Data Analytics
Τεχνικές Ανάλυσης Δεδομένων Υψηλής Κλίμακας
Big Data Mining Techniques
2020-2021

Report

Αθανασοπούλου Ελένη CS1200001
Σεμερτζάκης Γεώργιος AL1180015

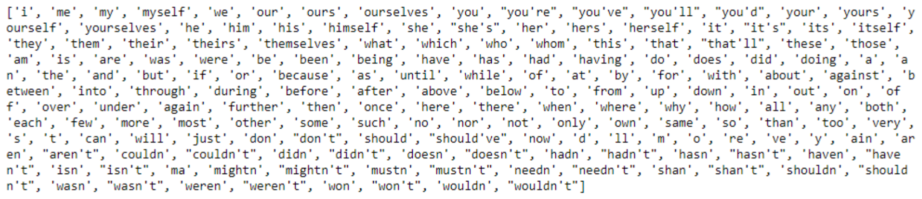
Requirement 1: Text Classification

Question 1a: WordCloud

A wordcloud is a visual representation of text data. It is typically used to depict keyword tags, which are usually single words. The importance of each tag is shown with font size or color. This format is useful for quickly perceiving the most prominent terms. Bigger terms means greater weight.

Python provides us the **wordcloud** library that contains *Wordcloud* method and helps us form wordclouds, which we visualize with the help of **matplotlib** library. *Wordcloud* consists of an amount of parameters, from which we chose to use *max_words*, *max_font_size*, *background_color* and *stopwords*. The first one defines the maximum number of words, the second one denotes maximum font size for the largest word, the third one is for the color of the background of wordcloud image and the fourth one defines the exclusion of the most common text words.

Term *stopwords* usually refers to the most common words in a language. This definition is enough to understand the necessity of their deletion since this kind of words do not yield any extra information. They may also confuse the algorithm which will end up making wrong predictions with high probability.



```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you'll", "you'd", 'your', 'yours', 'y
ourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "it's", 'its', 'itself',
'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'that', "that'll", 'these', 'those',
'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'having', 'do', 'does', 'did', 'doing', 'a', 'a
n', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'while', 'of', 'at', 'by', 'for', 'with', 'about', 'against', 'b
etween', 'into', 'through', 'during', 'before', 'after', 'above', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'of
f', 'over', 'under', 'again', 'further', 'then', 'once', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both',
'each', 'few', 'more', 'most', 'other', 'some', 'such', 'no', 'nor', 'not', 'only', 'own', 'same', 'so', 'than', 'too', 'very',
's', 't', 'can', 'will', 'just', 'don', "don't", 'should', "should've", 'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'ar
en', "aren't", 'couldn', "couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven', "have
n't", 'isn', "isn't", 'ma', 'mightn', "mightn't", 'mustn', "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "should
n't", 'wasn', "wasn't", 'weren', "weren't", 'won', "won't", 'wouldn', "wouldn't"]
```

Figure 1: English stopwords

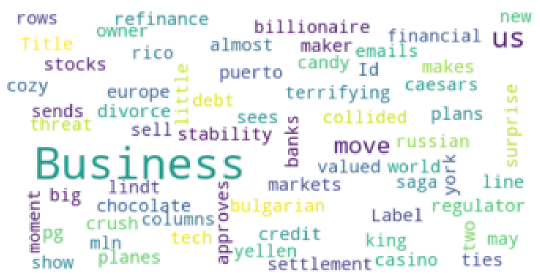


Figure 2: Business WordCloud

Above in Figure 2 is the wordcloud which describes first category of articles, business. We observe that the word ”business” is the most prominent due to its size. In addition, words like ”billionaire”, ”financial” and ”markets” are also significant and complete this category.

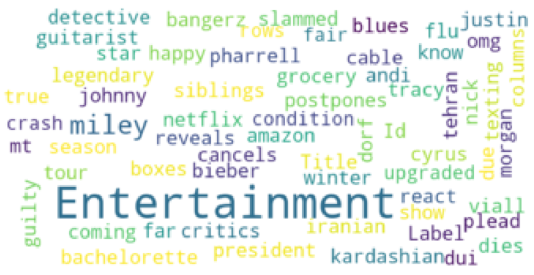


Figure 3: Entertainment WordCloud

The wordcloud of entertainment follows in Figure 3. It describes the second category of articles and we can see that the word ”entertainment” is the most important. Words like ”critics”, ”netflix” and ”amazon” determine the content of this category.

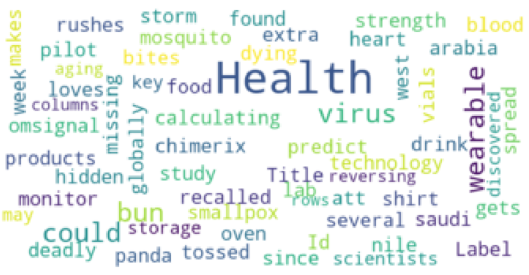


Figure 4: Health WordCloud

In Figure 4 we show the wordcloud of health. Again, the word ”health” is the biggest one because of its importance. Words like ”scientists”, ”virus” and ”blood” designate the content of this category.

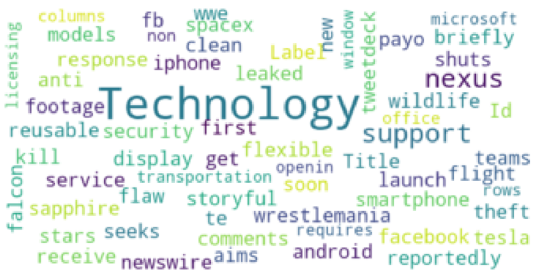


Figure 5: Technology WordCloud

Last but not least, in Figure 5 we illustrate the wordcloud of technology. It is the last category of the articles and we can see the great significance of ”technology”. This category is completed by words like ”iphone” and ”facebook”.

Question 1b: Classification Task

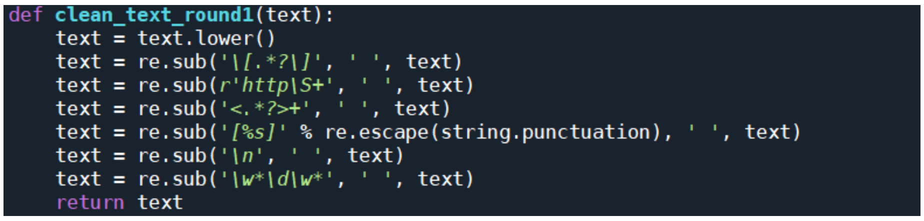
The majority of practical machine learning uses supervised learning. This kind of learning is where you have input data and output target data and you use an algorithm to learn the mapping function from the input to the output. The goal is to approximate the mapping function so well that when you have new input data, you can predict the output target data.

Supervised learning problems can be used in classification problems. A classification problem is when the output target data is a category. For instance, in this task the categories that are mentioned are ”business”, ”entertainment”, ”health” and ”technology”.

Text Preprocessing

Firstly, we will talk about some necessary steps needed for transferring text from human language to machine-readable format for further processing. After a text is obtained, we start with text normalization. This procedure is also called data cleaning. In our code, data preprocessing method is achieved in three separated functions.

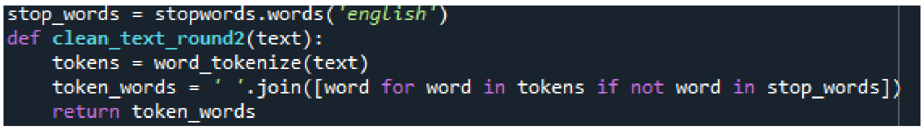
A text may contain either lowercase or uppercase letters, punctuations, tags or URLs, concatenated letters and integers or non-english words. This kind of words are useless for our task, so we removed them. Python provides us **re** library, which contains *sub* method. We use this method to subtract the aforementioned words. Figure 6 shows the function *clean_text_round1*.



```python
def clean_text_round1(text):
    text = text.lower()
    text = re.sub('\[.*?\]', ' ', text)
    text = re.sub(r'http\S+', ' ', text)
    text = re.sub('<.*?>+', ' ', text)
    text = re.sub('[%s]' % re.escape(string.punctuation), ' ', text)
    text = re.sub('\n', ' ', text)
    text = re.sub('\w*\d\w*', ' ', text)
    return text
```

Figure 6: Clean text round 1

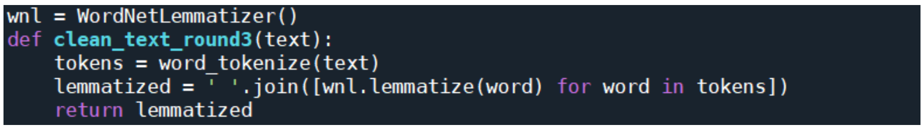
Stopwords which most probably are also contained in a text, are not needed since their appearance may cause problems in the predictions. Python provides us **nltk** library from which we obtain *stopwords* and *word_tokenize* method. The first one grants us the most common english words. The second one works like *split* method for strings. It converts a text into a list in which each element is a word of the text. Notice that this list may contain duplicates. In figure 7 we can see the function *clean_text_round2* that describes the above.



```python
stop_words = stopwords.words('english')
def clean_text_round2(text):
    tokens = word_tokenize(text)
    token_words = ' '.join([word for word in tokens if not word in stop_words])
    return token_words
```

Figure 7: Clean text round 2

It is possible that some words may appear in several inflected forms. For example, words like ”walk”, ”walked”, ”walking” may appear in a text. The base of each of these three words is the word ”walk”. Therefore we wanted only the base word to be taken into account. In Figure 8 we can see the function *clean_text_round3* which converts every word into its base form. By **nltk** library, we chose method *WordNetLemmatizer* which is useful to do the above.



```
wnl = WordNetLemmatizer()
def clean_text_round3(text):
    tokens = word_tokenize(text)
    lemmatized = ' '.join([wnl.lemmatize(word) for word in tokens])
    return lemmatized
```

Figure 8: Clean text round 3

Features

Word embedding is a technique which is used in order to represent the text as vectors. The more popular forms of word embeddings are BoW, which stands for Bag of Words, and TF-IDF, which stands for Term Frequency-Inverse Document Frequency. These methods prepare the text for machine learning model.

Dimensionality reduction can be achieved by simply dropping columns. SVD, which stands for Singular Value Decomposition, is one of several techniques that can be used to reduce the dimensionality.

-Bag of Words (BoW)

The BoW model is the simplest form of text representation in numbers. Like the term itself, we can represent a sentence as a bag of words vector (a string of numbers). In python we do this with the help of *CountVectorizer* which is obtained by **sklearn***.feature_extraction.text* method.

-Term Frequency-Inverse Document Frequency (TF-IDF)

Term frequency–inverse document frequency, is a numerical statistic that is intended to reflect how important a word is to a document in a collection or corpus.

Let’s first understand TF. It is a measure of how frequently a word appears in a document or equivalently in a row of the dataframe.

$$\text{TF}(\text{Document}, \text{Word}) = \frac{\text{Number of occurences of a word in a document}}{\text{Number of words in a document}}$$

IDF is a measure of how important a word is. We need IDF value because computing just the TF alone is not sufficient to understand the importance of words.

$$\text{IDF}(\text{Word}) = \log(\frac{\text{Number of documents}}{\text{Number of documents containing the word}})$$

We can now compute the TF-IDF score for each word in the corpus. Words with a higher score are more important, and those with a lower score are less important.

$$\text{TF-IDF}(\text{Word}) = \text{TF}(\text{Document}, \text{Word}) \cdot \text{IDF}(\text{Word})$$

In python we do this with the help of *TfidfVectorizer* which is obtained by **sklearn**.*feature_extraction.text* method.

-Singular Value Decomposition (SVD)

By choosing either BoW or TF-IDF feature we obtain a matrix A where the number of rows m equals to the number of documents and the number of columns n equals to the number of words. Let r be the rank of the matrix. It is known by Linear Algebra that SVD is a technique that decomposes the matrix A into a set of related matrices $A = U\Sigma V^T$, where:

- Σ is a $r \times r$ non-negative, decreasing order diagonal matrix. All elements not on the main diagonal are 0 and the elements of Σ are called singular values.
- U is a $m \times r$ orthonormal matrix and V is a $n \times r$ orthonormal matrix. Orthonormal matrices are orthogonal matrices where the columns are unit vectors. A matrix Q is called orthogonal if $Q \cdot Q^T = Q^T \cdot Q$ and $Q^T = Q^{-1}$.

The way to do this decomposition in python is to use *TruncatedSVD* method which is provided by **sklearn**.*decomposition*. Truncated SVD works on term BoW/TF-IDF matrices as returned by the aforementioned vectorizers.

Classifiers

-Support Vector Machines (SVMs)

In machine learning, SVMs are supervised learning models with associated learning algorithms that analyze data useful for classification analysis. Assume some given data points each belong to one of the categories that we mentioned above, the goal is to decide which category a new data point will be in. In the case of SVMs, a data point is viewed as a p-dimensional vector. A SVM constructs a hyperplane or set of hyperplanes in a high dimensional space. In our case, a p-dimensional vector is a transformed document from column "Content" according to a feature of our choice and p is the length of the vector. By using either BoW or TF-IDF feature, notice that p equals to the total number of words appearing in train set.

In python we use *SVC* classifier as obtained by **sklearn**.*svm*.

-Random Forest (RF)

A forest is composed of trees. A random forest creates decision trees on randomly selected data samples, gets prediction from each tree and selects the best solution by means of voting. In a classification problem each tree votes and the most popular class is chosen as the final result.

In python we use *RandomForestClassifier* classifier which is obtained by **sklearn**.*ensemble*.

-Multinomial Naive Bayes (MNB)

Naive Bayes classifiers are a family of simple probabilistic classifiers based on applying Bayes's theorem with strong independence assumptions between the features. Baye's theorem describes the probability of an event, based on prior knowledge of conditions that might be related to the event. Bayes' theorem is stated mathematically as $P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$, where A, B are the events.

MNB is Naive Bayes classifier for multinomial models. MNB classifier is suitable for classification with discrete features. For example, discrete features could be word counts for text classification. Multinomial distribution normally requires integer feature counts. However, in practice, fractional counts such as TF-IDF may also work.

In python we use *MultinomialNB* as obtained by **sklearn**.*naive_bayes*.

K-Fold Cross Validation

When we train a model, we split the dataset into two main sets: training and testing. The training set represents all the examples that a model is learning from, while the testing set simulates the testing examples. For our task, we worked on the train.csv file from datasets/q1 folder.

	Fold-1	Fold-2	Fold-3	Fold-4	Fold-5
Step-1	Train	Train	Train	Train	Test
Step-2	Train	Train	Train	Test	Train
Step-3	Train	Train	Test	Train	Train
Step-4	Train	Test	Train	Train	Train
Step-5	Test	Train	Train	Train	Train

Figure 9: A 5-fold representation of how each fold is used in the cross-validation process

Cross validation provides the ability to estimate the model performance on unseen data not used while training. Classification problems can use cross validation. It has two main steps, splitting the data into K subsets, called folds, and rotating the training and validation among them. The splitting technique can be varied and chosen based on the data's size. Cross validation ensures that the model has been tested on the full data without testing them at the same time. Variations are expected in each step of the validation. K-fold is characterizing how many folds you want to split your dataset into.

For our task we used *cross_validate* and *StratifiedKFold* methods which we accessed from **sklearn**.*model_selection*. The *cross_validate* provides an amount of parameters from which we chose to use *estimator*, *X*, *y*, *scoring* and *cv*. The first one denotes the estimator, which in our case is a classifier. The second one denotes the train data. The third one denotes the train target data. The fourth one represents a list which consists of the desired evaluation metrics. The fifth one determines the cross validation splitting strategy. We set *cv* as *StratifiedKFold(n_splits=5)* in our task, since we use 5-Fold cross validation. We use *StratifiedKFold* instead of *KFold* since the *estimator* is a classifier and y is multiclass, i.e. y has more than two classes.

The evaluation metrics we are interested in are accuracy, precision macro, recall macro and f1-score macro. Macro is used for models with more than two target classes. The definitions for each metric are broached below.

-Accuracy

It is the ratio of number of correct predictions to the total number of input samples.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

-Precision macro

Precision is a metric used in binary classification problems in order to find the proportion of positive predictions that was actually correct. Precision is defined as

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$

where True Positive is defined as the number of actual positives when they are predicted positive and False Positive is defined as the number of actual negatives when they are predicted positive.

Precision macro is performed by first computing the precision of each class, and then taking the average of all precisions.

-Recall macro

Recall is defined as

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$

where False Negative is defined as the number of actual positives when they are predicted negative.

Recall macro is performed by first computing the recall of each class, and then taking the average of all recalls.

-F1-score macro

F1-score is defined as

$$\text{F1-score} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision+Recall}}$$

F1-score macro is used to assess the quality of problems with multiple classes. The F1-score macro is defined as the mean of label-wise F1-scores:

$$\text{F1-score macro} = \frac{1}{N} \cdot \sum_{i=1}^{N} \text{F1-score(i)}$$

where i is the class and N denotes the number of classes.

Our Approach

We are given two csv files: train and test_without_labels. Our goal is to select the best model in order to predict the category for each article. For the main task, we are required to use four different methods so we can compute and evaluate the performance of every model and feature separately. Afterwards, we selected a new model and feature so we can have better results. Finally, we demonstrate a table of the results for each method.

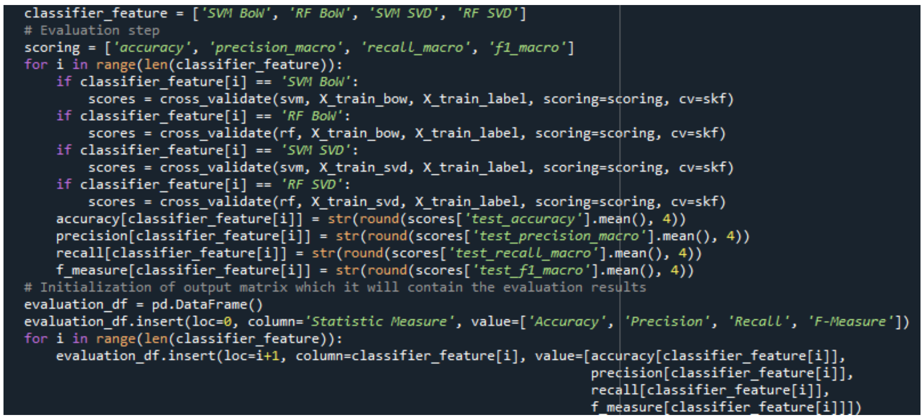
During the calculation of the performance we only deal with train.csv file. After the selection of the best method we also need test_without_labels.csv file. Furthermore, we are exclusively interested in ”Content” column, since the other columns do not give useful information. So this column is going to train each model. Also, the categories are matched with a unique label. In python *LabelEncoder*, which is obtained by **sklearn**.*preprocessing*, helps us make this matching.

Main Task

For this part, we are required to try classification with four methods. For the first one we selected SVM and we transformed the train data into vectors according to BoW feature. Then we did the same thing by selecting RF and BoW feature. For the third one we selected SVM and we transformed the train data into vectors according to SVD obtained by a BoW matrix. Finally, for the last one we selected RF and we converted the train data into vectors according to SVD obtained again by a BoW matrix.

For the BoW feature, we initialize the vectorizer mentioned in BoW section, we fit the train data into it and we transform the train data into vectors. After that, the vocabulary of the vectorizer is a dictionary which holds all different words as keys and the total number of their appearance in train data as values. The variable *X_train_bow* is a matrix of dimension number of rows of train set times number of words of vectorizer. This matrix is sparse. So, for the SVD feature, we initialize *TruncatedSVD*, we fit train data as obtained by BoW feature and we create new vectors. The variable *X_train_svd* is the new matrix which is less sparse than the previous one. The aforementioned procedure is called dimensionality reduction.

File ass1_ex1_wordclouds_&_eval_res.py contains the code for this task. Figure 10 shows a specific part of the code. For convenience, let <score_name> be one of ”accuracy”, ”precision_macro”, ”recall_macro” or ”f1_macro”. Then, observe that scores['test_<score_name>'] is a list of five elements of type float between 0 and 1. Every position of the list contains the performance of the model for a specific step during cross validation process. We add to final table the mean score of these elements for every evaluation metric.



```
classifier_feature = ['SVM BoW', 'RF BoW', 'SVM SVD', 'RF SVD']
# Evaluation step
scoring = ['accuracy', 'precision_macro', 'recall_macro', 'f1_macro']
for i in range(len(classifier_feature)):
    if classifier_feature[i] == 'SVM BoW':
        scores = cross_validate(svm, X_train_bow, X_train_label, scoring=scoring, cv=skf)
    if classifier_feature[i] == 'RF BoW':
        scores = cross_validate(rf, X_train_bow, X_train_label, scoring=scoring, cv=skf)
    if classifier_feature[i] == 'SVM SVD':
        scores = cross_validate(svm, X_train_svd, X_train_label, scoring=scoring, cv=skf)
    if classifier_feature[i] == 'RF SVD':
        scores = cross_validate(rf, X_train_svd, X_train_label, scoring=scoring, cv=skf)
    accuracy[classifier_feature[i]] = str(round(scores['test_accuracy'].mean(), 4))
    precision[classifier_feature[i]] = str(round(scores['test_precision_macro'].mean(), 4))
    recall[classifier_feature[i]] = str(round(scores['test_recall_macro'].mean(), 4))
    f_measure[classifier_feature[i]] = str(round(scores['test_f1_macro'].mean(), 4))
# Initialization of output matrix which it will contain the evaluation results
evaluation_df = pd.DataFrame()
evaluation_df.insert(loc=0, column='Statistic Measure', value=['Accuracy', 'Precision', 'Recall', 'F-Measure'])
for i in range(len(classifier_feature)):
    evaluation_df.insert(loc=i+1, column=classifier_feature[i], value=[accuracy[classifier_feature[i]],
                                                                       precision[classifier_feature[i]],
                                                                       recall[classifier_feature[i]],
                                                                       f_measure[classifier_feature[i]]])
```

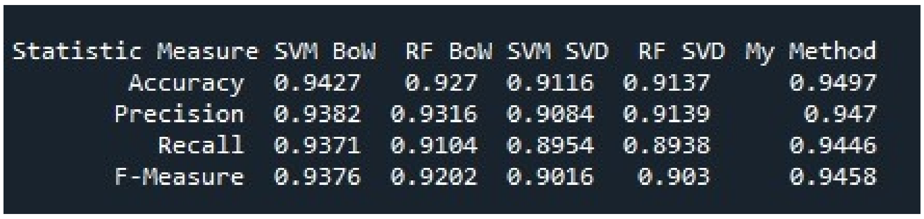
Figure 10: Evaluation results for given methods

Beat the Benchmark

We concluded that for ”My Method” it will be much more efficient to use MNB and TF-IDF feature. We operated in a similar way as in the other methods and we observed that the results of this method are much better. The whole code is in ass1_ex1_beat_the_benchmark&_csv.py file.

Evaluation Results

The following table contains the expected results.



```
Statistic Measure SVM BoW  RF BoW SVM SVD  RF SVD My Method
         Accuracy  0.9427   0.927  0.9116  0.9137    0.9497
        Precision  0.9382  0.9316  0.9084  0.9139     0.947
           Recall  0.9371  0.9104  0.8954  0.8938    0.9446
        F-Measure  0.9376  0.9202  0.9016   0.903    0.9458
```

Figure 11: Evaluation results for question 1b

Predictions

We chose method ”My Method” to make the predictions for articles in file without labels. Firstly, we fit the data and the target data from train.csv file into MNB classifier. This procedure is very important since the classifier needs to be familiar with the categories. Afterwards, we transform the test data from test_without_labels.csv file into vectors. Finally, model makes predictions on these unseen data vectors. Our code outputs the testSet_categories.csv file which contains the expected predictions.

Figure 12 is a countplot that illustrates the total number of appearances of each category in the output file.

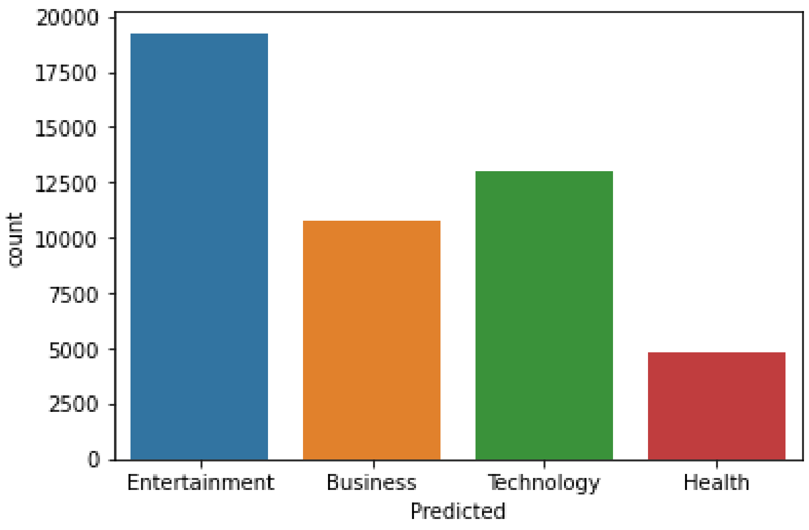


Figure 12: Visualization of predicted labels in question 1b

Requirement 2: Nearest Neighbor Search and Duplicate Detection

Question 2a: De-Duplication with Locality Sensitive Hashing

Exact Jaccard

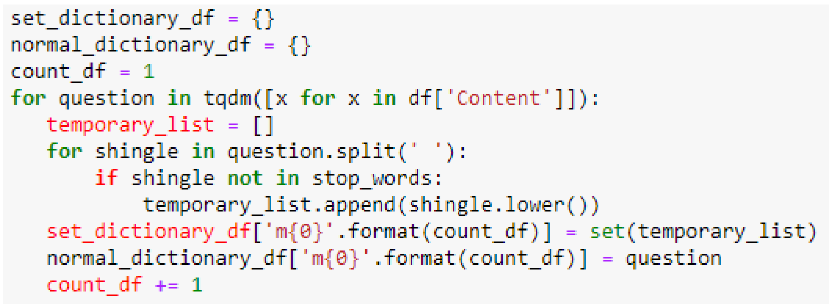
Jaccard coefficient measures the similarity between finite sets. It is defined as the size of the intersection divided by the size of union of two sets. Let A, B be two sets. Then

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Notice that $J(A, B) = 1$ if and only if sets A, B have exactly same elements and $J(A, B) = 0$ if and only if sets A, B have no element in common.

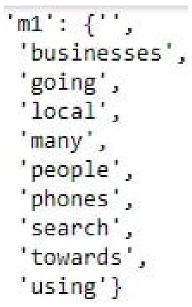
Since Jaccard computes the similarity between finite sets, we transformed every question of corpusTrain.csv file into a set of which elements are every word in the document/question. This procedure is called **Set Representation** of a document. Figure 13 shows this set construction for train file. We give an example of this transformation in Figure 14, which shows the form of the first question. We worked with the same mindset for corpusTest.csv file.

The code of Exact Jaccard follows in Figure 15 which stands for computing the similarity between questions of test set and questions of train set. Let n be the number of questions in test set and let m be the number of questions in



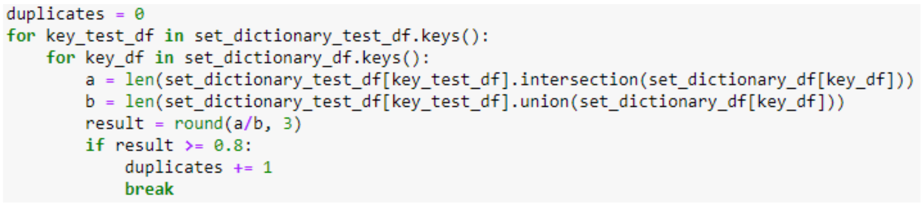
```
set_dictionary_df = {}
normal_dictionary_df = {}
count_df = 1
for question in tqdm([x for x in df['Content']]):
    temporary_list = []
    for shingle in question.split(' '):
        if shingle not in stop_words:
            temporary_list.append(shingle.lower())
    set_dictionary_df['m{0}'.format(count_df)] = set(temporary_list)
    normal_dictionary_df['m{0}'.format(count_df)] = question
    count_df += 1
```

Figure 13: Set representation for train set file



```
'm1': {'',
  'businesses',
  'going',
  'local',
  'many',
  'people',
  'phones',
  'search',
  'towards',
  'using'}
```

Figure 14: Form of first question of train set file



```
duplicates = 0
for key_test_df in set_dictionary_test_df.keys():
    for key_df in set_dictionary_df.keys():
        a = len(set_dictionary_test_df[key_test_df].intersection(set_dictionary_df[key_df]))
        b = len(set_dictionary_test_df[key_test_df].union(set_dictionary_df[key_df]))
        result = round(a/b, 3)
        if result >= 0.8:
            duplicates += 1
            break
```

Figure 15: Find duplicates

train set. Observe that this method checks all the questions of test set and at most the number of questions in train set. Finding only one question of train set which is similar to a question in test set with a threshold 0.8 is sufficient. As a result, the total running time of the algorithm is $\mathcal{O}(n \cdot m)$. This amount of time is confusing. On one hand, someone can say that this time is good enough but on the other hand they should consider that n, m may be exponentially large.

File ass1_ex2a_jaccard_exact.py contains the code for the aforementioned task.

LSH Jaccard

For starters, we perform the **Set Representation** procedure as before. Then, we compute the MinHash signatures for each question. MinHash creates a fixed length numeric "fingerprint" for each question. Furthermore, each number in this fingerprint is created by a random permutation of rows in the dictionary of words. To compare the similarity between two questions we compare their respective fingerprints and calculate their Jaccard Index, which is simply the number of integers that are in common between both signatures divided by the total number of integers in both signatures. LSH is a technique that hashes similar input items, with respect to a known threshold, into the same “buckets” with high probability. We use **datasketch** library in python code.

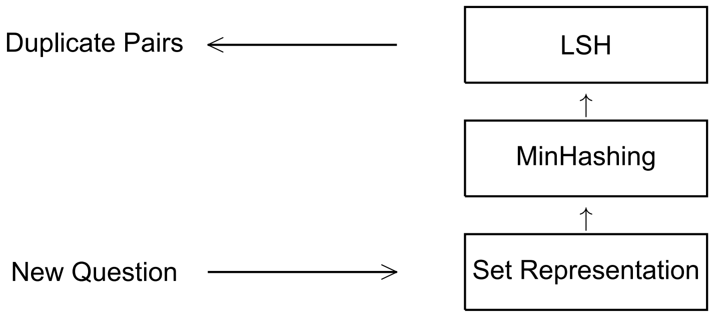
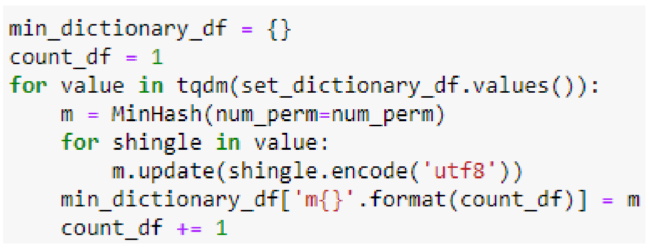


Figure 16: A quick overview of the process

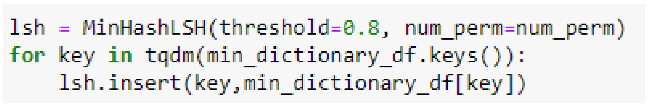
The number of permutations we want for the MinHash algorithm is expressed by *num_perm*. The higher the permutations are, the longer the runtime is. *Min_dictionary_df* maps question id to min hash signatures for the train set. We loop through all set representations of questions, compute the signatures and store them into *min_dictionary_df* dictionary. We repeat the same process for the test set and we create the *min_dictionary_test_df* dictionary.



```
min_dictionary_df = {}
count_df = 1
for value in tqdm(set_dictionary_df.values()):
    m = MinHash(num_perm=num_perm)
    for shingle in value:
        m.update(shingle.encode('utf8'))
    min_dictionary_df['m{}'.format(count_df)] = m
    count_df += 1
```

Figure 17: The mapping dictionary

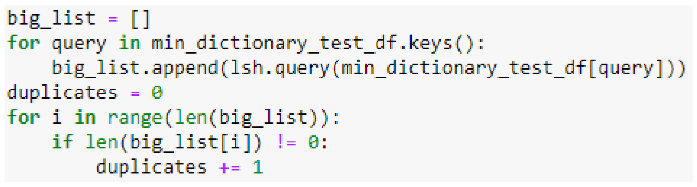
To create LSH index, we set the Jaccard similarity threshold as a parameter in MinHashLSH. We loop through the signatures or keys in the *min_dictionary_df* dictionary and store them as bands. These are stored in a dictionary format, where the key is a question and the values are all the questions deemed similar based on the threshold.



```
lsh = MinHashLSH(threshold=0.8, num_perm=num_perm)
for key in tqdm(min_dictionary_df.keys()):
    lsh.insert(key,min_dictionary_df[key])
```

Figure 18: Create LSH index method

Firstly, we initialize an empty list, defined as *big_list*. For every question of corpusTest.csv file, we store all the similar questions of corpusTrain.csv file as a form of a list into big_list. Thus, the length of big_list is equal to the total rows of corpusTest.csv file. Secondly, we initialize a variable, called *duplicates*, which will output the number of duplicates. We check every element/list of *big_list* and if the length of the element is not equal to zero then we increase variable duplicates by one.



```
big_list = []
for query in min_dictionary_test_df.keys():
    big_list.append(lsh.query(min_dictionary_test_df[query]))
duplicates = 0
for i in range(len(big_list)):
    if len(big_list[i]) != 0:
        duplicates += 1
```

Figure 19: Searching for duplicates

We consider that BuildTime is equal to the sum of time to create MinHash signatures on train set and time to create LSH index. In addition, we regard that QueryTime is equal to the sum of time to create MinHash signatures on test set and time it took to answer all the test set questions. All the times are measured in seconds.

Regarding the performance of the algorithm, when number of permutations is 16, BuildTime is 271, QueryTime took 7 and the number of duplicates are 1227. When the number of permutations is 32, BuildTime is 314, QueryTime took 8 and the duplicates are 1057. When the number of permutations is 64, BuildTime sums up to 398, QueryTime took 13 and duplicates are 1147. In the table below, we add the results we took when the number of permutations is 32.

File ass1_ex2a_jaccard_lsh.py contains the code for the aforementioned task.

Evaluation Results

The following table contains the expected results.

Type	Build Time	Query Time	Total Time	#Duplicates	Parameter
Exact-Jaccard	0	6627	6627	1101	-
LSH-Jaccard	314	8	322	1057	-

Figure 20: Evaluation results for question 2a

Question 2b: Same Question Detection

Classifier

-XGBoost

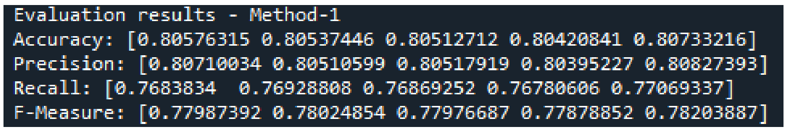
XGBoost is an optimized distributed gradient boosting library designed to be highly efficient and flexible. XGBoost provides a parallel tree boosting that solve many data science problems in a fast and accurate way. Boosting takes a more iterative approach. It's still technically an ensemble technique in that many models are combined together to perform the final one, but takes a more clever approach. Rather than training all of the models in isolation of one another, boosting trains models in succession, with each new model being trained to correct the errors made by the previous ones.

In python, we use *XGBClassifier* classifier as obtained by **xgboost** library.

Our Approach

Our goal for this task is to create a model that can answer if two questions are ultimately identical. We are given two csv files. One is the train set file that helps us train our algorithm and the other is the test without labels file that we will use to make predictions for the new data. We experiment with two different methods.

We are going to explain our first method. We begin with text preprocessing procedure on train and test set. Then, ”concat_questions” column is created by the concatenation of ”Question1” and ”Question2” columns. If we select this method for the predictions, then this column is going to be our train data. Also, we consider column 'IsDuplicate' as our target data. Afterwards, we transform every document in ”concat_questions” column into a vector according to TF-IDF feature. We select *RandomForestClassifier* and we proceed with 5-Fold cross validation. Figure 21 shows the evaluation results of this method.



```
Evaluation results - Method-1
Accuracy: [0.80576315 0.80537446 0.80512712 0.80420841 0.80733216]
Precision: [0.80710034 0.80510599 0.80517919 0.80395227 0.80827393]
Recall: [0.7683834  0.76928808 0.76869252 0.76780606 0.77069337]
F-Measure: [0.77987392 0.78024854 0.77976687 0.77878852 0.78203887]
```

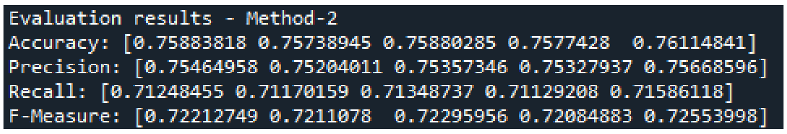
Figure 21: Evaluation results for RF classifier and TF-IDF feature with 5-Fold cross validation

For the second method, we proceed in a same way with a few differences. Instead of *RandomForestClassifier*, we select *XGBClassifier*. Furthermore, we transform our train data into vectors according to BoW feature. Below in Figure 22 we can see the evaluation results for this method.

File ass1_ex2b_eval_res.py contains the code for the aforementioned task.

Evaluation Results

The following table contains the expected results.



```
Evaluation results - Method-2
Accuracy: [0.75883818 0.75738945 0.75880285 0.7577428  0.76114841]
Precision: [0.75464958 0.75204011 0.75357346 0.75327937 0.75668596]
Recall: [0.71248455 0.71170159 0.71348737 0.71129208 0.71586118]
F-Measure: [0.72212749 0.7211078  0.72295956 0.72084883 0.72553998]
```

Figure 22: Evaluation results for XGB classifier and BoW feature with 5-Fold cross validation

Method	Precision	Recall	F-Measure	Accuracy
Method-1	0.8059	0.769	0.7801	0.8056
Method-2	0.754	0.713	0.7225	0.7588

Figure 23: Evaluation results for question 2b

Predictions

According to the evaluation results, we selected the first method to make the predictions for the similarity of two questions. We fit the data and the target data from train.csv file into RF classifier. Moreover, we create a new column for test set in the same matter as we do in train set and we transform the data into TF-IDF vectors. We proceed with the predictions by giving the previous vectors into the model. Our code outputs the duplicate_predictions.csv file which contains the predictions.

File ass1_ex2b_csv.py contains the code for this task.

Figure 24 is a countplot that illustrates the sum of each category from the output file.

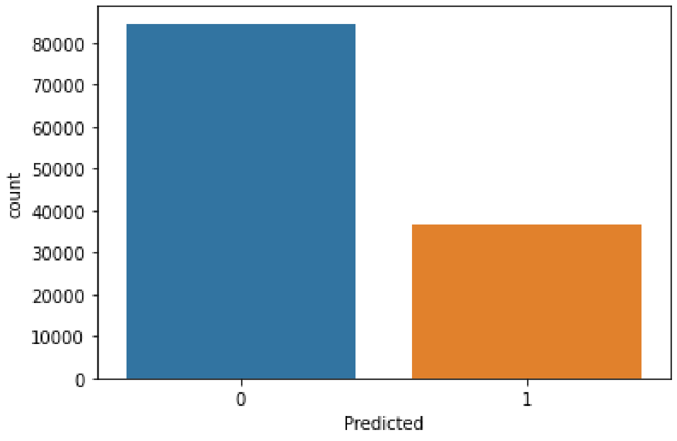


Figure 24: Visualization for predicted labels in question 2b

The End