Assignment in Data Analytics 2021-2022

Team Members:
Sofia Kalogiannidi ic121002-7115132100002

Dimitris Oreinos   ic121006- 7115132100006

Participated in Kaggle as Kalogiannidi_Oreinos

Requirement 1

Question 1

For the needs of Question1 we were asked to create 4 word clouds based on file “train.csv”. The steps made were:

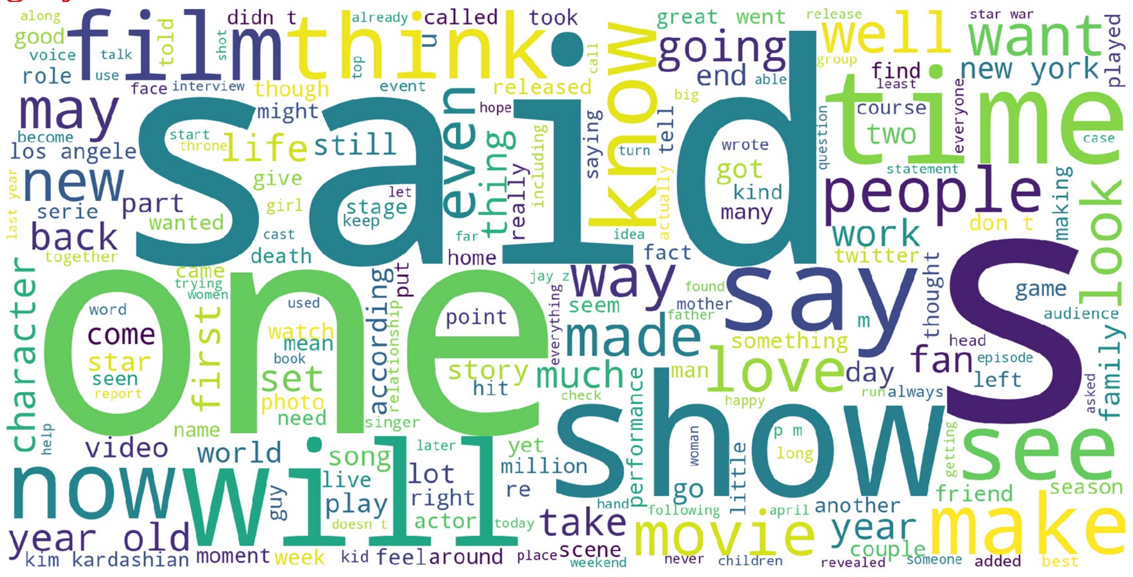
- use of pandas library in order to read the csv and create a dataframe with name df
- creation of four strings(one for each category).The strings are initially empty and their names are: words_Entertainment,words_Technology,words_Business,words_Health
- parse through df. Every time we find a content of Target 0 aka Entertainment, we add df[‘content’] to words_Entertainment. The same logic is followed for the rest of categories.
- Use of wordcloud library in order to create a word cloud for every category.The attributes for every wordcloud created are: height=1000,width=2000,background color=white .We also removed the stopwords for a more representative word cloud.
- Use of matplotlib library so as to plot the four word clouds.

We provide the word cloud designed for every category

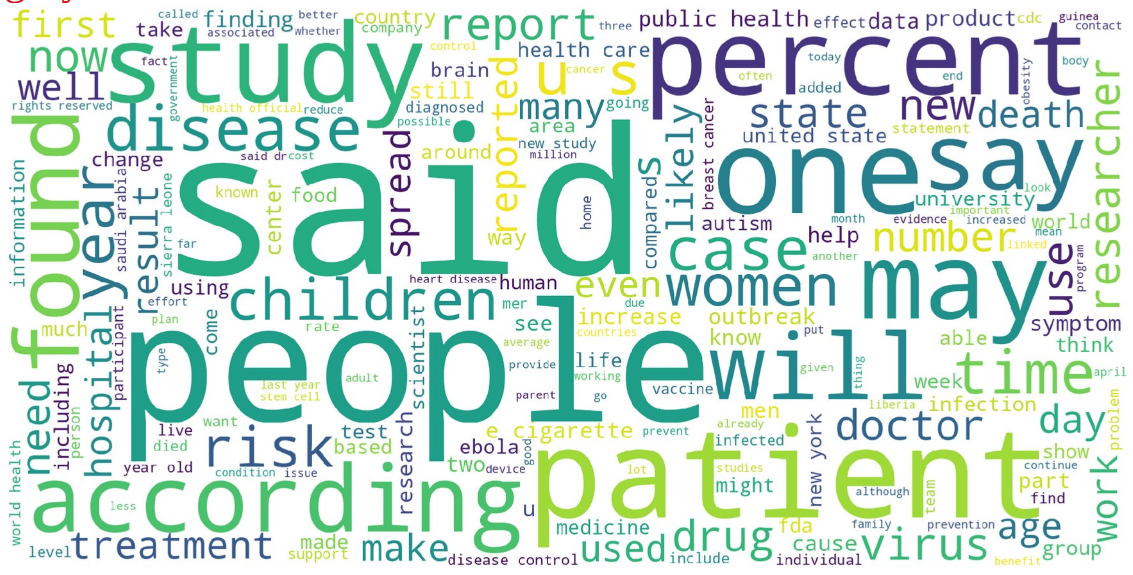
Category:Technology

Category:Business

Category:Entertainment



Category:Health



Question 2

In this Question, we were asked to perform text classification using KNN.We are going to describe the crucial parts of our work

1st step: Selection of features

We decided to use title and content as features to get better accuracy.What we actually did was to concatenate df[‘Content’]+ df[‘Title’] to a single string and then preprocess the resulting string with the procedure described in the next section.

2nd step: Preprocessing

In order to get better results in text classification, we realised the importance of preprocessing. The steps we took during preprocessing were the following:

- removal of all the numbers as well as punctuation, as we didn’t want them to reduce Jaccard_coefficient without actual reason
- conversion of all the words into lower case words. We wanted words such as “Company” and “company” in two different documents, to be translated both as “company” so as to contribute in the union.
- removal of white spaces
- removal of stopwords. Some words such as “the” will be appeared many times in the documents without showing something important about the document’s category. We didn’t want these words to be counted in the union, so we removed them with the help of stopwords in nltk.corpus library
- removal of single characters from the start
- substitution of multiple spaces with single space
- removal of prefixed ‘b’
- stemming. This step helped as to only consider the root of the words. For example words such as “companies” and “company” in two different documents were stemmed to “compan” and were included in the union during Jaccard Coefficient.

Implementation details:

1. For stemming we used WordNetLemmatizer() of nltk.stem
2. For the removal of punctuation and numbers, we imported re module in order to perform regular expression operations such as '[^a-zA-Z]’.
3. For the removal of white spaces, we used method strip() for strings

4. For the conversion into lower case, we used method lower() for strings
5. For tokenization,stemming and removal of stop words, we used library nltk.

The same preprocessing steps described were implemented for both train and test sets

Preprocessing Train Set (Duration)	Preprocessing Test Set(Duration)
6 minutes 29 seconds	2 minutes 47 seconds

<u>3rd step:Creating bag of words</u>

In order to use KNN classifier, we first had to convert texts to vectors.For that case we used the well-known method “Bag of words” and especially CountVectorizer of sklearn.feature_extraction.text

<u>4th step: Tf-idf</u>

After bag of words, we used Tf-idf. This technique helped as to quantify words by assigning a score and signify the importance of every term in a document and generally in corpus. At first, we didn’t include this step, but its addition added 0.8 to our accuracy score.

<u>5th step : Training the model</u>

Training is a crucial part during classification. We used train_test_split method in order to split the train set into train and test and get an accuracy score for that. More specifically, we split the train set into 80% training and 20% test.

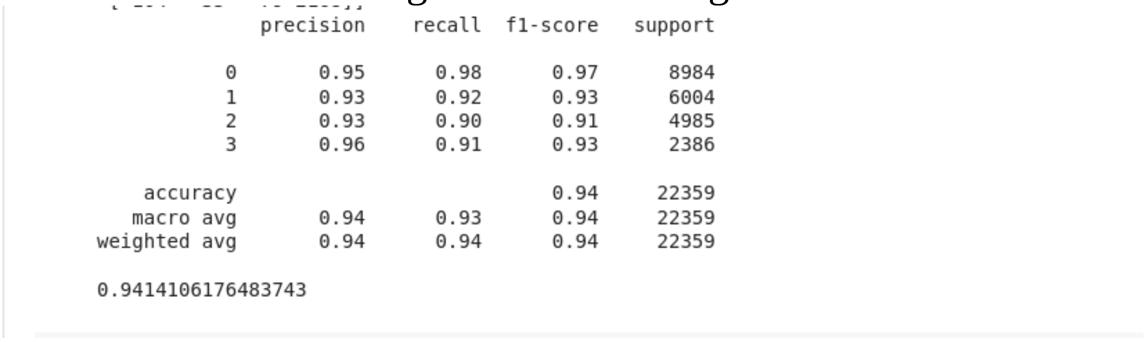
Prediction on 20% of Train Set(Duration)
48 minutes

In the following table, we provide the most important parameters we used.

Parameters	Value

CountVectorizer()	
max_features	1000
min_df	5
max_df	0.7
stop_words	stopwords.words('english')
train_test_split()	
test_size	0.2
random_state	42
KNN	
k	5
metric	jaccard

The results we had while training are the following



```
              precision    recall  f1-score   support

           0       0.95      0.98      0.97      8984
           1       0.93      0.92      0.93      6004
           2       0.93      0.90      0.91      4985
           3       0.96      0.91      0.93      2386

    accuracy                           0.94     22359
   macro avg       0.94      0.93      0.94     22359
weighted avg       0.94      0.94      0.94     22359

0.9414106176483743
```

6th step :Predicting Test Set

After training the model and preprocessing of the test set, the next step is to predict the test set.We stored all the titles and contents merged in a big list and afterwards, we used fit_transform into that list in order to create Xtest. The results of our predictions(targets) as well as the ids are stored in a list of tuples and are then written in a csv file called classification.csv.

Prediction of Test Set(Duration)
1 hour 45 minutes

Accuracy on Test Set
0.94028

Requirement 2

In this section,we were asked to speed up KNN algorithm by using lsh algorithm. We used MinHash from datasketch library.
At first,we take every review of train set and split it in order to make a list of shingles. Then,we preprocess that list so as to remove stop words and punctuation. For the preprocessing part we did the steps that were described analytically in the previous requirement with the aid of nltk library.
At first, we provide the table needed and then we are going to describe the details of our implementation

Type	BuildTime	QueryTime	TotalTime	Fraction of the true K most similar documents that are reported by LSH method as well	Parameters
Brute-Force-Jaccard	0	2 hours 36 minutes	2 hours 36 minutes	100%	
LSH-Jaccard	62.27 seconds	1067.36 seconds=17.7 minutes	18.82 minutes	60%	num_perm =16
LSH-Jaccard	94.95 seconds	233.87 seconds=3.89 minutes	5.48 minutes	16%	num_perm =32
LSH-Jaccard	158.03 seconds	508.81 seconds=8.48 minutes	11.1 minutes	33.3%	num_perm =64

Maximum Precision: 81.4 %

Functions created by us:
actual_jaccard(data1,data2): This function takes as arguments 2 lists of texts splitted into words and computes their actual jaccard distance by converting them into sets and dividing their intersection with their union.

preprocess(data): The preprocess steps described in previous requirement are performed by calling that function

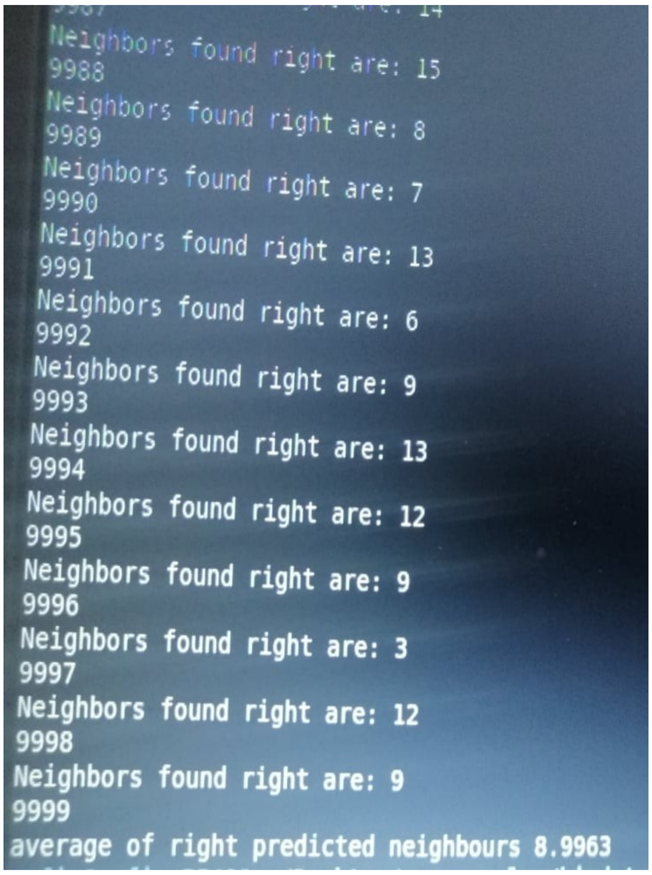
predict_sentiment(i,result,documents,documents_test,df):
This function takes as arguments the number of test row,the list of neighbours returned from lsh.query(result),the train and test documents and train dataframe. Then it creates a list of tuple pairs: (j,actual jaccard distances of documents_test[i],documents[j]) for every j in result list.
It stores this list with the aid of Sort_Tuple function which stores a list of tuples according to the second item of each tuple.
Then it keeps the 15 first tuples of that list(aka the 15 approximate nearest neighbors returned from lsh.query) and performs the well-known technique of KNN called “majority-voting”.It counts which sentiment (0 or 1) is the most common in the 15 approximate nearest neighbours and returns the result as prediction of sentiment.

KNN_search(i,documents,documents_test):
This function performs brute force search.It takes as arguments the number of test row and the documents of train and test set.Then it computes the actual jaccard distances of documents_test[i] ,documents[j] for every j in train dataset. In order to compare the real nearest neighbors with the approximate nearest neighbors and compute the fraction, this function stores the distances and returns the 15 first.

Computation of fraction approximate neighbors found right /nearest neighbors
In order to find that we call lsh.query(x) where x is the minhash for every test document.The result from lsh is then compared with the neighbors returned from function KNN_search(right).
The next step was to create a list with common neighbors found

```
    common = [c for c in right if c in result]
```

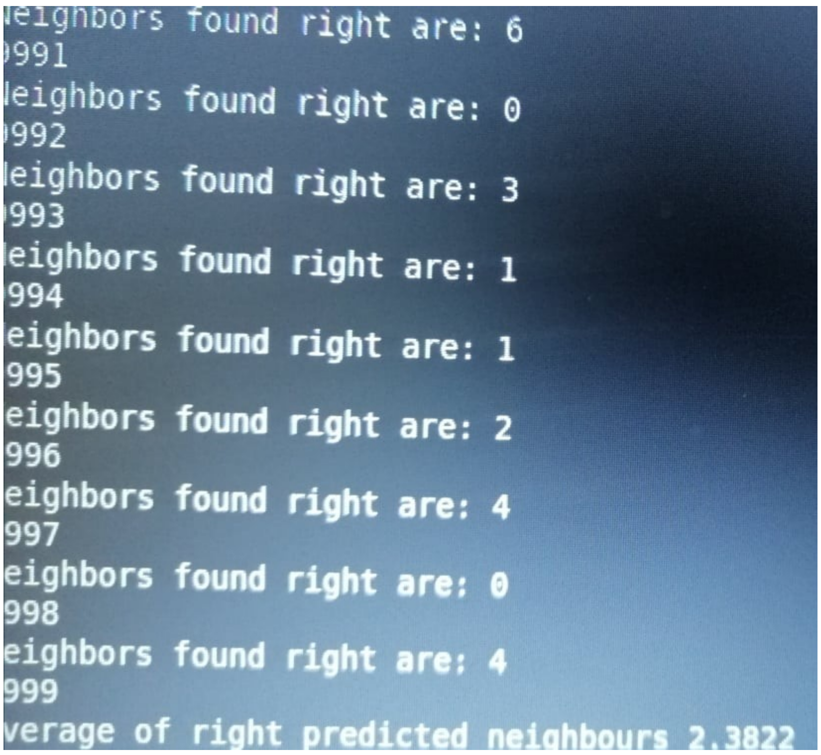
We sum the number of common neighbors for every test query and store it in variable total_right. Then,we divide total_right/test_size to find the average .As, you can see in screenshot below we find 9 /15 neighbors on average.(Num_perm=16)



```
Neighbors found right are: 15
9988
Neighbors found right are: 8
9989
Neighbors found right are: 7
9990
Neighbors found right are: 13
9991
Neighbors found right are: 6
9992
Neighbors found right are: 9
9993
Neighbors found right are: 13
9994
Neighbors found right are: 12
9995
Neighbors found right are: 9
9996
Neighbors found right are: 3
9997
Neighbors found right are: 12
9998
Neighbors found right are: 9
9999
average of right predicted neighbours 8.9963
```

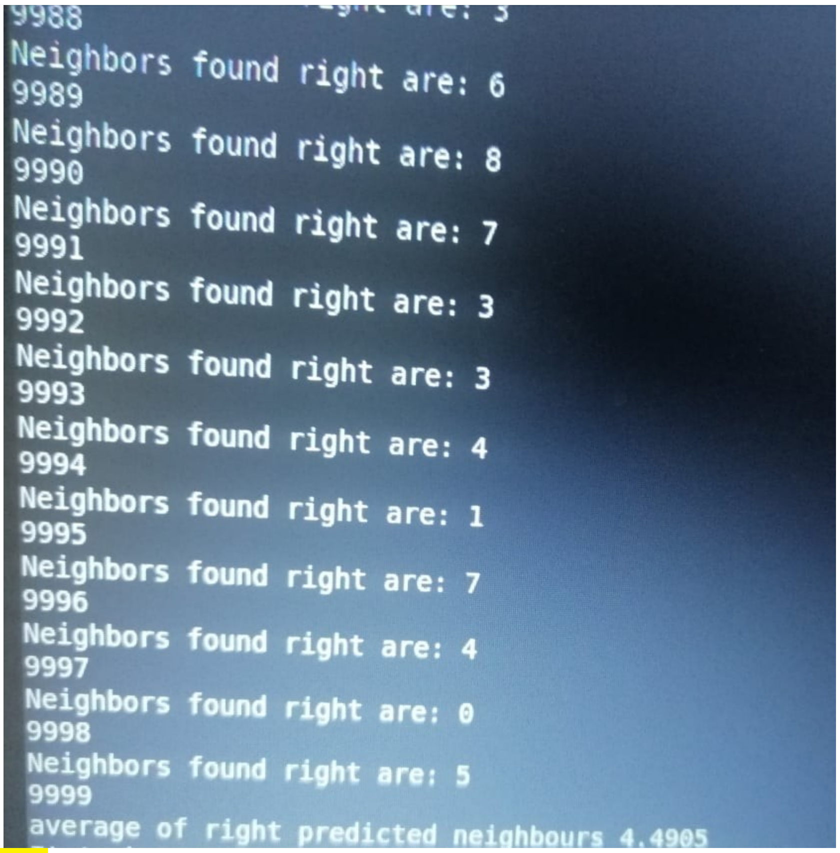
We also provide screenshots for Num_perm=32 and Num_perm=64

Num_perm=32 : find 2-3 right neighbors out of 15



```
leighbors found right are: 6
991
leighbors found right are: 0
992
leighbors found right are: 3
993
leighbors found right are: 1
994
eighbors found right are: 1
995
eighbors found right are: 2
996
eighbors found right are: 4
997
eighbors found right are: 0
998
eighbors found right are: 4
999
verage of right predicted neighbours 2.3822
```

Num_perm=64:find 4-5 neighbors out of 15



```
9988
Neighbors found right are: 6
9989
Neighbors found right are: 8
9990
Neighbors found right are: 7
9991
Neighbors found right are: 3
9992
Neighbors found right are: 3
9993
Neighbors found right are: 4
9994
Neighbors found right are: 1
9995
Neighbors found right are: 7
9996
Neighbors found right are: 4
9997
Neighbors found right are: 0
9998
Neighbors found right are: 5
9999
average of right predicted neighbours 4.4905
```

Requirement 3

For this exercise,we were asked to implement DTW. We are going to describe the steps taken before the computation of DTW distance, and the implementation we followed as well as the execution time.

1st step: Read the series - convert to float

The series in dtw_test.csv are represented as strings.So the first step we took was to convert them to lists of float numbers. More specifically:

1. We replaced ‘[‘ with ‘ ‘ in the initial string-serie
2. We replaced ‘]’ with ‘ ‘ in the initial string-serie
3. We replaced ‘,’ with ‘ ‘ .
4. We splitted the string
5. We created a new list where every word of the initial string is converted to float with eval().

We provide a part of our implementation:

```
reviewa= df["series_a"][i]
reviewa=reviewa.replace('[', '')
reviewa=reviewa.replace(',', '')
reviewa=reviewa.replace(']', '')
reviewa=reviewa.split()
reviewa=[eval(x) for x in reviewa]
```

This procedure is of course also repeated for seriesb of every row in the test file.

2nd step: Implementation of DTW function

Now, that the data are in the form we want, we are able to compute dtw. For this step we actually implemented the pseudocode provided in wikipedia in python language.The steps followed were:

1.Find the lengths of both series and name them n,m respectively
2.Initiliaze a cost matrix DTW[n+1,m+1] with zeros
3.Set DTW[0,0] aka the first cell equal to 0.
4.Set the DTW[i,j] equal to infinity
5. Start computing every cell of our cost martix with the following way:
   DTW[i,j]=distance(xi,yj)+min(DTW[i-1,j],DTW[i,j-1],DTW[i-1,j-1])
6.Return the last cell of the cost matrix aka DTW[n,m].This will be the DTW distance of the two time series.

Below, we provide the exact implementation in python

```
def DTWdistance(seriesa,seriesb):

    n=len(seriesa)
    m=len(seriesb)
    print(n,m)
    DTW = np.zeros((n+1, m+1))
    for i in range(n+1):
        for j in range(m+1):
            DTW[i,j]=np.inf
    DTW[0,0]=0

    for i in range(1,n+1):
        for j in range(1,m+1):
            dist=abs(seriesa[i-1]-seriesb[j-1])
            DTW[i,j]=dist+ np.min([DTW[i-1,j],DTW[i,j-1],DTW[i-1,j-1]])


    return DTW[n,m]
```

This is the implementation,we decided to keep. However, in Kaggle you can see that we made multiple submissions.The alternatives we tried were the following

1. DTW with window constraint w.We implemented the pseudocode given in Wikipedia, but the difference in our score with that version was really small
2. Time-normalised DTW. In that case, what we did was after finding the cost matrix like described above, starting from [n,m] cell and parsing the optimal path to [0,0] while keeping the total cost of the optimal path and its length k.Then we returned total_cost/k. This version didn’t have bad results,but the original DTW was better so we preferred it.
3. Squared euclidean distance. We also tried to use squared euclidean instead of euclidean distance as distance metric.This version gave us the worst results.
4. Tried our own version of fastDTW. This version gave us good results especially in time but we kept the most accurate version. In this alternative we performed DTW only if the initial arrays had lengths

less than 500.Otherwise, we shrinked them until that happened because DTW is slow in big time series

3rd step: Writing results into csv

We created a list of lists. For every row, we computed DTW between the two series and created a list [ID,DTWdistance(seriesa,seriesb)].

Then, we appended that list into a big list of lists called L. After the computation of DTW distance for every row of test file,we actually wrote L into a csv called “dtw.csv” with two columns “Id,distance”.In order to have the same format as in kaggle,we then dropped the first column of indices which is created automatically in csv files.

Computation time
1 hour 39 minutes