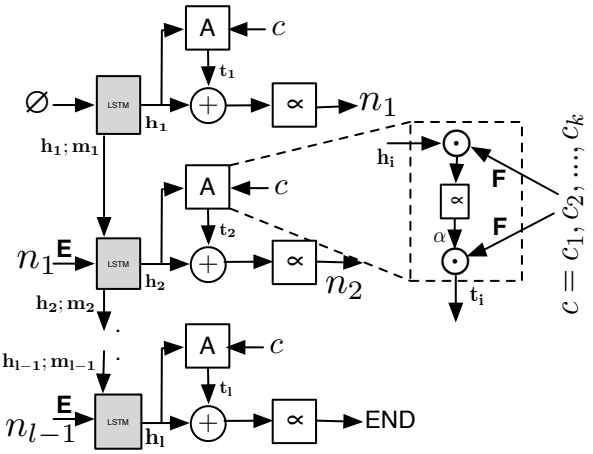

Figure 2: Generation of a title $n = n_1, \ldots,$ END given code snippet $c_1, ..., c_k$. The attention cell computes a distributional representation t_i of the code snippet based on the current LSTM hidden state h_i. A combination of t_i and h_i is used to generate the next word, n_i, which feeds back into the next LSTM cell. This is repeated until a fixed number of words or END is generated. $\propto$ blocks denote softmax operations.

$\mathbf{n_1}, \ldots, \mathbf{n_l} \in \{0,1\}^{|N|}$, where N is the vocabulary of the summaries. Our model computes the probability of n (scoring function s in Eq. 1) as a product of the conditional next-word probabilities

$$s(c, n) = \prod_{i=1}^{l} p(n_i | n_1, \ldots, n_{i-1})$$

with,

$$p(n_i | n_1, \ldots, n_{i-1}) \propto \mathbf{W} \tanh(\mathbf{W_1 h_i} + \mathbf{W_2 t_i})$$

where, $\mathbf{W} \in \mathbb{R}^{|N| \times H}$ and $\mathbf{W_1}, \mathbf{W_2} \in \mathbb{R}^{H \times H}$, H being the embedding dimensionality of the summaries. $\mathbf{t_i}$ is the contribution from the attention model on the source code (see below). $\mathbf{h_i}$ represents the hidden state of the LSTM cell at the current time step and is computed based on the previously generated word, the previous LSTM cell state $\mathbf{m_{i-1}}$ and the previous LSTM hidden state $\mathbf{h_{i-1}}$ as

$$\mathbf{m_i}; \mathbf{h_i} = f(\mathbf{n_{i-1}E}, \mathbf{m_{i-1}}, \mathbf{h_{i-1}}; \theta)$$

where $\mathbf{E} \in \mathbb{R}^{|N| \times H}$ is a word embedding matrix for the summaries. We compute f using the LSTM cell architecture used by Zaremba et al. (2014).

Attention The generation of each word is guided by a global attention model (Luong et al., 2015), which computes a weighted sum of the embeddings of the code snippet tokens based on the current LSTM state (see right part in Figure 2). Formally, we represent c as a set of 1-hot vectors $\mathbf{c_1}, \ldots, \mathbf{c_k} \in \{0,1\}^{|C|}$ for each source code token; C is the vocabulary of all tokens in our code snippets. Our attention model computes,

$$\mathbf{t_i} = \sum_{j=1}^{k} \alpha_{i,j} \cdot \mathbf{c_j F}$$

where $\mathbf{F} \in \mathbb{R}^{|C| \times H}$ is a token embedding matrix and each $\alpha_{i,j}$ is proportional to the dot product between the current internal LSTM hidden state $\mathbf{h_i}$ and the corresponding token embedding $\mathbf{c_j}$:

$$\alpha_{i,j} = \frac{\exp(\mathbf{h_i}^{\mathrm{T}} \mathbf{c_j F})}{\sum_{j=1}^{k} \exp(\mathbf{h_i}^{\mathrm{T}} \mathbf{c_j F})}$$

Training We perform supervised end-to-end training using backpropagation (Werbos, 1990) to learn the parameters of the embedding matrices $\mathbf{F}$ and $\mathbf{E}$, transformation matrices $\mathbf{W}$, $\mathbf{W_1}$ and $\mathbf{W_2}$, and parameters θ of the LSTM cell that computes f. We use multiple epochs of minibatch stochastic gradient descent and update all parameters to minimize the negative log likelihood (NLL) of our training set. To prevent over-fitting we make use of dropout layers (Srivastava et al., 2014) at the summary embeddings and the output softmax layer. Using pre-trained embeddings (Mikolov et al., (2013)) for the summary embedding matrix or adding additional LSTM layers did not improve performance for the GEN task. Since the NLL training objective does not directly optimize for our evaluation metric (METEOR), we compute METEOR (see Section 6.1.2) on a small development set (DEV) after every epoch and save the intermediate model that gives the maximum score, as the final model.

Decoding Given a trained model and an input code snippet c, finding the most optimal title entails generating the title n^* that maximizes $s(c, n)$ (see Eq. 1). We approximate n^* by performing beam search on the space of all possible summaries using the model output.

Implementation Details We add special START and END tokens to our training sequences and replace all tokens and output words occurring

with a frequency of less than 3 with an UNK token, making $|C| = 31,667$ and $|N| = 7,470$ for C# and $|C| = 747$ and $|N| = 2,506$ for SQL. Our hyper-parameters are set based on performance on the validation set. We use a minibatch size of 100 and set the dimensionality of the LSTM hidden states, token embeddings, and summary embeddings (H) to 400. We initialize all model parameters uniformly between -0.35 and 0.35. We start with a learning rate of 0.5 and start decaying it by a factor of 0.8 after 60 epochs if accuracy on the validation set goes down, and terminate training when the learning rate goes below 0.001. We cap the parameter gradients to 5 and use a dropout rate of 0.5.

We use the Torch framework[7] to train our models on GPUs. Training runs for about 80 epochs and takes approximately 7 hours. We compute METEOR score at every epoch on the development set (DEV) to choose the best final model, with the best results obtained between 60 and 70 epochs. For decoding, we set the beam size to 10, and the maximum summary length to 20 words.

6 Experimental Setup

6.1 GEN Task

6.1.1 Baselines

For the GEN task, we compare CODE-NN with a number of competitive systems, none of which had been previously applied to generate text from source code, and hence we adapt them slightly for this task, as explained below.

IR is an information retrieval baseline that outputs the title associated with the code c_j in the training set that is closest to the input code c in terms of token Levenshtein distance. In this case s from Eq.1 becomes,

$$s(c, n_j) = -1 \times \text{lev}(c_j, c), 1 \leq j \leq J$$

MOSES (Koehn et al., 2007) is a popular phrase-based machine translation system. We perform generation by treating the tokenized code snippet as the source language, and the title as the target. We train a 3-gram language model using KenLM (Heafield, 2011) to use with MOSES, and perform MIRA-based tuning (Cherry and Foster, 2012) of hyper-parameters using DEV.

SUM-NN is the neural attention-based abstractive summarization model of Rush et al. (2015). It uses an encoder-decoder architecture with an attention mechanism based on a fixed context window of previously generated words. The decoder is a feed-forward neural language model that generates the next word based on previous words in a context window of size k. In contrast, we decode using an LSTM network that can model long range dependencies and our attention weights are tied to the LSTM hidden states. We set the embedding and hidden state dimensions and context window size by tuning on our validation set. We found this model to generate overly short titles like *'sql server 2008'* when a length restriction was not imposed on the output text. Therefore, we fix the output length to be the average title length in the training set while decoding.

[7] http://torch.ch

6.1.2 Evaluation Metrics

We evaluate the GEN task using automatic metrics, and also perform a human study.

Automatic Evaluation We report METEOR (Banerjee and Lavie, 2005) and sentence level BLEU-4 (Papineni et al., 2002) scores. METEOR is recall-oriented and measures how well our model captures content from the references in our output. BLEU-4 measures the average n-gram precision on a set of reference sentences, with a penalty for overly short sentences. Since the generated summaries are short and there are multiple alternate summaries for a given code snippet, higher order n-grams may not overlap. We remedy this problem by using +1 smoothing (Lin and Och, 2004). We compute these metrics on the tuning set DEV and the held-out evaluation set EVAL.

Human Evaluation Since automatic metrics do not always agree with the actual quality of the results (Stent et al., 2005), we perform human evaluation studies to measure the output of our system and baselines across two modalities, namely naturalness and informativeness. For the former, we asked 5 native English speakers to rate each title against grammaticality and fluency, on a scale between 1 and 5. For *informativeness* (i.e., the amount of content carried over from the input code to the NL summary, ignoring fluency of the text), we asked 5 human evaluators familiar with C# and SQL to evaluate the system output by rating the factual overlap of the summary with the reference titles, on a scale between 1 and 5.

6.2 RET task

6.2.1 Model and Baselines

CODE-NN As described in Section 2, for a given NL question n in the RET task, we rank all code snippets c_j in our corpus by computing the scoring function $s(c_j, n)$, and return the query c_j^* that maximizes it (Eq. 2).

RET-IR is an information retrieval baseline that ranks the candidate code snippets using cosine similarity between the given NL question n and all summaries n_j in the retrieval set, based on their vector representations using TF-IDF weights over unigrams. The scoring function s in Eq. 2 becomes:

$$s(c_j, n) = \frac{\text{tf-idf}(n_j) \cdot \text{tf-idf}(n)}{\|\text{tf-idf}(n_j)\|\|\text{tf-idf}(n)\|}, 1 \leq j \leq J$$

6.2.2 Evaluation Metrics

We assess ranking quality by computing the Mean Reciprocal Rank (MRR) of c_j^*. For every snippet c_j in EVAL (and DEV), we use two of the three references (title and human annotation), namely $n_{j,1}, n_{j,2}$. We then build a retrieval set comprising $(c_j, n_{j,1})$ together with 49 random distractor pairs $(c', n'), c' \neq c_j$ from the test set. Using $n_{j,2}$ as the natural language question, we rank all 50 items in this retrieval set and use the rank of query c_j^* to compute MRR. We average MRR over all returned queries c_j^* in the test set, and repeat this experiment for several different random sets of distractors.

6.3 Tasks from Allamanis et al. (2015b)

Allamanis et al. (2015b) take a retrieval approach to answer C# related natural language questions (L to C), similar to our RET task. In addition, they also use retrieval to summarize C# source code (C to L) and evaluate both tasks using the MRR metric. Although they also use data from Stackoverflow, their dataset preparation and cleaning methods differs significantly from ours. For example, they filter out posts where the question has fewer than 2 votes, the answer has fewer than 3 votes, or the post has fewer than 1000 views. Additionally, they also filter code snippets that cannot be parsed by Roslyn (.NET compiler) or are longer than 300 characters. Thus, to directly compare with their model, we re-train our generation model on their dataset and use our model score for retrieval of both code and summaries.

	Model	METEOR	BLEU-4
C#	IR	7.9 (6.1)	13.7 (12.6)
	MOSES	9.1 (9.7)	11.6 (11.5)
	SUM-NN	10.6 (10.3)	19.3 (18.2)
	CODE-NN	**12.3 (13.4)**	**20.5 (20.4)**
SQL	IR	6.3 (8.0)	13.5 (13.0)
	MOSES	8.3 (9.7)	15.4 (15.9)
	SUM-NN	6.4 (8.7)	13.3 (14.2)
	CODE-NN	**10.9 (14.0)**	**18.4 (17.0)**

Table 3: Performance on EVAL for the GEN task. Performance on DEV is indicated in parentheses.

	Model	Naturalness	Informativeness
C#	IR	3.42	2.25
	MOSES	1.41	2.42
	SUM-NN	**4.61***	1.99
	CODE-NN	**4.48**	**2.83**
SQL	IR	3.21	2.58
	MOSES	2.80	2.54
	SUM-NN	4.44	2.75
	CODE-NN	**4.54**	**3.12**

Table 4: Naturalness and Informativeness measures of model outputs. Stat. sig. between CODE-NN and others is computed with a 2-tailed Student's t-test; $p < 0.05$ except for *.

7 Results

7.1 GEN Task

Table 3 shows automatic evaluation metrics for our model and baselines. CODE-NN outperforms all the other methods in terms of METEOR and BLEU-4 score. We attribute this to its ability to perform better content selection, focusing on the more salient parts of the code by using its attention mechanism jointly with its LSTM memory cells. The neural models have better performance on C# than SQL. This is in part because, unlike SQL, C# code contains informative intermediate variable names that are directly related to the objective of the code. On the other hand, SQL is more challenging in that it only has a handful of keywords and functions, and summarization models need to rely on other structural aspects of the code.

Informativeness and naturalness scores for each model from our human evaluation study are presented in Table 4. In general, CODE-NN performs well across both dimensions. Its superior performance in terms of informativeness further supports our claim that it manages to select content more effectively. Although SUM-NN performs similar to CODE-NN on naturalness, its output lacks content and has very little variation (see Section 7.4), which also explains its surprisingly low

	Model	MRR
C#	RET-IR	0.42 ± 0.02 (0.44 ± 0.01)
	CODE-NN	$\mathbf{0.58 \pm 0.01}$ $(\mathbf{0.66 \pm 0.02})$
SQL	RET-IR	0.28 ± 0.01 (0.4 ± 0.01)
	CODE-NN	$\mathbf{0.44 \pm 0.01}$ $(\mathbf{0.54 \pm 0.02})$

Table 5: MRR for the RET task. Dev set results in parentheses.

	Model	MRR
L to C	Allamanis	0.182 ± 0.009
	CODE-NN	$\mathbf{0.590 \pm 0.044}$
C to L	Allamanis	0.434 ± 0.003
	CODE-NN	$\mathbf{0.461 \pm 0.046}$

Table 6: MRR values for the Language to Code (L to C) and the Code to Language (C to L) tasks using the C# dataset of Allamanis et al. (2015b)

score on informativeness.

7.2 RET Task

Table 5 shows the MRR on the RET task for CODE-NN and RET-IR, averaged over 20 runs for C# and SQL. CODE-NN outperforms the baseline by about 16% for C# and SQL. RET-IR can only output code snippets that are annotated with NL as potential matches. On the other hand, CODE-NN can rank even unannotated code snippets and nominate them as potential candidates. Hence, it can leverage vast amounts of such code available in online repositories like Github. To speed up retrieval when using CODE-NN , it could be one of the later stages in a multi-stage retrieval system and candidates may also be ranked in parallel.

7.3 Comparison with Allamanis et al.

We train CODE-NN on their dataset and evaluate using the same MRR testing framework (see Table 6). Our model performs significantly better for the Language to Code task (L to C) and slightly better for the Code to Language task (C to L). The attention mechanism together with the LSTM network is able to generate better scores for (language, code) pairs.

7.4 Qualitative Analysis

Figure 3 shows the relative magnitudes of the attention weights ($\alpha_{i,j}$) for example C# and SQL code snippets while generating their corresponding summaries. Darker regions represent stronger weights. CODE-NN automatically learns to do high-quality content selection by aligning key summary words with informative tokens in the code snippet.

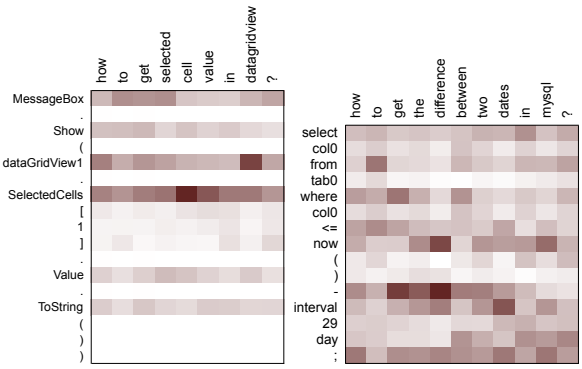

Figure 3: Heatmap of attention weights $\alpha_{i,j}$ for example C# (left) and SQL (right) code snippets. The model learns to align key summary words (like cell) with the corresponding tokens in the input (SelectedCells).

Table 8 shows examples of the output generated by our model and baselines for code snippets in DEV. Most of the models produce meaningful output for simple code snippets (first example) but degrade on longer, compositional inputs. For example, the last SQL query listed in Table 8 includes a subquery, where a complete description should include both summing and concatenation. CODE-NN describes the summation (but not concatenation), while others return non-relevant descriptions.

Finally, we performed manual error analysis on 50 randomly selected examples from DEV (Table 7) for each language. Redundancy is a major source of error, i.e., generation of extraneous content-bearing phrases, along with missing content, e.g., in the last example of Table 8 there is no reference to the concatenation operations present in the beginning of the query. Sometimes the output from our model can be out of context, in the sense that it does not match the input code. This often happens for low frequency tokens (7% of cases), for which CODE-NN realizes them with generic phrases. This also happens when there are very long range dependencies or compositional structures in the input, such as nested queries (13% of the cases).

8 Conclusion

In this paper, we presented CODE-NN , an end-to-end neural attention model using LSTMs to

Error	% Cases
Correct	37%
Redundancy	17%
Missing Info	26%
Out of context	20%

Table 7: Error analysis on 50 examples in DEV

generate summaries of C# and SQL code by learning from noisy online programming websites. Our model outperforms competitive baselines and achieves state of the art performance on automatic metrics, namely METEOR and BLEU, as well as on a human evaluation study. We also used CODE-NN to answer programming questions by retrieving the most appropriate code snippets from a corpus, and beat previous baselines for this task in terms of MRR. We have published our C# and SQL datasets, the accompanying human annotated test sets, and our code for the tasks described in this paper.

In future work, we plan to develop better models for capturing the structure of the input, as well as extend the use of our system to other applications such as automatic documentation of source code.

Acknowledgements

We thank Mike Lewis, Chloé Kiddon, Kenton Lee, Eunsol Choi and the anonymous reviewers for comments on an earlier version. We also thank Bill Howe, Dan Halperin and Mark Yatskar for helpful discussions and Miltiadis Allamanis for providing the dataset for the comparison study. This research was supported in part by the NSF (IIS-1252835), an Allen Distinguished Investigator Award, and a gift from Amazon.

References

Andi Albrecht. 2015. python-sqlparse.

Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. 2015a. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49.

Miltiadis Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015b. Bimodal modelling of source code and natural language. In *Proceedings of The 32nd International Conference on Machine Learning*, pages 2123–2132.

Gabor Angeli, Percy Liang, and Dan Klein. 2010. A simple domain-independent probabilistic approach to generation. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 502–512.

Method	Output

C# code	`var x = "FundList[10].Amount";`
	`int xIndex = Convert.ToInt32(`
	`  Regex.Match(x,@"\d+").Value);`

Gold	Identify the number in given string
IR	Convert string number to integer
MOSES	How to xIndex numbers in C#?
SUM-NN	How can I get the value of a string?
CODE-NN	How to convert string to int?

C# code	`foreach (string pTxt in xml.parent) {`
	`  TreeNode parent = new TreeNode();`
	`  foreach (string cTxt in xml.child) {`
	`  TreeNode child = new TreeNode();`
	`  parent.Nodes.Add(child);`
	`  }`
	`}`

Gold	Adding childs to a treenode dynamically in C#
IR	How to set the name of a tabPage programmatically
MOSES	How can TreeView nodes from XML parentText string to a treeview node
SUM-NN	How to get data from xml file in C#
CODE-NN	How to get all child nodes in TreeView?

C# code	`string url = baseUrl +`
	`  "/api/Entry/SendEmail?emailId="`
	`  + emailId;`
	`WebRequest req =`
	`  WebRequest.Create(url);`
	`req.Method = "GET";`
	`req.BeginGetResponse(null, null);`

Gold	Execute a get request on a web server and receive the response asynchronously
IR	How to download a file from another Sharepoint Domain
MOSES	How baseUrl emailId C how to a page in BeginGetResponse to
SUM-NN	How to get data from a file in C
CODE-NN	How to call a URL from a web api post ?
SQL Query	`SELECT * FROM table` `ORDER BY Rand() LIMIT 10`
Gold	Select random rows from mysql table
IR	How to select a random record from a mysql database?
MOSES	How to select all records in mysql ?
SUM-NN	How can I select random rows from a table
CODE-NN	How to get random rows from a mysql database?

SQL Query	`SELECT Group_concat(`
	`  Concat_ws(',', playerid, r1, r2)`
	`  SEPARATOR ';')`
	`FROM (SELECT playerid,`
	`  Sum(rank = 1) r1, Sum(rank < 5) r2`
	`  FROM result GROUP BY playerid) t;`

Gold	Get sum of group values based on condition and concatenate them into a string
IR	Mysql: counting occurences in a table, return as a single row
MOSES	Mysql query to get this result of the result of one column value in mysql
SUM-NN	How do i combine these two queries into one?
CODE-NN	How to get the sum of a column in a single query?

Table 8: Examples of outputs generated by each model for code snippets in DEV

Satanjeev Banerjee and Alon Lavie. 2005. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, volume 29, pages 65–72.

David L Chen, Joohyun Kim, and Raymond J Mooney. 2010. Training a multilingual sportscaster: Using perceptual context to learn language. *Journal of Artificial Intelligence Research*, pages 397–435.

Colin Cherry and George Foster. 2012. Batch tuning strategies for statistical machine translation. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 427–436.

Jacob Devlin, Hao Cheng, Hao Fang, Saurabh Gupta, Li Deng, Xiaodong He, Geoffrey Zweig, and Margaret Mitchell. 2015. Language models for image captioning: The quirks and what works. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 100–105.

Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. 2013. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 422–431.

Alessandra Giordani and Alessandro Moschitti. 2012. Translating questions to SQL queries with generative parsers discriminatively reranked. In *Proceedings of COLING 2012: Posters*, pages 401–410.

Sumit Gulwani and Mark Marron. 2014. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 803–814.

Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 223–226.

Kenneth Heafield. 2011. Kenlm: Faster and smaller language model queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, pages 187–197.

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

Andrej Karpathy and Fei-Fei Li. 2015. Deep visual-semantic alignments for generating image descriptions. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015*, pages 3128–3137.

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, et al. 2007. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th annual meeting of the ACL on interactive poster and demonstration sessions*, pages 177–180.

Ioannis Konstas and Mirella Lapata. 2013. A global model for concept-to-text generation. *Journal of Artificial Intelligence Research*, 48(1):305–346.

Georgia Koutrika, Alkis Simitsis, and Yannis E Ioannidis. 2010. Explaining structured queries in natural language. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 333–344.

Nate Kushman and Regina Barzilay. 2013. Using semantic unification to generate regular expressions from natural language. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 826–836.

Tao Lei, Fan Long, Regina Barzilay, and Martin Rinard. 2013. From natural language specifications to program input parsers. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1294–1303.

Fei Li and Hosagrahar V Jagadish. 2014. Nalir: An interactive natural language interface for querying relational databases. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 709–712.

Chin-Yew Lin and Franz Josef Och. 2004. Orange: a method for evaluating automatic evaluation metrics for machine translation. In *Proceedings of the 20th international conference on Computational Linguistics*, page 501.

Wei Lu and Hwee Tou Ng. 2011. A probabilistic forest-to-string model for language generation from typed lambda calculus expressions. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 1611–1622.

Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421.

Hongyuan Mei, Mohit Bansal, and Matthew R. Walter. 2016. What to talk about and how? selective generation using lstms with coarse-to-fine alignment. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. In *Proceedings of the International Conference on Learning Representations*.

Dana Movshovitz-Attias and William W. Cohen. 2013. Natural language models for predicting programming comments. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, pages 35–40.

Axel-Cyrille Ngonga Ngomo, Lorenz Bühmann, Christina Unger, Jens Lehmann, and Daniel Gerber. 2013. Sorry, i don't speak sparql: Translating sparql queries into natural language. In *Proceedings of the 22Nd International Conference on World Wide Web*, pages 977–988.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318.

Terence Parr. 2013. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf.

Sarah Rastkar, Gail C Murphy, and Gabriel Murray. 2010. Summarizing software artifacts: a case study of bug reports. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 505–514.

Sarah Rastkar, Gail C Murphy, and Alexander WJ Bradley. 2011. Generating natural language summaries for crosscutting source code concerns. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 103–112.

Ehud Reiter and Robert Dale. 2000. *Building natural language generation systems*. Cambridge University Press, New York, NY.

Alexander M. Rush, Sumit Chopra, and Jason Weston. 2015. A neural attention model for abstractive sentence summarization. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 379–389.

Alkis Simitsis and Yannis E. Ioannidis. 2009. Dbmss should talk back too. In *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Online Proceedings*.

Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 43–52.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958.

Amanda Stent, Matthew Marge, and Mohit Singhai. 2005. Evaluating evaluation methods for generation in the presence of variation. In *Computational Linguistics and Intelligent Text Processing*, pages 341–351.

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

Ilya Sutskever, Oriol Vinyals, and Quoc VV Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015*, pages 1556–1566.

Subhashini Venugopalan, Huijuan Xu, Jeff Donahue, Marcus Rohrbach, Raymond J. Mooney, and Kate Saenko. 2015. Translating videos to natural language using deep recurrent neural networks. In *In Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 1494–1504.

Tsung-Hsien Wen, Milica Gasic, Nikola Mrkšić, Pei-Hao Su, David Vandyke, and Steve Young. 2015. Semantically conditioned lstm-based natural language generation for spoken dialogue systems. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1711–1721.

Paul J Werbos. 1990. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560.

Yuk Wah Wong and Raymond J Mooney. 2007. Generation by inverting a semantic parser that uses statistical machine translation. In *In Proceedings of the 2007 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 172–179.

Wojciech Zaremba and Ilya Sutskever. 2014. Learning to execute. *CoRR*, abs/1410.4615.

Xingxing Zhang and Mirella Lapata. 2014. Chinese poetry generation with recurrent neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 670–680.