Final Project Report

Naive Bayes Spam Filter

1 Design and Implementation

1.1 corpus.py

This part aims to access to and preprocess the raw emails.

Firstly, the tokenizer is implemented. As a clever tokenizing tool, **NLTK** is used here to segment the sentences and tokenize words and all the tokens are stored in a list. Additionally, the **stops words**, whose list comes from NLTK, and all the **punctuation**, whose list was created by myself, are **filtered out**, due to the reason that they seldom convey meanings and will **negatively affect** the accuracy.

Secondly, to deal with a single email, the first step is to read in the file and extract the main body of the content. Briefly, traverse each line in the file give the path, if it starts with **"subject" or "cc"**, it's certainly not the main body and thus can be ignored. After traversing each line, only main body lines are kept and combined together as a string, and finally tokenize it to a list.

Thirdly, training the spam filter requires a large amount of data, thus each email in a large data set needs to be considered. Therefore, **os** is imported to assist **listing the files** in the directories and **generating the path** to each file automatically, so that the function *read_file* can be applied to each file to get the token list. Furthermore, the token list and the matched label that is obtained by the **ending of file name** form a **training instance tuple**. Each instance tuple is then appended to the **training list**.

1.2 nb.py

This module is responsible for training and testing spam filter.

In the *train* function, a set called **"vocab"** is created to hold all the unique words that appear in the training set. A dictionary called **"email_dict"** maps each word to a list of all the emails that contain that word. Then loop over each email in the training set, add its words to the vocab set and add the email to the email_dict for each word it contains. Furthermore, loop over each word in the vocab set, use a list comprehension to count the number of spam and ham emails containing that word. Then use those counts to calculate **P(w_i|"spam")** and **P(w_i|"ham")**, and add those probabilities to the global variables.

The *classify* function then is used to calculate the **spam score** of each input email (list of tokens). Because *train* function has learned the **statistics of training set**, so this part only finds out the **P(w_i|"spam")** and **P(w_i|"ham")** and multiply them. But the tricky thing is that those words never appear in the training data don't have probabilities, so I assign the probability value of 0.5 to them.

1.3 main.py

This module is designed for **interacting with users**.

There are basically three functionalities. Firstly, the users are able to **train a spam filter** from a dataset. So the first step is to **require a path to the training set** from the user and then use *read_dataset* and *train* functions to train it.

The second function is to allow users to do the **classification**. In this case both **single email classification** and **batch classification** should be considered. When classifying just one email, the user should provide the **path to this email** and then call *read_file* and *classify* functions to obtain **spam score** and its **classified label**. The output format is "file name+spam score+class label", so **os.path.basename(file_path)** is used to get the **file name** from its path. Apart from printing the result,

another option is to **print and save it to a file**. Thus, the program will create a file named **"spamfilter.txt"** and write the result into it and save it in the same path with the **main.py** file.

The process is quite similar for the batch classification, except that **os.listdir(path)** should be used to traverse all of the files in the given directory.

Especially, I make the classifying part an endless circle, because it can avoid the situation that the user want to **reuse the testing function** and they need to wait long for the training process.

Finally, all of the above functionalities are accompanied by **user instructions**.

2 Documentation

This part is for instructing the users of spam filter to run and use the program. To be concise, follow the steps.

STEP 1: Make sure that you've downloaded NLTK library.

Otherwise, use `pip install nltk` to install it and include `nltk.download('punkt')` as well as `nltk.download('stopwords')` in the first lines of **corpus.py** file.

STEP 2: Open and run main.py.

Nothing to extend.

STEP 3: Input the path to the training set.

```
Please enter the path to the training data. Example: E:\Codes\Programming for CL\project\data\train.
E:\Codes\Programming for CL\project\xiaopiliang\
```

The program assumes **by default** that the training data directory contains **two directories**: **ham and spam**, each containing multiple files representing ham and spam emails respectively.

STEP 4: Decide how many email(s) you want to classify.

```
What do you want to classify? (Enter A, B or E)
(A) an email
(B) a batch of emails
(E) terminate the whole program
```

Enter A, B or E according to your demand. If your entry is incorrect, you'll see the reminder.

STEP 5: Enter the path to the testing file (directory).

When you choose to classify one email, just give the **path** to that email.

When you want to classify a batch of emails, input the **path to the directory** instead, but notice that all the training files should be **in one directory**, which means there are no other directories within the training directory. If your entry is incorrect, you'll see the reminder.

```
Please enter the path to a testing email:
E:\Codes\Programming for CL\project\data\test\ham\0688.2000-03-22.farmer.ham.txt
```

STEP 6: Choose if you want to save the result to a file and check the result.

Choose A or B. If "yes", then you will find the result file named **"spamfilter.txt"** in the same directory with your main.py file. If no, the result will be **printed directly**. This part suits for both single and batch classification.

```
Do you want to save the result to a file?
(A) yes
(B) no, only print it
A
Result saved. File name: spamfilter.txt.
```

```
Do you want to save the result to a file?
(A) yes
(B) no
B
0688.2000-03-22.farmer.ham.txt 0.42944830601687367 ham
```

STEP 7: Terminate or continue to test other emails.

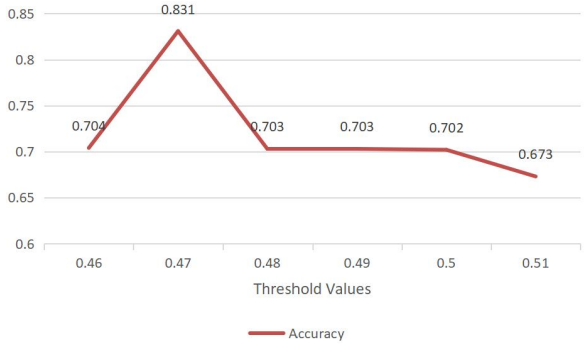
The program won't end by itself, when everything is done, choose **E** to end the program, but remember that next time it will need to train the spam filter again.

3 Extension

3.1 Experiment with different spam score threshold values

I tested the spam filter with different threshold values. The results are shown in the table below.

Threshold Values	0.46	0.47	0.48	0.49	0.5	0.51
Accuracy	0.704	0.831	0.703	0.703	0.702	0.673



Accuracy is defined as the number of correct classification over all classified emails. The calculation of accuracy with code is shown in the following picture.

```
f = open('rs1.txt', 'r', encoding='utf-8')
lines = f.readlines()
line_num = 0
correct = 0
for line in lines:
    line_num += 1
    if line.split()[0].split('.')[-2] == line.split()[-1]:
        correct += 1
print(correct/line_num)
f.close()
```

According to the experiment data it can be seen that the accuracy peaks at 83.1% with threshold value of 0.47.

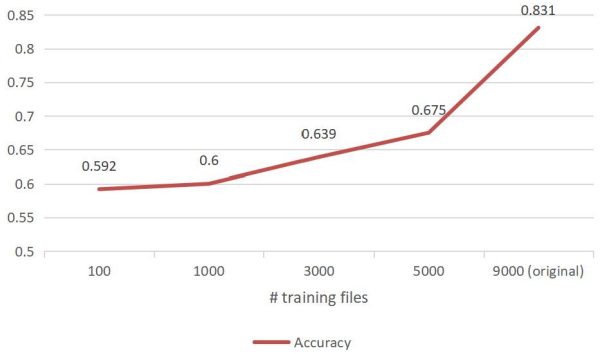
3.2 Filter stop word

See *1 Design and Implementation corpus.py*.

3.3 Change the size of the training data

A certain number of training files are selected from the whole training set and see whether the size of the training data will have an influence on the accuracy. The threshold value is set to be 0.47.

# training files	100	1000	3000	5000	9000 (original)
Accuracy	0.592	0.600	0.639	0.675	0.831



The size of the training data also has a huge impact on the accuracy. It's obvious that the bigger size of the training data can usually yield better results, the reason is probably that the the variety of training examples can train a more "experienced" model.