**Machine Learning Fundamentals**

**Final Project**

Manciulea Teodor - Nicolae

**1.** The first task consists in finding and implementing a statistical model in order to filter the racist tweets from a Twitter posts dataset. The dataset was downloaded from Kaggle.

**a. Naive Bayes Implementation:**

This task is actually a classification application and it is very similar to the Spam Filter, for example, which can be easily solved using the Naive Bayes model.

tweets\_train = [re.sub('[0-9]+', '', i) for i in tweets\_train]  
tweets\_validation = [re.sub('[0-9]+', '', i) for i in tweets\_validation]

At the first step, the data was read from the .csv file which contains the labels (whether the tweet is racist or not) and the tweets, as two columns in a table. For this, the **pandas**

library had to be imported. This process was performed in the function **get\_data**. Also, in this function the data was split using the **StratifiedShuffleSplit**  class, from **sklearn.model\_selection**  library. This method was chosen especially because of the implementation method. When it comes to Naive Bayes, it is important for the apparition probabilities to be balanced in all the classes and in the both of the data sets (train and test).

def get\_data(filename):  
 data\_frame = pd.read\_csv(filename, delimiter = ',')  
 labels = data\_frame['label'].values  
 tweets = data\_frame['tweet'].values  
  
# print(data\_frame)

df\_stratif = StratifiedShuffleSplit(n\_splits = 1, test\_size=0.2)  
  
 for train\_index, validation\_index in df\_stratif.split(tweets, labels):  
 tweets\_train, tweets\_validation = tweets[train\_index], tweets[validation\_index]  
 labels\_train, labels\_validation = labels[train\_index], labels[validation\_index]  
  
 return tweets\_train, labels\_train, tweets\_validation, labels\_validation

After the function call, in order to clean the tweets a bit, all the numbers in the data set were removed, because they have no use for this application. In order to perform this, the **re** library was imported.

tweets\_train = [re.sub('[0-9]+', '', i) for i in tweets\_train]  
tweets\_validation = [re.sub('[0-9]+', '', i) for i in tweets\_validation]

The next step consists of turning our tweets into features. There are plenty of possibilities to perform this process, but in this case, the **CountVectorizer** was chosen. It was imported from **sklearn.feature\_extraction**  library. This method provides a number of apparitions for every token from the tweets section. In order to clean the data, the tokens were lowercased and the stop words were canceled. After creating the object, it has to be trained and then, using the **fit\_transform** method, is must be provided the matrix which contains the number of apparitions. The model was trained on the train data, and the on the test data there was only applied the **transform** method.

count\_vectorizer = CountVectorizer(lowercase=True, analyzer = 'word', stop\_words='english')  
x\_train = count\_vectorizer.fit\_transform(tweets\_train)  
x\_validation = count\_vectorizer.transform(tweets\_validation)

Now it is time to apply the Naive Bayes model. It is going to be applied the Multinomial Naive Bayes method, because the features we work with are discrete. Also, the correction is performed with an alpha parameter of value 0.001 in order to have a value quite close to 0. The prediction was performed on both train and test data, mostly because we want to evaluate the model and observe the difference between the performance of the algorithm on the train set and the test set.

model = MultinomialNB(alpha = 0.001)  
model.fit(x\_train, labels\_train)  
  
predictions\_validation = model.predict(x\_validation)  
predictions\_train = model.predict(x\_train)

If we run the code, the next results are shown:

**accuracy = 0.9913958308889671**

**precision recall f1-score**

**0 1.00 0.99 1.00**

**1 0.92 0.96 0.94**

**accuracy = 0.9585484123259815**

**precision recall f1-score**

**0 0.97 0.98 0.98**

**1 0.75 0.62 0.68**

The first results are for the train set and the second ones for the test set. It can be observed quite a low precision and recall on the test set.

In order to obtain a better score, I have tried to clean the data more by lemmatizing it and replacing the **CountVectorizer** with **TfidfVectorizer**, from the same library. The lemmatizing process transforms the words to their original state from the dictionary, hence the number of features could be reduced into less features with a higher apparition frequency. Now, instead of counting the number of apparitions with **CountVectorizer**, the algorithm also counts the number of ''non-apparitions'' with the **TfidfVectorizer.** I chose to use this method because I have noticed some tokens which are repeated frequently in every tweet and which are not stop words, for example, "@user".

In order to tokenize and lemmatize the words, the next function was implemented:

def word\_processing(sentence):  
 results = ''  
  
 for token in sentence:  
 if len(token) > 3:  
 results = results + " " + str(token.lemma\_)  
 return results

Also, the tokens with less than three letters were canceled, because they cannot be relevant for our application.

For the next steps, some manipulation of the data was necesarry:

doc\_list1 = []  
doc\_list2 = []  
tweets\_train = []  
tweets\_validation = []  
  
words\_train, labels\_train, words\_validation, labels\_validation = get\_data('train.csv')  
  
words\_train = [re.sub('[0-9]+', '', i) for i in words\_train]  
tweets\_validation = [re.sub('[0-9]+', '', i) for i in tweets\_validation]  
  
nlp = spacy.load('en\_core\_web\_sm')  
  
for i in words\_train:  
 i = str(i)  
 doc1 = nlp(i)  
 doc\_list1.append(doc1)  
  
for j in words\_validation:  
 j = str(j)  
 doc2 = nlp(j)  
 doc\_list2.append(doc2)  
  
for i in doc\_list1:  
 tweets\_train.append(word\_processing(i))  
  
for j in doc\_list2:  
 tweets\_validation.append(word\_processing(j))

Next, **TfidfVectorizer** was trained and the TfIdf scores matrix was generated with the fit\_transform method:

tfidf\_vectorizer = TfidfVectorizer(stop\_words='english')  
x\_train = tfidf\_vectorizer.fit\_transform(tweets\_train)  
x\_validation = tfidf\_vectorizer.transform(tweets\_validation)

After applying the same **MultinomialNB** model, the following results were obtained:

**accuracy = 0.9934686534475341**

**precision recall f1-score**

**0 0.99 1.00 1.00**

**1 0.98 0.93 0.95**

**accuracy = 0.9608947286094166**

**precision recall f1-score**

**0 0.97 0.99 0.98**

**1 0.85 0.54 0.66**

On the test set, it can be observed an increase of the precision, but the recall is not better. I have performed some changes to the alpha parameter and by changing its value to 0.01, the recall improves to a value of 59%, but the precision decreases to 81%.

**b. Support Vector Machine Implementation:**

I thought it would be interesting to compare the Naive Bayes method with the Support Vector Machine method. I have noticed that in the NB case, I obtained a sort of overfitting in terms of F1 score (most of all recall) on the positive class (racist tweets) and I presumed this happened because of the generative nature of this classifier. I thought I should try to classify the tweets using a discriminative classifier, like the SVM.

The steps of the implementation are the same. It only differs the model. Although, an observation could be the fact that for SVM is not compulsory to use the stratified shuffle. The model does not require such a balanced split of data.

model = SVC(C = 0.8, kernel = 'linear')  
model.fit(x\_train, labels\_train)  
  
predictions\_validation = model.predict(x\_validation)  
predictions\_train = model.predict(x\_train)

Without lemmatizing the tokens or using **TfidfVectorizer**, with the SVM method, better results are obtained:

**accuracy = 0.9960890140404396**

**precision recall f1-score**

**0 1.00 1.00 1.00**

**1 1.00 0.95 0.97**

**accuracy = 0.9641795714062256**

**precision recall f1-score support**

**0 0.97 0.99 0.98**

**1 0.84 0.60 0.70**

The linear kernel was chosen because of the considerable number of features this application presents. Also, the hyper parameter C has an influence on the predictions. Bay increasing it, the precision decreases and the recall increases. (Ex: C = 1 => P = 78%, R = 64%)

By applying the same data cleaning process of lemmatizing and canceling of the words with less than three letters and applying TfIdf instead of Count Vectorizer, the following results were obtained:

**accuracy = 0.9957761351636748**

**precision recall f1-score**

**0 1.00 1.00 1.00**

**1 0.99 0.95 0.97**

**accuracy = 0.9644924135773503**

**precision recall f1-score**

**0 0.97 0.99 0.98**

**1 0.80 0.66 0.72**

These results were obtained for C = 3. Also for C = 2, the same score was obtained, but with different values of the precision and the recall on the positive class (P = 80%, R = 66%).