Team Dev Hour
Jerel Santos
Caden Silberlicht
Drew Price
437 Final Project

I. Google API

(Code Block 1)

```
# Mount drive

from google.colab import drive

drive.mount('/content/drive')
```

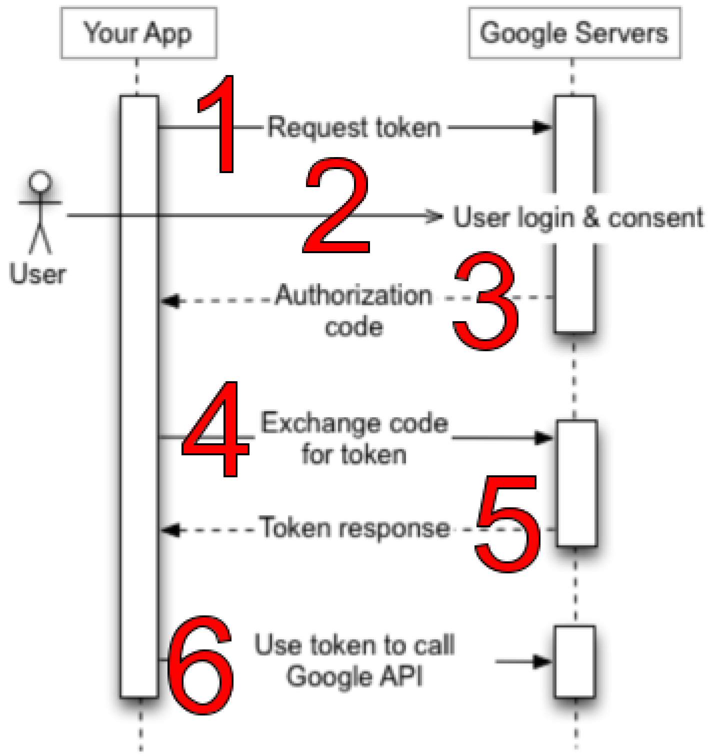
The purpose of Code Block 1 is to mount our Drive so that we can get the credentials .json file given to us when creating our Google Cloud Project, which allows us access to Google's API.

(Code Block 2)

```
# Install correct versions so that OAuth Flow has run_console()
method

!pip install 'google-api-python-client==1.7.2'

!pip install 'google-auth==1.8.0'

!pip install 'google-auth-httplib2==0.0.3'

!pip install 'google-auth-oauthlib==0.4.1'

!pip install nltk==3.5
```

This next code block, Code Block 2, has us revert certain libraries to previous versions so that we have access to certain methods. Specifically, 'flow.run_console()' used in Code Block 3.

On Google API's documentation, the following workflow was given in the following image we'll refer to each red number as a step in the workflow process:

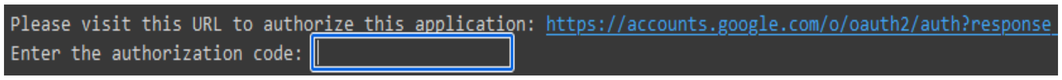


(Code Block 3)

```
      # Log into gmail to scrape it for email data

      from google_auth_oauthlib.flow import InstalledAppFlow

      from googleapiclient.discovery import build



      # Get the credentials json file and establish scope

      credentials =
'/content/drive/MyDrive/client_secret.apps.googleusercontent.com.json'

      scope = ['https://www.googleapis.com/auth/gmail.readonly']



      # Use flow and scope to authorize gmail

      flow = InstalledAppFlow.from_client_secrets_file(credentials, scope)

      creds = flow.run_console()
```

In the above code block, we first import the libraries necessary for completing authorization (they are reverted back to older versions because of Code Block 2). After this, we get the credentials.json file from our mounted Drive from Code Block 1. We then specify the scope, which for the purpose of this project would only be ‘gmail.readonly’.

After this, we then verify our project using the credentials we grabbed from the .json file. After verifying, we can run ‘flow.run_console()’, which requests a token from Google's servers and completes Step 1 of the workflow. The following appears in the terminal:

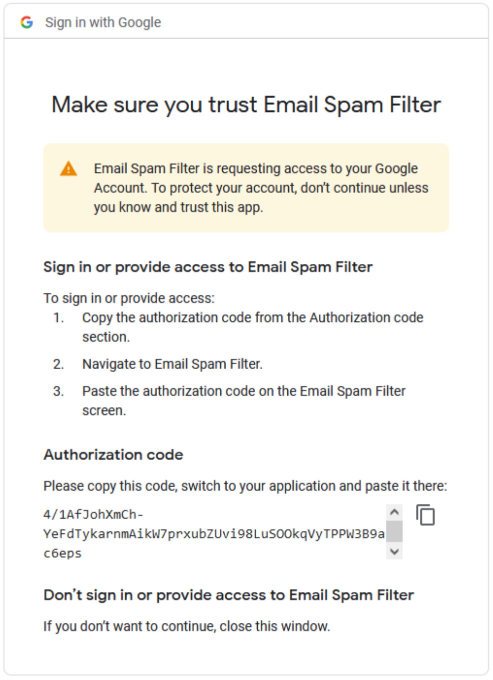


```
Please visit this URL to authorize this application: https://accounts.google.com/o/oauth2/auth?response
Enter the authorization code:
```

When clicking the link, you will be prompted to login, where the user will have to login and consent to the scopes of the project (Step 2 of the workflow).

NOTE: Because we couldn’t get a license for our Google Cloud project, our project is in “testing”, so only specific users can login. More details about this is in our project write-up.

After the user logs in, they will be given an authorization code that can be pasted in the terminal back in the Collab notebook (Step 3 and 4 of the workflow):



Sign in with Google

Make sure you trust Email Spam Filter

Email Spam Filter is requesting access to your Google Account. To protect your account, don't continue unless you know and trust this app.

Sign in or provide access to Email Spam Filter

To sign in or provide access:
1. Copy the authorization code from the Authorization code section.
2. Navigate to Email Spam Filter.
3. Paste the authorization code on the Email Spam Filter screen.

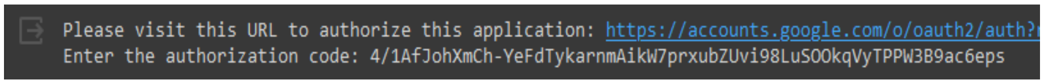
Authorization code

Please copy this code, switch to your application and paste it there:

```
4/1AfJohXmCh-
YeFdTykarnmAikW7prxubZUvi98LuSOOkqVyTPPW3B9a
c6eps
```

Don't sign in or provide access to Email Spam Filter

If you don't want to continue, close this window.



```
Please visit this URL to authorize this application: https://accounts.google.com/o/oauth2/auth?
Enter the authorization code: 4/1AfJohXmCh-YeFdTykarnmAikW7prxubZUvi98LuSOOkqVyTPPW3B9ac6eps
```

Once completed, the token response (Step 5 of the workflow) is stored in the 'creds' variable defined in the last line of Code Block 3. Now any future call to Google's API in the next parts of the code will fulfill Step 6 of the workflow.

II. Retrieving Emails

Once we are connected to a user's Gmail, we will next gather non-SPAM emails using the following operations, as well as finding SPAM emails. We then will combine all the messages to one list and shuffle. In order to shuffle, we use the **random** library from python.

```
service = build('gmail', 'v1', credentials=creds)

# Call the Gmail API to find non-SPAM and SPAM emails
results = service.users().messages().list(userId='me', maxResults=100).execute()
messages = results.get('messages', [])
results = service.users().messages().list(userId='me', labelIds=['SPAM'], maxResults=50).execute()
spam_messages = results.get('messages', [])

#combine SPAM and non-SPAM to one list and randomize order
messages = messages + spam_messages
random.shuffle(messages)
```

(Code block 4)

III. Cleaning Data

Our next step is to clean our data. We are going to start by importing all dependencies:

```
#breaking down emails
import numpy as np
import pandas as pd
import nltk
import string
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

#download necessary nltk dependencies
nltk.download("stopwords")
stop_words = set(stopwords.words("english"))
```

(Code block 5)

The **nltk** library, or natural language toolkit, must be downloaded to tokenize our text. We also import **string** for a couple operations on strings to be done. **numpy** is imported to later make a matrix to pass to our NB classifier.

We need to create some variables to store data:

```
x = []
y = []
word_token={}
itr = 0
```

(Code block 5)

x stores the IDs of the tokens in lists for each message, **y** indicates with a 1 or 0 if a message is SPAM or not, **word_token** will provide the corresponding IDs of previously detected words. **itr** will be used to create unique IDs.

Our next step is to iterate through each message and store data to be moved to the ML algorithm. We will first extract the text from the message with the following commands:

```
for message in messages:

  #extract the contents of the message
  msg = service.users().messages().get(userId='me', id=message['id']).execute()
  text = msg['snippet']
```

(Code block 5)

(In the loop) After this, we will take raw text, and use the **string** library to remove all non-alphabetic symbols from the text, then make all letters lowercase.

```
text = "".join([i.lower() for i in text if (i.isalpha() or i == " ")])
words = word_tokenize(text)
```

(Code block 5)

Once we have done this, we can tokenize the text, where **words** is a list of all tokens.

(In the loop) To populate **x**, we must loop through each word in **words**. We check if a word is a stopword (words providing no value to the classifier) or has a length of less than 2. If this is the case, they will not be accepted.

```python
#loop over each word in the tokenized message
w = []
for word in words:
  #accept words only if they have a length of 2 or greater and not stopword and append
  if word not in stop_words and len(word) > 2:
    #check if the word is already assigned an ID
    if word not in word_token.keys():
      #assign an ID to the word
      word_token[word] = itr
      itr+=1
    w.append(word_token[word])
x.append(w)
```

(Code block 5)

Once accepted, we check if the word is already in the **word_token** dictionary. If it is not, we will add the word to **word_token** with a new ID and increment **itr**. After, the ID corresponding is appended to a temporary list, **w**, and once the loop is executed, we will append **w** to **x**.

Below we will use the **.get** function to determine if the message is SPAM, and append the result to **y**.

```python
#append a value indicating if a message is SPAM or not
if 'SPAM' in msg.get('labelIds', []):
  y.append(1)
else:
  y.append(0)
```

(Code base 5)

The final step to cleaning and preparing our data is creating a bag-of-words. We will first create a list to hold the rows of our numpy arrays. Iterating through each **tokens** in **x**, we will create a numpy array of length **itr** and set all values to zero. For each ID in **tokens**, we add 1 to that index of the numpy array.

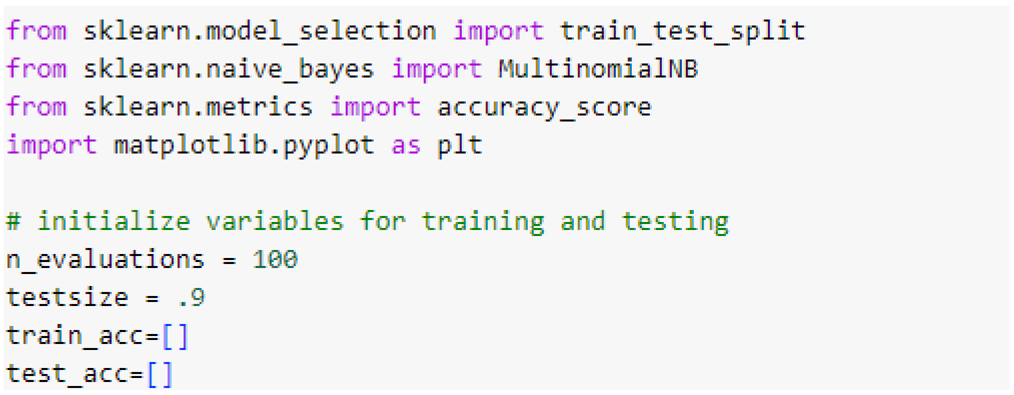
```python
#creating bag-of-words:
to_numpy = []
for tokens in x:
  arr=np.zeros(itr)
  for token in tokens:
    arr[token] +=1
  to_numpy.append(arr)

#stack rows to create a numpy matrix
bag_of_words = np.row_stack(to_numpy)
```

(Code base 5)

At the end of the loop, we appended to **to_numpy** and finally use **.row_stack** from the **numpy** library to convert **to_numpy** to a numpy matrix.

IV. The Naive Bayes Classifier

The last step of our project is to split the parsed data into training and testing data to be run in a Naive Bayes Classifier. We start this process by importing all necessary libraries:

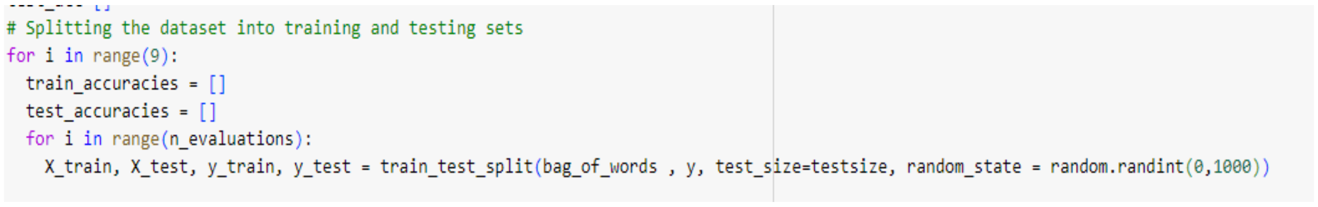


```
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt

# initialize variables for training and testing
n_evaluations = 100
testsize = .9
train_acc=[]
test_acc=[]
```

(code block 6)

Each of these imports us used for training the model with Naive Bayes. The **train_test_split** import is used to split the data that was collected in the previous section into training and testing data. The **MultinomialNB** import is a function from sklearn that is used to calculate the Naive Bayes algorithm without us having to manually create a function for it. The **accuracy_score** is used to find the accuracy of the testing data and **matplotlib** is for graphing our results. We then initialize the amount of evaluations we want our model to run along with the testing size and lists to store the training and testing accuracy.

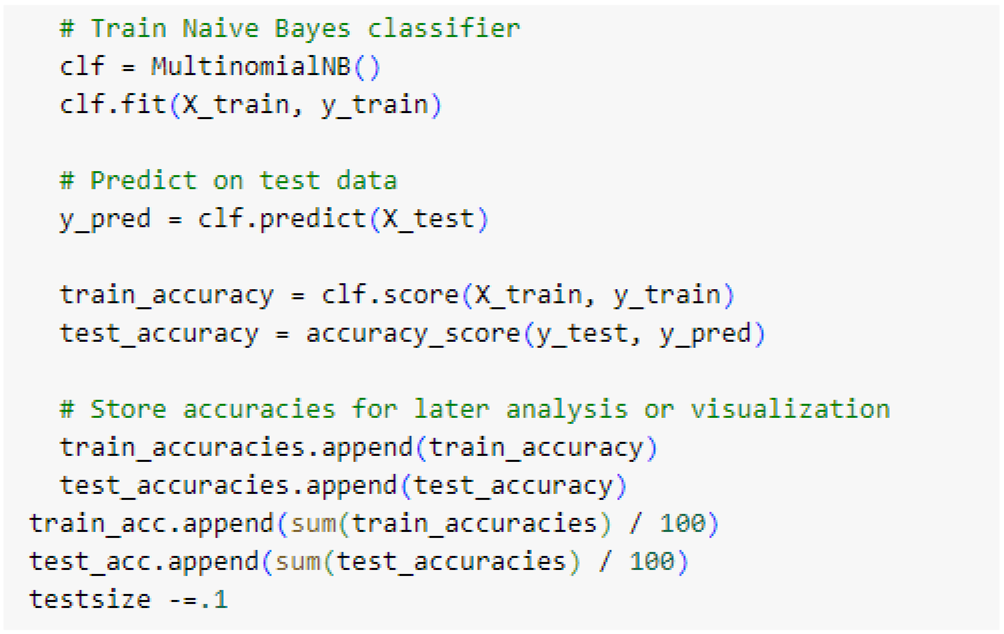
The next step is to start the process of training our data. We first create a loop that will iterate through nine times. This is because we decided to start by using only 10% of the data for training and incrementing up until we are using 90% training and 10% testing data. We then start the splitting of training and testing data that will iterate n_evaluation times (100 times).



```
# Splitting the dataset into training and testing sets
for i in range(9):
  train_accuracies = []
  test_accuracies = []
  for i in range(n_evaluations):
    X_train, X_test, y_train, y_test = train_test_split(bag_of_words , y, test_size=testsize, random_state = random.randint(0,1000))
```

(code block 6)

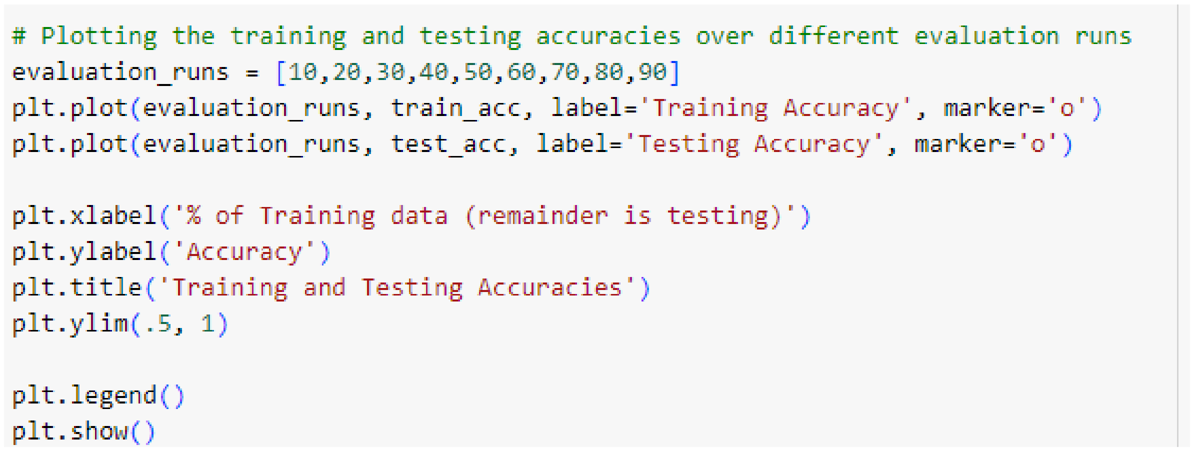
Once we have the data prepared, we turn to the Naive Bayes Classifier (NBC) to train the data. We fit the training data and then predict on the test data, storing it in y_pred. We find the training accuracy by finding the score of the training data on the NBC and find the testing accuracy by comparing the y_test to the y_pred. Then, we store the accuracies into a list for visualization analysis and subtract the size of the testing data to match the increase size of the training data.



```
  # Train Naive Bayes classifier
  clf = MultinomialNB()
  clf.fit(X_train, y_train)

  # Predict on test data
  y_pred = clf.predict(X_test)

  train_accuracy = clf.score(X_train, y_train)
  test_accuracy = accuracy_score(y_test, y_pred)

  # Store accuracies for later analysis or visualization
  train_accuracies.append(train_accuracy)
  test_accuracies.append(test_accuracy)
train_acc.append(sum(train_accuracies) / 100)
test_acc.append(sum(test_accuracies) / 100)
testsize -=.1
```

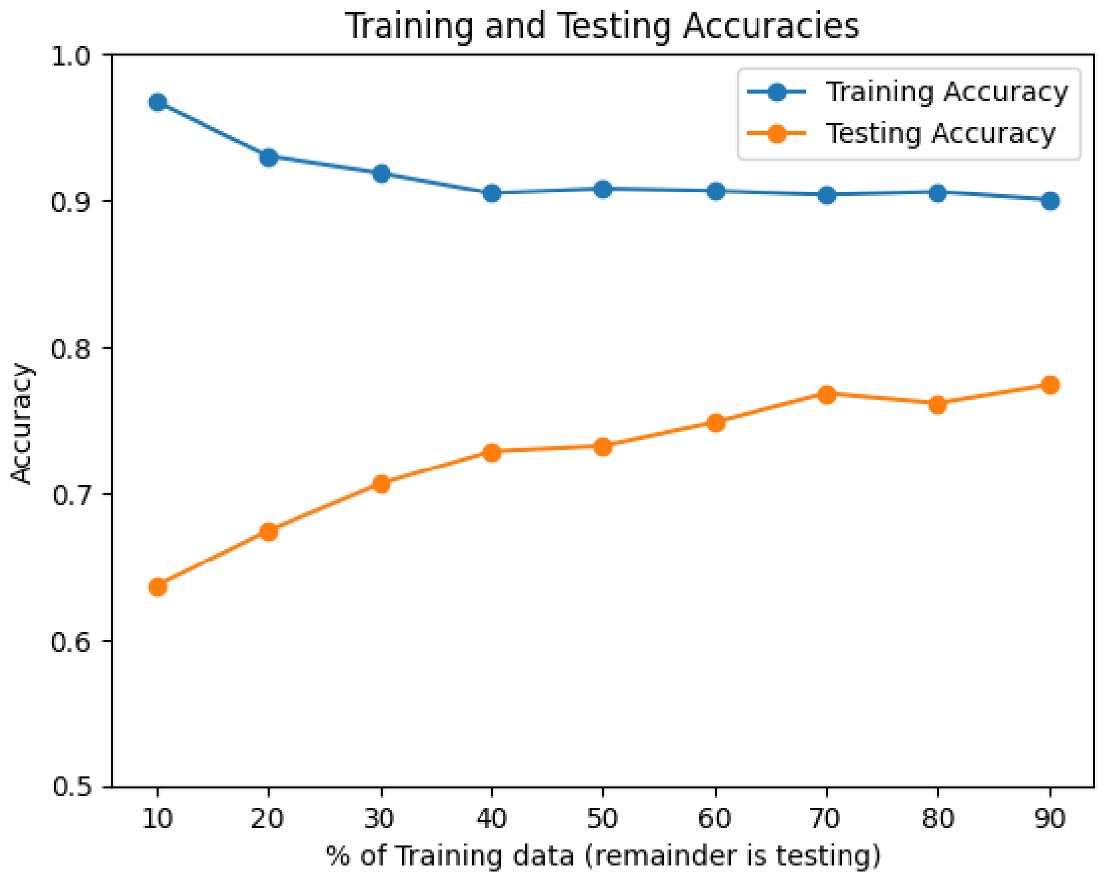
(code block 6)

Finally, we manually set the amount of evaluation runs so we are able to plot those values on the graph shown below.



```
# Plotting the training and testing accuracies over different evaluation runs
evaluation_runs = [10,20,30,40,50,60,70,80,90]
plt.plot(evaluation_runs, train_acc, label='Training Accuracy', marker='o')
plt.plot(evaluation_runs, test_acc, label='Testing Accuracy', marker='o')

plt.xlabel('% of Training data (remainder is testing)')
plt.ylabel('Accuracy')
plt.title('Training and Testing Accuracies')
plt.ylim(.5, 1)

plt.legend()
plt.show()
```

(code block 6)



(graph varies in output depending on the randomness of the data)