Τεχνικές Ανάλυσης Δεδομένων Υψηλής Κλίμακας

ΜΑΛΩΝΑΣ ΚΩΝΣΤΑΝΤΙΝΟΣ
ID: 7115112200020
EMAIL: cs22200020@di.uoa.gr

Exercise 1

Data import

The first step is to import our **train** and **test data** from the csv files and transform them into **pandas** dataframes. We print the five (5) first lines of each dataframe with the head() method to get an idea of the data we will be working with.

Exploratory Data Analysis

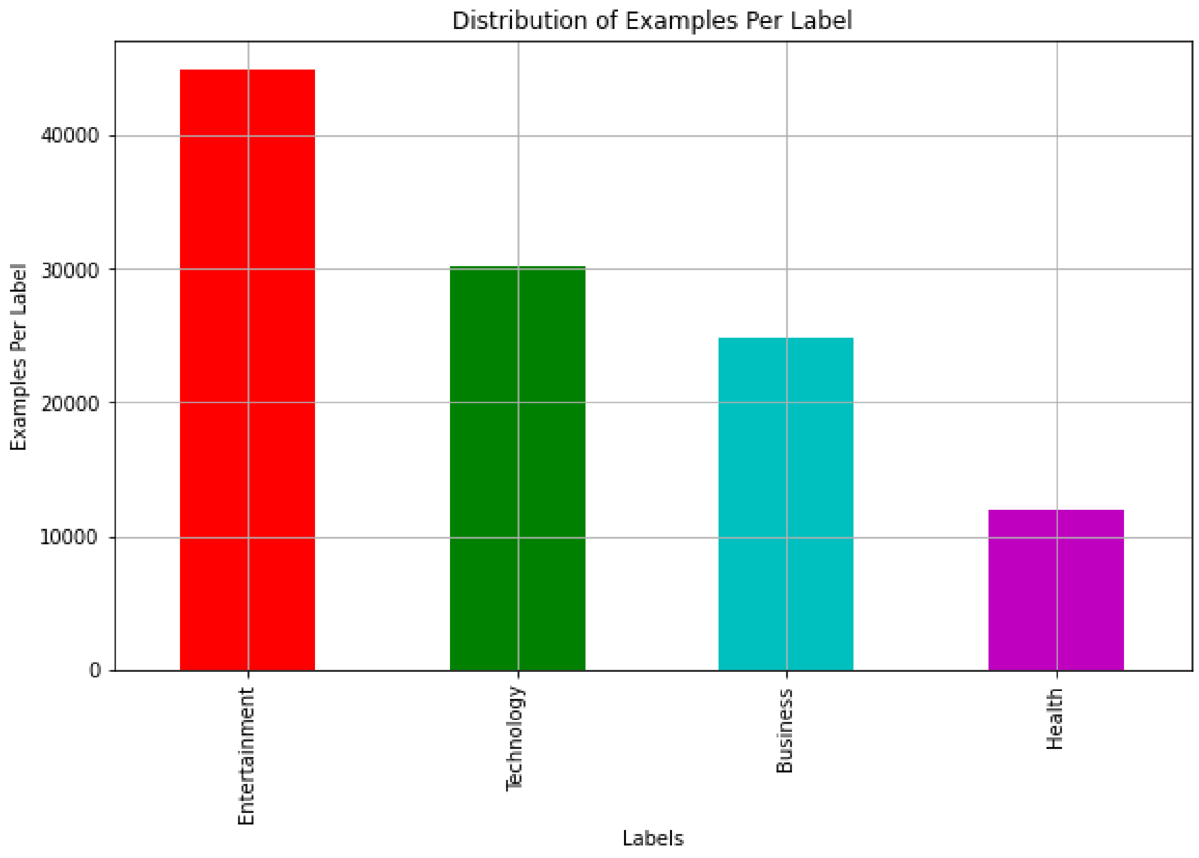
We perform some **EDA** to study our data in more depth. We observe that our train data consists of **111795** data points and four (4) features, which are Id (each article has a unique id), Title (the title of the article), **Content** (the text of the article) and **Label** (the category of the article) which is the target variable. Label column can take four (4) values which are Entertainment, Technology, Business, Health. When we print how many examples each Label has we get the following. We observe that most articles are classified as **Entertainment** articles.

```
Entertainment    44834
Technology       30107
Business         24834
Health           12020
Name: Label, dtype: int64
```

We also check if we have any nan values in our dataframe and we get the following, which indicates that we don't have any.

```
Id         0
Title      0
Content    0
Label      0
dtype: int64
```

Also we have plotted a **barplot** to show how many examples each Label contains and we got the following plot.



Data preprocessing

For the data preprocessing purpose we define the **Text_Preprocessin class**. The aforementioned class has the following methods:

- **convert_text_to_lowercase**: For converting the text in Title and in Content columns to lowercase
- **remove_punctuation**: For removing any punctuation and special characters
- **remove_stopwords**: For removing the english stopwords
- **remove_extras_spaces**: For removing extra spaces
- **lemmatize**: For lemmatization purposes
- **encode_labels**: For turning categorical values into numerical
- **get_df**: For returning the new and processed dataframe

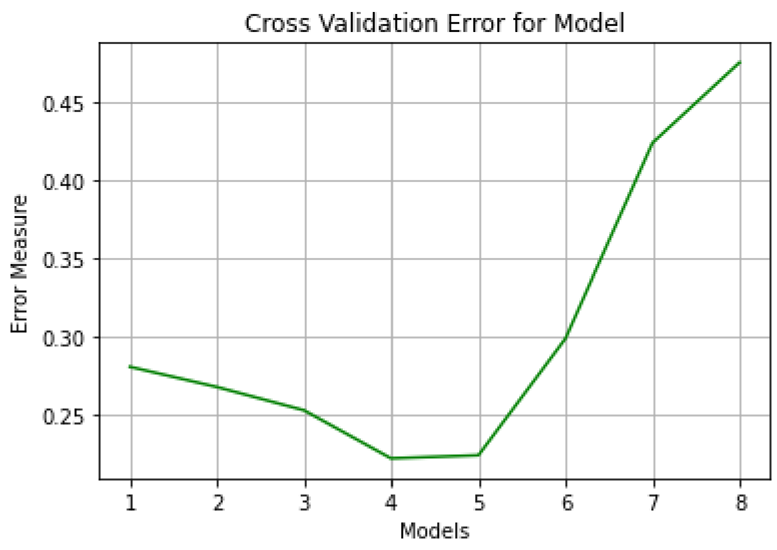
We process with the aforementioned methods both our train and our test dataframes. We also **split** our train data into train and test sets so we can train our models, get predictions and see what accuracy each model scores. The last and one of the most important steps of the data preprocessing phase is to transform the categorical data into numerical with the **TfidfVectorizer** library and normalizing them.

Experimenting with different models

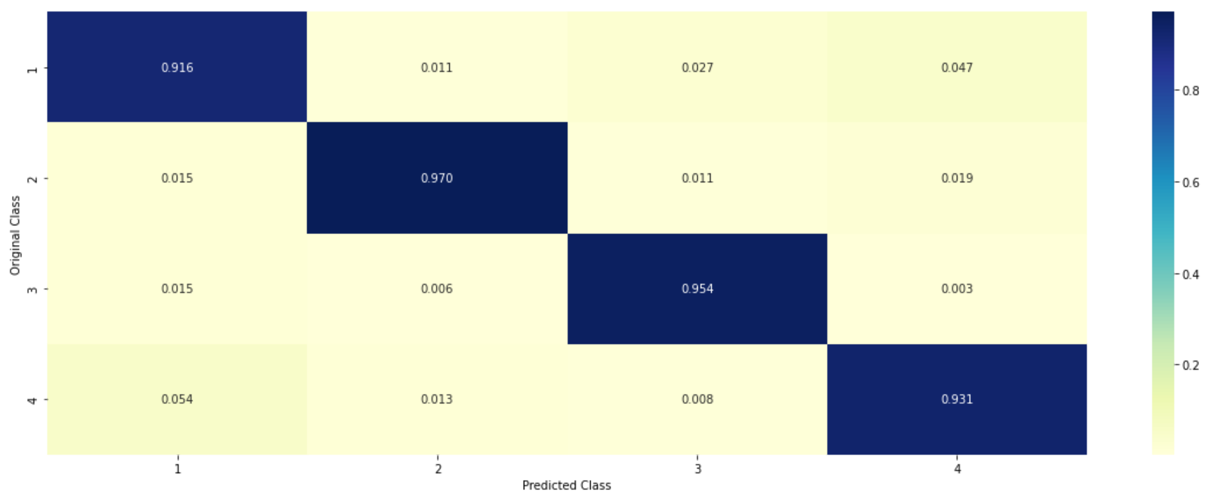
Multinomial Naive Bayes classifier

Content feature

The first model we trained is a **Multinomial Naive Bayes** model. We trained that model first with the **Content** feature and then with the **Title** feature to see which one produces the most accurate predictions and choose that one for later experimentation. We have combined every model with the **CalibratedClassifierCV** for better accuracy. Also we have experimented with the hyperparameter **alpha** of **MultinomialNB** of **scikit-learn** to find the model that gives us the best results. For every model we print the **Log-Loss**. From the following line plot we see that the fourth model gives us the best results using the Content feature.



Below we observe the percent of correct classified examples for each class.

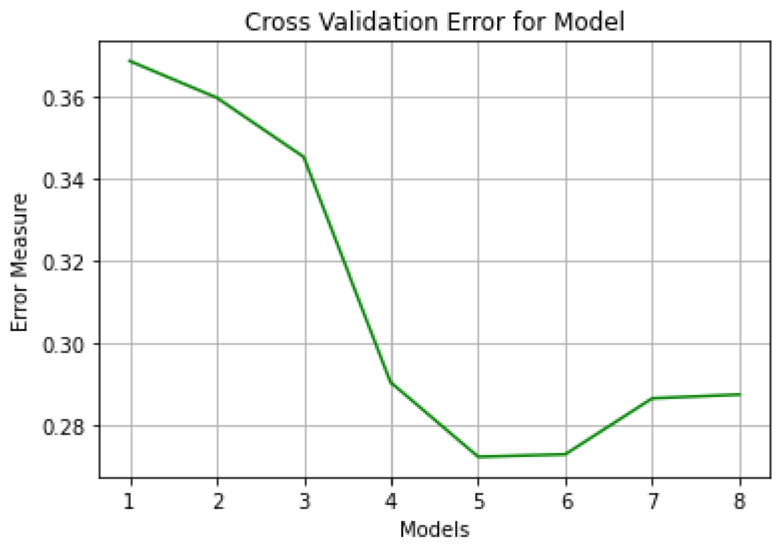


The **train accuracy** the Multinomial Naive Bayes classifier gave us for training with the content feature was **96.58%**. Also we print the **classification report** to see how our best model scored for each label. For example, for our model we got the following classification report. The **best model** will be selected based on the **f1-score** the **classification reports** will give us.

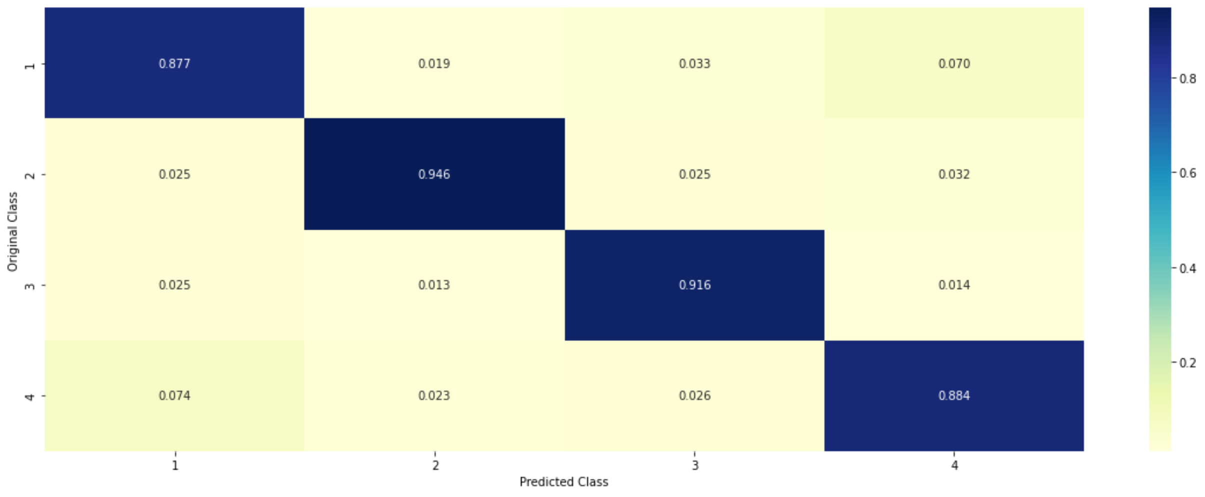
	precision	recall	f1-score	support
Entertainment	0.90	0.91	0.91	4889
Technology	0.98	0.96	0.97	9109
Business	0.93	0.95	0.94	2350
Health	0.92	0.93	0.93	6011
accuracy			0.94	22359
macro avg	0.93	0.94	0.94	22359
weighted avg	0.94	0.94	0.94	22359

Title feature

We performed the same operations with the **Title feature** and we got **93.72% accuracy**, so the later experiments will be performed only for the Content feature. Below we see that the fifth model gave us the best results.



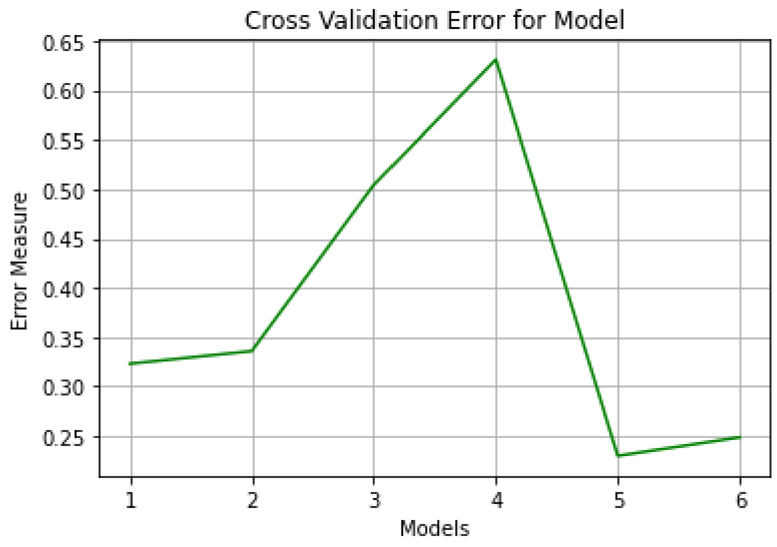
Below we see the confusion matrix for the Title feature. We can observe that the percentage of correct classified examples is lower.



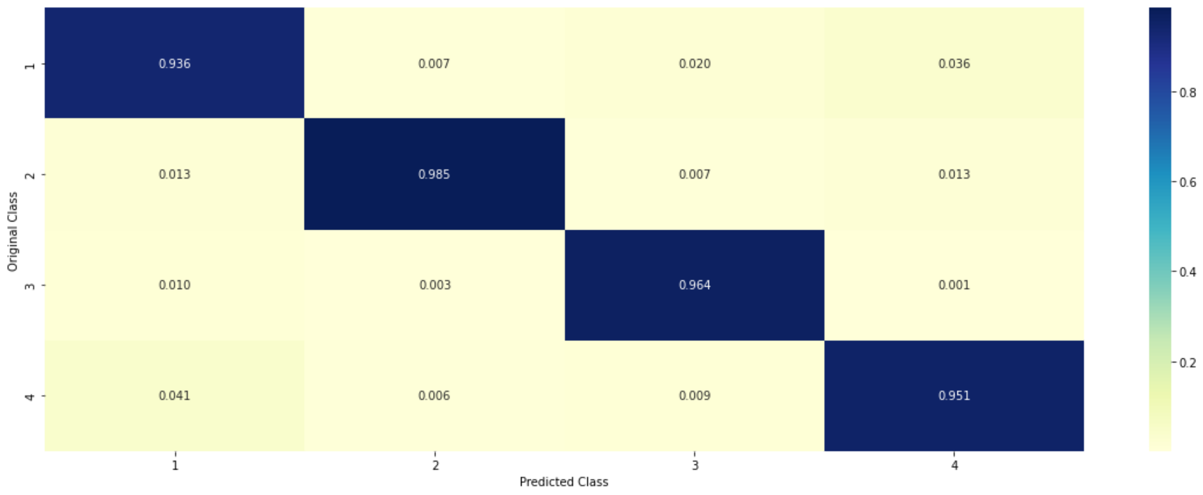
	1	2	3	4
1	0.877	0.019	0.033	0.070
2	0.025	0.946	0.025	0.032
3	0.025	0.013	0.916	0.014
4	0.074	0.023	0.026	0.884

SGD classifier

The next classifier we tested from the scikit-learn library was the **SGDClassifier**. We trained six (6) different models using that classifier, using different hyperparameters like **loss** (hinge, modified_huber), **penalty** (l1, l2) and **alpha** (0.0001, 0.0002). The accuracy we got from that classifier was **96.70%**. The best model is the fifth one with values **modified_huber** for the loss hyperparameter, **l2** for the penalty and **0.0001** for the alpha.



We also generate a confusion matrix where we can see the precision regarding each predicted class.



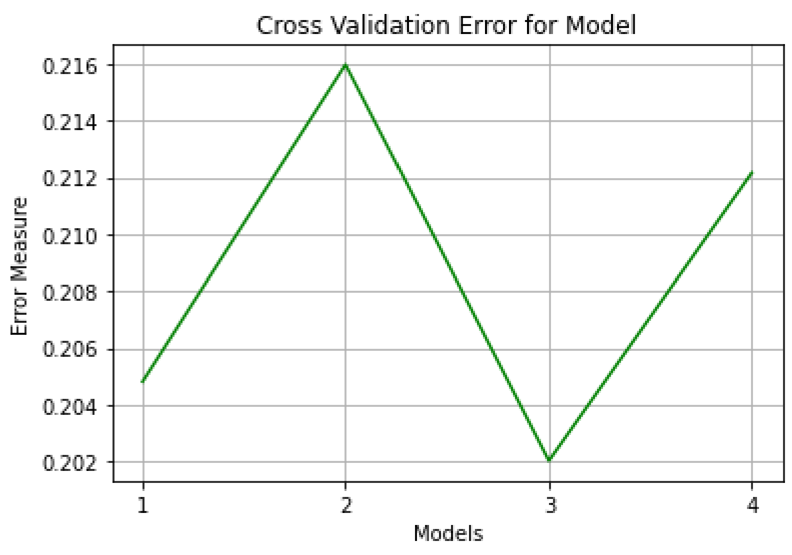
Original Class \ Predicted Class	1	2	3	4
1	0.936	0.007	0.020	0.036
2	0.013	0.985	0.007	0.013
3	0.010	0.003	0.964	0.001
4	0.041	0.006	0.009	0.951

Below we see the values of the classification report that we generated for the **SGD classifier**. We will make our final predictions with that classifier since it has the highest **f1-score**.

	precision	recall	f1-score	support
Entertainment	0.94	0.94	0.94	4965
Technology	0.98	0.98	0.98	8942
Business	0.97	0.96	0.97	2412
Health	0.95	0.95	0.95	6040
accuracy			0.96	22359
macro avg	0.96	0.96	0.96	22359
weighted avg	0.96	0.96	0.96	22359

Random Forest classifier

We also performed experiments with the Random Forest classifier (RandomForestClassifier) with the hyperparameters **n_estimators** (100, 150) and **criterion** (gini, entropy). The best model was the one with **150 n_estimators** and the value **gini** for the **criterion** hyperparameter with **log-loss** of **0.19** and train accuracy of **99.83%**.



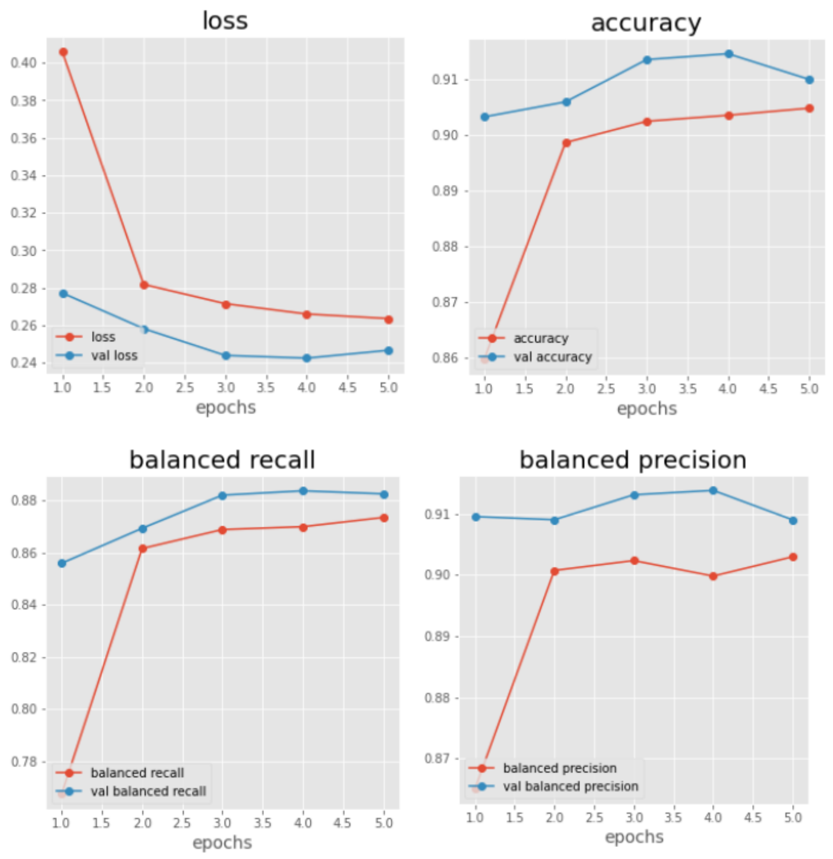
	precision	recall	f1-score	support
Entertainment	0.89	0.91	0.90	4876
Technology	0.97	0.96	0.97	9053
Business	0.93	0.94	0.93	2378
Health	0.92	0.92	0.92	6052
accuracy			0.94	22359
macro avg	0.93	0.93	0.93	22359
weighted avg	0.94	0.94	0.94	22359

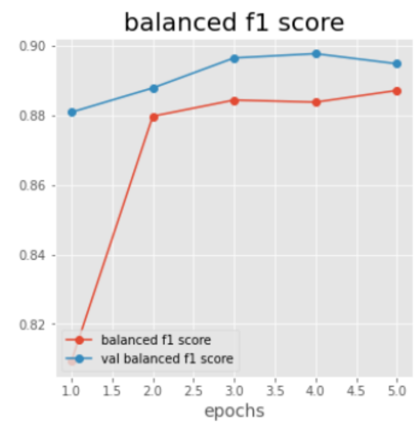
Experiments with Bert model and LSTM model

The code from the notebook for the following experiments executed in Google Collab and the notebook is saved as BERT_LSTM.ipynb

Bert model

We have also experimented with **Bert for text classification** model. We imported the model with **tensorflow hub**. We defined a model as the preprocessor and encoder layers followed by a dropout and a dense layer with a **softmax** activation function and an output space dimensionality equal to the number of classes we want to predict and we trained with the Content feature for **five** (5) **epochs**. The greatest accuracy we got was **90%**. Below the plots we generated for loss, accuracy, balanced recall, balanced precision and balanced f1 score. Observe that the **validation scores** are higher than the **training scores**, something that indicates that we might fit our model with too many training examples.





LSTM

We have also trained an **LSTM** neural network with **one input layer** that uses **100 length vectors** to represent each word. One hidden layer with **100 neurons** and one output layer that creates **four** (4) **output values**. The activation function we used was the **softmax** for multi-class classification. Our model gave us a maximum validation accuracy of **41%** so we did not continue with further experimentation. We trained our model for **five** (5) **epochs** and because of the **early stopping** technique it stopped at the **fourth epoch**. Below you see the results of the training for each epoch.

```
Epoch 1/5
1054/1054 [==============================] - 1585s 2s/step - loss: 0.0000e+00 - accuracy: 0.3992 - val_loss: 0.0000e+00 - val_accuracy: 0.4120
Epoch 2/5
1054/1054 [==============================] - 1509s 1s/step - loss: 0.0000e+00 - accuracy: 0.3992 - val_loss: 0.0000e+00 - val_accuracy: 0.4120
Epoch 3/5
1054/1054 [==============================] - 1489s 1s/step - loss: 0.0000e+00 - accuracy: 0.3992 - val_loss: 0.0000e+00 - val_accuracy: 0.4120
Epoch 4/5
1054/1054 [==============================] - 1469s 1s/step - loss: 0.0000e+00 - accuracy: 0.3992 - val_loss: 0.0000e+00 - val_accuracy: 0.4120
```

Best model selection

The best model was the **SGD classifier** trained with the **Content feature**. We selected that classifier based on the **f1-score** we got from the classification report table (**96%**)

Exercise 2

For that exercise we had to compare the **brute-force Knn** with the combination of **Lsh** and **Knn**. Before we implement our algorithms we preprocess our data, by removing punctuation, special characters, turning everything to lowercase, removing stopwords and performing lemmatization. After that we transformed our data to tfidf matrices.

Knn brute-force

For the **brute-force Knn** algorithm we used the **NearestNeighbors** method from **scikit-learn**. After training our model we construct the **results_dict** dictionary where each dictionary key (id of document) has as value the document id's (15 id's) of its neighbors.

Lsh-Jaccard

For the implementation of LSH we have used the **MinHash** library. First we created the **hashes** for the train and the test set and for each hash in the test set we used the **query** command to get the most similar hashes. After that inside the for loop for each hash of the test set we create a **Knn instance** and train it with the values of the most similar hashes. Finally we find the percent of the hashes returned from the Lsh approach. We also used different portions of our dataset (1000, 5000, 10000, 20000 e.t.c) examples.

Lsh-Cosine

For the implementation of the **Lsh-Cosine** we created hashes for each bucket, which are a string representation of the **dot product** of each train row e.g **'0111000100'** and inside the buckets we inserted the **indexes** of the similar rows. We did that both for the training and the test sets. Then for each hash of the test set that also exists in the train set we took the similar train data and trained a **Knn instance** and made predictions, for each row that exists in the same bucket. Finally we calculate how many similar results the two methods returned.

Type	**BuildTime (minutes)**	**QueryTime (minutes)**	**Total time**	**fraction of the true K most similar documents that are reported by LSH method as well**	**Parameters (different row for different K or for different number of permutations, etc)**
Brute-Force-Cosine	0	1.6	1.6	100%	K = 15 train_examples_used= 40000 (all)
LSH-Cosine	0	1.49	1.49	0.96%	buckets = 10 train_examples =40000 (all)
LSH-Cosine	0	46	46	15.24%	buckets = 4 train_examples =40000 (all)
Brute-Force-Cosine	0	13.7sec	13.7sec	100%	K = 15 train_examples_used= 10000
LSH-Cosine	0	18.4 sec	20.4 sec	0.59%	buckets = 10 train_examples =10000
LSH-Cosine	0	11.7	11.7	19.3%	buckets = 4 train_exam

					ples =10000
Brute-Force-Jaccard	0	14.16	14.16	100%	K = 15 train_examples_used= 10000
LSH-Jaccard	26.76	25	51.76	10.7%	threshold = 0.8 permutations = 128 train_examples_used = 10000
LSH-Jaccard	22.57	147.59	170.16	19.07%	threshold = 0.5 permutations = 50 train_examples_used = 10000
Brute-Force-Jaccard	0	9.5	9.5	100%	K = 15 train_examples_used= 5000
LSH-Jaccard	6.56	1.6	8.16	22.45%	threshold = 0.8 permutations = 128 train_examples_used = 5000
Brute-Force-Jaccard	0	0.7	0.7	100%	K = 15 train_examples_used= 1000
LSH-Jaccard	1.87	0.20	2.07	94.19%	threshold = 0.8 permutations = 28 train_examples_used= 1000

Exercise 3

For that exercise we implemented the DTW algorithm. First I imported the data which were in str format and I preprocessed them and transformed them into lists of numbers. Below is a step by step explanation of the **dtw** function that implements the DTW algorithm.

1. The function takes two sequences s and t as input.
2. The number of elements in each sequence is determined using len() and stored in variables n and m.
3. A 2D array dtw_matrix is created with n+1 rows and m+1 columns. Each element in the matrix is initialized to np.inf (a special value for positive infinity in the NumPy library).
4. The first element in the matrix (0,0) is set to 0.
5. The next two nested for-loops iterate through the dtw_matrix from (1,1) to (n,m), where n is the number of elements in s and m is the number of elements in t.
6. In each iteration, the cost between elements s[i-1] and t[j-1] is computed using abs() (the absolute value of the difference between the two elements).
7. The last_min value is then calculated as the minimum of the three values in the cells immediately above, left, and left-above the current cell.
8. The value of the current cell is then set to the sum of cost and last_min.
9. Finally, the dtw_matrix is returned as the result of the function.

Below is the function written in Python. The time for the termination of DTW was **74 minutes**.

```
def dtw(s, t):
    n, m = len(s), len(t)
    dtw_matrix = np.zeros((n+1, m+1))
    for i in range(n+1):
        for j in range(m+1):
            dtw_matrix[i, j] = np.inf
    dtw_matrix[0, 0] = 0

    for i in range(1, n+1):
        for j in range(1, m+1):
            cost = abs(s[i-1] - t[j-1])
            last_min = np.min([dtw_matrix[i-1, j], dtw_matrix[i,
j-1], dtw_matrix[i-1, j-1]])
            dtw_matrix[i, j] = cost + last_min
    return dtw_matrix
```