**CODE ANALYSIS**

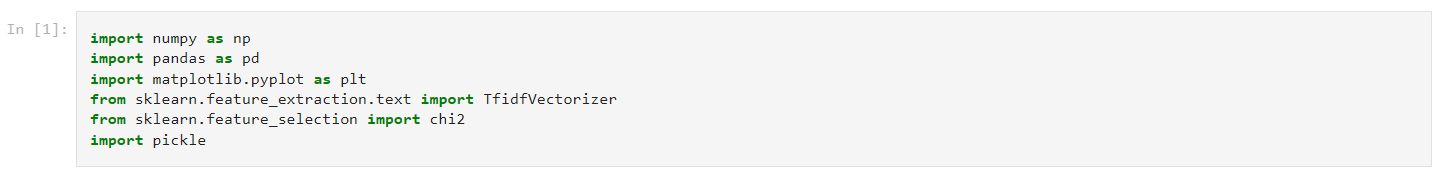
**Preparation Steps**

In this section, we look at each part of the code and understand the steps covered above. For this document, we stick to the text classification steps since it requires proper detailing. For processing steps, refer to the steps covered above and the code in link below.

For the BBC News classification code, please refer to the code link below.

Link for prediction steps code: <https://github.com/dhrubasil/BBC-News-Classification/blob/master/News%20Classificaiton.ipynb>

**(A). Importing important packages and libraries:**



We have imported ‘**numpy**’ and ‘**pandas**’ for basic data manipulation in our project.

We have imported ‘**TfidVectorizer**’ to convert a collection of raw documents to a matrix of TF-IDF features to measure how relevant a word is to a document; it is equivalent to ‘**CountVectorizer**’.

We have imported ‘**chi2**’ to compute chi-squared stats between each non-negative feature such as term counts in document classification relative to the classes.

We have imported ‘**pickle**’ for serializing and de-serializing a Python object structure. Any object on python can be pickled so that it can be saved on disk. At first Python pickle serialize the object and then converts the object into a character stream so that this character stream contains all the information necessary to reconstruct the object in another python script.

**(B). Loading the given Dataset from Kaggle:**

Graphical user interface, text, application

Description automatically generated

This training dataset is already in CSV format, and it has 1490 different texts, each labeled under one of 5 categories: entertainment, sport, tech, business, or politics. The dataframe has three columns, which is ‘*category*’ will be our **label**, ‘*ArticleId*’ and ‘*text*’ which will be our **input data** for **BERT**.

**(C). Performing Exploratory Data Analysis (EDA) of the dataset:**

Graphical user interface, text, application, email

Description automatically generated

Using ‘**value\_counts()**’ we can see the unique values of the different categories and its label encoding (3: sport with 346 texts, 0: business with 336 texts, 2: politics with 274 texts, 4: entertainment with 273 texts and 1: tech with 261 texts).

A picture containing application

Description automatically generated

We have checked if there’s any null or missing values in the training dataset using ‘**isnull().sum()**’.

**(D). TF-IDF:**

**Graphical user interface, text, application

Description automatically generated**

Here, we have first initialized the TfidVectorizer function with these parameters:

1. **sublinear\_tf = True**: To apply sublinear tf scaling, ie., replace tf with ‘1+log(tf)’.

It is not always true that multiple occurrences of a term in a document mean more significance of that term in proportion to the number of occurrences (ex., if a word ‘country’ occurs for 20 times, it doesn’t indicate that the word will have 20 times the relevance). It’s a method of **reducing the impact of repeated terms**.

Sublinear tf-scaling is modification of term frequency, which calculates weight as following,

a. if

b. then return (0)

c. else return

2. **min\_df = 7**: ‘min\_df’ is used for removing terms that appear too infrequently. Here, we ignore terms that appear in less than 7 documents.

3. **norm = l2**: Sum of squares of vector elements is 1. The cosine similarity between two vectors is their dot product when l2 norm has been applied.

4. **encoding='utf-8'**: The file has been encoded in ‘utf-8’ format to work upon.

5. **ngram\_range = (1, 3)**: An n-gram is just a string of n words in a row. Here, the parameter ‘ngram\_range= (1,3), where 1 is the minimum and 3 is the maximum size of ngrams we will include in our features.

6. **lowercase = True**: It is used to convert all the characters to lowercase before tokenizing.

7. **stop\_words = 'english'**: Some of the words detected in the vocabulary of unique n-grams have little value, such as “would”, “you”, or “your”. These are so-called “stop words” and can safely be removed from the data. Stop words generally don’t contribute very much and can massively bloat the size of your dataset, which increases model training times and causes various other issues. As a result, it’s a common practice to remove stop words. The easiest way is to do this automatically by passing the argument (stop\_words='english').

After initializing the ‘Tfidf’, we **train the dataset** by transforming the dataframe using ‘**fit\_transform()**’ and ‘**toarray()**’ method converts the efficient representation of a sparse matrix that sklearn uses to our ordinary readable dense ndarray representation. Furthermore, we save the training data using ‘**pickle**’.

Graphical user interface, text

Description automatically generated

A picture containing graphical user interface

Description automatically generated

A picture containing background pattern

Description automatically generated

Now, we will look for top 5 categories where for each category, we find words that are highly correlated to it using ‘**chi square**’ test which is a statistical method to check the relationship (or correlation) between the variables. We converts the indices to ‘**feature\_names**’ and get a list of **unigram, bigrams, and trigrams** (in increasing order of chi-squared stat values). Finally, printed the unigram, bigrams and trigrams with the highest value of chi-squared stat values.

**(E). Splitting the dataset into Training and Test data and building the Random Forest Classifier:**



We have imported ‘**RandomForestClassifier**’ & ‘**train\_test\_split**’ from scikit-learn library for classification and training, testing, and splitting the pre-processed data.

We have imported ‘cross\_val\_score’ from scikit-learn library to evaluate the test model over multiple folds of our dataset.

The final pre-processing step is to split our processed dataset into ‘**training**’ and ‘**test**’ sets. ‘test\_size’ parameter specifies the ratio of the test set, in which we used to split up **25% of the data into the test set and 75% for training**. We have created our **random forest classifier** (**model**) and then train it on the train set.

The ‘**fit**’ method of this class is called to **train the algorithm on the training data**, which is passed as parameter to the ‘fit’ method.

**(F). Evaluating the Model:**

Graphical user interface, text, application, email

Description automatically generated

After training the random forest model, we saved the model using ‘**pickle**’. After our classifier is trained, we made predictions on our test data. To make predictions, the ‘**predict**’ method of the ‘RandomForestClassifier’ class is used.

We have imported ‘**classification\_report**’ to evaluate and analyze the random forest model where in classification, some commonly used metrics are **‘confusion matrix’, ‘accuracy score’, ‘precision’, ‘recall’ & ‘F1-score or F-measure’** which can be used using the Scikit-learn ‘**metric**’ library (**accuracy\_score, confusion\_matrix, classification\_report**).

From the classification report of 373 test instances, we can see that the **accuracy of the model is 95%,** which is quite good. From the **weighted average**: the model is **95% precise** with **95% recall** and **f1-score of 0.95**.