Automatically Generating Comments for Arbitrary Source Code

01.02.2019 by Jessica Moore

```
public static void main(String[] args) throws Exception {
  LOGGER.info("starting {}",Main.class.getSimpleName());
  TimeZone.setDefault(getTimeZone("UTC"));
  DateTimeZone.setDefault(UTC);
  ServiceManager serviceManager=new ServiceManager().addModules(
         ApiModule.class,MetricsModule.class).addServices(getEnabledWebServices()).addService
         getEnabledMetricsServices()).addServices(EventLogService.class);
  serviceManager.start();
  LOGGER.info("started {}",Main.class.getSimpleName());
}
```

starts the services using the provided input.

application entry point.

One of these comments was generated by a computer. Can you tell which one?

Comments — or Lack Thereof

Commenting code is boring. It's tedious and time-consuming and almost universally disliked. Why should we spend our time writing comments when we could be doing something so much more interesting? Like, for example, designing a model to write our comments for us.

At this point, it's a well-established fact that programmers regularly fail to comment their code or do so in such a way that the comments are effectively indecipherable. But, beyond being a courtesy to other developers (and oneself, 6 months later), effective comments play a crucial role in the software engineering process. They drive down development time, enable more effective bug detection, and facilitate better code reuse. As much as we would like to avoid writing them, source code comments are incredibly important.

But, like all good programmers, we know that the best way to solve any specific problem is to first solve a more general problem. So, instead of accepting the inevitability of having to comment our own code, we set out to design an algorithm that would take the code we've written and generate appropriate comments. And we think we're on to something. The first comment in the example above was generated by our system. The second was written by a human. Did you choose correctly?

Data — the Main Ingredient

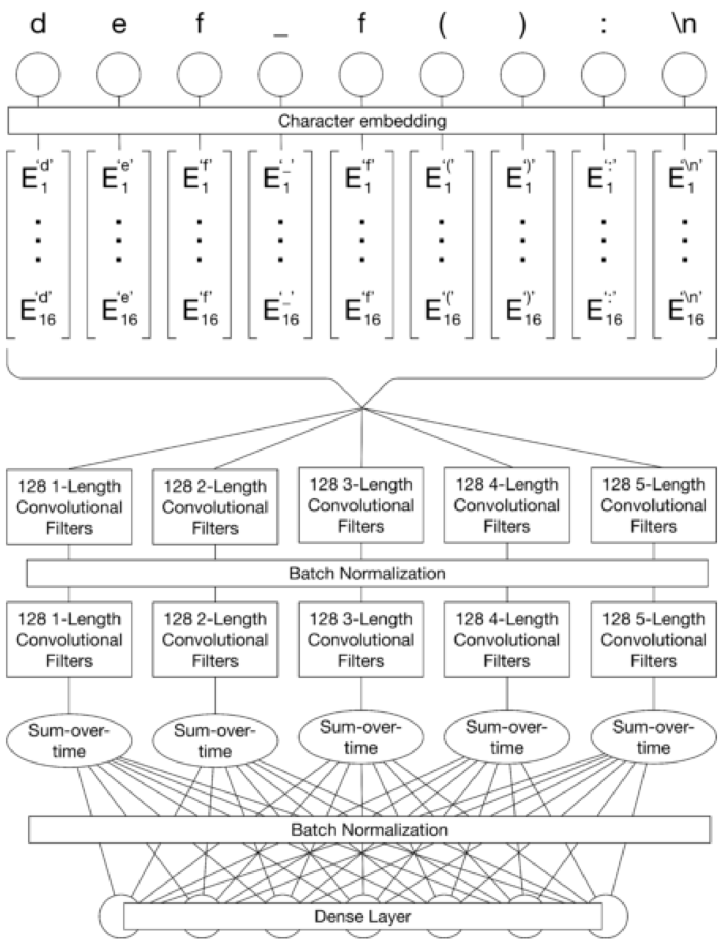
Unfortunately, in order to train a model to write source code comments, you need both source code and comments. Fortunately, there are a number of developers that have been so kind as to provide us with their code and the associated comments in the form of open-source repositories. We use Doxygen to identify code/comment pairs in a large number of repos containing code written in Java, C++, and Python. We use the first sentence of each extracted comment, as it is usually the one that best describes the associated code. We also filter out code/comment pairs where the comments are too long (more than 50 words), too short (fewer than 3 words), or contain phrases like "thanks to" or "created by", since these are all typically uninformative. Similarly, we filter out code/comment pairs where the code is very long (more than 4096 characters), since these code blocks cannot usually be summarized well by a single sentence. We ultimately use about 85% of the code/comment pairs initially collected.

LANGUAGE	COLLECTED	USED
C++	1,050,077	918,583
Java	14,709,616	12,461,021
Python	1,325,845	1,121,421

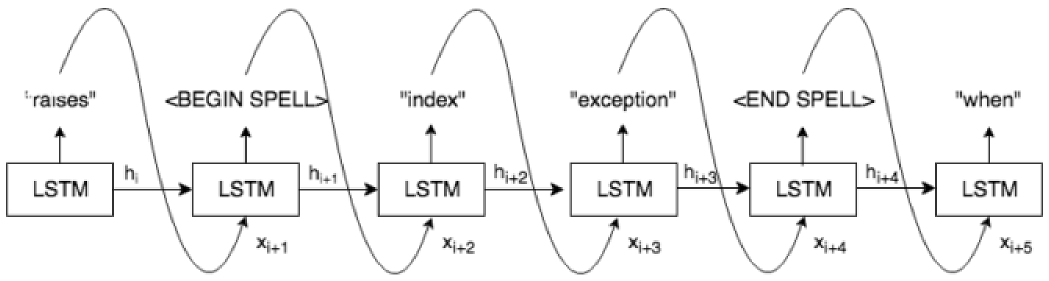
Modeling — the Secret Sauce

Previous source code summarization models require an explicit parsing of the code to be summarized. They also employ a fixed vocabulary composed of the most common words in training data comments. We'd like to avoid both of these modeling constraints. Really, they just make everything harder. Parsing code is time-consuming and failure-prone, especially when the code of interest is a work-in-progress. And, a fixed, word-based vocabulary will limit the model's flexibility, preventing it from generalizing effectively to new code written for new problem domains.

We can do better. We introduce a novel encoder-decoder model designed to overcome both of these limitations. Specifically, we construct a deep convolutional encoder (pictured below) that directly ingests source code. While current models focus on source code's structure and the precise tokens it uses, our model ingests source code at a character-level, avoiding those formalisms. This code-as-text approach enables the model to identify complex natural language constructs in the source code itself, which generalizes well across samples and languages.


d
e
f
_
f
(
)
:
\n
Character embedding
E'd' 1 ... E'd' 16
E'e' 1 ... E'e' 16
E'f' 1 ... E'f' 16
E'_' 1 ... E'_' 16
E'f' 1 ... E'f' 16
E'(' 1 ... E'(' 16
E')' 1 ... E')' 16
E':' 1 ... E':' 16
E'\n' 1 ... E'\n' 16
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
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
Sum-over-time
Sum-over-time
Sum-over-time
Sum-over-time
Sum-over-time
Batch Normalization
Dense Layer

In order to enable the model to predict rare and unique terms, we teach it to spell! That is, we construct an LSTM decoder that can yield, at each step, a full word, a sub-word (e.g. "ing", "ly"), or a letter, as depicted below. For a given example, we train the model to predict the sequence of word components that make up the comment. The model learns the relationship between the input code and output tokens, as well as how to combine output tokens in a reasonable manner. Our methodology significantly increases the range of terms that the model can potentially generate, enabling it to predict relatively rare words. However, we avoid the significant increase in vocabulary size (and, therefore, weight space and propensity to overfit) that would usually accompany the inclusion of many rare words in the vocabulary.



Results — Will It Blend?

Outlining a modeling approach is all well-and-good in the abstract, but what happens when the rubber meets the road (or the data meets the GPU, or something…)? As it turns out, that's actually a pretty hard question to answer quantitatively. There are some well-defined evaluation metrics in the machine translation literature that have historically been used to evaluate source code summaries. In keeping with previous work on this subject, we use BLEU scores (bilingual evaluation understudy scores) to quantitatively evaluate our model. The results suggest that our method performs at least as well as previous state-of-the-art techniques. See the poster linked at the end of this post for more details.

Calculating BLEU scores — and most other machine translation evaluation metrics — involves comparing n-grams in the model's prediction to those in some "ground truth". For us, the "ground truth" is the actual comment associated with a piece of code. However, this approach to evaluating predictions does not always translate especially well to source code summarization, because source code can correctly be summarized in a wide variety of different ways. Really, the best way to evaluate the effectiveness of the model is to look at some results for yourself. Below, we present a few examples of actual and predicted comments for each of the three languages on which we train and evaluate the model.

LANGUAGE	ACTUAL COMMENT	PREDICTED COMMENT
C++	convert payload into string .	returns a string containing the contents of the payload .
C++	insert a row cut unless it is a duplicate – cut may get sorted .	creates a new matrix .
C++	construct an instruction text to the ui .	creates a new text object .
C++	resumes the thread .	starts a new thread .
C++	set the maximal width of a tab .	sets the width of the tab .

Java	the class errormanager handles errors of the file parsing .	this class is used to report the error messages .
Java	calculate the distance in metres to the rhumb line defined by coords a and b .	returns the distance between two points .
Java	calls the specified method with the given params .	returns a string containing the string representation of the string .
Java	change the length of the file .	sets the length of the file .
Java	custom layout for an item representing a bookmark in the browser .	adds a bookmark to the list .
Python	a foundation class from which atom : title , summary , etc .	represents an atom .
Python	render one or more cells .	renders the contents of the given rectangle .
Python	get the identifier assigned to the message .	returns the id of the underlying dbm .
Python	abstraction for a collection of reactions .	returns a list of action actions .
Python	create the property instance :	the meta object literal for the '

Not bad, eh? While certainly not perfect, our model-generated comments do tend to capture the key themes present in the original comment. And, the model does all of this without an explicit parsing of the code, without a large output vocabulary, and (most importantly) without any human input. Moving forward, we plan to expand the scope of the code summarized by the model. We'd like to be able to meaningfully summarize documents, folders, and even entire repositories. Just think, you'll never again have to wade through foreign code because the repository lacks a coherent "Read Me"! Stay tuned for further work on that issue.

Acknowledgements

This project was sponsored by the Air Force Research Laboratory (AFRL) as part of the DARPA MUSE program. This and other tools developed using the MUSE Corpus can be found at https://muse-

portal.net/.

This work was initially presented at the Women in Machine Learning Workshop at NeurIPS. Additional details can be found on the poster here. Updates have been made since this presentation.