A CNN for Language-Agnostic Source Code Summarization

Jessica Moore
jessica.moore@twosixlabs.com

Ben Gelman
ben.gelman@twosixlabs.com

David Slater
david.slater@twosixlabs.com

This work was sponsored by the Air Force Research Laboratory.

Introduction

Software engineers spend more than half of their productive time on program comprehension tasks, such as reading documentation or trying to understand a colleague's code. Comments enable programmers to understand code more rapidly, prevent them from duplicating existing functionality, and aid them in fixing (or preventing) bugs. Unfortunately, many codebases suffer from a lack of thorough documentation. In order to ameliorate this problem, **we propose a novel technique for language-agnostic, automatic summarization of source code**.

```
def fib(n):
    if n <= 1:
        return n
    else:
        return(fib(n-1) +
                    fib(n-2))
```

"Recursively find n-th Fibonacci number"

Prior Work

Prior methods for generating natural language summaries of source code involved leveraging intermediate representations, template-based approaches, and information retrieval techniques.

Recently, authors have applied deep-learning models reminiscent of those used to translate between two natural languages. Two recent papers define the state-of-the-art for source code summarization:

- **Iyer et al., 2016 (CODE-NN)** – Code is tokenized and fed into CODE-NN, an LSTM-based neural network with attention.
- **Hu et al., 2018 (DeepCom)** – Code is converted into abstract syntax trees. A novel traversal method is applied to generate sequences, which are used as input to an LSTM-based encoder/decoder model.

Both works take a **language-specific** approach and use **"closed" vocabularies** (i.e., all words that can be predicted are known in advance).

Data Sources

We evaluate our model using two primary data sources:

1. **Hu et al., 2018** – Previous authors collected a large parallel corpus of Java methods and comments from 9,714 GitHub projects.

Language	Examples
Java	558,108

2. **Multilanguage Corpus** – We created a database of unique code/comment pairs from a large number of open-source projects.

Language	Examples
C++	910,169
Java	12,314,470
Python	1,166,922

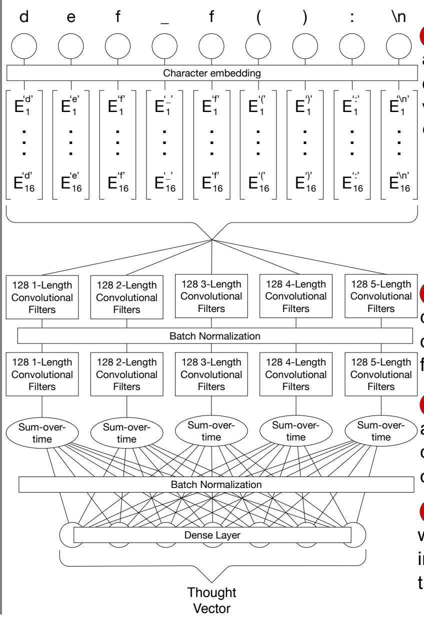
Methodology

Hu et al., 2018 – We use the train / validate / test splits provided by the authors.

Multilanguage Corpus – We use the first sentence in the description of a piece of source code as the comment. We exclude code/comment pairs that do not appear to be informative, based on either their content or their length.

We hold out 50,000 examples from each language as a test set. The remaining examples are split 80%-20% into training and validation sets. During training and validation, batches contains only one language, where languages are chosen with equal probability.

Convolutional Encoder


d e f _ f ( ) : \n
Character embedding
$E_1^{'d'}$ … $E_{16}^{'d'}$
128 1-Length Convolutional Filters
128 2-Length Convolutional Filters
128 3-Length Convolutional Filters
128 4-Length Convolutional Filters
128 5-Length Convolutional Filters
Batch Normalization
Sum-over-time
Batch Normalization
Dense Layer
Thought Vector

1 The characters from a given code snippet are converted to real-valued vectors using a *character embedding.*

2 Stacked convolutional filters learn different-length features from the code.

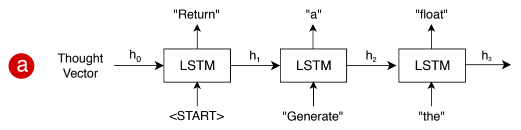
3 Summary statistics are derived from convolutional filter outputs.

4 Using a dense layer, we obtain the model's internal representation of the code.

LSTM Decoder

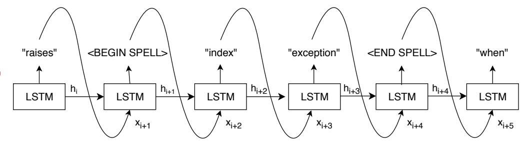
Our LSTM decoder translates the "thought vector" from the encoder into natural language. The initial hidden state is set equal to the thought vector. At the first time step, the LSTM receives a "START" token as input. For subsequent time steps, during training (a) we use "teacher forcing," in which the LSTM receives the true prior word as input; during testing (b), we instead provide the LSTM with the predicted prior word.

a



Unlike previous machine translation models, our model can predict any word by leveraging a vocabulary consisting of word, subwords, and single characters. At each time step, the model predicts one element from the vocabulary. Sequential predictions can be combined to form a single word, using special <BEGIN SPELL> and <END SPELL> tokens.

b



To create the model's vocabulary, we generate a set of all comments in the training data. We then split compound words (e.g., "IndexException" and "guiFrame") into their components, and split words into their roots and suffixes. The final vocabulary is made up of the most common word components, as well as all letters and punctuation marks.

Results

We use the bilingual evaluation understudy (BLEU) score to compare our predicted summaries to true source code comments. We use BLEU-4, meaning the BLEU score includes 4-grams and below. Consistent with Hu et al., we do not use smoothing to resolve the lack of higher order n-gram overlap. Mathematically,

$$BLEU = B * e^{\sum_{n=1}^{4} w_n * \log(p_n)}$$

$$B = \begin{cases} 1 & \text{if } c > r \\ e^{(1-r/c)} & \text{if } c \leq r \end{cases}$$

where p_n is the proportion of n-grams in the prediction that are in the true comment, w_n is the weight associated with those n-grams, c is the length of the prediction, and r is the length of the true comment.

We present BLEU scores and examples for each data source.

1. **Hu et al., 2018**

Language	Our Results (BLEU)	Hu et al. (BLEU)	Words per Comment	Comment Entropy (Bits)
Java	38.63	38.17	11.61	104.92

Actual Comment	Predicted Comment
serialize an object to an outputstream.	writes the given objects to the output stream.
warning log	prints a message at verbose priority.

2. **Multilanguage Corpus**

Language	Our Results (BLEU)	Words per Comment	Comment Entropy (Bits)
C++	40.86	11.22	103.57
Java	42.34	10.62	97.35
Python	26.53	24.36	224.78

Actual Comment	Predicted Comment
return the real table name of the object.	return the name of the object.
receive a message from the IO components in the channel stack	called when a message is received.
a function to unload the config object.	the config file.

Conclusions \ Future Work

We present a novel encoder/decoder model capable of summarizing arbitrary source code. We demonstrate results comparable to the state-of-the-art for a single-language (Java), while avoiding the cumbersome parsing required by previous models. We also present the first results on a corpus containing multiple programming languages and provide the first baselines for summarization of C++ and Python code.

Possible extensions of this work include:

- Summarization at the class, document, or project level
- Incorporation of a "copying" mechanism, to enable direct references to proper names (e.g., variable and class names)
- Comparison to a closed-vocabulary model with the same encoder, on the basis of compute time and accuracy

There are a number of opportunities to integrate automatic source code summarization into the software development lifecycle, such as incorporating it into IDEs and version control systems.