Term Project

CAP-5638 Pattern Recognition: Fall 2017

Predicting Daily Stock Market Performance
Using Sentiment Analysis of Economic News

Vishwas Gurudatta Torvi
Yevgeniy Ryasnianskiy

Abstract

The goal of this project is to develop a system that will provide insight into future stock market performance. Predicting market performance more accurately than existing systems will allow for more profitable trading and more accurate analysis of the economic situation.

Dataset

We synthesize our dataset by combining two existing datasets. The first dataset contains the text of every business or economic news story published by Bloomberg and Reuters from 2006 to 2013. The second dataset contains open and close prices of the S&P 500 index. We extract the titles from the news dataset and use only these as our document text. We do this to make the problem tractable. We expect the most important words will be contained in the headline.

News Dataset

https://github.com/philipperemy/financial-news-dataset

Market Dataset

https://drive.google.com/file/d/181VKUJnYByJIHGqGe1PNp-klBKP8N0RO/view?usp=sharing

Preprocessing

The text of stories that were published are grouped daily into documents and tokenized into meaningful vector representation of documents. To generate this vector representation, we apply ***Tf-Idf transformer*** on a title were each word has been stemmed and were stop words has been removed.

The formula that is used to compute the Tf-Idf transformer is,

tf-idf(d, t) = tf(t) * idf(d, t),
and idf(d, t) = log [n / df(d, t)] + 1

where, n is the total number of documents,
df(d, t) is the document frequency.

To remove stop words, we classified words into parts of speech using an existing classifier and removed all words that did not belong to useful parts of speech. To stem the words, we also used an existing classifier based on Snowball. Both libraries were part of NLTK for Python.

The difference between the opening and closing prices of the S&P 500 index were the basis for the category that is assigned to each document. For each day where this difference is greater than 6 basis points we classify the document as 2, where the difference is less than -6 basis points we classify the document as 0, otherwise documents are classified as 1. This translates to a classification of 2 on days when the stock market increased by a significant margin, 1 on days were there was no significant movement, and 0 when the stock market decreased by significant margin. The resultant document vectors and classes are paired daily.

For Recurrent Neural Network, we group the vectors into batches of the 4 preceding days.

Classifiers:

a) **Ada-boost of SVM:**
   Here we performed a grid search on all permutations of a set of possible parameters. Three-fold cross validation was performed to choose the best estimator on the training set. Classifying on the test data yields accuracy not more than priors.

b) **LSTM Recurrent Neural Network:**
   Here we trained the neural network using the Adam Optimizer with a binary cross entropy loss function. The network struggled to find the good fit in cases were the batch size, as defined in the preprocessing section, was much above four. The network managed to fit the train set without improving classification accuracy above priors at any point during the training on the validation data.

c) **Linear Batch Support Vector Machine:**
   Here we performed a grid search on all permutations of a set of possible parameters. Three-fold cross validation was performed to choose the best estimator on the training set. Classifying on the test data yielded accuracy about 11-12% above priors. These results will be discussed in detail in the Experimental Results Section.

d) **Deep Neural Network:**
   Here we trained the neural network using the Adam Optimizer with a binary cross entropy loss function. The network managed to fit the train set without improving classification accuracy above priors at any point during the training on the validation data.

e) **Random Forest of Decision Trees:**
   Here we performed a grid search on all permutations of a set of possible parameters. Three-fold cross validation was performed to choose the best estimator on the training set. Classifying on the test data yields accuracy about 6-7% above priors. These results will be discussed in detail in the Experimental Results Section.

f) Multinomial Bayesian Classifier:
We trained our dataset on Multinomial Bayesian Classifier. Classification Accuracy was not above priors.

Experimental Results

The only classifiers that produced result above priors were the Linear Batch Support Vector Machine and Random Forest of Decision Trees. Here we will discuss the performance of these classifiers:

For both of these classifier, a parameter search for initially performed to find the most suitable parameters for the preprocessing functions. Here we varied maximum document frequency, maximum features, N-Gram Count, whether we factored in inverse document frequency, and the type of normalization.

For the Linear batch SVM, we varied the number of training iterations, the error function, and the learning rate along with the parameters just mentioned above. A random search was performed on a distribution of potentially suitable parameter.

For the Random Forest of Decision Trees, we varied the maximum depth, number of base estimators along with the parameters just mentioned above. A grid search on all possible permutations of a set of potentially suitable parameters was performed.

Our Priors on the test data were 102, 62, 87 out of 251 for a percentage of 40.1%,24.7%, 34.6% for classes 2,1, and 0 respectively

Linear Batch SVM achieved a classification accuracy of 51.5% on the test dataset.

Best Estimators for Linear Batch SVM with which we achieved above,

Best parameters set: alpha: 0.001, iterations: 700, penalty: 'elasticnet', tf-idf normalization: 'l2', use idf: True, maximum document Frequency: 0.92, maximum features: 32000, ngram range: (1, 2).

Random Forest of Decision Trees achieved a classification accuracy of 46.22% on the test dataset.

Best Estimators for Random Forest of Decision Tree with which we achieved above,

Best parameters set: maximum depth: 5, estimators: 20, tf-idf normalization: 'l2', use idf: True, maximum document frequency: 0.75, maximum features: 80000, ngram range: (1, 2)

Code

1) Linear Batch SVM
Note: We used Pandas Library for DataFrame creation, Scikit Learn Library for Tf-Idf transforming, SGD Classification, and RandomizedSearchCV.

```
import logging
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
```

```
from sklearn.linear_model import SGDClassifier
from sklearn.pipeline import Pipeline
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint as sp_randint
from time import time
import os

print(__doc__)

logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s %(levelname)s %(message)s')

base = 'C:\\Users\\Resnus\\Documents\\My
Files\\FSU\\Current\\Pattern\\Project\\Classified'
if 1:
    Test2 = base + '\\20122\\'
    Test1 = base + '\\20121\\'
    Test0 = base + '\\20120\\'
    AllTest = [[Test2, '2'], [Test1, '1'], [Test0, '0']]


    Train2 = base + '\\20112\\'
    Train1 = base + '\\20111\\'
    Train0 = base + '\\20110\\'
    AllTrain = [[Train2, '2'], [Train1, '1'], [Train0, '0']]

    dirWalk = AllTrain + AllTest
    data = []
    for aDir, sType in dirWalk:
        for fDir in os.walk(aDir):
            for ffDir in fDir[1:]:
                for fffDir in ffDir:
                    file = open(aDir + fffDir, 'r')
                    data += [[file.read(), sType]]
                    file.close()
data = np.array(data)
data1 = []
data2 = []
for i in range(len(data)):
    if i%4==0:
        data2 += [data[i]]
    else:
        data1 += [data[i]]
```

```
d1 = open('data1.txt','w')
data1 = np.array(data1)
d1.write(str(data1))
d1.close()
d2 = open('data2.txt','w')
data2 = np.array(data2)
d2.write(str(data2))
d2.close()

xTrain = data1[:,0]
yTrain = data1[:,1]
xTest = data2[:,0]
yTest = data2[:,1]


if __name__ == "__main__":
    pipeline = Pipeline([
            ('vect', TfidfVectorizer()),
            ('clf', SGDClassifier()),
        ])

    parameters = {
        'vect__max_df': np.arange(0.25,1.0,0.00001),
        'vect__max_features': sp_randint(6000, 80000),
        'vect__ngram_range': ((1, 2), (2, 2), (1, 3), (2, 3), (3, 3), (1, 4), (2, 4), (3, 4)),
        'vect__norm': ('l2',),
        'clf__max_iter':sp_randint(200, 800),
        'clf__alpha': (1/np.logspace(1, 4, 10000)),
        'clf__penalty':('elasticnet',),a
    }

    t0 = time()
    print("Performing grid search...")
    print("pipeline:", [name for name, _ in pipeline.steps])
    print("parameters:")
    print(parameters)
    grid_search = RandomizedSearchCV(pipeline, param_distributions=parameters,
n_jobs=8, n_iter=3000,verbose=3, pre_dispatch=2*8,cv=4)
    grid_search.fit(xTrain, yTrain)
    print("done in %0.3fs" % (time() - t0))
    print()

    print("Best score: %0.3f" % grid_search.best_score_)
    print("Best parameters set:")
```

```
best_parameters = grid_search.best_estimator_.get_params()
for param_name in sorted(parameters.keys()):
    print("\t%s: %r" % (param_name, best_parameters[param_name]))

print('Training Set')
print('-----------------------')
prediction = grid_search.best_estimator_.predict(xTrain)
print('predictions = ',np.sum(prediction == yTrain)/len(yTrain))

print(prediction)
print(yTrain)

print('Test Set')
print('-----------------------')
prediction = grid_search.best_estimator_.predict(xTest)
print('predictions = ',np.sum(prediction == yTest)/len(yTest))

print(prediction)
print(yTest)
```

2) Ada-boost of SVM

Note: We used Pandas Library for DataFrame creation, Scikit Learn Library for Tf-Idf transforming, SGD Weak Classifier, AdaBoost Classifier, and Grid Search CV.

```
from __future__ import print_function

from pprint import pprint
from time import time
import logging
import numpy as np
from sklearn.tree import DecisionTreeClassifier
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.linear_model import SGDClassifier
from sklearn.linear_model import Perceptron
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import AdaBoostClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler
from sklearn.ensemble import RandomForestClassifier
import os

#print(__doc__)
scaler = MinMaxScaler(feature_range=(0,1))
```

```
# Display progress logs on stdout
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s %(levelname)s %(message)s')

if 1:
    Test2 = 'C:\\TermProject\\Classified\\20122\\'
    Test1 = 'C:\\TermProject\\Classified\\20121\\'
    Test0 = 'C:\\TermProject\\Classified\\20120\\'
    AllTest = [[Test2, '2'], [Test1, '1'], [Test0, '0']]


    Train2 = 'C:\\TermProject\\Classified\\20112\\'
    Train1 = 'C:\\TermProject\\Classified\\20111\\'
    Train0 = 'C:\\TermProject\\Classified\\20110\\'
    AllTrain = [[Train2, '2'], [Train1, '1'], [Train0, '0']]

    dirWalk = AllTrain + AllTest
    data = []
    for aDir, sType in dirWalk:
        for fDir in os.walk(aDir):
            for ffDir in fDir[1:]:
                for fffDir in ffDir:
                    file = open(aDir + fffDir, 'r')
                    data += [[file.read(), sType]]
                    file.close()
data = np.array(data)
#print(data)
data1 = []
data2 = []
for i in range(len(data)):
    if i%2==0:
        data2 += [data[i]]
    else:
        data1 += [data[i]]
d1 = open('data1.txt','w')
data1 = np.array(data1)
d1.write(str(data1))
d1.close()
d2 = open('data2.txt','w')
data2 = np.array(data2)
d2.write(str(data2))
d2.close()

if 1:
```

```
    #
################################################################################
#########
    # Define a pipeline combining a text feature extractor with a simple
    # classifier
    pipeline = Pipeline([
        ('vect', CountVectorizer()),
        ('tfidf', TfidfTransformer()),
        ('clf', AdaBoostClassifier()),
    ])

    be = ()
    pen = 'elasticnet'
    niter = 40
    alph = 0.001
    #class_weights = list(np.ones(len(data1[:,1]),dtype='float64')/len(data1[:,1]))
    be += (SGDClassifier(max_iter=niter,alpha=alph,penalty=pen),)

    # uncommenting more parameters will give better exploring power but will
    # increase processing time in a combinatorial way
    parameters = {
        'vect__max_df': (0.5, 0.75, 1.25),
        'vect__max_features': (10000,40000),
        'vect__ngram_range': ((1, 1), (1, 2)), # unigrams or bigrams
        'tfidf__use_idf': (False,True),
        'tfidf__norm': ('l2','l1'),
        'clf__max_depth':(5,10),
        'clf__n_estimators':(10,20,60,240,640,1600),
        'clf__algorithm':('SAMME',),
        'clf__base_estimator':be,
    }

    if __name__ == "__main__":
        # multiprocessing requires the fork to happen in a __main__ protected
        # block

        # find the best parameters for both the feature extraction and the
        # classifier
        grid_search = GridSearchCV(pipeline, parameters, n_jobs=3, verbose=1)

        print("Performing grid search...")
        print("pipeline:", [name for name, _ in pipeline.steps])
        print("parameters:")
        print(parameters)
```

```
        t0 = time()
        #scaler.fit(data1[:,0])
        grid_search.fit(data1[:,0], data1[:,1].astype(int))
        print("done in %0.3fs" % (time() - t0))
        print()

        print("Best score: %0.3f" % grid_search.best_score_)
        print("Best parameters set:")
        best_parameters = grid_search.best_estimator_.get_params()
        for param_name in sorted(parameters.keys()):
            print("\t%s: %r" % (param_name, best_parameters[param_name]))
        prediction = grid_search.best_estimator_.predict(data2[:,0])
print('predictions = ',np.sum(prediction == data2[:,1].astype(int))/len(data2))
```

3) LSTM Recurrent Neural Network

Note: We used Pandas Library for DataFrame creation, Keras Library for LSTM Neural Network.

```
import numpy as np
import matplotlib.pyplot as plt
from pandas import read_csv
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.callbacks import EarlyStopping
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error

para= 6

train = read_csv('./ClassifiedRnn/trainREC.csv',sep=',')
test = read_csv('./ClassifiedRnn/testREC.csv',sep=',')

train = np.array(train)
test = np.array(test)[:len(test)-1]
trainX =  np.array(train[:,1:])
trainY =  np.array(train[:,0])
testY =  np.array(test[:,0])
testX =  np.array(test[:,1:])

feat = len(trainX[0])
classes = 2

trainSort = np.append(trainY, testY)
```

```
trainSort.sort()

breakPts = {}
for i in range(0, classes):
    breakPts[i] = trainSort[(len(trainSort)//classes)*i]

lenX1 = len(test)
lenX2 = feat

trainXMod = []
trainYMod = []
for i in range(para-1,len(trainX)):
    trainXModTmp = []
    trainYModTmp = []

    for j in range(para-1,-1,-1):
        trainXModTmp += [trainX[i-j]]

        cls = 0
        for bp in breakPts:
            if(trainY[i-j] >= breakPts[bp]):
                cls = bp

        clss = []
        for bp in breakPts:
            if cls == bp:
                clss += [1]
            else:
                clss += [0]
        if j == 0:
            for k in range(para-1,-1,-1):
                trainYModTmp += [clss]

    trainXMod += [trainXModTmp]
    trainYMod += [trainYModTmp]

testXMod = []
testYMod = []
for i in range(para-1,len(testX)):
    testXModTmp = []
    testYModTmp = []

    for j in range(para-1,-1,-1):
        testXModTmp += [testX[i-j]]
```

```
        cls = 0
        for bp in breakPts:
            if(testY[i-j] >= breakPts[bp]):
                cls = bp

        clss = []
        for bp in breakPts:
            if cls == bp:
                clss += [1]
            else:
                clss += [0]

        if j == 0:
            for k in range(para-1,-1,-1):
                testYModTmp += [clss]

    testXMod += [testXModTmp]
    testYMod += [testYModTmp]

trainX=np.array(trainXMod)
testX=np.array(testXMod)

trainY = np.array(trainYMod)
testY = np.array(testYMod)

lenX1 = para
lenX2 = feat

model = Sequential()
n1 = 1200
n2 = 400
model.add(LSTM(n1, input_shape=(lenX1, lenX2), return_sequences=True))
model.add(LSTM(n2, input_shape=(n1, lenX1), return_sequences=True))
model.add(LSTM(n2, input_shape=(n2, lenX1), return_sequences=True))
model.add(LSTM(n2, input_shape=(n2, lenX1), return_sequences=True))
model.add(LSTM(n2, input_shape=(n2, lenX1), return_sequences=True))
model.add(Dense(len(breakPts)))
stop = [EarlyStopping(monitor='categorical_accuracy', mode='max', min_delta=0.02,
patience=100)]
model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['categorical_accuracy'])
trainSize = 1
bsize = 5
```

```
maxEpochs = 100
bestAcc = 0
for i in range(maxEpochs//trainSize):
    print(i * trainSize)
    model.fit(trainX, trainY, batch_size=bsize, epochs=trainSize,shuffle=False,
validation_data=(testX, testY))
    evalsTe = model.evaluate(testX, testY)[1]
    evalsTr = model.evaluate(trainX, trainY)[1]
    if min(evalsTe,evalsTe) > bestAcc:
        bestAcc = min(evalsTe,evalsTe)
        model.save_weights('model_weights.h5')
model.load_weights('model_weights.h5')

predictions = model.predict(testX, batch_size=bsize)
peval = model.predict(trainX, batch_size=bsize)
evalsTe = model.evaluate(testX, testY)
evalsTr = model.evaluate(trainX, trainY)

predictionindex = []
for a in range(0,len(predictions)):
    predictionindex += [np.argmax(predictions[a][-1])]

pevalindex = []
for a in range(0,len(peval)):
    pevalindex += [np.argmax(peval[a][-1])]


testindex = []
for a in range(0,len(testY)):
    testindex += [np.argmax(testY[a,0])]

trainindex = []
for a in range(0,len(trainY)):
    trainindex += [np.argmax(trainY[a,0])]

testindex = np.array(testindex)
predictionindex = np.array(predictionindex)
trainindex = np.array(trainindex)
pevalindex = np.array(pevalindex)

mid = classes//2
redTest = []
redPred = []
redTrain = []
```

```
redPeval = []

for arrI, arrO in [[testindex,redTest], [predictionindex, redPred], [trainindex, redTrain],
[pevalindex,redPeval]]:
    for i in range(len(arrI)):
        if arrI[i] >= mid:
            arrO += [1]
        else:
            arrO += [0]

#testindex = np.array(redTest)
#predictionindex = np.array(redPred)
#trainindex = np.array(redTrain)
#pevalindex = np.array(redPeval)

xte = np.linspace(1,len(testindex), num=len(testindex))
xtr = np.linspace(1,len(trainindex), num=len(trainindex))

ssize = 20
salpha = 0.5


plt.figure(1)
plt.scatter(xte,testindex, c='r',alpha=salpha, s=ssize)
plt.scatter(xte,predictionindex, c='b',alpha=salpha, s=ssize)
plt.figure(2)
plt.scatter(xtr,trainindex, c='r',alpha=salpha, s=ssize)
plt.scatter(xtr,pevalindex, c='b',alpha=salpha, s=ssize)
plt.show()

accTe = np.sum(testindex==predictionindex)
accTr = np.sum(trainindex==pevalindex)

print('Train Acc:', accTr)
print('Train Acc%:', accTr/len(trainindex) * 100)
print('Test Acc:', accTe)
print('Train Acc%:', accTe/len(testindex) * 100)
```

4) Deep Neural Network

Note: We used Pandas Library for DataFrame creation, Keras Library for Deep Neural Network.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
from pandas import read_csv
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.callbacks import EarlyStopping
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
import keras

para= 1

train = read_csv('./ClassifiedRnn/trainREC.csv',sep=',')
test = read_csv('./ClassifiedRnn/testREC.csv',sep=',')

train = np.array(train)
test = np.array(test)[:len(test)-1]
trainX =  np.array(train[:,1:])
trainY =  np.array(train[:,0])
testY =  np.array(test[:,0])
testX =  np.array(test[:,1:])

feat = len(trainX[0])
classes = 3

trainSort = np.append(trainY, testY)
trainSort.sort()

breakPts = {}
for i in range(0, classes):
    breakPts[i] = trainSort[(len(trainSort)//classes)*i]

print(breakPts)
for i in range(len(trainY)):
    for j in reversed(list(breakPts)):
        if trainY[i] >= breakPts[j]:
            trainY[i] = j
            break
    for j in reversed(list(breakPts)):
        if testY[i] >= breakPts[j]:
            testY[i] = j
            break

trainY = keras.utils.to_categorical(trainY, num_classes=classes)
```

```
testY = keras.utils.to_categorical(testY, num_classes=classes)

lenX1 = len(test)
lenX2 = feat

trainX=np.array(trainX).reshape(lenX1,lenX2)
testX=np.array(testX).reshape(lenX1,lenX2)

trainY = np.array(trainY).reshape(lenX1,classes)
testY = np.array(testY).reshape(lenX1,classes)

model = Sequential()
n1 = 4000
n2 = 500
model.add(Dense(n1, input_shape=(lenX2,)))
model.add(Dense(n2))
model.add(Dense(n2))
#model.add(LSTM(n2, input_shape=(n2,1), return_sequences=True))
#model.add(LSTM(n2, input_shape=(n2,1), return_sequences=True))
model.add(Dense(classes))
stop = [EarlyStopping(monitor='categorical_accuracy', mode='max',
min_delta=0.002, patience=5)]
model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['categorical_accuracy'])
bsize = 5
model.fit(trainX, trainY, batch_size=bsize, epochs=100,shuffle=False,
validation_data=(testX, testY), callbacks=stop)
predictions = model.predict(testX, batch_size=bsize)
peval = model.predict(trainX, batch_size=bsize)
evalsTe = model.evaluate(testX, testY)
evalsTr = model.evaluate(trainX, trainY)

predictionindex = []
for a in range(0,len(predictions)):
    predictionindex += [np.argmax(predictions[a])]

pevalindex = []
for a in range(0,len(peval)):
    pevalindex += [np.argmax(peval[a])]


testindex = []
for a in range(0,len(testY)):
    testindex += [np.argmax(testY[a])]
```

```
  trainindex = []
  for a in range(0,len(trainY)):
     trainindex += [np.argmax(trainY[a])]

  xte = np.linspace(1,len(testindex), num=len(testindex))
  xtr = np.linspace(1,len(trainindex), num=len(trainindex))

  ssize = 20
  salpha = 0.5

  testindex = np.array(testindex)
  predictionindex = np.array(predictionindex)
  trainindex = np.array(trainindex)
  pevalindex = np.array(pevalindex)

  plt.figure(1)
  plt.scatter(xte,testindex, c='r',alpha=salpha, s=ssize)
  plt.scatter(xte,predictionindex, c='b',alpha=salpha, s=ssize)
  plt.figure(2)
  plt.scatter(xtr,trainindex, c='r',alpha=salpha, s=ssize)
  plt.scatter(xtr,pevalindex, c='b',alpha=salpha, s=ssize)
  plt.show()

  accTe = np.sum(testindex==predictionindex)
  accTr = np.sum(trainindex==pevalindex)

  print('Train Acc:', accTr)
  print('Train Acc%:', accTr/len(trainindex) * 100)
  print('Test Acc:', accTe)
print('Train Acc%:', accTe/len(testindex) * 100)
```

5) Random Forest of Decision Trees

Note: We used Pandas Library for DataFrame creation, Scikit Learn Library for Tf-Idf transforming, Random Forest Classifier, and Grid Search CV.

```
from __future__ import print_function

from pprint import pprint
from time import time
import logging
import numpy as np
from sklearn.tree import DecisionTreeClassifier
from sklearn.feature_extraction.text import CountVectorizer
```

```
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.linear_model import SGDClassifier
from sklearn.linear_model import Perceptron
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import AdaBoostClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler
from sklearn.ensemble import RandomForestClassifier
import os

#print(__doc__)
scaler = MinMaxScaler(feature_range=(0,1))
# Display progress logs on stdout
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s %(levelname)s %(message)s')

if 1:
    Test2 = 'C:\\TermProject\\Classified\\20122\\'
    Test1 = 'C:\\TermProject\\Classified\\20121\\'
    Test0 = 'C:\\TermProject\\Classified\\20120\\'
    AllTest = [[Test2, '2'], [Test1, '1'], [Test0, '0']]


    Train2 = 'C:\\TermProject\\Classified\\20112\\'
    Train1 = 'C:\\TermProject\\Classified\\20111\\'
    Train0 = 'C:\\TermProject\\Classified\\20110\\'
    AllTrain = [[Train2, '2'], [Train1, '1'], [Train0, '0']]

    dirWalk = AllTrain + AllTest
    data = []
    for aDir, sType in dirWalk:
        for fDir in os.walk(aDir):
            for ffDir in fDir[1:]:
                for fffDir in ffDir:
                    file = open(aDir + fffDir, 'r')
                    data += [[file.read(), sType]]
                    file.close()
data = np.array(data)
#print(data)
data1 = []
data2 = []
for i in range(len(data)):
    if i%2==0:
        data2 += [data[i]]
```

```
    else:
        data1 += [data[i]]
d1 = open('data1.txt','w')
data1 = np.array(data1)
#print(data1)
d1.write(str(data1))
d1.close()
d2 = open('data2.txt','w')
data2 = np.array(data2)
d2.write(str(data2))
d2.close()

if 1:
    #
################################################################################
#########
    # Define a pipeline combining a text feature extractor with a simple
    # classifier
    pipeline = Pipeline([
        ('vect', CountVectorizer()),
        ('tfidf', TfidfTransformer()),
        #('clf', AdaBoostClassifier()),
        ('clf', RandomForestClassifier()),
    ])

    pen = 'elasticnet'
    niter = 40
    alph = 0.001


    # uncommenting more parameters will give better exploring power but will
    # increase processing time in a combinatorial way
    parameters = {
        'vect__max_df': (0.5, 0.75, 1.25),
        'vect__max_features': (10000,40000),
        'vect__ngram_range': ((1, 1), (1, 2)), # unigrams or bigrams
        'tfidf__use_idf': (False,True),
        'tfidf__norm': ('l2','l1'),
        'clf__max_depth':(5,10),
        'clf__n_estimators':(10,20,60,240,640,1600),
    }

    if __name__ == "__main__":
        # multiprocessing requires the fork to happen in a __main__ protected
```

```
    # block

    # find the best parameters for both the feature extraction and the
    # classifier
    grid_search = GridSearchCV(pipeline, parameters, n_jobs=3, verbose=1)

    print("Performing grid search...")
    print("pipeline:", [name for name, _ in pipeline.steps])
    print("parameters:")
    print(parameters)
    t0 = time()
    grid_search.fit(data1[:,0], data1[:,1].astype(int))
    print("done in %0.3fs" % (time() - t0))
    print()

    print("Best score: %0.3f" % grid_search.best_score_)
    print("Best parameters set:")
    best_parameters = grid_search.best_estimator_.get_params()
    for param_name in sorted(parameters.keys()):
        print("\t%s: %r" % (param_name, best_parameters[param_name]))

    prediction = grid_search.best_estimator_.predict(data2[:,0])
print('predictions = ',np.sum(prediction == data2[:,1].astype(int))/len(data2))
```

6) Multinomial Bayesian Classifier

Note: We used Pandas Library for DataFrame creation, Scikit Learn Library for Tf-Idf transforming, Multinomial Bayesian Classifier, and Grid Search CV.

```
from __future__ import print_function

from pprint import pprint
from time import time
import logging
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
#from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import GridSearchCV
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline
import os

print(__doc__)
```

```
# Display progress logs on stdout
logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s %(levelname)s %(message)s')

if 1:
    Test2 = 'C:\\TermProject\\Classified\\20122\\'
    Test1 = 'C:\\TermProject\\Classified\\20121\\'
    Test0 = 'C:\\TermProject\\Classified\\20120\\'
    AllTest = [[Test2, '2'], [Test1, '1'], [Test0, '0']]


    Train2 = 'C:\\TermProject\\Classified\\20112\\'
    Train1 = 'C:\\TermProject\\Classified\\20111\\'
    Train0 = 'C:\\TermProject\\Classified\\20110\\'
    AllTrain = [[Train2, '2'], [Train1, '1'], [Train0, '0']]

    dirWalk = AllTrain + AllTest
    data = []
    for aDir, sType in dirWalk:
        for fDir in os.walk(aDir):
            for ffDir in fDir[1:]:
                for fffDir in ffDir:
                    file = open(aDir + fffDir, 'r')
                    data += [[file.read(), sType]]
                    file.close()
data = np.array(data)
print(data)
data1 = []
data2 = []
for i in range(len(data)):
    if i%2==0:
        data2 += [data[i]]
    else:
        data1 += [data[i]]
d1 = open('data1.txt','w')
data1 = np.array(data1)
print(data1)
d1.write(str(data1))
d1.close()
d2 = open('data2.txt','w')
data2 = np.array(data2)
d2.write(str(data2))
d2.close()
#print(data1[:,0])
```

```
#print(data1[:,1])

if 1:
    #
##########################################################################
#########
    # Define a pipeline combining a text feature extractor with a simple
    # classifier
    pipeline = Pipeline([
        ('vect', CountVectorizer()),
        ('tfidf', TfidfTransformer()),
        ('clf', MultinomialNB()),
    ])

    # uncommenting more parameters will give better exploring power but will
    # increase processing time in a combinatorial way
    parameters = {
        'vect__max_df': (0.5, 0.75, 1.0),
        'vect__max_features': (None, 5000, 10000, 50000),
        'vect__ngram_range': ((1, 1), (1, 2)), # unigrams or bigrams
        'tfidf__use_idf': (True, False),
        'tfidf__norm': ('l1', 'l2'),
        'clf__alpha': (0.5, 0.1, 0.001),
    }

    if __name__ == "__main__":
        # multiprocessing requires the fork to happen in a __main__ protected
        # block

        # find the best parameters for both the feature extraction and the
        # classifier
        grid_search = GridSearchCV(pipeline, parameters, n_jobs=-1, verbose=1)

        print("Performing grid search...")
        print("pipeline:", [name for name, _ in pipeline.steps])
        print("parameters:")
        pprint(parameters)
        t0 = time()
        grid_search.fit(data1[:,0], data1[:,1].astype(int))
        print("done in %0.3fs" % (time() - t0))
        print()

        print("Best score: %0.3f" % grid_search.best_score_)
        print("Best parameters set:")
```

```
best_parameters = grid_search.best_estimator_.get_params()
for param_name in sorted(parameters.keys()):
    print("\t%s: %r" % (param_name, best_parameters[param_name]))

prediction = grid_search.best_estimator_.predict(data2[:,0])
print('predictions = ',np.sum(prediction == data2[:,1].astype(int))/len(data2))
```