CIS 521-Artificial Intelligence

Homework 8: Language Models [105 points]

Instructions

In this assignment, you will gain experience working with Markov models on text.

A skeleton file homework8.py containing empty definitions for each question has been provided. Since portions of this assignment will be graded automatically, none of the names or function signatures in this file should be modified. However, you are free to introduce additional variables or functions if needed.

You may import definitions from any standard Python library, and are encouraged to do so in case you find yourself reinventing the wheel.

You will find that in addition to a problem specification, most programming questions also include a pair of examples from the Python interpreter. These are meant to illustrate typical use cases, and should not be taken as comprehensive test suites.

Once you have completed the assignment, you should submit your file on Gradescope. You may submit as many times as you would like before the deadline, but only the last submission will be saved.

0. Style [5 points]

Your code should follow the proper Python style guidelines set forth in PEP 8, which was written in part by the creator of Python. Our autograders will automatically scan your submission for style errors using the `pycodestyle` library on default settings. If your submission contains **any** style errors, the autograder will show you some of them and you will not receive these 5 points. You can use `pycodestyle` or any other tool you like to make sure that your submission conforms to PEP 8 guidelines.

1. N-Gram Model [95 points]

In this section, you will build a simple language model that can be used to generate random text resembling a source document. Your use of external code should be limited to built-in Python modules, which excludes, for example, NumPy and NLTK.

1. **[5 points]** Write a simple tokenization function `tokenize(text)` which takes as input a string of text and returns a list of tokens derived from that text. Here, we define a token to be a contiguous sequence of non-whitespace characters, with the exception that any punctuation mark should be treated as an individual token. *Hint: Use the built-in constant `string.punctuation`, found in the `string` module.*

```
>>> tokenize("  This is an example.  ")
['This', 'is', 'an', 'example', '.']
```

```
>>> tokenize("'Medium-rare,' she said.")
["'", 'Medium', '-', 'rare', ',', "'", 'she', 'said', '.']
```

2. **[10 points]** Write a function `ngrams(n, tokens)` that produces a list of all n-grams of the specified size from the input token list. Each n-gram should consist of a 2-element tuple (context, token), where the context is itself an $(n-1)$-element tuple comprised of the $n-1$ words preceding the current token. The sentence should be padded with $n-1$ `"<START>"` tokens at the beginning and a single `"<END>"` token at the end. If $n=1$, all contexts should be empty tuples. You may assume that $n \geq 1$.

```
>>> ngrams(1, ["a", "b", "c"])
[((), 'a'), ((), 'b'), ((), 'c'), ((), '<END>')]
```

```
>>> ngrams(2, ["a", "b", "c"])
[(('<START>',), 'a'), (('a',), 'b'), (('b',), 'c'),
(('c',), '<END>')]
```

```
>>> ngrams(3, ["a", "b", "c"])
[(('<START>', '<START>'), 'a'), (('<START>', 'a'), 'b'),
(('a', 'b'), 'c'), (('b', 'c'), '<END>')]
```

3. **[10 points]** In the `NgramModel` class, write an initialization method `__init__(self, n)` which stores the order n of the model and initializes any necessary internal variables. Then write a method `update(self, sentence)` which computes the n-grams for the input sentence and updates the internal counts. Lastly, write a method `prob(self, context, token)` which accepts an $(n-1)$-tuple representing a context and a token, and returns the probability of that token occuring, given the preceding context.

```
>>> m = NgramModel(1)
>>> m.update("a b c d")
>>> m.update("a b a b")
>>> m.prob((), "a")
0.3
>>> m.prob((), "c")
0.1
>>> m.prob((), "<END>")
0.2
```

```
>>> m = NgramModel(2)
>>> m.update("a b c d")
```

```
>>> m.update("a b a b")
>>> m.prob(("<START>",), "a")
1.0
>>> m.prob(("b",), "c")
0.3333333333333333
>>> m.prob(("a",), "x")
0.0
```

4. **[20 points]** In the `NgramModel` class, write a method `random_token(self, context)` which returns a random token according to the probability distribution determined by the given context. Specifically, let $T = \langle t_1, t_2, \cdots, t_n \rangle$ be the set of tokens which can occur in the given context, sorted according to Python's natural lexicographic ordering, and let $0 \leq r < 1$ be a random number between 0 and 1. Your method should return the token t_i such that

   $$\sum_{j=1}^{i-1} P(t_j \mid \text{context}) \leq r < \sum_{j=1}^{i} P(t_j \mid \text{context}).$$

   You should use a single call to the `random.random()` function to generate r.

```
>>> m = NgramModel(1)
>>> m.update("a b c d")
>>> m.update("a b a b")
>>> random.seed(1)
>>> [m.random_token(())
     for i in range(25)]
['<END>', 'c', 'b', 'a', 'a', 'a', 'b', 'b', '<END>',
'<END>', 'c', 'a', 'b', '<END>', 'a', 'b', 'a', 'd',
'd', '<END>', '<END>', 'b', 'd', 'a', 'a']
```

```
>>> m = NgramModel(2)
>>> m.update("a b c d")
>>> m.update("a b a b")
>>> random.seed(2)
>>> [m.random_token(("<START>",)) for i in range(6)]
['a', 'a', 'a', 'a', 'a', 'a']
>>> [m.random_token(("b",)) for i in range(6)]
['c', '<END>', 'a', 'a', 'a', '<END>']
```

5. **[20 points]** In the `NgramModel` class, write a method `random_text(self, token_count)` which returns a string of space-separated tokens chosen at random using the `random_token(self, context)` method. Your starting context should always be the $(n-1)$-tuple `("<START>", ..., "<START>")`, and the context should be updated as tokens are generated. If $n = 1$, your context should always be the empty tuple. Whenever the special token `"<END>"` is encountered, you should reset the context to the starting context.

```
>>> m = NgramModel(1)
>>> m.update("a b c d")
```

```
>>> m.update("a b a b")
>>> random.seed(1)
>>> m.random_text(13)
'<END> c b a a a b b <END> <END> c a b'
```

```
>>> m = NgramModel(2)
>>> m.update("a b c d")
>>> m.update("a b a b")
>>> random.seed(2)
>>> m.random_text(15)
'a b <END> a b c d <END> a b a b a b c'
```

6. **[15 points]** Write a function `create_ngram_model(n, path)` which loads the text at the given path and creates an n-gram model from the resulting data. Each line in the file should be treated as a separate sentence.

```
# No random seeds, so your results may vary
>>> m = create_ngram_model(1, "frankenstein.txt"); m.random_text(15)
'beat astonishment brought his for how , door <END> his . pertinacity to I
felt'
>>> m = create_ngram_model(2, "frankenstein.txt"); m.random_text(15)
'As the great was extreme during the end of being . <END> Fortunately the
sun'
>>> m = create_ngram_model(3, "frankenstein.txt"); m.random_text(15)
'I had so long inhabited . <END> You were thrown , by returning with
greater'
>>> m = create_ngram_model(4, "frankenstein.txt"); m.random_text(15)
'We were soon joined by Elizabeth . <END> At these moments I wept bitterly
and'
```

7. **[15 points]** Suppose we define the perplexity of a sequence of m tokens $\langle w_1, w_2, \cdots, w_m \rangle$ to be

$$\sqrt[m]{\frac{1}{P(w_1, w_2, \cdots, w_m)}}.$$

For example, in the case of a bigram model under the framework used in the rest of the assignment, we would generate the bigrams $\langle (w_0 = \langle \text{START} \rangle, w_1), (w_1, w_2), \cdots, (w_{m-1}, w_m), (w_m, w_{m+1} = \langle \text{END} \rangle) \rangle$, and would then compute the perplexity as

$$\sqrt[m+1]{\prod_{i=1}^{m+1} \frac{1}{P(w_i \mid w_{i-1})}}.$$

Intuitively, the lower the perplexity, the better the input sequence is explained by the model. Higher values indicate the input was "perplexing" from the model's point of view, hence the term perplexity.

In the `NgramModel` class, write a method `perplexity(self, sentence)` which computes the n-grams for the input sentence and returns their perplexity under the current model. *Hint: Consider performing an intermediate computation in log-space and re-exponentiating at the end, so as to avoid numerical overflow.*

```
>>> m = NgramModel(1)
>>> m.update("a b c d")
>>> m.update("a b a b")
>>> m.perplexity("a b")
3.815714141844439
```

```
>>> m = NgramModel(2)
>>> m.update("a b c d")
>>> m.update("a b a b")
>>> m.perplexity("a b")
1.4422495703074083
```

2. Use AI to Write Harry Potter Fanfiction [0 points - just for fun]

Botnik Studios used text prediction to create three pages of Harry Potter Fanfiction entitled "Harry Potter and the Portrait of What Looked Like a Large Pile of Ash,". I think that we should try too. So I downloaded an archive of Harry Potter fanfiction. There is *a lot* of Harry Potter fanfiction. There is more than a quarter of a billion words of it.

You can find the archive on Eniac. To download it, you must connect to Penn's network, which you can do by following the instructions here. Then run the following command (replacing "PENN-KEY" with your Penn key):

```
scp PENN-KEY@eniac.seas.upenn.edu:
~cis521/data/HarryPotterFanfiction.com.sentences.utf8.txt.gz .
```

Then, locally, you can unzip it by running

```
gunzip HarryPotterFanfiction.com.sentences.utf8.txt.gz
```

The file is large (1.4GB) and contains 25 million lines and 279,000,000 words. For testing purposes, you might want to take the first 1 million lines:

```
head -1000000 HarryPotterFanfiction.com.sentences.utf8.txt >
smaller-HarryPotterFanfiction.com.sentences.utf8.txt
```

This file may contain copyrighted materials, so please don't redistrbute it. We are using it for non-commerical research purposes, which is fair use.

How well did your automatically generated Harry Potter sentences turn out? Post your favorites to Piazza.

3. Feedback [5 points]

1. **[1 point]** Approximately how many hours did you spend on this assignment?
2. **[2 point]** Which aspects of this assignment did you find most challenging? Were there any significant stumbling blocks?
3. **[2 point]** Which aspects of this assignment did you like? Is there anything you would have changed?