Assignment4

February 16, 2021

1 DAT405 Introduction to Data Science and AI

Group 6:

Name	Contribution
1. Himanshu Chuphal (guschuhi@student.gu.se)	10 H
2. Claudio Aguilar Aguilar(claagu@student.chalmers.se)	10 H

1.1 Assignment 4: Spam classification using Naïve Bayes

The exercise takes place in a notebook environment where you can chose to use Jupyter or Google Colabs. We recommend you use Google Colabs as it will facilitate remote group-work and makes the assignment less technical. Hints: You can execute certain linux shell commands by prefixing the command with `!`. You can insert Markdown cells and code cells. The first you can use for documenting and explaining your results the second you can use writing code snippets that execute the tasks required.

In this assignment you will implement a Naïve Bayes classifier in Python that will classify emails into spam and non-spam ("ham") classes. Your program should be able to train on a given set of spam and "ham" datasets. You will work with the datasets available at https://spamassassin.apache.org/old/publiccorpus/. There are three types of files in this location: - easy-ham: non-spam messages typically quite easy to differentiate from spam messages. - hard-ham: non-spam messages more difficult to differentiate - spam: spam messages

Execute the cell below to download and extract the data into the environment of the notebook – it will take a few seconds. If you chose to use Jupyter notebooks you will have to run the commands in the cell below on your local computer, with Windows you can use 7zip (https://www.7-zip.org/download.html) to decompress the data.

[1]:
```
#Download and extract data
!wget https://spamassassin.apache.org/old/publiccorpus/20021010_easy_ham.tar.bz2
!wget https://spamassassin.apache.org/old/publiccorpus/20021010_hard_ham.tar.bz2
!wget https://spamassassin.apache.org/old/publiccorpus/20021010_spam.tar.bz2
!tar -xjf 20021010_easy_ham.tar.bz2
!tar -xjf 20021010_hard_ham.tar.bz2
```

```
!tar -xjf 20021010_spam.tar.bz2
```

```
--2021-02-16 20:13:42--
https://spamassassin.apache.org/old/publiccorpus/20021010_easy_ham.tar.bz2
Resolving spamassassin.apache.org (spamassassin.apache.org)… 207.244.88.140,
95.216.26.30, 2a01:4f9:2a:1a61::2
Connecting to spamassassin.apache.org
(spamassassin.apache.org)|207.244.88.140|:443… connected.
HTTP request sent, awaiting response… 200 OK
Length: 1677144 (1.6M) [application/x-bzip2]
Saving to: ‘20021010_easy_ham.tar.bz2.3’

20021010_easy_ham.t 100%[===================>]   1.60M  3.74MB/s    in 0.4s

2021-02-16 20:13:42 (3.74 MB/s) - ‘20021010_easy_ham.tar.bz2.3’ saved
[1677144/1677144]

--2021-02-16 20:13:42--
https://spamassassin.apache.org/old/publiccorpus/20021010_hard_ham.tar.bz2
Resolving spamassassin.apache.org (spamassassin.apache.org)… 95.216.26.30,
207.244.88.140, 2a01:4f9:2a:1a61::2
Connecting to spamassassin.apache.org
(spamassassin.apache.org)|95.216.26.30|:443… connected.
HTTP request sent, awaiting response… 200 OK
Length: 1021126 (997K) [application/x-bzip2]
Saving to: ‘20021010_hard_ham.tar.bz2.3’

20021010_hard_ham.t 100%[===================>] 997.19K   972KB/s    in 1.0s

2021-02-16 20:13:44 (972 KB/s) - ‘20021010_hard_ham.tar.bz2.3’ saved
[1021126/1021126]

--2021-02-16 20:13:44--
https://spamassassin.apache.org/old/publiccorpus/20021010_spam.tar.bz2
Resolving spamassassin.apache.org (spamassassin.apache.org)… 207.244.88.140,
95.216.26.30, 2a01:4f9:2a:1a61::2
Connecting to spamassassin.apache.org
(spamassassin.apache.org)|207.244.88.140|:443… connected.
HTTP request sent, awaiting response… 200 OK
Length: 1192582 (1.1M) [application/x-bzip2]
Saving to: ‘20021010_spam.tar.bz2.3’

20021010_spam.tar.b 100%[===================>]   1.14M  2.77MB/s    in 0.4s

2021-02-16 20:13:45 (2.77 MB/s) - ‘20021010_spam.tar.bz2.3’ saved
[1192582/1192582]
```

The data is now in the three folders `easy_ham`, `hard_ham`, and `spam`.

[2]:
```
!ls -lah
```

```
total 16M
drwxr-xr-x 1 root root 4.0K Feb 16 20:13 .
drwxr-xr-x 1 root root 4.0K Feb 16 18:34 ..
-rw-r--r-- 1 root root 1.6M Jun 29  2004 20021010_easy_ham.tar.bz2
-rw-r--r-- 1 root root 1.6M Jun 29  2004 20021010_easy_ham.tar.bz2.1
-rw-r--r-- 1 root root 1.6M Jun 29  2004 20021010_easy_ham.tar.bz2.2
-rw-r--r-- 1 root root 1.6M Jun 29  2004 20021010_easy_ham.tar.bz2.3
-rw-r--r-- 1 root root 998K Dec 16  2004 20021010_hard_ham.tar.bz2
-rw-r--r-- 1 root root 998K Dec 16  2004 20021010_hard_ham.tar.bz2.1
-rw-r--r-- 1 root root 998K Dec 16  2004 20021010_hard_ham.tar.bz2.2
-rw-r--r-- 1 root root 998K Dec 16  2004 20021010_hard_ham.tar.bz2.3
-rw-r--r-- 1 root root 1.2M Jun 29  2004 20021010_spam.tar.bz2
-rw-r--r-- 1 root root 1.2M Jun 29  2004 20021010_spam.tar.bz2.1
-rw-r--r-- 1 root root 1.2M Jun 29  2004 20021010_spam.tar.bz2.2
-rw-r--r-- 1 root root 1.2M Jun 29  2004 20021010_spam.tar.bz2.3
drwxr-xr-x 1 root root 4.0K Feb 10 14:40 .config
drwx--x--x 2  500  500 184K Oct 10  2002 easy_ham
drwx--x--x 2 1000 1000  20K Dec 16  2004 hard_ham
drwxr-xr-x 1 root root 4.0K Feb 10 14:40 sample_data
drwxr-xr-x 2  500  500  36K Oct 10  2002 spam
```

###1. Preprocessing: 1. Note that the email files contain a lot of extra information, besides the actual message. Ignore that and run on the entire text. 2. We don't want to train and test on the same data. Split the spam and the ham datasets in a training set and a test set. (`hamtrain`, `spamtrain`, `hamtest`, and `spamtest`) **0.5p**

[3]:
```
#pre-processing code here
#imported python modules
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import os
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.naive_bayes import BernoulliNB
from sklearn.feature_extraction.text import CountVectorizer
from sklearn import metrics
from sklearn.metrics import confusion_matrix, classification_report
```

[4]:
```
#common file handling for shared data
ROOT = os.getcwd()
EASY_HAM_PATH = os.path.join(ROOT, "easy_ham")
HARD_HAM_PATH = os.path.join(ROOT, "hard_ham")
SPAM_PATH = os.path.join(ROOT, "spam")
```

```python
# get files for each folder
easy_ham_files = [fname for fname in sorted(os.listdir(EASY_HAM_PATH))]
hard_ham_files = [fname for fname in sorted(os.listdir(HARD_HAM_PATH))]
spam_files = [fname for fname in sorted(os.listdir(SPAM_PATH))]
print('Total Files in each directory: hard_ham_files: {:d}, easy_ham_files: {:
 ↪d}, spam_files: {:d} '.format(len(hard_ham_files),
                                          len(easy_ham_files),
                                          len(spam_files)))
#function to decode the email files
def get_emails(files, directory, message_type):
    emails  = []
    for filename in files:
      if filename is not None:
        with open(os.path.join(ROOT, directory, filename), "rb") as f:
          email = f.read()
          #decode the email with labels for each type of message
          emails.append({'message': email.decode('latin-1'),
                         'message_type': message_type })
        f.close()
    return pd.DataFrame(emails) #return the data frame
```

```
Total Files in each directory: hard_ham_files: 250, easy_ham_files: 2551,
spam_files: 501
```

[5]:
```python
#get data frames
df1 = get_emails(easy_ham_files, "easy_ham", "ham")
df2 = get_emails(hard_ham_files, "hard_ham", "ham")
df3 = get_emails(spam_files, "spam", "spam")
frames = [df1, df2, df3] # concatnating all the emails data frames
all_mail_data = pd.concat(frames)
hamtrain, hamtest, spamtrain, spamtest =␣
 ↪train_test_split(all_mail_data['message'],
                                                              ␣
 ↪all_mail_data['message_type'],
                                                               test_size=0.20,
                                                               random_state=1)
print('Total emails:',all_mail_data.shape[0])
print('Ham :: train data set{:d}, test set: {:d} '.format(hamtrain.shape[0],␣
 ↪hamtest.shape[0]))
print('Spam :: train data set {:d}, test set: {:d} '.format(spamtrain.shape[0],␣
 ↪spamtest.shape[0]))
```

```
Total emails: 3302
Ham :: train data set2641, test set: 661
Spam :: train data set 2641, test set: 661
```

###2. Write a Python program that: 1. Uses four datasets (`hamtrain`, `spamtrain`, `hamtest`, and `spamtest`) 2. Trains a Naïve Bayes classifier (e.g. Sklearn) on `hamtrain` and `spamtrain`,

that classifies the test sets and reports True Positive and False Negative rates on the `hamtest` and `spamtest` datasets. You can use `CountVectorizer` to transform the email texts into vectors. Please note that there are different types of Naïve Bayes Classifier in SKlearn (Documentation here). Test two of these classifiers that are well suited for this problem - Multinomial Naive Bayes
- Bernoulli Naive Bayes.

```
- updated with True Positive and True Negative
```

[6]:
```
#Code here
email_fit = CountVectorizer().fit_transform(all_mail_data['message'])
#Learn the vocabulary dictionary from the messages and return document-term␣
 ↪matrix.
y = all_mail_data['message_type']
# splitting the data into 20% test and 80% train data
hamtrain, hamtest, spamtrain, spamtest = train_test_split(email_fit,
                                                          y,
                                                          test_size=0.20,
                                                          random_state=1)
```

#Multinomial Naive Bayes

[7]:
```
#2 Multinomial Naive Bayes
def MNB_model(hamtrain, spamtrain, hamtest, spamtest):
  print("\nMultinomial Naive Bayes Model Classifer::")
  MNB = MultinomialNB()
  MNB.fit(hamtrain, spamtrain)
  MNB_predict = MNB.predict(hamtest)
  #binary classification, the count of true negatives , false negatives, true␣
 ↪positives and false positives.
  tn, fp, fn, tp = confusion_matrix(spamtest,
                                    MNB_predict,
                                    normalize='true').ravel()# from matrix to␣
 ↪array
  matrix = classification_report(spamtest, MNB_predict)
  print("True Positive : {:.3f} ".format(tp))
  print("True Negative: {:.3f} ".format(tn))
  #printing False rates too
  print("False Positive : {:.3f} ".format(fp))
  print("False Negative: {:.3f} ".format(fn))

MNB_model(hamtrain, spamtrain, hamtest, spamtest)
```

```
Multinomial Naive Bayes Model Classifer::
True Positive : 0.897
True Negative: 0.991
False Positive : 0.009
False Negative: 0.103
```

2 Bernoulli Naive Bayes.

[8]:
```
# 2 Bernoulli Naive Bayes.
def BNB_model(hamtrain, spamtrain, hamtest, spamtest):
  print("\nBernoulli Naive Bayes Classifer::")
  BNB = BernoulliNB()
  BNB.fit(hamtrain, spamtrain)
  BNB_predict = BNB.predict(hamtest)
  #binary classification, the count of true negatives, false negatives, true␣
 ↪positives and false positives.
  tn, fp, fn, tp = confusion_matrix(spamtest,
                                    BNB_predict,
                                    normalize='true').ravel()# from matrix to␣
 ↪array
  matrix = classification_report(spamtest, BNB_predict)
  print("True Positive : {:.3f} ".format(tp))
  print("True Negative: {:.3f} ".format(tn))
  print("False Positive : {:.3f} ".format(fp))
  print("False Negative: {:.3f} ".format(fn))

BNB_model(hamtrain, spamtrain,  hamtest, spamtest)
```

```
Bernoulli Naive Bayes Classifer::
True Positive : 0.320
True Negative: 0.989
False Positive : 0.011
False Negative: 0.680
```

a) Explain how the classifiers differ. What different interpretations do they have? 1p

- sklearn.metrics.confusion_matrix helps to evaluate the accuracy of a classifications and Normalize flag Normalizes confusion matrix over the true (rows), predicted (columns) conditions.
- In general, we can say that a better model algorithm makes good prediction should indicate::

1) always give high rates for True positive and True Negatives, and
2) low values for False Positives and False Negatives

Having said that, as per the results we got before for MNB (Multinomial Naive Bayes) and BNB (Bernoulli Naive Bayes model) models, we can say that MNB model does better on predicting true positive rates and true negatives in the case ham emails.

MNB model predictions gave less false rates, i.e it is better on predicting the spam emails as well.

On the other hand, the BNB model did poor at predicting true positives (0.32) and gave a high value for False Negative(0.68).

MNB vs BNB:

In general the differences between the two classifiers are:

Multinomial Naive Bayes considers multiple features and how many times they occur in the dataset. **Bernoulli Naive Bayes** only considers a single feature and how many times that single feature occurs and does not occur in the dataset.

The **Multinomial Naive Bayes** classifier will classify the emails based on the counts of multiple keywords(ham,spam), while **Bernoulli Naive Bayes** will only focus on the counts of a single keyword(ham or spam) and also how many times that single keyword does not occur in the mails.

This leads to the **Bernoulli Naive Bayes** classifier penalising an email that does not contain a specific feature, which leads to less true positives which can be seen in the prints.

Hence, **Multinomial Naive Bayes** classifier comes in handy When you have multiple features to worry about while **Bernoulli Naive Bayes** is best suited when there is only one feature to worry about.

2.0.1 3.Run your program on

- Spam versus easy-ham
- Spam versus (hard-ham + easy-ham).
- Discuss your results **2.5p**

[9]:
```
#Code to report results here
#Spam versus easy-ham
frames_spam_easy_ham = [df1, df3] # df1 -> easy_ham and df3->spam
all_mail_data = pd.concat(frames_spam_easy_ham)
email_fit = CountVectorizer().fit_transform(all_mail_data['message'])
y = all_mail_data['message_type']
#using the same test split as before
hamtrain, hamtest, spamtrain, spamtest = train_test_split(email_fit,
                                                          y,
                                                          test_size=0.20,
                                                          random_state = 1)

#MNB
print("Spam versus easy-ham Rates")
MNB_model(hamtrain, spamtrain,  hamtest, spamtest)
#BNB
BNB_model(hamtrain, spamtrain,  hamtest, spamtest)


print()
#Spam versus (hard-ham + easy-ham)
print("Spam versus (hard-ham + easy-ham Rates)")
# df1 -> easy_ham and df3->spam and df2->hard_ham
frames_spam_easy_ham_hard_ham = [df1, df2, df3]
```

```
all_mail_data = pd.concat(frames_spam_easy_ham_hard_ham)
email_fit = CountVectorizer().fit_transform(all_mail_data['message'])
y = all_mail_data['message_type']
hamtrain, hamtest, spamtrain, spamtest = train_test_split(email_fit,
                                                          y,
                                                          test_size=0.20,
                                                          random_state = 1)

#MNB
MNB_model(hamtrain, spamtrain,  hamtest, spamtest)

#BNB
BNB_model(hamtrain, spamtrain,  hamtest, spamtest)
```

```
Spam versus easy-ham Rates

Multinomial Naive Bayes Model Classifer::
True Positive : 0.935
True Negative: 0.998
False Positive : 0.002
False Negative: 0.065

Bernoulli Naive Bayes Classifer::
True Positive : 0.565
True Negative: 0.996
False Positive : 0.004
False Negative: 0.435

Spam versus (hard-ham + easy-ham Rates)

Multinomial Naive Bayes Model Classifer::
True Positive : 0.897
True Negative: 0.991
False Positive : 0.009
False Negative: 0.103

Bernoulli Naive Bayes Classifer::
True Positive : 0.320
True Negative: 0.989
False Positive : 0.011
False Negative: 0.680
```

#Discussion

- Similar to previous discussion and based on results from the previous cell, we can say that the **Bernoulli Naive Bayes model** performed better this time at predicting both ham (higher True Positive : 0.565) and spam (lower False Negative: 0.435) in case Spam versus easy-ham : probably because the model got higher *proportion* of the training data which was spam emails and got less ham message-type to train on.

- Comparing **Multinomial Naive Bayes model**, which gave us better rates before as well and again this time there was not much of the difference between comparing rates in spam vs. easy-ham and Spam versus (hard-ham + easy-ham Rates) and we got slightly better TP and TN rates in case Spam versus easy-ham Rates (probably because of the same reason as mentioned in the previous explanation).
- Also, seems like when both the models only got to train on the easy-spam, all the rates got somewhat better, hard-ham are non-spam messages which more difficult to differentiate. Therefore, probably the models performed better in differentiating with Spam versus easy-ham scenario.

3 4. To avoid classification based on common and uninformative words it is common to filter these out.

a. Argue why this may be useful. Try finding the words that are too common/uncommon in the dataset. **1p**

```
[10]: #find the words that are too common
def count_common_uncommon_words(message, n=None):
  vector = CountVectorizer().fit(message) #fit the dataset
  matrix_of_words = vector.transform(message)
  # each word divided into columns for each row(file in the dataset) with their␣
 ↪occurence in each file.
  sum_of_words = matrix_of_words.sum(axis=0)
  # sum for each column(each occurence for a word in all files)
  frequency_of_words = [(word, sum_of_words[0, index])
                        for word, index in vector.vocabulary_.items()]
  frequency_of_words_sorted = sorted(frequency_of_words, key=lambda x: x[1],␣
 ↪reverse=True)
  # sort the list of occurences for each word in descending order
  return frequency_of_words_sorted[:n]

# COMMON WORDS
print("list of the 10 most common words:")
#10 most common words in messages
common_words = count_common_uncommon_words(all_mail_data['message'], 10)
#loop through the most common words and print them out with their nr of␣
 ↪occurences
for (word, frequency) in common_words:
  print(word, frequency)

print("\n")

# UNCOMMON WORDS
print("list of the 10 most uncommon words:")
```

```
uncommon_words = count_common_uncommon_words(all_mail_data['message'])
#loop through the most uncommon words and print them out with their nr of␣
 ↪occurences
for (word, frequency) in uncommon_words[:-11:-1]: # take the 10 last elements␣
 ↪backwards
  print(word, frequency)
```

```
list of the 10 most common words:
com 69898
the 40824
to 38179
http 34048
from 28715
td 28399
2002 28275
3d 25415
for 23845
net 22839


list of the 10 most uncommon words:
bm7 1
5999 1
xbr 1
bowled 1
vascular 1
cardio 1
aranthopanacis 1
pseudoslellariae 1
corni 1
mouton 1
```

In General:

By filtering out these words it could lead to better predictions since we are cutting the amount of data and training the model on more relevant words. Common words can be hard to classify since they appear in almost every mail and can mislead the calculation.

Problem with uncommon words:

When the training set contains an uncommon word that only appear in one message type(ham emails) and no times in the other one(spam emails). Then the model would be trained to interpret this uncommon word as spam in this case.

Problem with common words:

Having alot of common words that appear in almost every email makes it harder to classify an email. Therefore it is a good idea to filter these out because they could otherwise mislead the calculation.

b. Use the parameters in Sklearn's `CountVectorizer` to filter out these words. Update

the program from point 3 and run it on your data and report and discuss your results. You have two options to do this in Sklearn: either using the words found in part (a) or letting Sklearn do it for you. Argue for your decision-making. 1p

[11]:
```
#lets go for the method were sklearn does it for us
#SPAM VS EASY-HAM
frames_spam_easy_ham = [df1, df3] # df1 -> easy_ham and df3->spam
all_mail_data = pd.concat(frames_spam_easy_ham)
#remove words that appear are in less than 10% and more than 80% of the␣
 ↪messages.
email_fit = CountVectorizer(min_df=0.1, max_df=0.8).
 ↪fit_transform(all_mail_data['message'])
y = all_mail_data['message_type']
hamtrain, hamtest, spamtrain, spamtest = train_test_split(email_fit,
                                                           y,
                                                           test_size=0.20,
                                                           random_state =1)
#MNB
print("SPAM VS EASY-HAM")
MNB_model(hamtrain, spamtrain,  hamtest, spamtest)
#BNB
print()
BNB_model(hamtrain, spamtrain,  hamtest, spamtest)
```

```
SPAM VS EASY-HAM

Multinomial Naive Bayes Model Classifer::
True Positive : 0.954
True Negative: 0.938
False Positive : 0.062
False Negative: 0.046


Bernoulli Naive Bayes Classifer::
True Positive : 0.981
True Negative: 0.893
False Positive : 0.107
False Negative: 0.019
```

[12]:
```
#SPAM VS (HARD-HAM + EASY-HAM)
#Spam versus (hard-ham + easy-ham). - Already done
print("Spam versus (hard-ham + easy-ham Rates)")
# df1 -> easy_ham and df3->spam and df2->hard_ham
frames_spam_easy_ham_hard_ham = [df1, df2, df3]
all_mail_data = pd.concat(frames_spam_easy_ham_hard_ham)
#remove words that appear are in less than 10% and more than 80% of the␣
 ↪messages.
```

```
email_fit = CountVectorizer(min_df=0.1, max_df=0.8).
 ↪fit_transform(all_mail_data['message'])
y = all_mail_data['message_type']
hamtrain, hamtest, spamtrain, spamtest = train_test_split(email_fit,
                                                          y,
                                                          test_size=0.20,
                                                          random_state =1)

#MNB
print("SPAM VS (HARD-HAM + EASY-HAM)")
MNB_model(hamtrain, spamtrain,  hamtest, spamtest)
#BNB
print()
BNB_model(hamtrain, spamtrain,  hamtest, spamtest)
```

```
Spam versus (hard-ham + easy-ham Rates)
SPAM VS (HARD-HAM + EASY-HAM)

Multinomial Naive Bayes Model Classifer::
True Positive : 0.918
True Negative: 0.929
False Positive : 0.071
False Negative: 0.082


Bernoulli Naive Bayes Classifer::
True Positive : 0.969
True Negative: 0.858
False Positive : 0.142
False Negative: 0.031
```

Discussion of the results

Spam Vs Easy Ham: After using Sklearn's CountVectorizer to filter out uncommon/common words, both the models performed better. **MNB:** did even better in terms of true positives and true negatives. But we see major improvements in case of **BNB:**, the Binomial Naive Bayes classifier perfomed much better in true positives and better in False positives (True Positive : 0.981, True Negative: 0.893)

Spam versus (hard-ham + easy-ham) :

We see the same improvements in this case as well for both the models. Both **MNB** and **BNB** did well in terms of true positives and true negatives but for the same reasons as explained in section **3(Discussion)**, the rates are slightly better in case of Spam versus Vs. easy-ham than in Spam versus (hard-ham + easy-ham).

- Using our own list of common words to do the filtering would be more tailor made to our case but also too specific since we compare with words that we have selected. This would take a longer time to run since we would have to loop through all emails and see if they contain a word in our list. In our method we decide how many uncommon/common words that we want to filter out and add to our list, but in sklearns algorithm you can consider all uncommon/common words that does not surpass the threshold of the parameters min_df and max_df. This results in a faster and more effective way and is also proven to work which is why we went with sklearns algorithm instead.

3.0.1 5. Eeking out further performance

a. Use a lemmatizer to normalize the text (for example from the `nltk` library). For one implementation look at the documentation (here). Run your program again and answer the following questions: - Why can lemmatization help? - Does the result improve from 3 and 4? Discuss. **1.5p**

[13]:
```
#Lemmatize the words in the messages
import nltk
nltk.download('punkt')
nltk.download('wordnet')
from nltk import word_tokenize
from nltk.stem import WordNetLemmatizer

#create a lemmatizer
class LemmaTokenizer:
    def __init__(self):
        self.wnl = WordNetLemmatizer()
    def __call__(self, doc):
        return [self.wnl.lemmatize(t) for t in word_tokenize(doc)]
```

```
[nltk_data] Downloading package punkt to /root/nltk_data…
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data…
[nltk_data]   Package wordnet is already up-to-date!
```

[14]:
```
#create a vector with the lemmatizer as the tokenizer
vect = CountVectorizer(tokenizer=LemmaTokenizer())

#SPAM VS EASY-HAM
frames_spam_easy_ham = [df1, df3] # df1 -> easy_ham and df3->spam
all_mail_data = pd.concat(frames_spam_easy_ham)

#remove words that appear are in less than 10% and more than 80% of the␣
 ↪messages.
email_fit = vect.fit_transform(all_mail_data['message'])
y = all_mail_data['message_type']
hamtrain, hamtest, spamtrain, spamtest = train_test_split(email_fit,
                                                          y,
                                                          test_size=0.20,
                                                          random_state = 1)
#MNB
print("LemmaTokenizer SPAM VS EASY-HAM")
```

```
MNB_model(hamtrain, spamtrain,  hamtest, spamtest)

#BNB
print()
BNB_model(hamtrain, spamtrain,  hamtest, spamtest)
```

```
LemmaTokenizer SPAM VS EASY-HAM

Multinomial Naive Bayes Model Classifer::
True Positive : 0.870
True Negative: 0.998
False Positive : 0.002
False Negative: 0.130


Bernoulli Naive Bayes Classifer::
True Positive : 0.639
True Negative: 0.994
False Positive : 0.006
False Negative: 0.361
```

[ ]:
```
#create a vector with the lemmatizer as the tokenizer
vect = CountVectorizer(tokenizer=LemmaTokenizer())

#SPAM VS (HARD-HAM + EASY-HAM)
# df1 -> easy_ham and df3->spam and df2->hard_ham
frames_spam_easy_ham_hard_ham = [df1, df2, df3]
all_mail_data = pd.concat(frames_spam_easy_ham_hard_ham)
#remove words that appear are in less than 10% and more than 80% of the␣
 ↪messages.
email_fit = vect.fit_transform(all_mail_data['message'])
y = all_mail_data['message_type']
hamtrain, hamtest, spamtrain, spamtest = train_test_split(email_fit,
                                                          y,
                                                          test_size=0.20,
                                                          random_state =1)
#MNB
print("LemmaTokenizer SPAM VS (HARD-HAM + EASY-HAM)")
MNB_model(hamtrain, spamtrain, hamtest, spamtest)

print()
BNB_model(hamtrain, spamtrain, hamtest, spamtest)
```

Lemmatization can help by recognizing different tenses of the same verb and thus, condensing the related words so we dont have much **variability** which would reduce the train vectors and affect the precision(more false positives) and recall(more true positives). This is done by reducing all the different forms of a word to one stem root. Words that are derived from each other will be mapped to the base word when having the same meaning.

This is also good when you want to search for a specific word through the emails, then by condensing all the words to the steem root it would facilitate our search since we would only need to search for the stem of that word.

Discussion of results

Spam vs easy-ham:

Using lemmatizer to normalize the text, in case of **MNB model**, we see the same true positive and true negative rates as comapred to Q3 and Q4 with an exception of true positive rates have gone down a bit by 4-5% with lemmatizer and a slight increase in False Negative. Even though this is a small difference, we don't really know the reason why MNB did somewhat poor with lemmatizer (probably how Lemmatization is optimizing the messages is making a difference).

In case of **BNB model**, we see that it has performed better in terms of true positives and true negatives compared to Q3 but not as good as Q4(where we used CountVectorizer to filter out common/uncommon words). We see the similar difference in terms of false negative rates.

Spam VS (Hard-Ham + Easy-Ham):

Using lemmatizer to normalize the text, in case of **MNB model** with Spam VS (Hard-Ham + Easy-Ham), we see better performance with improved true positive and true negative rates as comapred to Q3 and Q4.

In case of **BNB model** with lemmatizer, we see that it has performed better in terms of true positives and true negatives (not big difference) as compared to Q3 but not as good as Q4 (where we used CountVectorizer to filter out common/uncommon words). But we see high false negative rates as compared to Q4.

- In conclsion based on the overall comparions in Q3, 4 and 5, we can say that **BNB model** shows much better performance as we optimize the messages with CountVectorizer and lemmatizer (with CountVectorizer being a better option). On the other hand, we see almost the same performance in case of MNB model whilst doing the same optimizations.

b) The split of the data set into a training set and a test set can lead to very skewed results. Why is this, and do you have suggestions on remedies? 1p

It's because there could be an imbalance when distributing the proportions of the classes when we split the data which leads to a train and test set with different data distributions. A model that is trained on different data than the test set will lead to skewed results.

The remedy to this is to use **stratification** which will lock the distributions of classes in the train and test sets. In the train_test_split there is a parameter that is called stratify where you pass the dataset with the class label. Which in our case would look like stratify=all_mail_data['message']. By adding the last parameter to our train_test_split we will now have identical distributions and our model won't need to worry about facing the problem of validation against our imbalanced test set and give us a wrong impression of the performance, which would instead happen if we would've had a training set that were mostly spam messages while our test set were mostly ham messages.

b) Part2 :What do you expect would happen if your training set were mostly spam messages while your test set were mostly ham messages?

- if your training set were mostly spam messages while your test set were mostly ham messages, it would be probably difficult to find out weather a ham email message is ham or spam as here the used model has been only trained on features of spam emails data. We could clearly see the difference in that case if we split the test data using stratification i.e. identical distributions (as explained in the previous section). We can check True Positives and True Negative rates for the same.

```
[ ]: frames_spam_easy_ham_hard_ham = [df1, df2, df3]
     all_mail_data = pd.concat(frames_spam_easy_ham_hard_ham)
     email_fit = CountVectorizer().fit_transform(all_mail_data['message'])
     y = all_mail_data['message_type']
     hamtrain, hamtest, spamtrain, spamtest = train_test_split(email_fit,
                                                               y,
                                                               test_size=0.20,
                                                               random_state =1,
                                                               stratify=y)
     print()
     #MNB
     print("With Stratification::")
     MNB_model(hamtrain, spamtrain,  hamtest, spamtest)

     #BNB
     print()
     BNB_model(hamtrain, spamtrain,  hamtest, spamtest)
```

c. Re-estimate your classifier using `fit_prior` parameter set to `false`, and answer the following questions: - **What does this parameter mean?**

[**Answer**] Source : class sklearn.naive_bayes.MultinomialNB(, *alpha=1.0, fit_prior=True, class_prior=None)[source], here* ***fit_prior*** *bool, default=True Whether to learn class prior probabilities or not. If false, a uniform prior will be used. With* ***fit_prior*** *set to* True*, includes previously known probabilities on the dataset, which is default True and the default value is being used in this notebook so far.

When set to false, it probably uses use an uniform prior with no regularization and with which we might get some inconsistent and unexpected results. We can use some other prior but not uniform ones unless we know that the bounds are representing true constraints.

- **How does this alter the predictions? Discuss why or why not. 0.5p**

```
[ ]: #2 Multinomial Naive Bayes
     def MNB_model_2(hamtrain, spamtrain,  hamtest, spamtest):
       print("MNB_model::")
       MNB = MultinomialNB(fit_prior=False)
       MNB.fit(hamtrain, spamtrain)
       MNB_predict = MNB.predict(hamtest)
       tn, fp, fn, tp = confusion_matrix(spamtest, MNB_predict, normalize='true').
     ↪ravel()# from matrix to array
       matrix = classification_report(spamtest, MNB_predict)
```

```
  return tn, fp, fn, tp

tn, fp, fn, tp= MNB_model_2(hamtrain, spamtrain,  hamtest, spamtest)
print("True Positive : {:.2f} ".format(tp))
print("True Negative: {:.2f} ".format(tn))
#printing False rates too
print("False Positive : {:.2f} ".format(fp))
print("False Negative: {:.2f} ".format(fn))
print()


# 2 Bernoulli Naive Bayes.
def BNB_model_2(hamtrain, spamtrain,  hamtest, spamtest):
  print("BNB_model::")
  BNB = BernoulliNB(fit_prior=False)
  BNB.fit(hamtrain, spamtrain)
  BNB_predict = BNB.predict(hamtest)
  # true and false positive and negative
  tn, fp, fn, tp = confusion_matrix(spamtest, BNB_predict, normalize='true').
↪ravel()
  matrix = classification_report(spamtest, BNB_predict)
  return tn, fp, fn, tp

tn, fp, fn, tp = BNB_model_2(hamtrain, spamtrain,  hamtest, spamtest)
print("True Positive : {:.2f} ".format(tp))
print("True Negative: {:.2f} ".format(tn))
#printing False rates too
print("False Positive : {:.2f} ".format(fp))
print("False Negative: {:.2f} ".format(fn))
```

- Comparing with the results where we had the default value of *fit_prior* (True), we see that Multinomial Naive Bayes Model Classifer is performing better with improved true positive and true negative rates but for some reasons we don't see the same improvements in the Bernoulli Naive Bayes Classifer with true positives are somewhat lower than before.

4 with default, fit_prior=True

- MNB results with fit_prior=True

True Positive : 0.897 and True Negative: 0.991

- BNB results with fit_prior=True

True Positive : 0.320 and True Negative: 0.989

5 with fit_prior=False

- MNB results with fit_prior=False

True Positive : 0.94 and True Negative: 0.99

- BNB results with fit_prior=False

True Positive : 0.27 and True Negative: 0.99

d. The python model includes smoothing (`alpha` parameter), explain why this can be important. - What would happen if in the training data set the word 'money' only appears in spam examples? What would the model predict about a message containing the word 'money'? Does the prediction depend on the rest of the message and is that reasonable? Explain your reasoning 1p

- alphafloat, default=1.0 :: Additive (Laplace/Lidstone) smoothing parameter (0 for no smoothing).
- It represents the additive smoothing parameter. If you choose 0 as its value, then there will be no smoothing. If we choose a value of alpha!=0 (not equal to 0), the probability will no longer be zero even if a word is not present in the training dataset.

As alpha increases, the likelihood probability moves towards uniform distribution. Most of the time, alpha = 1 is being used to remove the problem of zero probability. The laplace smoothing is a smoothing technique that helps to tackle the problem of zero probability in the Naïve Bayes machine learning algorithm. It preferred to use alpha=1.

if in the training data set the word '**money**', and since it only appears in spam, true negative rate would increase. The predications can vary based on the training data set proportion. If we do it using stratication with equal proportion then it would matter that much. The prediction would not depend on on the rest of the message as the model is trained to differentiate based on the word "Money" here (not anything else) and this is not reasonable since we need to take into account the rest of the words in the message.

5.0.1 What to report and how to hand in.

- You will need to clearly report all results in the notebook in a clear and appropriate way, either using plots or code output (f.x. "print statements").
- The notebook must be reproducible, that means, we must be able to use the `Run all` function from the `Runtime` menu and reproduce all your results. **Please check this before handing in.**
- Save the notebook and share a link to the notebook (Press share in upper left corner, and use `Get link` option. **Please make sure to allow all with the link to open and edit.**
- Edits made after submission deadline will be ignored, graders will recover the last saved version before deadline from the revisions history.
- **Please make sure all cells are executed and all the output is clearly readable/visible to anybody opening the notebook.**