A Problematic Spam Filter
ISTA 331 Hw2, Due Thursday 2/20 at 11:59 pm

Introduction. The first AI application that impacted everyone's daily life was the spam filter. In this assignment, you will use Naïve Bayes to implement a simple spam filter, a technique still in use in many filters, https://en.wikipedia.org/wiki/Naive_Bayes_spam_filtering. Ours is overly simple, resulting in some serious drawbacks, but the basics are there.

Instructions. Download `get_data.py`, put it in your hw2 folder, and run it. Create a module named `hw2.py`. Below is the spec for two classes containing 10 methods and a `main` function. Implement them and upload your module to the appropriate D2L Assignments folder.

Testing. Download `hw2_test.py` and the auxiliary files and put them in the same folder as your `hw2.py` module. Run it from the command line to see your current correctness score. Each of the 11 methods/functions is worth 9% of your correctness score. You can examine the test module in a text editor to understand better what your code should do. The test module is part of the spec. The test file we will use to grade your program will be different and may uncover failings in your work not evident upon testing with the provided file. Add any necessary tests to make sure your code works in all cases.

Documentation. Your module must contain a header docstring containing your name, your section leader's name, the date, `ISTA 331 Hw2`, and a brief summary of the module. Each function must contain a docstring. Each function docstring should include a description of the function's purpose, the name, type, and purpose of each parameter, and the type and meaning of the function's return value.

Grading. Your module will be graded on correctness, documentation, and coding style. Code should be clear and concise. You will only lose style points if your code is a real mess. Include inline comments to explain tricky lines and summarize sections of code.

Collaboration. Collaboration is allowed. You are responsible for your learning. Depending too much on others will hurt you on the tests. "Helping" others too much harms them in reality. Cite any sources/collaborators in your header docstring. Leaving this out is dishonest.

Resources.
https://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html
https://scikit-learn.org/stable/modules/feature_extraction.html
https://stackoverflow.com/questions/24647400/what-is-the-best-stemming-method-in-python
https://en.wikipedia.org/wiki/Stemming
https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html
https://en.wikipedia.org/wiki/Confusion_matrix
https://docs.python.org/3/library/collections.html

Necessary import statements:

```
from sklearn.feature_extraction import stop_words
from nltk.stem import SnowballStemmer
from sklearn.metrics import confusion_matrix
```

Also import `math`, `string`, `random`, and `os`.

Class, method, and function specifications.

`class LabeledData:`

Look at the test file in an editor. Implement the functions in the order

- `parse_line`
- `parse_message`
- `__init__`

The point of `LabeledData` objects is to hold an *N* x 1 data matrix **X** of emails (as a list of strings) and a labels vector (also a list) **y**, also of length *N*, containing `0`'s where the corresponding emails are ham and `1`'s where they are spam.

`__init__`: This instance method takes two strings with default arguments of `'data/2002/easy_ham'` and `'data/2002/spam'`, respectively. The first one is a relative path to a corpus of ham emails; the second is a relative path to a corpus of spam emails. The default corpuses will be our training data. It also takes a data matrix and a labels vector with default arguments of `None`. Name their parameters `X` and `y`, respectively. The instance variables in the objects created by this method will also be named `X` and `y`.

- If the parameter `X` is `None`, set the instance variable `X` to a list of parsed emails from the ham directory followed by the spams. In other words, you will have to traverse the filenames in those directories, pass each filename to `parse_message` (below), and append the parsed email to the list you are building. `y` should be a list of the same length as `X` containing `0`'s in the same positions as the ham emails in `X` and `1`'s corresponding to the spams. If the parameter `X` is not `None`, then instead assign the instance variables to the corresponding parameters.

`parse_message`: This instance method takes the name of a file containing an email with the following format:

- A line starting with `From` that has no colon.
- A bunch of header lines that start with a word followed by a colon, then more text.
- The subject line, which we want to grab, that starts with `Subject:`
- More header lines.
- A blank line.
- The rest of the email, from which we want to extract the text, except for header lines from previous emails, which have the format: whitespace, a word ending in a colon, more text.

Example:

```
From exmh-workers-admin@redhat.com  Thu Aug 22 12:36:23 2002
Return-Path: <exmh-workers-admin@example.com>
Delivered-To: zzzz@localhost.netnoteinc.com
Cc: exmh-workers@example.com
Subject: Re: New Sequences Window
In-Reply-To: 1029945287.4797.TMDA@deepeddy.vircio.com
Date: Thu, 22 Aug 2002 18:26:25 +0700

    Date:        Wed, 21 Aug 2002 10:54:46 -0500
    Message-ID:  1029945287.4797.TMDA@deepeddy.vircio.com
```

You can see as many examples as you like in your data folders. You will return a string containing a cleaned version of the email. Use the following steps:

- Open the email file with the following keyword arguments: `errors = 'ignore', encoding = 'ascii'`. This will strip all non-ASCII characters (for test repeatability, because special character handling can vary by OS and Python version).
- Ignore all of the header except for the subject line. (Hint: you can look for the first blank line to locate the end of the header.)
- Concatenate the tokens from the subject line to the string you're building separated by spaces except for the `Re:`'s (make sure you skip `re`: no matter what case it is).
- For each of the rest of the lines, pass the line to static method `parse_line` and concatenate a space and the returned value onto the string your building, unless the string is empty. If the string is empty and the returned value is not, set the string to the returned value. Call the static method thusly: `LabeledData.parse_line(line)`.

`parse_line`: This is a static method takes a line from an email and returns stripped line, unless it's a header line, in which case, return the empty string. To make a static method, put `@staticmethod` on the line before the method signature and do not include the `self` parameter. This is a way of including a function definition within a class that is not tied directly to specific instances.

```
class NaiveBayesClassifier:
```

`__init__`: This instance method takes a `LabeledData` object, a pseudocount with a default value of `0.5`, and a hyperparameter called `max_words` that limits the number of tokens used in classifying an email with default value `50`. To initialize your `NaiveBayesClassifier`:

- Assign the `LabeledData` object to an instance variable called `labeled_data` and the token limit to an instance variable called `max_words`.
- Assign a `SnowballStemmer('english')` object to an instance variable called `stemmer`.
- Count and store the number of spams and hams.

- Store a word count dictionary, obtained by calling the `count_words` instance method, in an instance variable called `word_probs`.
- Finally, process the word counts into estimated probabilities. Loop over the dictionary, replacing all of the frequencies with estimated probabilities according to the following formula (k is the pseudocount):

$$\frac{freq+k}{N+2k}$$

`tokenize`: This instance method takes a string representing an email and returns a set representing a vector of lowercased, stemmed tokens, excluding any stop words. Make the token vectors case-insensitive by lowercasing the email message. Replace all occurrences of "`n't`" with the empty string (the stemmer doesn't handle these well). Replace every character in `string.punctuation` with the empty string. Replace all digits with the empty string. Finally, split the resulting string, pass the individual words to the instance's `stemmer`, and add the returned value to a `set`. Create and return a `set` containing all of the stemmed tokens except those that are in `stop_words.ENGLISH_STOP_WORDS`.

`count_words`: This instance method returns a word count dictionary that maps tokens to 2-element lists. The first element is the frequency of spam emails that contain the token, the second the frequency of ham emails. Go through every email in the `LabeledData` object, tokenize it, and then loop through the resulting set of tokens incrementing your dictionary appropriately for every token in the message. Return the dictionary.

`get_tokens`: When classifying a message, we won't use every token in the message, but rather just a randomly chosen subset. This instance method takes a token vector representing an email and returns a random sample of the tokens in it. You can use `random.sample` to select a subset. The sample should be of size minimum of `max_words` and the length of the message. Select your sample from a sorted list of the keys (sorting isn't important for performance, it's just for test repeatability here).

`spam_probability`: This instance method takes a string representing an unlabeled, unclassified email and returns an estimate probability that it is spam. Tokenize the message. Return `1` if none of the tokens are keys in `word_probs` (**What does this mean we are assuming about a message when it contains no tokens we've seen before?**). Initialize variables to hold the logs of our estimated spam and ham probabilities. Get a random sample of the tokens. For each of the tokens in the sample, if the token is in the `word_probs` dictionary, update the log probability variables (this is where we're doing the joint conditional probability part of our Naïve Bayes algorithm). Turn the log probabilities into probabilities. Return:

$$\frac{spam\ probability * 0.5}{spam\ probability * 0.5 + ham\ probability * 0.5}$$

What are we assuming about the odds that a new email is spam/ham? What is strange about this formula the way it is written (code it up this

way anyway so that your result matches mine)? Why do we have `max_words`? What happens as the number of tokens we use to calculate the probability gets bigger and bigger?

`classify`: This instance method takes a string representing an email and returns `True` if our classifier estimates that probability (using `spam_probability`) that it is spam is at least 50%, `False` otherwise.

`predict`: This instance method takes an *N* x 1 data matrix (list) of email messages and returns a list of Booleans representing our predictions as to the spamminess of each message. (Just call `classify` on every member of the list).

`main`: Make a training dataset by calling `LabeledData` with no arguments. Make a testing dataset by calling `LabeledData` with the arguments `'data/2003/easy_ham'` and `'data/2003/spam'`. Make a classifier with the training data, setting `max_words` to `25`. Use the data matrix in the test data to get a predictions list. Create a confusion matrix using the test data labels and the predictions. Print it. Your confusion matrix should resemble the following:

```
[[2008  493]
 [   0  501]]
```

Make sure you remove that line before you turn in your code. Print your test's accuracy score in the following format:

```
accuracy: 83.58%
```

Include this code at the bottom of your module:

```
if __name__ == "__main__":
    main()
```

Run your code with several different values of `max_words`. What happens as `max_words` starts to get pretty big? Why?