Assignment 3—Due 03/01/2023 at 11:59 PM

Submission. Your GitHub classroom submission will look *something like* the following structure:

```
assignment3/
    naive_bayes/
        bin/
            __init__.py
            main.py
        data/
            johnson/…
            kennedy/…
            unlabeled/…
        nb/
            __init__.py
            nb.py
        utils/
            __init__.py
            load_data.py
        tests/
            data/
                docs.txt
            __init__.py
            test_naive_bayes.py
        .gitignore
        pytest.ini
        README.md
        setup.py
        environment.yml
    index/
        index.py
```

Problem 1. In this problem, you will take your Naive Bayes from Assignment 2 (I will upload my solution) and refactor your code—rewriting your code to improve the quality of the software itself without altering any external behaviors. Your work in this problem will be contained in the `naive_bayes` folder within the `assignment3` folder. I've written a template that you can follow to structure your project, but you are welcome to change it as you see fit. There is no one "right" way to architect a Python project; the way I've done it here is just a reflection of some of my preferences and a tie-in to our discussions in class. The complexity in this problem is less algorithmic or theoretical, and we instead are concerned with careful project organization and being able to import code we've created in one part of the project to some other part.

Now, let's take a closer look at each of the files under the `naive_bayes` directory. The first we see is the folder `bin`; let's return to that at the end. We start with `data`, which contains our raw text data. Next is the `nb` directory, where we house all the code for our `NaiveBayes` class. The file `nb.py` is a Python module that should contain the functions that deal with the training and testing of the Naive Bayes classifier. The class and methods you create in this file will be used in other parts of the project; for instance, you'll import your class to test it within the `tests` directory.

Within the `utils` directory, we'll place our code for pulling text from the raw text files. Since you'll mainly be copy-pasting code and wrapping it in classes, this folder should reasonably contain just a single function like the one you've already written to build the dataframe from the text files. Remember, in Assignment 2, we performed whitespace tokenization as a subroutine within our training and test code. In other words, looking at the code, we see tokenization was treated as simply as `for token in text.split()`, which splits along whitespace. That's fine for our purposes here, too, but it's probably the case that you'd want a separate `Tokenizer` class, perhaps under the `utils` directory (since that's where we put other data preprocessing code).

The next part of the project is the `tests` directory, which you'll read about in Part (B) of this problem. You'll also notice the `pytest.ini` file, which is described in Part (B), as well.

After that, we'll use a `.gitignore` file to tell git to ignore any `__pycache__` files, which are compiled files from your Python code automatically generated by Python to make your code start a little faster. We'll also want to tell git to ignore the `data` directory at the top level. It's a good practice to avoid including large data files when you push to git. So, there should be two lines in this `.gitignore` file, one is __pycache__ and the other is `data/`.

Your `README.md` file should contain a high-level description of what your code does, as well as an explanation of how to use it.

The `setup.py` file should orchestrate the correct installation of your project. You can copy and paste the code at the bottom of this assignment for your `setup.py`.

The `environment.yml` file should be the same as the `environment.yml` file we use for this class.

So far, we've separately looked at all the moving pieces. Now the question is how do we tie them all together? That's done within the `bin` folder, where we store the command-line interface (e.g., using `argparse`) in our `main.py` module.

Now that we've taken an overview of the project structure, let's be more precise about a few issues below.

(A) Following the examples of Python project structuring, take the Naive Bayes code you wrote to form a `NaiveBayes` class. This class should contain two methods, a `train` method for fitting the model and a `test` method for running the model against unseen data. A useful strategy might be reading the source code within the `scikit-learn` library for the `GaussianNB` class. Note that this class features much more functionality than we're interested in here, but the spirit and organization of the class might be a helpful reference. Other examples from class should be helpful, too. Your class should be created in the file called `nb.py`. For this problem, it's important to recognize the hard work is already done—you just need to organize the training and test code from Assignment 2 into methods of the `NaiveBayes` class.

(B) Once you've written your class, you should write some unit tests, which are pieces of code that check code that you've written elsewhere. In particular, create a `tests` directory within the `naive_bayes` directory. Within this directory, create a python module called `test_naive_bayes.py` with a function named `test_nb_class`. The `pytest` framework looks for filenames that start or end with `test`, so the filename `test_naive_bayes.py` is not arbitrary. We want to create three unit tests within this file. The first two should test whether the priors and likelihoods created by your class match what you'd expect. The third should check whether the sum along the axis for each class should sum to 1. Let's think about this for a moment. Along a particular axis corresponding to a given class, every element in that dimension corresponds to a probability estimate for a word in that class. If we add all of these probability estimates together, that sum over all words for that particular class should be 1.

To make testing your work tractable, you'll want a very simple test case that you can work out by hand and then verify your work with `pytest`. For this problem, use the following documents from Jurafsky and Martin Section 4.3 in your testing:

document 1 (class 1): Chinese Beijing Chinese
document 2 (class 1): Chinese Chinese Shanghai
document 3 (class 1): Chinese Macao
document 4 (class 2): Tokyo Japan Chinese

So, within the `tests` folder, we have another folder `data` to store data we've created for the sole purpose of testing our code. In that data folder should be a raw text file called something like `docs.txt` containing the four "documents" above. This is precisely how we perform testing in real projects—creating small, understandable test cases that can be assessed directly. The point of this testing is to make sure your Naive Bayes code is running as expected. For this example, we're not writing tests to verify the `load_data.py` module is creating the `DataFrames` correctly, but you'd check that functionality as well in a real project. So, what you can do is just directly put that text data into a `DataFrame` and run your Naive Bayes code from there. Your priors and likelihoods should match those in the textbook. Note that to import your class in the testing folder, you will have to install your `nb.py` as a package. Then, you can import from this file in your `test_naive_bayes.py` file.

The `pytest.ini` file is a configuration file that tells `pytest` how to run its tests. The contents of that file should be

```
[pytest]
testpaths = tests
```

This tells `pytest` to search the `tests` directory when it's looking for tests. Once you've written your testing code, run the tests by executing `pytest tests/` in the root directory. If your code and example are written correctly, the three tests should all pass.

(C) Let's put everything together. In `main.py`, use `argparse` just as we did in Assignment 2, now with your OOP design. This should come together relatively quickly if your class works. In other words, you should perform the same steps as in Assignment 2—but this time using the NaiveBayes class.

(a) Ask the user for an input directory to load the data from.
(b) Build the DataFrame from the text within the data subfolders.
(c) Produce estimates for the priors and likelihoods.
(d) Run the `train` method in your `NaiveBayes` class.
(e) Run the `test` method in your `NaiveBayes` class.
(f) Print out the predictions.

Problem 2. In this problem, we'll write a Python program to build an inverted index. As you work through each of these individual problems, you are not required to structure this as a Python project, in contrast to what you did in Problem 1. So, all of this code can be stored in a single `index.py` file. This problem is adapted from a similar problem by Doug Oard. Your file should be run using

```
python index.py [path_to_collection]
```

(A) First, read in the documents sampled from the *20 Newsgroups* dataset. The file to download from Canvas is called `20newsgroups-initial.xml`. This is a dataset that uses newsgroup categories as labels. Then, write code to parse the XML content. You may want to use an XML parser, such as `ElementTree` (`import xml.etree.ElementTree as ET`). There's another file called `20newsgroups-additional.xml` that you can index after you have a working program. The point is to verify that your program can handle other documents in the same exact format (error checking that you haven't written your code specifically to the data at hand or introduced some idiosyncratic processing).

(B) In this part of the problem, we'll write a tokenizer to tokenize the extracted text. You should write an abstract base class (ABC) that acts as the interface to your tokenizer. As the ABC for your concrete tokenizers that you are about to implement, what are the universal properties of a tokenizer that you should include in your ABC? Now, you will write a few different tokenizers. One should simply tokenize by whitespace, one should use an off-the-shelf tokenizer, and a final one should tokenize text into n-grams, where *n* is a parameter of that tokenizer function. Once you've tokenized the text, take the output of that process and perform basic normalization by lowercasing and removing punctuation:

```
tokenizer_string.lower().strip(string.punctuation)
```

(C) Stem the tokens using the Porter Stemmer from `nltk`.

(D) At this point, we're ready to build the index. Each postings list should contain the (docid, term frequency) pairs we saw in class. Use the following pseudo code to write your indexer:

```
index = dict()
for text_body in xml: # for the raw text you're pulling out of the xml
```

```
            list_of_tokens = tokenize(text_body)
            list_of_stemmed_tokens = stem(list_of_tokens)
            term_frequency = Counter(list_of_stemmed_tokens)
            for token in term_frequency:
                 if token in index:
                      index[token] += [docid, tf[token]]]
                 else:
                      index[token] = [docid, tf[token]]]
```

Notice that we haven't serialized the index—that is, it hasn't been compressed in any way. We aren't too concerned about that here, but real indexes take care to find a balance between reducing space and ease of decompression.

(E) Using your index, what document ids are returned for the query "system"? Note that this search should be case-insensitive, so that documents containing "system" and "System" are both returned. Note that in this question, we are not concerned with ranking the documents in any order. We're simply returning the documents that contain the query terms, but we're checking for the presence of those terms using our inverted index.

(F) Using your index, what document ids are returned for the query "compatibility"?

(G) Using your index, what document ids are returned for the query terms "system" and "compatibility"? There should be at least one document returned.

For Problem 1: Copy and paste the following code for your `setup.py` file. Only change the regions marked in bold.

```
import setuptools

with open('README.md', 'r') as f:
    long_description = f.read()

setuptools.setup(
    name=naive_bayes,
    version='0.0.1',
    author='your name',
    author_email='your email',
    description='brief description',
    long_description=long_description,
    long_description_content_type='text/markdown',
    packages=setuptools.find_packages(),
    python_requires='>=3.6',
)
```

Grading Rubric.

Problem	Maximum Score	Your Score	Comments
Problem 1	4		
Problem 2	2		
Total	6		