Programming for Computational Linguistics 2018/2019

Final Project: Full Specification

In this document, we will fully enumerate the required components of your n-gram language model. You will be graded on how well your program meets this specification.

Your program will be split into three separate modules: `corpus.py`, `lm.py`, and `main.py` (although you may implement additional modules to use if you would like). `corpus.py` will be responsible for reading text files, and processing and tokenizing text. `lm.py` will implement the core logic of the language model, and will be responsible for predicting tokens and generating text. `main.py` will be responsible for user interaction — users of the language model will envoke `main.py`, and all functionality will be exposed through this module.

1 `corpus.py`

This module should implement the following functions:

1.1 `tokenize(text)`

This function should accept as input a string `text`, and should return a list of tokens, with each token consisting of a string. The exact implementation is up to you — you can do this as simply or as complicatedly as you like. As long as you generate a list of tokens in some manner, you will receive full credit for this section. However, cleverer tokenization techniques may yield subjectively better final text, which will factor into the subjective output evaluation.

1.2 `detokenize(tokens)`

This function should accept as input a list of tokens, and should return a single string consisting of all of those tokens together. This could be as simple as `' '.join(tokens)`, but could be more sophisticated to account for punctuation, capitalization, etc. As with `tokenize(text)`, you will get full credit so long as this function is present and works, but more clever implementations may yield subjectively better text.

2 `lm.py`

This module should contain a class `LanguageModel`. This class will deal only with token sequences as provided by `corpus.py`, not directly with untokenized text. That class should implement the following methods:

2.1 `__init__(self, n)`

The constructor should create a new, untrained `LanguageModel`, and initialize whatever is needed. The constructor must have the parameter `n`, to select which n the language model should use for its n-grams. You may also provide other parameters if you think they might be useful to you, but they must have default values — it should be possible to instantiate a `LanguageModel` by calling `LanguageModel(n)` with just a single argument for n.

2.2 `train(self, token_sequences)`

This method should train your language model, such that it learns the n-gram statistics of `token_sequences`. This method alone will not return anything, but after this method is called, the methods `p_next` and `generate` should work properly, according to the statistics of `token_sequences`. You should assume that this method will always be called at least once, after an object is instantiated, and before `p_next` or `generate` are called. `token_sequences` is a list of token sequences, one for each text, and each token sequence is a list of strings, one string per token (with no EOS at the end). Thus, `token_sequences` is a list of list of strings. If this method is called multiple times, your language model should remember the statistics of all token sequence it has seen, and agregate them. If, for example, `lm` is a `LanguageModel`, and `seqs_1` and `seqs_2` are both lists of token sequences, then the single line

```
lm.train(seqs_1 + seqs_2)
```

should be exactly equivalent to

```
lm.train(seqs_1)
lm.train(seqs_2)
```

2.3 `p_next(self, tokens)`

Return the estimated probability distribution for the next word that occurs after the token sequence `tokens`. `tokens` is a list of zero or more strings, representing the text so far. The return value should be a dict that represents a probability distribution. The keys of the returned dict are words which might follow `tokens`, and the values are the probabilities for each word, as floats. The values of the dict should sum to 1. Words with zero probability do not need to be present as keys in the dict. As an example, if our language model `lm` says that, after the tokens ⟨*cat*, *and*⟩, the next token is 60% likely to be *mouse* and 40% likely to *dog*, `lm.p_next(['cat', 'and'])` should return `{'mouse': 0.6, 'dog': 0.4}`. The EOS symbol should be represented by `None`.

2.4 `generate(self)`

Generate a random token sequence according to the underlying probability distribution. This method should take no parameters, and should return a full generated text, as a list of tokens (strings). The returned list should *not* end in an EOS symbol.

3 `main.py`

This module will be responsible for user interaction — users of the language model will envoke `main.py`, and all functionality will be exposed through this file. When run, it should provide the user instructions, and ask what they would like to do. This program should allow the user to:

- Create a new language model with a user-specified n
- Load texts from a file, and train the language model on those texts
- Generate a text from the language model, and print it to the screen
- Generate a user-specified number of texts from the language model, and write them to a file
- Exit the program

The exact interface is up to you, but your program should print enough instructions to make it self explanatory. We should be able to run this program, without reading your report, and figure out how to do what we want (However, you should still fully document the user interface in your report).