Instructions

In this assignment, you will gain experience working with Markov models and hidden Markov models.

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.

You are strongly encouraged to follow the Python style guidelines set forth in PEP 8, which was written in part by the creator of Python. However, your code will not be graded for style.

Once you have completed the assignment, you should submit your file on Eniac using the following turnin command, where the flags -c and -p stand for “course” and “project”, respectively.

`turnin -c cis391 -p hw8 homework8.py`

You may submit as many times as you would like before the deadline, but only the last submission will be saved. To view a detailed listing of the contents of your most recent submission, you can use the following command, where the flag -v stands for “verbose”.

`turnin -c cis391 -p hw8 -v`

1 Markov Models [45 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", ",", ",", ",", ",", ",", "she", ",", "said", ","]
```

2. [5 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 $n \geq 1$.

```python
>>> ngrams(1, ["a", "b", "c"])
[(((), 'a'), ((), 'b'), ((), 'c'), ((), '<END>')]
>>> ngrams(2, ["a", "b", "c"])
[(('START',), 'a'), (('a',), 'b'), (('b',), 'c'), (('<END>',), '<END>')]
>>> ngrams(3, ["a", "b", "c"])
[(('START', 'START'), 'a'), (('a', 'START'), 'b'), (('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 occurring, given the preceding context.

```python
>>> 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
```

4. **[10 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 $T = \{t_1, t_2, \ldots , t_n\}$ 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

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

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

```python
>>> 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', 'b', '<END>', '<END>', '<END>', '<END>', 'c', 'b', 'b', '<END>', '<END>', '<END>', 'a', 'a', 'a', 'a', 'b', 'd', 'd', '<END>', '<END>', '<END>', 'c', 'b', 'a', 'a', 'a', 'a', '<END>']
```

5. **[10 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>"), or the empty tuple if $n = 1$, and your context should be updated as tokens are generated. Whenever the special token "<END>" is encountered, you should reset the context to the starting context.

```python
>>> 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', 'b', '<END>', '<END>', 'c', 'b', '<END>', '<END>', '<END>', '<END>', 'c', 'b', 'a', 'a', 'a', 'a', '<END>']
```
6. **[5 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.

```python
>>> 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 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'
```

2 Hidden Markov Models **[50 Points]**

In this section, you will develop a hidden Markov model for part-of-speech (POS) tagging, using the Brown corpus as training data. The tag set used in this assignment will be the universal POS tag set, which is composed of the twelve POS tags `NOUN` (noun), `VERB` (verb), `ADJ` (adjective), `ADV` (adverb), `PRON` (pronoun), `DET` (determiner or article), `ADP` (preposition or postposition), `NUM` (numeral), `CONJ` (conjunction), `PRT` (particle), `.` (punctuation mark), and `X` (other).

As in the previous section, your use of external code should be limited to built-in Python modules, which again excludes packages such as NumPy and NLTK.

1. **[5 Points]** Write a function `load_corpus(path)` that loads the corpus at the given path and returns it as a list of POS-tagged sentences. Each line in the file should be treated as a separate sentence, where sentences consist of sequences of whitespace-separated strings of the form "token=POS". Your function should return a list of lists, with individual entries being 2-tuples of the form (token, POS).

```python
>>> c = load_corpus("brown_corpus.txt")
>>> c[1402]
[('It', 'PRON'), ('made', 'VERB'), ('him', 'PRON'), ('human', 'NOUN'), ('.', '.')]
```

2. **[10 Points]** In the Tagger class, write an initialization method `__init__(self, sentences)` which takes a list of sentences in the form produced by `load_corpus(path)` as input and initializes the internal variables needed for the POS tagger. In particular, if \( \{t_1, t_2, \ldots, t_n\} \) denotes the set of tags and \( \{w_1, w_2, \ldots, w_m\} \) denotes the set of tokens found in the input sentences, you should at minimum compute:

- The initial tag probabilities \( \pi(t_i) \) for \( 1 \leq i \leq n \).
- The transition probabilities \( a(t_i \rightarrow t_j) \) for \( 1 \leq i, j \leq n \).
- The emission probabilities \( b(t_i \rightarrow w_j) \) for \( 1 \leq i \leq n \) and \( 1 \leq j \leq m \).
You should use Laplace smoothing where appropriate to ensure that your system can handle novel inputs. Your initialization method should take no more than a few seconds to complete when given the full Brown corpus as input.

3. [10 Points] In the Tagger class, write a method `most_probable_tags(self, tokens)` which returns the list of the most probable tags corresponding to each input token. In particular, the most probable tag for a token $w_j$ is defined to be the tag with $\arg\max_i b(t_i \rightarrow w_j)$.

```python
>>> c = load_corpus("brown_corpus.txt")
>>> t = Tagger(c)
>>> t.most_probable_tags(
... ["The", "man", "walks", "."]
["DET", "NOUN", "VERB", "."]
```

4. [25 Points] In the Tagger class, write a method `viterbi_tags(self, tokens)` which returns the most probable tag sequence as found by Viterbi decoding. Recall from lecture that Viterbi decoding is a modification of the Forward algorithm, adapted to find the path of highest probability through the trellis graph containing all possible tag sequences. Computation will likely proceed in two stages: you will first compute the probability of the most likely tag sequence, and will then reconstruct the sequence which achieves that probability from end to beginning by tracing backpointers.

```python
>>> c = load_corpus("brown_corpus.txt")
>>> t = Tagger(c)
>>> s = "I am waiting to reply".split()
>>> t.most_probable_tags(s)
["PRON", "VERB", "VERB", "PRT", "NOUN"]
>>> t.viterbi_tags(s)
["PRON", "VERB", "VERB", "PRT", "VERB"]
```

3 Feedback [5 Points]

1. [1 Point] Approximately how long did you spend on this assignment?

2. [2 Points] Which aspects of this assignment did you find most challenging? Were there any significant stumbling blocks?

3. [2 Points] Which aspects of this assignment did you like? Is there anything you would have changed?