CMPSC/DS 442: Homework 5

TO PREPARE AND SUBMIT HOMEWORK

Follow these steps exactly, so the Gradescope autograder can grade your homework. Failure to do so will result in a zero grade:

1. You *must* download the homework template file `homewor5_starter.py` from Canvas. Each template file is a python file that gives you a headstart in creating your homework python script with the correct function names for autograding. Make sure to check you have additional data files other than the `homewor5_starter.py`:
    - `brown_corpus.txt`
    - `homework5_data.zip`
2. You *must* rename the file by replacing `starter` with your PSU email id. For example, if your PSU email id is abcd1234, you would rename your file as `homework5_abcd1234.py`.
3. Upload your `homework5_abcd1234.py` file to Gradescope by the due date.
4. Make sure your file can be imported before you submit; the autograder imports your file. If it won't import, you will get a zero.

Instructions

In this assignment, you will work on two parts: implement 1) a basic spam filter using naive Bayes classification and 2) hidden Markov models for part-of-speech tagging.

A skeleton file `homework5_starter.py` containing empty definitions for each question has been provided. A zip file called `homework5_data.zip` has also been provided that contains the input train and test data. Since portions of this assignment will be graded automatically, none of the names or function signatures in the skeleton template 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. If you are unsure where to start, consider taking a look at the data structures and functions defined in the `collections`, `email`, `math`, and `os` modules.

You will find that in addition to a problem specification, most programming questions also include one or two examples from the Python interpreter. These are meant to illustrate typical use cases to clarify the assignment, and are not comprehensive test suites. In addition to performing your own testing, you are strongly encouraged to verify that your code gives the expected output for these examples before submitting.

It is highly recommended that you 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.

1. Spam Filter [50 points]

In this section, you will implement a minimal system for spam filtering. You should unzip the `homework5_data.zip` file in the same location as your skeleton file; this will create a `homework5_data/train` folder and a `homework5_data/dev` folder. You will begin by processing the raw training data. Next, you will proceed by estimating the conditional probability distributions of the words in the vocabulary determined by each document class. Lastly, you will use a naive Bayes model to make predictions on the publicly available test set, located in `homework5_data/dev`.

1. **[3 points]** Making use of the `email` module, write a function `load_tokens(email_path)` that reads the email at the specified path, extracts the tokens from its message, and returns them as a list.

   Specifically, you should use the `email.message_from_file(file_obj)` function to create a `message` object from the contents of the file, and the `email.iterators.body_line_iterator(message)` function to iterate over the lines in the message. Here, tokens are considered to be contiguous substrings of non-whitespace characters.

```
>>> ham_dir = "homework5_data/train/ham/"
>>> load_tokens(ham_dir+"ham1")[200:204]
['of', 'my', 'outstanding', 'mail']
>>> load_tokens(ham_dir+"ham2")[110:114]
['for', 'Preferences', '-', "didn't"]
```

```
>>> spam_dir = "homework5_data/train/spam/
>>> load_tokens(spam_dir+"spam1")[1:5]
['You', 'are', 'receiving', 'this']
>>> load_tokens(spam_dir+"spam2")[:4]
['<html>', '<body>', '<center>', '<h3>']
```

2. **[15 points]** Write a function `log_probs(email_paths, smoothing)` that returns a dictionary from the words contained in the given emails to their Laplace-smoothed log-probabilities. Specifically, if the set V denotes the vocabulary of words in the emails, then the probabilities should be computed by taking the logarithms of

$$P(w) = \frac{\text{count}(w) + \alpha}{\left(\sum_{w' \in V} \text{count}(w')\right) + \alpha(|V| + 1)}, \qquad P(\langle \text{UNK} \rangle) = \frac{\alpha}{\left(\sum_{w' \in V} \text{count}(w')\right) + \alpha(|V| + 1)}$$

   where w is a word in the vocabulary V, α is the smoothing constant (typically in the range $0 < \alpha \leq 1$), and <UNK> denotes a special word that will be substituted for unknown tokens at test time. The output can be slightly different but it has to be ± 0.05 to get full credit.

```
>>> paths = ["homework5_data/train/ham/ham%d" % i
...   for i in range(1, 11)]
>>> p = log_probs(paths, 1e-5)
>>> p["the"]
-3.6080194731874062
>>> p["line"]
-4.272995709320345
```

```
>>> paths = ["homework5_data/train/spam/spam%d" % i
...  for i in range(1, 11)]
>>> p = log_probs(paths, 1e-5)
>>> p["Credit"]
-5.837004641921745
>>> p["<UNK>"]
-20.34566288044584
```

3. **[5 points]** Write an initialization method `_init_(self, spam_dir, ham_dir, smoothing)` in the `SpamFilter` class that creates two log-probability dictionaries corresponding to the emails in the provided spam and ham directories, then stores them internally for future use. Also compute the class probabilities *P(spam)* and *P(¬spam)* based on the number of files in the input directories.

4. **[12 points]** Write a method `is_spam(self, email_path)` in the `SpamFilter` class that returns a Boolean value indicating whether the email at the given file path is predicted to be spam. Tokens which were not encountered during the training process should be converted into the special word `"<UNK>"` in order to avoid zero probabilities.

   Recall from the lecture slides that for a given class *c E {spam, ¬ spam}*,

$$P(c \mid \text{document}) \sim P(c) \prod_{w \in V} P(w \mid c)^{\text{count}(w)},$$

   where the normalization constant *1 / P(document)* is the same for both classes and can therefore be ignored. Here, the count of a word is computed over the input document to be classified.

   These computations should be computed in log-space to avoid underflow.

```
>>> sf = SpamFilter("homework5_data/train/spam",
...  "homework5_data/train/ham", 1e-5)
>>> sf.is_spam("homework5_data/train/spam/spam1")
True
>>> sf.is_spam("homework5_data/train/spam/spam2")
True
```

```
>>> sf = SpamFilter("homework5_data/train/spam",
...  "homework5_data/train/ham", 1e-5)
>>> sf.is_spam("homework5_data/train/ham/ham1")
False
>>> sf.is_spam("homework5_data/train/ham/ham2")
False
```

5. **[15 points]** Suppose we define the spam indication value of a word w to be the quantity

$$\log\left(\frac{P(w \mid \text{spam})}{P(w)}\right).$$

Similarly, define the ham indication value of a word w to be

$$\log\left(\frac{P(w \mid \neg\text{spam})}{P(w)}\right).$$

Write a pair of methods `most_indicative_spam(self, n)` and `most_indicative_ham(self, n)` in the `SpamFilter` class which returns the `n` most indicative words for each category, sorted in descending order based on their indication values. You should restrict the set of words considered for each method to those which appear in at least one spam email and one ham email. *Hint: The probabilities computed within the `_init_(self, spam_dir, ham_dir, smoothing)` method are sufficient to calculate these quantities.*

```
>>> sf = SpamFilter("homework5_data/train/spam",
...  "homework5_data/train/ham", 1e-5)
>>> sf.most_indicative_spam(5)
['<a', '<input', '<html>', '<meta','</head>']
```

```
>>> sf = SpamFilter("homework5_data/train/spam",
...  "homework5_data/train/ham", 1e-5)
>>> sf.most_indicative_ham(5)
['Aug', 'ilug@linux.ie', 'install', 'spam.', 'Group:']
```

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 previous assignments, your use of external code should be limited to built-in Python modules, which **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).

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

```
>>> c = load_corpus("brown_corpus.txt")
>>> c[1799]
[('The', 'DET'), ('prospects', 'NOUN'),
 ('look', 'VERB'), ('great',
 'ADJ'), ('.', '.')]
```

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 $1 \le i \le n$, where $\pi(t_i)$ is the probability that a sentence begins with tag t_i.
    - The transition probabilities $a(t_i \rightarrow t_j)$ for $1 \le i, j \le n$, where $a(t_i \rightarrow t_j)$ is the probability that tag t_j occurs after tag t_i.
    - The emission probabilities $b(t_i \rightarrow w_j)$ for $1 \le i \le n$ and $1 \le j \le m$, where $b(t_i \rightarrow w_j)$ is the probability that token w_j is generated given tag t_i.

    It is imperative that you use Laplace smoothing where appropriate to ensure that your system can handle novel inputs, but the exact manner in which this is done is left up to you as a design decision. Your initialization method should take no more than a few seconds to complete when given the full Brown corpus as input.

3. **[15 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 wj is defined to be the tag with index $i^* = argmax_i\, b(t_i \rightarrow wj)$.

```
>>> c = load_corpus("brown_corpus.txt")
>>> t = Tagger(c)
>>> t.most_probable_tags(
...  ["The", "man", "walks", "."])
['DET', 'NOUN', 'VERB', '.']
```

```
>>> c = load_corpus("brown_corpus.txt")
>>> t = Tagger(c)
>>> t.most_probable_tags(
...  ["The", "blue", "bird", "sings"])
['DET', 'ADJ', 'NOUN', 'VERB']
```

4. **[20 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 the 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.

```
>>> 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']
```

```
>>> c = load_corpus("brown_corpus.txt")
>>> t = Tagger(c)
>>> s = "I saw the play".split()
>>> t.most_probable_tags(s)
['PRON', 'VERB', 'DET', 'VERB']
>>> t.viterbi_tags(s)
['PRON', 'VERB', 'DET', 'NOUN']
```