Project 5: Spam Filter

Due: Wednesday Oct. 28th, 11:59am (Eastern Time)

Deliverables

- `writeup.tex` and the corresponding compiled `writeup.pdf` with answers to all questions.
- A completed `hamspam.py` with all required functions filled in.

1 Naive Bayes

In this assignment, you implement a basic spam filter using naive Bayes classification.

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.

In this lab, you will implement a minimal system for spam filtering. You will begin by processing the raw training data (in `./data/train-ham/`[1] and `./data/train-spam/`). 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 `./data/dev-ham/` and `./data/dev-spam/`.

All of the data you'll need for this project is outlined below and already included in the associated GitHub repo:

Data

- A skeleton file is available in the `hamspam.py` Python file of the Lab5 **GitHub repo**.
- `data` is a directory containing the training and dev files categorized as either ham or spam. The same data is also available on the department machines at: `/data/cs65-F20/spamham/`.

[1] Yes, it's actually called **ham**!

1.1 load_tokens()

Questions

1. 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. Give a couple sentence description of your approach in `writeup.tex`.

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

1.2 log_probs()

Questions

2. 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. Provide a one/two sentence description of your implementation in `writeup.tex` and point me to the relevant line numbers in your code.

Specifically, if the set V denote the vocabulary of words in the emails, then the probabilities should be computed by taking the logarithms of:

$$P(w) = \frac{count(w) + \alpha}{\sum_{w' \in V} count(w') + \alpha|V|} \tag{1}$$

where w is a word in the vocabulary V and α is the smoothing constant[2]. Be sure to add an <*UNK*> token (whose count in the training data is 0) to the vocabulary to handle novel items in the development and test data.

[2] it's up to you to figure out a reasonable value!

```
>>> paths = ["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 = ["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
```

1.3 __init__

Questions

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

   Describe your approach in one/two sentences in `writeup.tex` and provide a pointer to the relevant line numbers for the corresponding implementation.

1.4 is_spam()

Questions

4. 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. Again, provide a brief description in `writeup.tex` and a pointer to the relevant line numbers.

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 class that for a given class $c \in \{spam, \neg spam\}$:

$$P(c|document) \approx P(c) \prod_{w \in V} P(w|c)^{count(w)} \tag{2}$$

(In principle we should be dividing by the normalization constant $P(document)$ but since it's the same for both classes it can safely be ignored). Here the count of a word is computed over the input document to be classified (not the count in the training data).

Remember: these computations should be done in log-space to avoid underflow.

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

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

1.5 most_indicative{spam,ham}()

Suppose we define the spam indication value of a word w to be the quantity:

$$log(\frac{P(w|spam)}{P(w)}) \tag{3}$$

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

$$log(\frac{P(w|\neg spam)}{P(w)}) \tag{4}$$

Questions

5. Write a pair of methods `most_indicative_spam(self, n)` and `most_indicative_ham(self, n)` in the `SpamFilter` class which return 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 both at least one spam email and at least one ham email. As with the other questions, provide a very brief description in `writeup.tex` and a pointer to the relevant line numbers.

Hint: The probabilities computed within the `__init__(self, spam_dir, ham_dir, smoothing)` method are sufficient to calculate these quantities.

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

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

2 Feedback

Please provide answers to these in `writeup.tex` just like the rest of the questions. There are no right or wrong answers here of course :)

Questions

6. Approximately how many hours did you spend on this assignment?

7. Which aspects of this assignment did you find most challenging? Were there any significant stumbling blocks?

8. Which aspects of this assignment did you like? Is there anything you would have changed?

3 Optional Extra credit: Extending the Spam Filter with Feature Engineering

Let me know if you complete this extra credit assignment by adding an additional section at the end of the `writeup` file with the relevant answers.

So far you have already implemented a baseline system for spam detection using simple single-token features in a Naive Bayes framework. Your task for this extra credit is to modify and extend that system in order to improve its performance. Since the baseline classifier achieved accuracies in excess of 97% on the development set, here we'll aim to achieve an accuracy of 99% or higher on the dev set.

The only interface your code will be required to support is as follows, provided in the skeleton file:

1. I should be able to initialize your spam filter using the statement `SpamFilter(spam_dir, ham_dir)`, where `spam_dir` and `ham_dir` are strings pointing to the directories containing the training spam and non-spam emails. If your classifier takes any additional parameters, such as smoothing constants, they should be hard-coded in.
2. If `spam_filter` is an instance of your original `SpamFilter` class, I should be able to classify an email using the statement `spam_filter.is_spam(email_path)`, where a return value of `True` indicates that the email is spam, and a return value of `False` indicates that the email is not spam.

You are encouraged to borrow liberally from the code you wrote for the main assignment, though some non-trivial changes will have to be made. In particular, while I required that you maintain a Naive Bayes framework, you will want to make use of features beyond single tokens. To get you started, a few ideas are listed below, but you will also want to consider features of your own creation in order to maximize performance and robustness.

Additional features might include:

- Bigram features formed from pairs of consecutive words.
- Capitalization features, such as all-lowercase, all-uppercase, first-letter-capitalized, etc.
- Punctuation features, indicating whether a token consists of only punctuation.
- Number features, indicating whether a token is a number.

- Features related to the length of a token.
- Features derived from the email headers.

Moreover, you might also consider using tokenization methods that are more sophisticated than simply splitting on whitespace, or introducing multiple smoothing constants for different components of your system. Finally, you may want to play around with the value of the smoothing constant, to find a value that better optimizes spam detection.

To help ensure that you have a comprehensive understanding of the techniques you are applying, you will need to limit your use of external code to built-in Python modules (just like with the main assignment). This excludes, for example, the NLTK platform. Before you begin writing your system extensions, think carefully about how this should be integrated into the Naive Bayes model, and what additional infrastructure they may require.

In order to get credit for this Bonus Assignment, you will need to add an additional section to your `writeup` report. A bonus section with appropriate detail will be approximately one page in length and include at least the following information:

- The general design of your system.
- The features you experimented with.
- How you handled pre-processing and tokenization.
- The experiments you performed and the results you obtained, including charts or plots as needed.
- An analysis of the specific errors made by your system on the development data.

Have fun!