CS 272: Statistical NLP: Winter 2020

Homework 4: Generating Function Descriptions

Sameer Singh
https://canvas.eee.uci.edu/courses/22668/

Code documentation is one of the the most important aspects of programming, helping not only for end-users, but also for future reference for the developer, and for other developers in the same team. However, most developers find the process quite tedious, which results in missing or poor quality code documentation. In this homework, we will use neural networks to automatically generate comments, only based on the function signature. The submissions are due by midnight on **March 20, 2020**, with a maximum of *3 late days allowed*.

1 Task: Text Description of Functions

Machine translation is the task of designing a model which automatically translates text from one language (which we refer to as the *source* language S) to another language (which we refer to as the *target* language T). In our setting, the source language will be a *function signature* either for Java or Python, and the target language will be English (see Figure 1 for examples). To train such a model, we assume we have access to a *parallel corpus* $C = \{<\mathbf{s},\mathbf{t}> | \mathbf{s} \in S, \mathbf{t} \in T\}$ of pairs of sequences, where $\mathbf{t}$ is the textual description (doctstring) of the function signature $\mathbf{s}$. Thus, there will be two corpuses, C_{java} and C_{python}, but we omit the code language when clear from context. Our goal is then to find models that maximize probability $P(\mathbf{t}|\mathbf{s})$ for the sequences in this corpus.

There are certain differences between Java and Python that may be relevant for this task. Java code, in general, tends to be much more verbose, due to a deeper hierarchy of packages and type annotations on the return value and the arguments, even if we leave aside coding practices. Thus, in general we expect Java function signatures to contain more meaningful information, and thus produce more accurate descriptions. However, there may be other factors, such as the quality of docstrings between the languages, that might affect the metrics as well. Given that we will be evaluating using the BLEU score, it's possible docstrings from one language inherently are easier than the other, for example, if they have a much smaller vocabulary.

1.1 Datasets

The data archive is available on Canvas, which is a processed subset of the data originally released here: https://github.com/yakazimir/Code-Datasets/. The dataset is a *parallel corpus* containing pairs of functions and their *automatically extracted* docstrings, for two languages: Python and Java. The function details consist of the full signature, i.e. the path of the package, the return type, the name of the function, and the types and names of the arguments. It is followed by a short text snippet describing the function. Both of them have been tokenized and preprocessed to make it easier to use (even if it sometimes introduces inaccuracies), and since they are automatically extracted, the quality of the generated comments is quite noisy. If you study the file (as you should), you will also see that there is another column, containing the last two tokens from the description; more on this this later. Data is split into train/test/validation.

Input: Function Signature	**Output:** Function Docstring
`util.collection boolean add(object e)`	Ensures that this collection contains the specified element (optional operation).
`numpy.array_repr(arr)`	Return the string representation of an array.

Figure 1: **Examples from the Dataset** showing (processed) source code signature functions and their description, for two different languages (Java and Python). The original documentation for these functions is here: https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html#add-E- and https://docs.scipy.org/doc/numpy/reference/generated/numpy.array_repr.html, respectively.

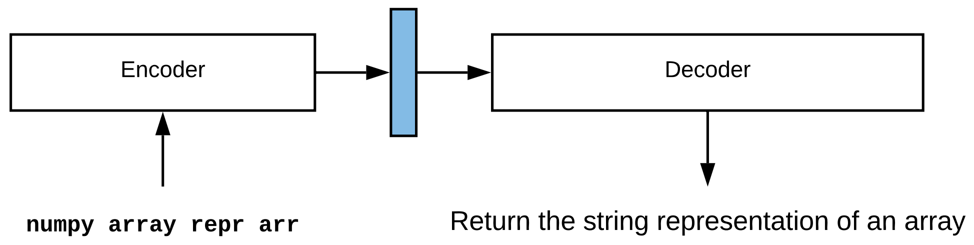

Figure 2: **Naive Sequence to Sequence Model** showing the input function signature, which is encoded into a single vector, followed by a decoder, that generates the output one token at a time.

1.2 (Naive) Sequence-to-Sequence Model

As a reminder, we need a model that estimates $P(\mathbf{t}|\mathbf{s})$. At its core, this problem entails mapping a sequence of inputs (tokens in the source code) to a sequence of outputs (words in English). One difficulty that arises in this task is that there is not a one-to-one correspondence between the input and output sequence. That is, the sequences are typically of different lengths and the word alignment may be non-trivial (e.g. words that are direct translations of each other may not occur in the same order).

The sub-field of neural machine translation parameterizes this probability distribution as a neural network. I will briefly describe the neural machine translation model in this section, for more details see: `https://arxiv.org/pdf/1409.3215.pdf`. We will use a more flexible architecture known as a sequence-to-sequence model, shown in Figure 2. This model is composed of two parts, an *encoder* and a *decoder*. The *encoder* takes as its input the sequence of words in the source language, and outputs a single vector; we have seen a few examples of these, such as the maxpool-ed vector for CNNs or the final hidden states of the RNN layers. The *decoder* is a RNN network (using other kinds of decoders is beyond the scope of the assignment), including an fully connected layer (w/ softmax activation) used to define a probability distribution over the next word in the generated sequence. In this way, the *decoder* essentially functions as a neural language model for the target language. The key difference is that the *decoder* uses the output of the *encoder* as its initial hidden state, as opposed to a vector of zeros.

1.3 (Informed) Sequence-to-Sequence Model

As we will see later, the above model may not be very good at generating accurate descriptions, since it is trying to produce accurate and coherent statements using nothing but the signature. In real world, however, the developer might be able to provide a few words as a "hint", and we want to be able to generate a docstring that uses those words. Thus, we call the above model that does not use any extra information as *naive* because it only takes the function signature into account, while calling a model that takes a "hint" into account as *informed*.

We assume that the extra information consists of two words only, which are the last two words of the actual docstring, i.e. we are giving the model the last two words of the docstring as the "hint". In this homework, you have to create a simple extension to the naive model to incorporate this information. This *informed* model contains an additional encoder, called *extra encoder* that encodes this additional information as another vector. This vector needs to be concatenated with the vector of the function signature, and fed to the decoder, which will generate the sequence as before. The architecture of this model is shown in Figure 3, demonstrating how the extra information flows to the decoder, and thus should improve generation.

1.4 Source Code

I have released the initial source code, available at `https://github.com/sameersingh/uci-statnlp/tree/master/hw4`. You will need to uncompress the data archive and put it in the `data/` folder for the code to work. The source code contains the following:

- **`informed_seq2seq.py`**: The main model file containing an implementation of the **naive** sequence to sequence model, simlar to the default sequence decoder in AllenNLP available here: `https://github.com/allenai/allennlp/blob/master/allennlp/models/encoder_decoders/simple_seq2seq.py`. We have included additional hooks into the code for supporting the "informed" version, however the implementation of the extra part is mostly incomplete; the additional hooks exist to support the extra column in the data,

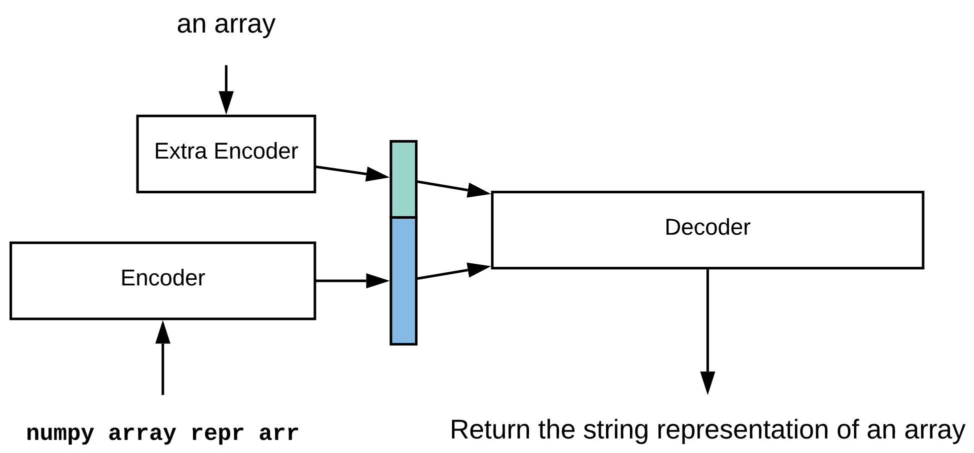

Figure 3: ***Informed* Sequence to Sequence Model** showing the additional encoder that encodes the two tokens of the "hint" into a single vector, which is concatenated with the input vector. The decoder takes both these vector, and uses them to generate the output text. You can pick different options for the encoders and the decoder.

the extra fields in the configuration, etc. This is the only Python file you need to change. Details about what you need to implement is in the sections below.

- **`naive_{java,python}.json`**: The configuration files for training and running the naive sequence to sequence model, on respective language. These configuration files "turn off" the extra embedding part, and run the model in `informed_seq2seq.py`. If your data files are in the right place, you should be able to execute the following command:

```
   allennlp train config/naive_{java,python} -s models/naive_{java,python}
       ↪ --include-package informed_seq2seq --include-package informed_seq2seq_reader
```

  This takes about 40 mins and give you a BLEU score of around 3% for Java, and 15 mins/1% respectively for Python, confirming that Python docstring are more difficult (at least for default configuration).
- **`informed_{java,python}.json`**: Configuration files that "enable" the extra encoder/embedding part, which, if your implementation is correct, should provide much higher gains. The command to run it will be identical to the one above, with a different configuration file.
- **`informed_seq2seq_{predictor,reader}.json`**: Two simple files that provides a way to read the the data and set up a *predictor* that will be used to set up the demo. You should not need to change these files.

2 What to Submit?

Prepare and submit a single write-up (**PDF, maximum 5 pages**) and your modified `.py`, your *custom* configuration files, and any other modified files (compressed in a single `zip` or `tar.gz` file; we will not be compiling or executing it, nor will we be evaluating the quality of the code) to Canvas. The following describes what to include in the write-up.

2.1 Implementation of the *Informed* Model (40 points)

Your primary goal is to add relevant lines to the current implementation so that the extra information is encoded, concatenated, and sent to the decoder. For this purpose, we have included some simple markups in the source code, indicating where you need to add/change the lines, tagged by `'# TODO:'`s. There are only a few changes, each one more than a few lines of code.

Implement this new model, and run the `informed_{java,python}.json` configuration files. If your implementation is correct, you should obtain around 7% BLEU score for Java and 4% BLEU score for Python (2.5-4x that of naive!). Report the four numbers your models have obtained in the report (2 for naive, and 2 for informed).

2.2 Customizing all the Configurations (30 points)

You now have results from models that only use the default configuration, but have not been modified much. You should now try to tune the encoder and decoders, such as picking a more sophisticated encoder (like in HW3), and changing the various dimension sizes, etc. to improve the results. You should be able to obtain substantial gains with these *customized* informed models. Compare their BLEU scores, in a table, with the naive version that uses the same configuration but turns off the informed part.

2.3 Analysis of Predictions (25 points)

BLEU score is not a perfect metric, and thus it is not clear whether the models actually perform better than each other or not. In this section, using only your customized models for one language of your choice (between Java and Python), provide an analysis that provides some insights into the following:

- What are the differences between naive and informed? Try to find one example where naive and informed are both correct, one where both are incorrect, and one example where informed is better. How do you think the “hint” helps in the last case?
- How much can the “hint” be pushed? Does the model learn to always end with the hint? What if you give it a related word rather than the true hint, e.g, “a list” for Figure 3. What if you give it completely a completely different, unrelated “hint”? How do you explain the model’s behavior?
- Take 10 random examples from the test data, and manually judge the quality of the output in terms of grammaticality and meaning equivalence, in terms of a 1-5 rating, when compared to the true docstring. How does the naive model compared to informed in the average rating for each metric?

To perform this analysis, you will need to run the interactive demo of your model, using the following:

```
python -m allennlp.service.server_simple --archive-path [SAVED_MODEL_DIR]/model.tar.gz
    ↪ --predictor informed_seq2seq_predictor --include-package informed_seq2seq      --
    ↪ include-package informed_seq2seq_reader --include-package informed_seq2seq_predictor
    ↪ --title "Code to Text" --field-name source --field-name extra
```

3 Statement of Collaboration (5 points)

It is **mandatory** to include a *Statement of Collaboration* in each submission, with respect to the guidelines below. Include the names of everyone involved in the discussions (especially in-person ones), and what was discussed.

All students are required to follow the academic honesty guidelines posted on the course website. For programming assignments, in particular, I encourage the students to organize (perhaps using Campuswire) to discuss the task descriptions, requirements, bugs in my code, and the relevant technical content *before* they start working on it. However, you should not discuss the specific solutions, and, as a guiding principle, you are not allowed to take anything written or drawn away from these discussions (i.e. no photographs of the blackboard, written notes, etc.). Especially *after* you have started working on the assignment, try to restrict the discussion to Campuswire as much as possible, so that there is no doubt as to the extent of your collaboration. You are free to discuss the numbers you are getting with others, and again, I encourage you to use Campuswire to post your generated text and compare them with others.

Acknowledgements

The homework was implemented primarily with the help of Yoshitomo Matsubara, with the datasets released by Kyle Richardson, and the help of Matt Gardner.