NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

Project Report

Text Classification Project - Winter 2018/19

Maroulis Nikolaos - Μαρούλης Νικόλαος
AM: M1605
Msc. At Department of informatics (Data Science)

Course:	**Big Data Mining Techniques (M118)**
Professor:	**Dimitrios Gounopoulos**

February 2019

Introduction

All the requested tasks have been answered and will be further explained below. For the project implementation, Python 3.6, Jupyter Notebook and many python libraries were used (especially Sklearn libraries), which will be mentioned in the corresponding section. Also, Pandas and Numpy Library was used for storing and processing the dataset.

The files in the deliverable are:

- wordcloud.ipynb: A Jupyter Notebook file which contains the code for the first part of the exercise, the wordcloud creation.
- 5 .png files with each category wordcloud.
- similarity.ipynb: A jupyter Notebook file with the implementation of the cosine similarity.
- duplicatePairs.csv: The requested csv output file. It contains the ids of the documents that have cosine similarity greater than 0.7
- classification.ipynb: A Jupyter Notebook file which contains the code for all the text classification part, where various techniques are being tested.
- EvaluationMetric_10fold.csv: Contains the requested results in csv format
- roc_10fold.png: A png file of the results
- test_prediction.ipynb: code for the testSet_categories.csv creation.
- testSet_categories.csv: contains the class predictions for the given documents based on my ‘Beat the Benchmark’ model.

WordCloud Creation

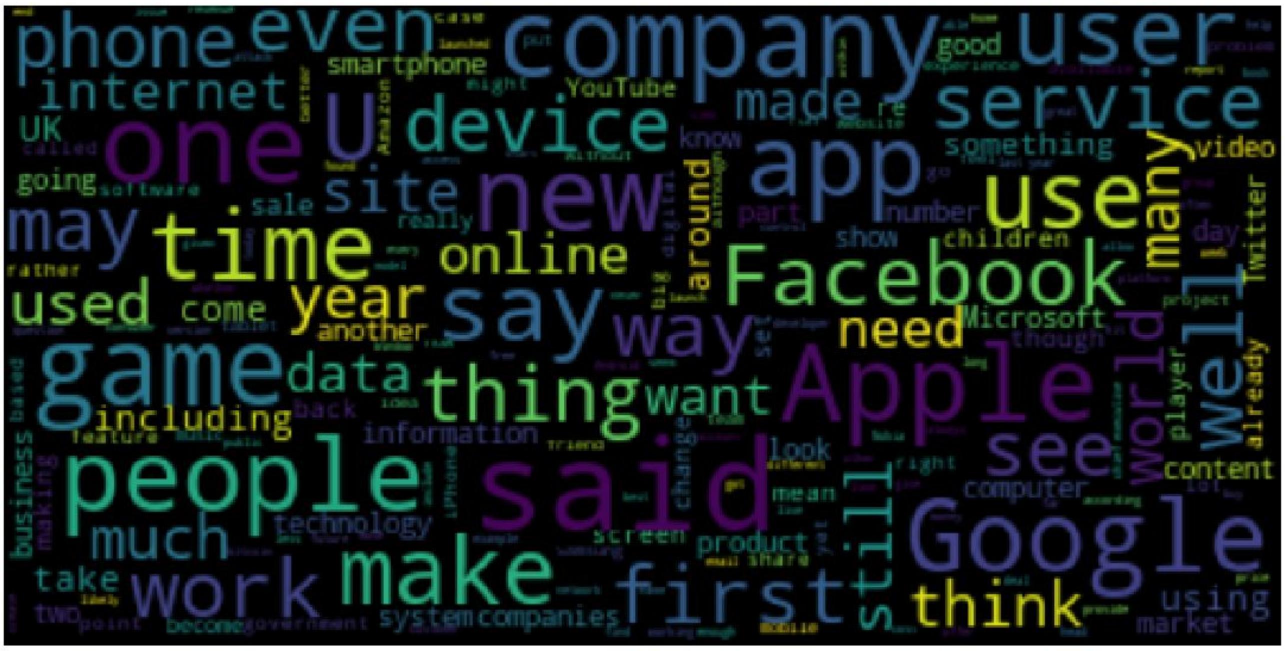
What are Word Clouds?

Word clouds (also known as text clouds or tag clouds) work in a simple way: the more a specific word appears in a source of textual data (such as a speech, blog post, or database), the bigger and bolder it appears in the word cloud.

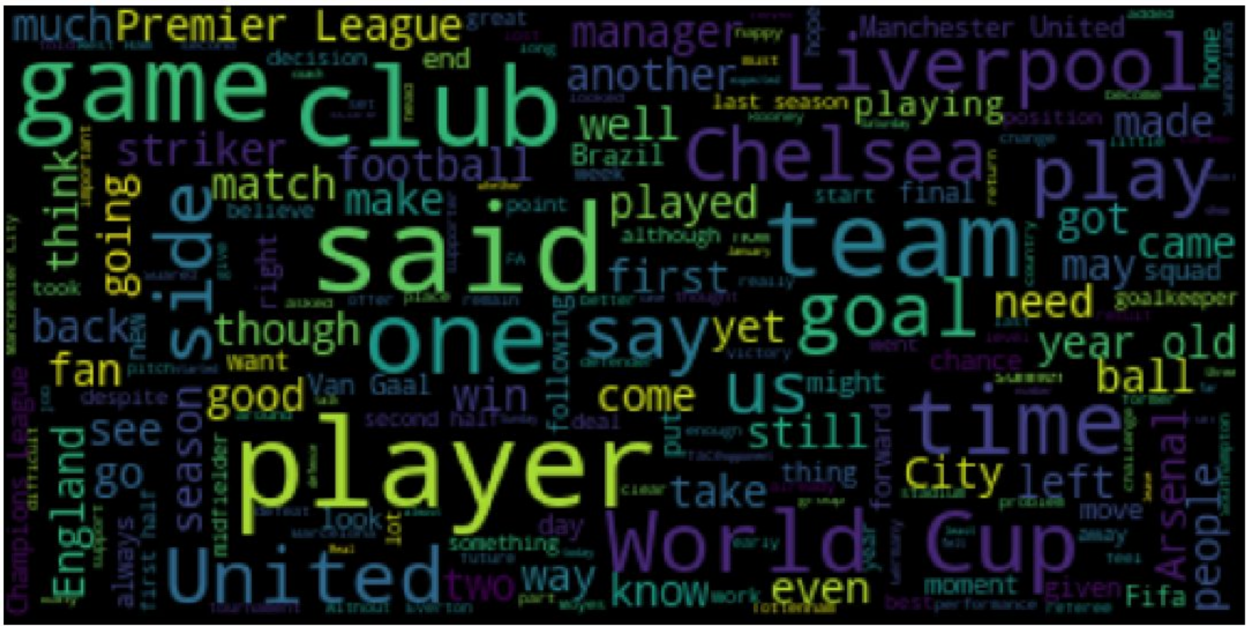
For my implementation I used the WordCloud Library (can be found here https://github.com/amueller/word_cloud).
First I got a statistic about how many words appear in each category:
Politics: 2148114,
Football: 1907737,
Business: 1759077,
Film: 1615972,
Technology: 1160208

A moderate cleaning takes places, which doesn't consider stop words as words, because they are really frequent, so they would appear a lot. Some words that don't have a very specific meaning are shown, but in order to tackle that issue, I would have to exclude each specific word, which is rather inconvenient.
The Word Clouds are presented below.

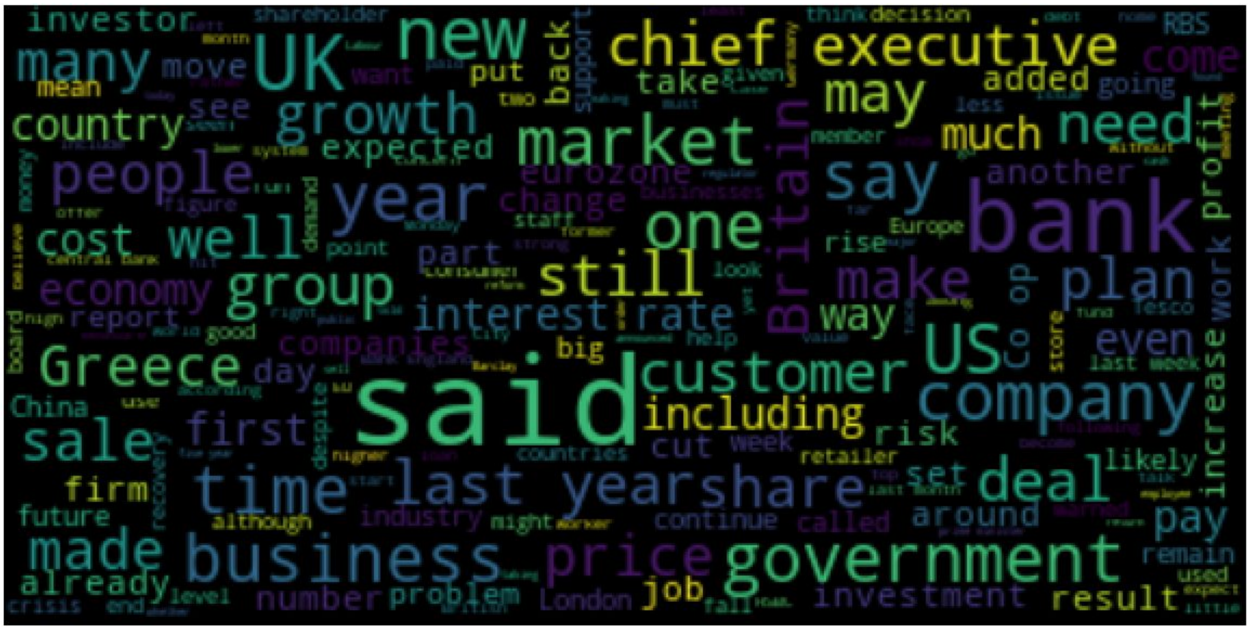
Technology



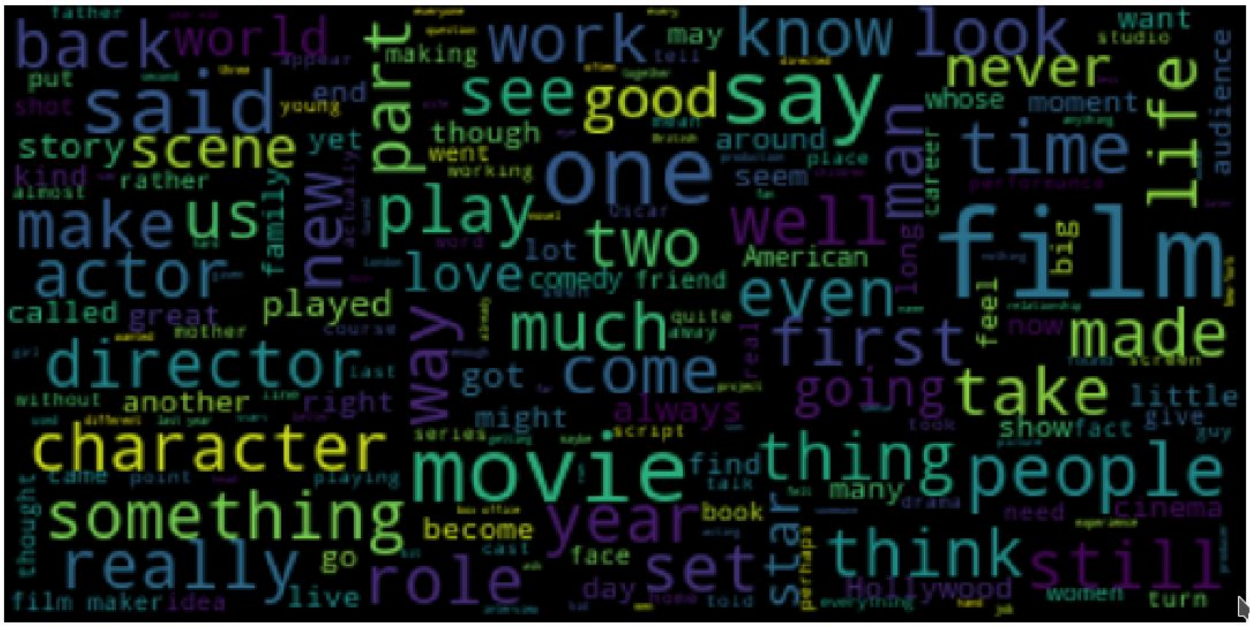
Football



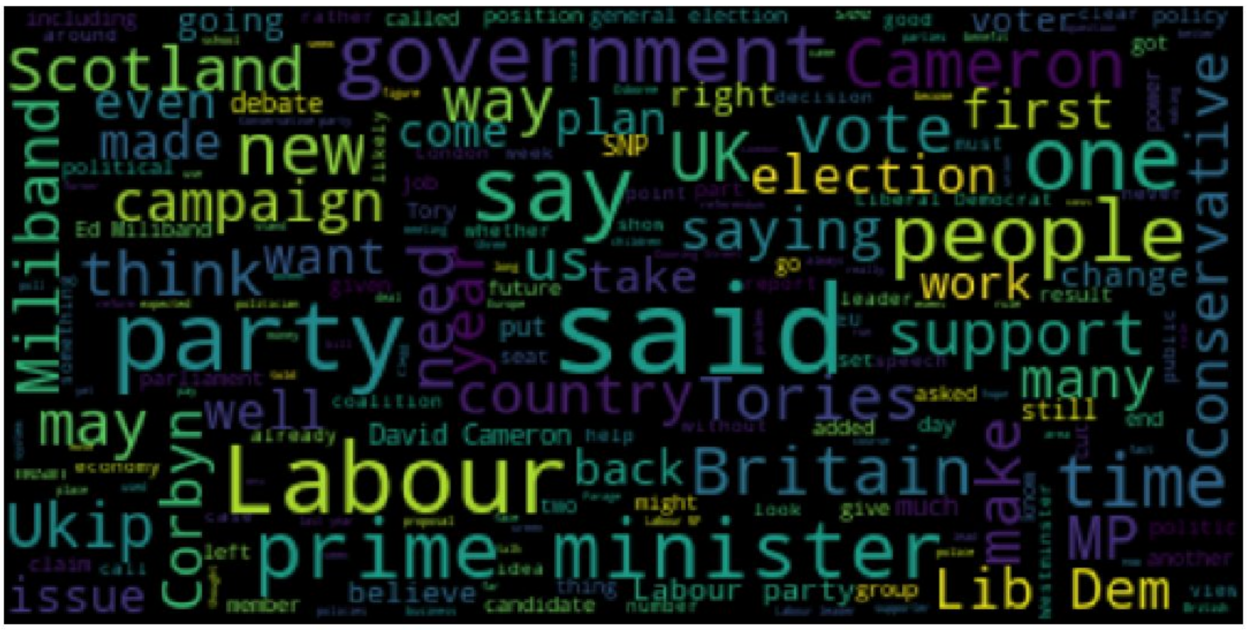
Business



Film



Politics



Duplicates Detection

For the implementation of this section I used two basic libraries, the <u>cosine_similarity</u> library and the <u>TfidfVectorizer</u> from sci-kit learn. First just for analytical purposes I found how many documents of the training set belong in each category:

Business	2735
Film	2240
Football	3121
Politics	2683
Technology	1487

First let's define what Tf-idf Vectorizer does.

What is **Tf-idf** ?

Computers are good with numbers, but not that much with textual data. One of the most widely used techniques to process textual data is TF-IDF. From our intuition, we think that the words which appear more often should have a greater weight in textual data analysis, but that's not always the case. Words such as "the", "will", and "you", called **stopwords**, appear the most in a corpus of text, but are of very little significance. Instead, the words which are rare are the ones that actually help in distinguishing between the data, and carry more weight.

TF-IDF stands for “Term Frequency — Inverse Data Frequency”. First, we will learn what this term means mathematically.

Term Frequency (tf): gives us the frequency of the word in each document in the corpus. It is the ratio of number of times the word appears in a document compared to the total number of words in that document. It increases as the number of occurrences of that word within the document increases. Each document has its own tf.

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{i,j}}$$

Inverse Data Frequency (idf): used to calculate the weight of rare words across all documents in the corpus. The words that occur rarely in the corpus have a high IDF score. It is given by the equation below.

$$idf(w) = log(\frac{N}{df_t})$$

Combining these two we come up with the TF-IDF score (w) for a word in a document in the corpus. It is the product of tf and idf:

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

$tf_{i,j}$ = number of occurrences of i in j
df_i = number of documents containing i
N = total number of documents

So in simple terms if a word comes up a lot in all the Documents it is considered less important than others. Because the term "the" is so common, term frequency will tend to incorrectly emphasize documents which happen to use the word "the" more frequently, without giving enough weight to the more meaningful terms "brown" and "cow". The term "the" is not a good keyword to distinguish relevant and non-relevant documents and terms, unlike the less-common words "brown" and "cow". Hence an inverse document frequency factor is incorporated which diminishes the weight of terms that occur very frequently in the document set and increases the weight of terms that occur rarely.

What is a Text Vectorizer?

To represent documents in vector space, we first have to create mappings from terms to term IDS. We call them terms instead of words because they can be arbitrary n-grams not just single words. We represent a set of documents as a sparse matrix, where each row corresponds to a document and each column corresponds to a term. This can be done in 2 ways: using the vocabulary itself or by feature hashing.

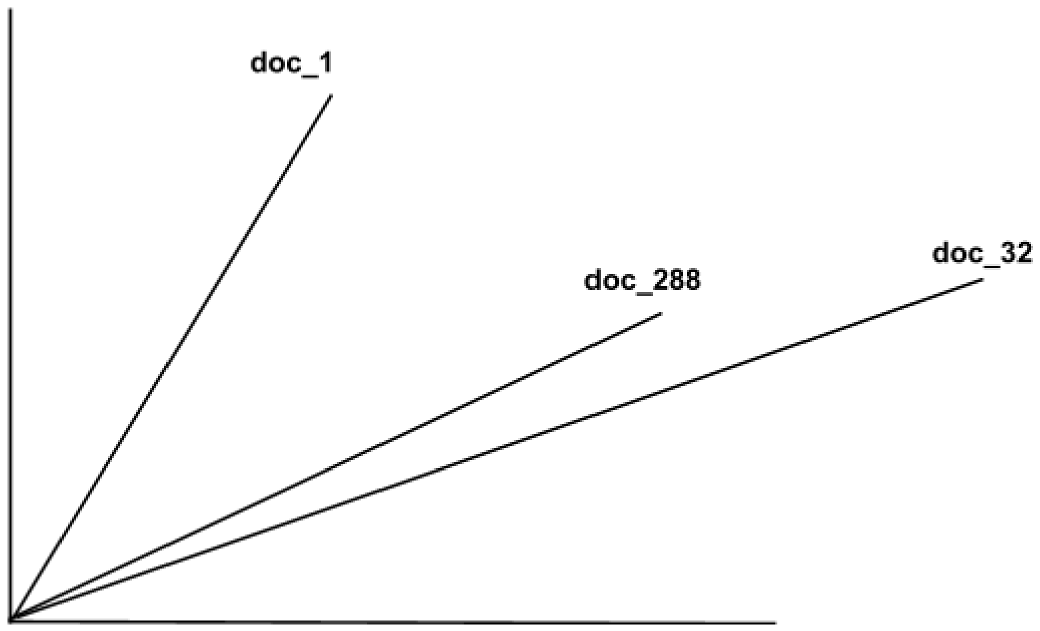
Initial Document Corpus

Document ID	Content
doc_1	Sed ut perspiciatis unde omnis...
doc_2	Ut enim ad minima veniam, quis nostrum..
…..	…..
doc_n	Quis autem vel eum iure reprehenderit….

So we Start with a Corpus of documents where each row represents a document with (let’s say) 2 columns, one the document id and the other the text content. We want to represent that in the vector space so we create a Matrix where each row is the document, but there we have a column for each word (term) in the *entire Corpus. So a simple solution is to have 1 in the column that the term appears in the corresponding document. The tf-idf vectorizer, instead of just filling 1, or the number of occurrences in the corresponding field, additionally, adds a weight to the term based on its frequency (low weight if it appears a lot).*

Term Matrix

	t1	t2	t3	t4	t5	t6	…..	t_m
doc_1	0	0	0	1	1	0		1
doc_2	1	1	0	0	0	0		0
…..								
doc_n	0	0	0	0	0	1		0



Document Representation in the Vector Space

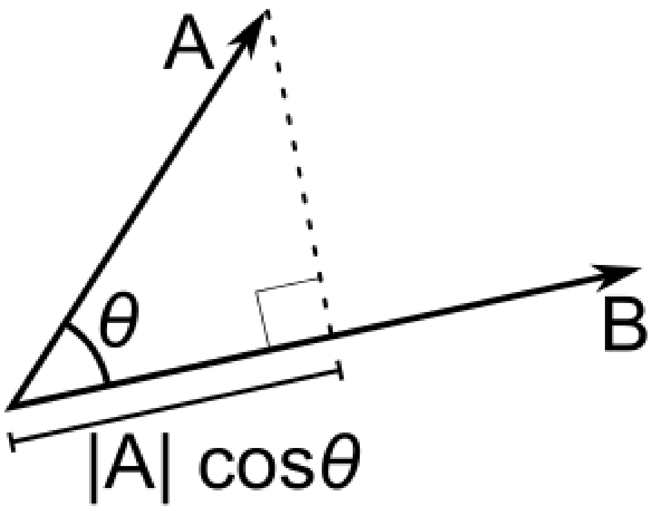
What is the Cosine Similarity?

The cosine similarity between two vectors (or two documents on the Vector Space) is a measure that calculates the cosine of the angle between them. This metric is a measurement of orientation and not magnitude, it can be seen as a comparison between documents on a normalized space because we're not taking into the consideration only the magnitude of each word count (tf-idf) of each document, but the angle between the documents. What we have to do to build the cosine similarity equation is to solve the equation of the dot product for the $\cos\theta$:

$$\vec{a} \cdot \vec{b} = \|\vec{a}\|\|\vec{b}\| \cos\theta$$

$$\cos\theta = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\|\|\vec{b}\|}$$

A graphical representation:



The projection of the vector A into the vector B. By Wikipedia.

So now that we have the matrix, we take each term vector for each document and calculate the cosine similarity. We set a threshold **θ** and if it is greater than 0.7 we assume that the two documents are similar.

```
similarity_df = train_df['Content'].head(np.size(train_df,0)).values

tfidf_vectorizer = TfidfVectorizer()

tfidf_matrix = tfidf_vectorizer.fit_transform(similarity_df)

cos_similarity_matrix = cosine_similarity(tfidf_matrix, tfidf_matrix)
```

What the code above does, is to create the term matrix and the calculate the cosine similarity for all the documents with each other and create a matrix with these similarities. I then iterate the matrix to find similarities greater than 0.7 (as shown below)

```
for i in range(tfidf_matrix.shape[0]):
    for c in range(len(cos_similarity_matrix[i])):
        if cos_similarity_matrix[i][c] > sim_threshold  and c != i:
            if([c,i,cos_similarity_matrix[i][c]] not in similarity_vector): # prevent duplicate entry
                similarity_vector.append([i,c,cos_similarity_matrix[i][c]])
```

Then I append the results in duplicatePairs.csv, containing the document IDs of the similar document and the similarity indicator.

Document Classification

What is Classification?

Classification is the process of predicting the class of given data points. Classes are sometimes called as targets/ labels or categories. Classification predictive modeling is the task of approximating a mapping function (f) from input variables (X) to discrete output variables (y). Classification belongs to the category of supervised learning where the targets also provided with the input data.

In this project the Task is to classify text documents into one of the following Categorie ***Film***, ***Politics***, ***Football***, ***Business*** and ***Technology.*** Given a training set where each entry contains a Document ID, its Text Content and the given Label (Class), we train a model and test it to take its accuracy.

Required Steps to Classify a document corpus:

1) Split dataset to Training and Testing Data.
2) Vectorize / Transform: Convert a collection of text documents to a matrix of token counts.
3) Train a model given the training data and test it with the Test data. (10 fold cross validation)
4) Aggregate Results (Accuracy, Recall, Precision, F1-Score) and compare each model’s performance.

What is K-fold Cross Validation?

Cross-validation is a resampling procedure used to evaluate machine learning models on a limited data sample. The procedure has a single parameter called k that refers to the number of groups that a given data sample is to be split into. As such, the procedure is often called k-fold cross-validation. When a specific value for k is chosen, it may be used in place of k in the reference to the model, such as k=10 becoming 10-fold cross-validation. Cross-validation is primarily used in applied machine learning to estimate the skill of a machine learning model on unseen data. That is, to use a limited sample in order to estimate how the model is expected to perform in general when used to make predictions on data not used during the training of the model.
The general procedure is as follows:

1. Shuffle the dataset randomly.
2. Split the dataset into k groups
3. For each unique group:
    1. Take the group as a hold out or test data set
    2. Take the remaining groups as a training data set
    3. Fit a model on the training set and evaluate it on the test set
    4. Retain the evaluation score and discard the model
4. Summarize the skill of the model using the sample of model evaluation scores

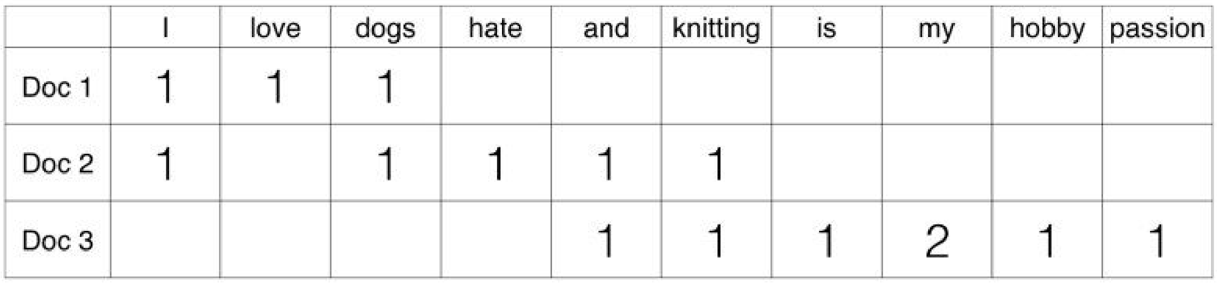
For the purposes of the project I implemented my own 10-fold cross validation algorithm, in order to be more flexible with the inner operations.

Text Transformation Techniques

Bag of Words (BoW)

In practice, the Bag-of-words model is mainly used as a tool of feature generation. After transforming the text into a "bag of words", we can calculate various measures to characterize the text. The most common type of characteristics, or features calculated from the Bag-of-words model is term frequency, namely, the number of times a term appears in the text. This I vector representation does not preserve the order of the words in the original sentences. This is just the main feature of the Bag-of-words model. This kind of representation has several successful applications. However, term frequencies are not necessarily the best representation for the text. Common words like "the", "a", "to" are almost always the terms with highest frequency in the text. Thus, having a high raw count does not necessarily mean that the corresponding word is more important. To address this problem, one of the most popular ways to "normalize" the term frequencies is to weight a term by the inverse of document frequency, or tf–idf.

For the Project Implementation I used **CountVectorizer** (from the sklearn library), which creates a BoW representation of the document in the vector space.
> CountVectorizer(analyzer ='word', lowercase=True).



	I	love	dogs	hate	and	knitting	is	my	hobby	passion
Doc 1	1	1	1							
Doc 2	1		1	1	1	1				
Doc 3					1	1	1	2	1	1

A BoW matrix example

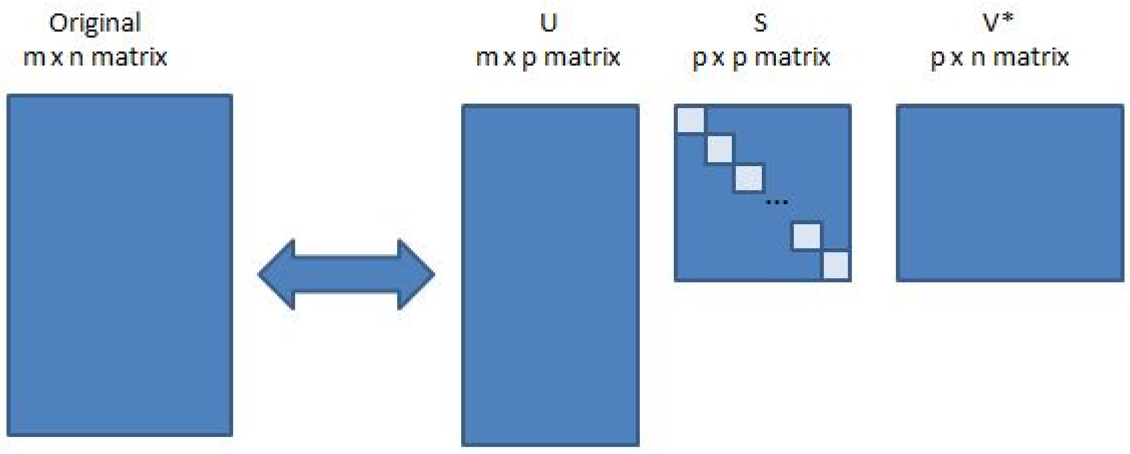
Singular Value Decomposition (SVD)

SVD, or Singular Value Decomposition, is one of several techniques that can be used to reduce the dimensionality, i.e., the number of columns, of a data set. Why would we want to reduce the number of dimensions? In predictive analytics, more columns normally means more time required to build models and score data. If some columns have no predictive value, this means wasted time, or worse, those columns contribute noise to the model and reduce model quality or predictive accuracy. Dimensionality reduction can be achieved by simply dropping columns, for example, those that may show up as collinear with others or identified as not being particularly predictive of the target as determined by an attribute importance ranking technique. But it can also be achieved by deriving new columns based on linear combinations of the original columns. In both cases, the resulting transformed data set can be provided to machine learning algorithms to yield faster model build times, faster scoring times, and more accurate models. While SVD can be used for dimensionality reduction, it is often used in digital signal processing for noise reduction, image compression, and other areas.

For the purposes of the Project I used the **TruncatedSVD** (from Sklearn),

> TruncatedSVD(n_components=component_num)

Where the attribute n_components is the desired dimensionality of output data, which must be strictly less than the number of features. In order to keep variance above 90% I wrote a function which calculates the right n_components, which is around **593.** This function will be shown in the next section.



SVD Dimensionality Reduction

Word to Vec (W2V)

Word2vec is a group of related models that are used to produce word embeddings. These models are shallow, two-layer neural networks that are trained to reconstruct linguistic contexts of words. Word2vec takes as its input a large corpus of text and produces a vector space, typically of several hundred dimensions, with each unique word in the corpus being assigned a corresponding vector in the space. Word vectors are positioned in the vector space such that words that share common contexts in the corpus are located in close proximity to one another in the space.

Word2vec can utilize either of two model architectures to produce a distributed representation of words: continuous bag-of-words (CBOW) or continuous skip-gram. In the continuous bag-of-words architecture, the model predicts the current word from a window of surrounding context words.

For the Project's purposes I used the Word2Vec model from Gensim models.
> model = Word2Vec(train_df['Content'], workers=num_workers, size=num_features, min_count = min_word_count, window = context, sample = downsampling)
Where

- *train_df['Content']*: The pandas dataframe column that contains all the corpus documents, it is given as input in order to **train the w2v model.**
- *workers*: number of threads in order to be parallel (chosen value 4)
- size: The size of each created word vector (chosen value 400)
- min_word_count: ignores all words with lower frequency than that. (chosen value 140)
- window_size: The maximum distance between the current and predicted word within a sentence.(chosen value 10)
- sample: The threshold for configuring which higher-frequency words are randomly downsampled. (chosen value 1e-3)

These values were chosen by me, I'm sure there can be a better configuration which gives better results. My main issue was that W2V model creates a **vector for each word**, but I needed **a vector for each document**. So i created the functions

MeanEmbeddingVectorizer and ***TfidfEmbeddingVectorizer***, which iterate each documents' words and takes each word's vector and take the mean. So each document's vector is the mean of all its' words' vectors. The TfidfEmbeddingVectorizer also uses Tf-Idf. I also experimented a little with the model and which can give the neighbors of words etc. An example is the code below:
> w2v_model.most_similar('youtube', topn=10), which finds the top 10 neighbor words for the word youtube (cosine similarity), the result:

```
('videos', 0.7882595062255859),
('channels', 0.7623605728149414),
('spotify', 0.7340512871742249),
('facebook', 0.7319403886795044),
('video', 0.7314403057098389),
('minecraft', 0.7295858263969421),
('content', 0.7102934718132019),
('ads', 0.7091004848480225),
('twitch', 0.7027051448822021),
('adverts', 0.6964372992515564)
```

Classifiers

Support Vector Machine (SVM)

In machine learning, support-vector machines (SVMs) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier (although methods such as Platt scaling exist to use SVM in a probabilistic classification setting). An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall.

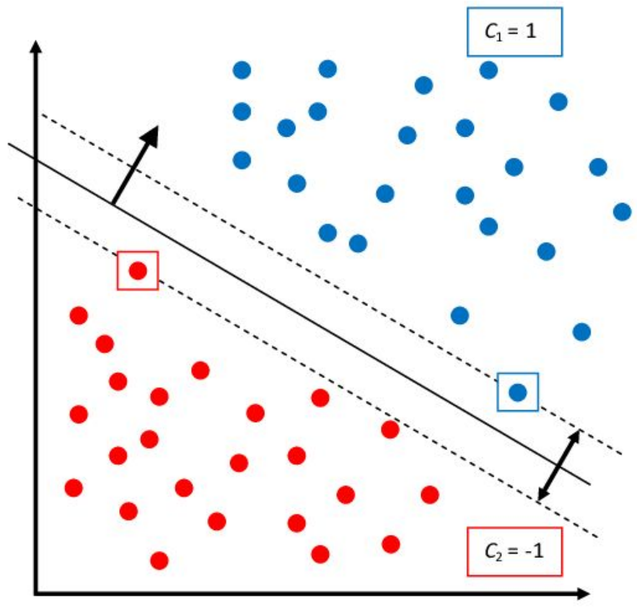
So in order to train an **SVM model for text classification,** I need to train it with data it can work with. That's why I used the Vectorizer to turn the docs into term vectors, because SVM works with vectors, separating them ideally.

Important Note: Because the size of term vector columns is greater than the training set feature vectors, they are linearly separable. (12266 feature vectors and many more columns -> terms)

I used the Sklearn Library for SVM

> svm.SVC(kernel='linear', C=1.0, decision_function_shape='ovr').

- kernel: Defines the **kernel function** to be used. A kernel function (or kernel trick), adds an additional dimension to the features in order to make them linearly separable by a hyperplane.
- C: Penalty Parameter for mismatched data (outliers)
- decision_function_shape: Because we have a multi-class dataset, we have to choose what the hyperplane separates. ovr means 'One vs Rest', so it looks at one class and all the other also as one.



A 2D Representation of an SVM Model. The feature vectors in squares are the support vectors and the dotted lines the margins.

Random Forests

Random forests or random decision forests are an ensemble learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. Random decision forests correct for decision trees' habit of overfitting to their training set. Decision trees are a popular method for various machine learning tasks. Tree learning come closest to meeting the requirements for serving as an off-the-shelf procedure for data mining, because it is invariant under scaling and various other transformations of feature values, is robust to inclusion of irrelevant features, and produces inspectable models.

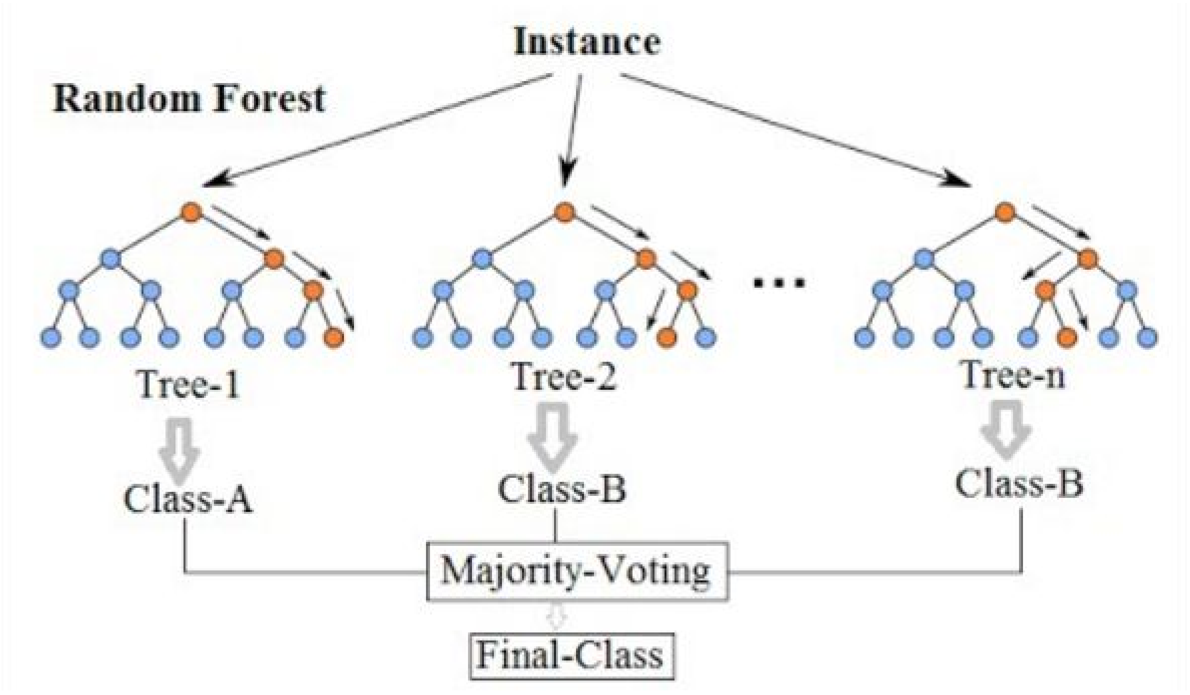
However, they are seldom accurate. In particular, trees that are grown very deep tend to learn highly irregular patterns: they overfit their training sets, i.e. have low bias, but very high variance. Random forests are a way of averaging multiple deep decision trees, trained on different parts of the same training set, with the goal of reducing the variance. This comes at the expense of a small increase in the bias and some loss of interpretability, but generally greatly boosts the performance in the final model.

I used the Sklearn Library for Random Forests

> RandomForestClassifier(n_estimators=240, max_depth=30, random_state=0)

- n_estimators: number of decision trees
- max_depth: The max depth of each decision tree.

These settings for the random forests were tested by me, having already tried various combination and I think they are close to the optimal ones. (but not the optimal)



Random Forest Decision Trees illustration

Model Performance Evaluation

Performance Indicators

1. Accuracy

Accuracy measures how many of the predicted classes of the test dataset were correctly classified. So it’s the fraction

Accuracy = Correctly_Predicted_Classes / Total_Number_of_Test_Data

e.g. if the test set has 100 feature vectors and we classify correctly the 92 of them, the total accuracy is 92%.

2. Precision

Precision talks about how precise/accurate the model is out of those predicted positive, how many of them are actual positive. Precision is a good measure to determine, when the costs of False Positive is high.

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

3. Recall

Recall calculates how many of the Actual Positives our model capture through labeling it as Positive (True Positive). Applying the same understanding, we know that Recall shall be the model metric we use to select our best model when there is a high cost associated with False Negative.

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

4. F1-Score

F1 Score might be a better measure to use if you need to seek a balance between Precision and Recall and there is an uneven class distribution. It’s just a simple fraction of precision and recall.

$$\text{F1} = 2 \text{ x } \frac{Precision * Recall}{Precision + Recall}$$

For The Project Implementation I used the sklearn metrics API. More specifically in my code below I implemented a 10-fold cross validation algorithm, where in each iteration I segment the train dataset into a training and test set (9/10 of the dataset are the training and 1/10 the test set) taking a different chunk of the data each time. I then train the model using the training set (model.fit(train_X, ttrain_Y)) and then I predict the labels for the test set (predicted = model.predict(test_X)) and store it in the predicted list.

Then by using :

precision += precision_score(test_Y, predicted, average='macro')

recall += recall_score(test_Y, predicted, average='macro')

acc += np.mean(predicted == test_Y)

f1_sc += f1_score(test_Y, predicted, average='macro')

I calculate the metrics required (macro means that it calculates metrics for each label) and finally by

result.append([acc/10, recall/10, precision/10, f1_sc/10])

return result

I return the mean of these measurements. The full code is shown below.

```
def ten_fold_cross_validation_metrics(model, model_name):
    """ten fold cross validation using the input model using sklearn metrics"""

    fold_size = int (np.size(train_df,0) / 10 )
    result = []
    acc = 0
    precision = 0
    recall = 0
    f1_sc = 0

    for i in range(10):

        from_ = i*fold_size
        to_ = (i+1)*fold_size
        if i == 9:
            to_ = 12266

        test_X = train_df['Content'][from_:to_] # Validation Set
        test_Y = train_df['Category'][from_:to_]
        train_set = train_df.drop(train_df.index[from_:to_])
        train_X = train_set['Content']
        train_Y = train_set['Category']
        _ = model.fit(train_X, train_Y)
        predicted = model.predict(test_X)
        precision += precision_score(test_Y, predicted, average='macro')
        recall += recall_score(test_Y, predicted, average='macro')
        acc += np.mean(predicted == test_Y)
        f1_sc += f1_score(test_Y, predicted, average='macro')
        print('10fcv - Iteration',str(i+1))
    result.append([acc/10, recall/10, precision/10, f1_sc/10])

    return result
```

Sklearn Pipeline

The Sklearn Pipeline API was used in the project, which sequentially applies a list of transforms and a final estimator. Intermediate steps of the pipeline must be ‘transforms’, that is, they must implement fit and transform methods. The final estimator only needs to implement fit.

Experiment Results

In this section I will demonstrate my experiments, showing the requested methods, as well as, my own experimentations.

SVM with Bag of Words

```
svm_bow = Pipeline([
    ('vect', CountVectorizer(analyzer ='word', lowercase=True)),
    ('svm', svm.SVC(kernel='linear', C=1.0, decision_function_shape='ovr')),])
```

Metric	Accuracy	Precision	Recall	F1-Score
Result	0.95206	0.94985	0.94844	0.94894

Random Forests with Bag of Words

```
"""
    number of decision trees: 240
    max_depth in tree: 30
"""
random_forest_bow = Pipeline([
    ('vect', CountVectorizer(analyzer ='word', lowercase=True)),
    ('random_forest', RandomForestClassifier(n_estimators=240, max_depth=30,
random_state=0)),])
```

Metric	Accuracy	Precision	Recall	F1-Score
Result	0.94390	0.94397	0.93197	0.93651

SVM using SVD

Below is the function for calculating the number of features for the SVD in order to keep 90% of the variance.

```
def find_svd_feature_size(var_ratio):

    total_variance = 0.0  # Set initial variance explained so far
    n_components = 0 # Set initial number of features
    # For the explained variance of each feature:
    for explained_variance in var_ratio:

        total_variance += explained_variance # Add the explained variance to the total
        n_components += 1  # Add one to the number of components

        # If variance >= 90%
        if total_variance >= 0.9:
            break
    return n_components
```

And the transformation - model code.

```
svm_svd = Pipeline([
    ('vect', CountVectorizer(analyzer='word')),
    ('svd', TruncatedSVD(n_components=component_num)),
    ('svm', svm.SVC(kernel='linear', C=1.0, decision_function_shape='ovr')),])
```

Metric	Accuracy	Precision	Recall	F1-Score
Result	0.94293	0.94159	0.93733	0.93918

Random Forests using SVD

```
random_forest_svd = Pipeline([
    ('vect', CountVectorizer()),
    ('svd', TruncatedSVD(n_components=component_num)),
    ('random_forest', RandomForestClassifier(n_estimators=240, max_depth=30,
    random_state=0)),])
```

Metric	Accuracy	Precision	Recall	F1-Score
Result	0.86605	0.88016	0.83771	0.84797

W2V Model

Below is the code of the w2v model training (explained above), where the training data is the document corpus.

```
num_features = 400          # Word vector dimensionality
min_word_count = 140        # Minimum word count
num_workers = 4             # Number of threads to run in parallel
context = 10                # Context window size
downsampling = 1e-3         # Downsample setting for frequent words

model = Word2Vec(train_df['Content'] , workers=num_workers,size=num_features,
min_count = min_word_count, window = context, sample = downsampling)

w2v = dict(zip(model.wv.index2word, model.wv.syn0))
```

The output of the model above is a vector for each word, so for the needs of the project, which is document classification, I needed to have a vector for each document, instead of a vector for each word. So I implemented the 2 functions below, which take all the words in each document and take the mean of all its' word vectors. The one called **MeanEmbeddingVectorizer** does just that, while the other **TfidfEmbeddingVectorizer** also uses Tf-Idf transformation.

```
class TfidfEmbeddingVectorizer(object):
    def __init__(self, word2vec):
        self.word2vec = word2vec
        self.word2weight = None
        self.dim = len(next(iter(word2vec.items())))

    def fit(self, X, y):
        tfidf = TfidfVectorizer(analyzer=lambda x: x)
        tfidf.fit(X)
        # if a word was never seen - it must be at least as infrequent
        # as any of the known words - so the default idf is the max of known idf's

        max_idf = max(tfidf.idf_)
        self.word2weight = cl.defaultdict( lambda: max_idf, [(w, tfidf.idf_[i]) for w, i in
        tfidf.vocabulary_.items()])

        return self

    def transform(self, X):
        return np.array([ np.mean([self.word2vec[w] * self.word2weight[w] for w in words
                          if w in self.word2vec] or [np.zeros(self.dim)], axis=0) for words in X])
```

SVM using the W2V model and TfidfEmbeddingVectorizer

```
w2v_svm = Pipeline([
    # ("word2vec vectorizer", MeanEmbeddingVectorizer(w2v)),
    ("word2vec vectorizer", TfidfEmbeddingVectorizer(w2v)),
    ('svm', svm.SVC(kernel='linear', C=1.0, decision_function_shape='ovr')),])
```

Metric	Accuracy	Precision	Recall	F1-Score
Result	0.71245	0.72638	0.67376	0.67575

Random Forest using the W2V model and TfidfEmbeddingVectorizer

```
w2v_rf = Pipeline([
    ("word2vec vectorizer", MeanEmbeddingVectorizer(w2v)),
    ('random_forest', RandomForestClassifier(n_estimators=240, max_depth=30, 
random_state=0)),])
```

Metric	Accuracy	Precision	Recall	F1-Score
Result	0.70169	0.70994	0.66521	0.66907

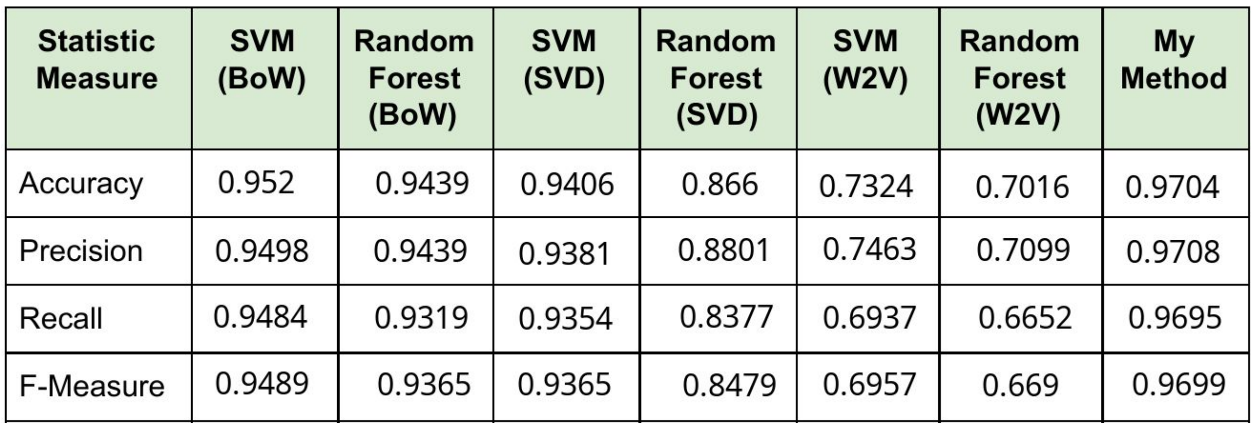
Beat the BenchMark Technique

I have tried various implementations (shown below), but the one I kept getting the better results was SVM. So I experimented with SVM, by tuning the Penalty value, kernel tricks etc. And I arrived at the conclusion that the **linear kernel** is the best for text classification, while the penalty value is optimal at 0.92. This test was done by the trial and error method. Also the CountVectorizer parameters were changed a bit, like making all letters to lowercase and removal of stop words. Finally a Tf-Idf transformation was used in order to minimize the weight of frequently used words. Some promising implementations will be shown below in the experimental ways, which came pretty close to be the ‘beat the benchmark’ algorithm.

```
text_clf_svm = Pipeline([
    ('vect', CountVectorizer(analyzer='word', stop_words='english', lowercase=True)),
    ('tfidf', TfidfTransformer()),
    ('svm', svm.SVC(kernel='linear', C=0.92, decision_function_shape='ovr')),])
```

Metric	Accuracy	Precision	Recall	F1-Score
Result	0.9704	0.97084	0.96951	0.96997

The table below illustrates the results of all the requested models, which can also be found in the EvaluationMetric_10fold.csv and .png.



Statistic Measure	SVM (BoW)	Random Forest (BoW)	SVM (SVD)	Random Forest (SVD)	SVM (W2V)	Random Forest (W2V)	My Method
Accuracy	0.952	0.9439	0.9406	0.866	0.7324	0.7016	0.9704
Precision	0.9498	0.9439	0.9381	0.8801	0.7463	0.7099	0.9708
Recall	0.9484	0.9319	0.9354	0.8377	0.6937	0.6652	0.9695
F-Measure	0.9489	0.9365	0.9365	0.8479	0.6957	0.669	0.9699

Experimental Classification Models

In order to test these models, I used a simple split, where 66.6% of the training dataset is used as the training set of the model and 33.3% is used as the test set.

This implementation is shown below:

```
def simple_split(model, model_name):
    """ 0.33 split for test data """
    train_X = train_df['Content'][0:8177]
    train_Y = train_df['Category'][0:8177]

    test_X = train_df['Content'][8177:12266]
    test_Y = train_df['Category'][8177:12266]

    _ = model.fit(train_X, train_Y)
    predicted = model.predict(test_X)
    result = np.mean(predicted == test_Y)
    res = '<split> Accuracy Score for ' + model_name + ' is ' + str(np.mean(predicted ==
test_Y))
    return res
```

Multinomial Naive Bayes Classifier with BoW and Tf-Idf

The general term Naive Bayes refers the the strong independence assumptions in the model, rather than the particular distribution of each feature. A Naive Bayes model assumes that each of the features it uses are conditionally independent of one another given some class. The multinomial approach simply lets us know that each feature's distribution is a multinomial distribution, rather than some other distribution.

The sklearn naive_bayes MultinomialNB API was used.

```
nb = Pipeline([
    ('vect', CountVectorizer(stop_words='english')),
    ('tfidf', TfidfTransformer()),
    ('naive_bayes', MultinomialNB(fit_prior=False)),])

print(simple_split(nb, 'Naive Bayes with BoW'))
```

ACCURACY 94.34%

K-nearest Neighbors (KNN) Classifier with BoW and Tf-Idf

KNN is a lazy learning model, which means it doesn't exactly create a model. The way it works is simple, it just takes the square root sum of all the Euclidean Distances between all the attributes of the test set feature vector with each feature vector in the Train Dataset. The feature vector with the smallest Euclidean distance from the test feature vector is considered as the closest neighbor, so the class that the test sample belongs, is its' neighbor's class. The K in KNN is the implementation where we choose the K (Integer > 0) closest neighbors, and pick the class where the majority of those neighbors have. For the implementation of KNN in this Project the optimal number of neighbors I found was **13.** The sklearn KNeighbors Classifier API was used.

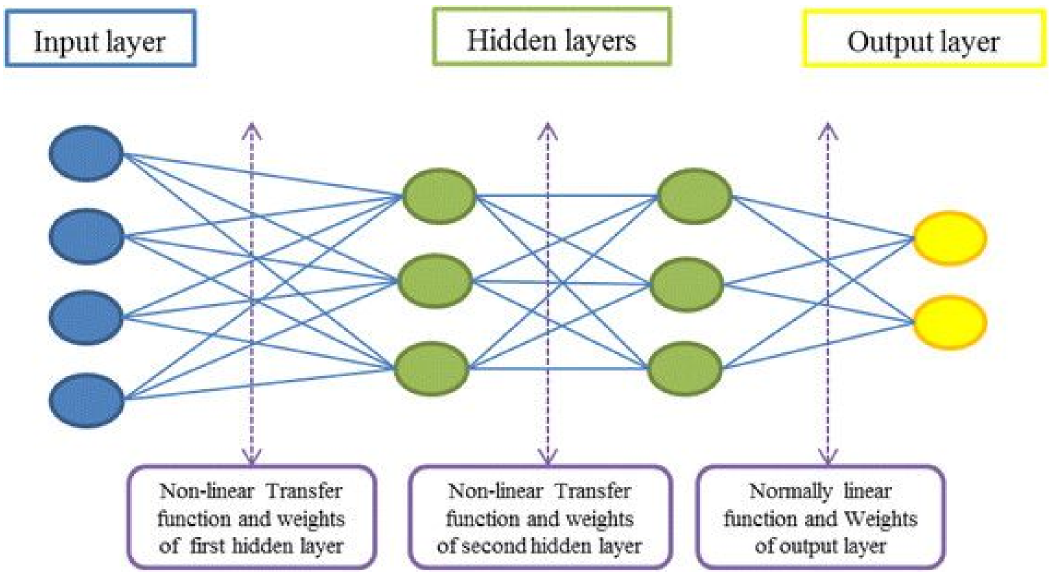
```
"""
K Neighrest Neighbors
number of neighbors: 13
"""
knn = Pipeline([
    ('vect', CountVectorizer(stop_words='english')),
    ('tfidf', TfidfTransformer()),
    ('knn', KNeighborsClassifier(n_neighbors=13)),])
```

ACCURACY 95.05%

Multi Layer Perceptron (MLP) Classifier with BoW and Tf-Idf

An MLP is a deep, artificial neural network. It is composed of more than one perceptron. They are composed of an input layer to receive the signal, an output layer that makes a decision or prediction about the input, and in between those two, an arbitrary number of hidden layers that are the true computational engine of the MLP. MLPs with one hidden layer are capable of approximating any continuous function. (shown below)

The sklearn neural network API MLPClassifier was used.


Input layer
Hidden layers
Output layer
Non-linear Transfer function and weights of first hidden layer
Non-linear Transfer function and weights of second hidden layer
Normally linear function and Weights of output layer

The **activation function** takes the input of the neuron and applies a function on it. I chose **RELU** which keeps only the positive inputs of the neuron.

Gradient Descent is a procedure where at the Back Propagation phase, the weights of the neural network are updated. I used the solver= 'adam' which is the **Stochastic Gradient Descent** approach.

Learning rate is how much the weights change after each back propagation iteration. I chose 0.0003 which is among the standard standard.

Hidden Layer Size indicates the nodes in each hidden layer, I set hidden 3 layers of 50 nodes

Max_iter indicates the number of iterations the gradient descent doesn't find a better solutions before it stops, I set it to 150.

```
"""
MLP Classifier with Stohastic Gradient Decent
RELU Activation function
alpha: L2 penalty
"""

lr_ = 0.001 # learning rate
l2_ = 0.0001 # l2 penalty

mlp = Pipeline([
    ('vect', CountVectorizer(stop_words='english')),
    ('tfidf', TfidfTransformer()),
    ('mlp', MLPClassifier(hidden_layer_sizes=(50,50,50 ), activation='relu',
solver='adam', alpha=l2_, batch_size='auto', learning_rate='constant',
learning_rate_init=lr_, max_iter=150)),])
```

ACCURACY 96.67%

SVM using Stochastic Gradient Descent with BoW and Tf-Idf

I used the sklearn SGDClassifier API, which uses Stochastic Gradient Descent . All the concept have been analyzed before, but in more detail, loss='hinge' means it uses linear SVM and penalty l2 is the regularization technique. This particular implementation produces results equal to the 'Beat the BenchMark' implementation, while the training time takes far less time to complete.

```
_clf_svm = Pipeline([('vect', CountVectorizer(stop_words='english')),
                     ('tfidf', TfidfTransformer()),
                     ('clf-svm', SGDClassifier(loss='hinge', penalty='l2', alpha=1e-4,
                     max_iter=120,  tol=1e-3,random_state=42, n_iter_no_change= 40)),])
```

ACCURACY 97%

Quadratic Discriminant Analysis using SVD

QDA assumes the predictor variables X are drawn from a multivariate Gaussian (aka normal) distribution and require the number of predictor variables (p) to be less then the sample size (n). The reason I used SVD with this implementation is that **it cannot work with sparse matrices.** The library used is shown in the code.

```
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis

qda = Pipeline([
    ('vect', CountVectorizer(stop_words='english')),
    ('tfidf', TfidfTransformer()),
    ('svd', TruncatedSVD(n_components=component_num)),
    ('qda', QuadraticDiscriminantAnalysis()),])
```

ACCURACY 95.57%

Gaussian Naive Bayes using SVD

The reason I used SVD with this implementation is that **it cannot work with sparse matrices.** The library used is shown in the code.

```
from sklearn.naive_bayes import GaussianNB

nb = Pipeline([
    ('vect', CountVectorizer(stop_words='english')),
    ('tfidf', TfidfTransformer()),
    ('svd', TruncatedSVD(n_components=component_num)),
    ('g_naive_bayes', GaussianNB()),])
```

ACCURACY 86.25%

AdaBoost classifier

An AdaBoost classifier is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases. The library used is shown in the code.

```
from sklearn.ensemble import AdaBoostClassifier

abc = Pipeline([
    ('vect', CountVectorizer(stop_words='english')),
    ('tfidf', TfidfTransformer()),
    ('abc', AdaBoostClassifier(n_estimators=240)),])
```

ACCURACY 87.25%

Multinomial Naive Bayes Classifier with SnowballStemmer

This is a Vectorizer (instead of CountVectorizer), form the nltk library that creates a term matrix using **stemming**. Stemming is the process of reducing a word to its word stem that affixes to suffixes and prefixes or to the roots of words known as a lemma. Stemming is important in natural language understanding (NLU) and natural language processing (NLP). The result is surprisingly good. The documentation of this API is not totally explanatory.

```
from nltk.stem.snowball import SnowballStemmer

stemmer = SnowballStemmer("english", ignore_stopwords=True)
class StemmedCountVectorizer(CountVectorizer):
    def build_analyzer(self):
        analyzer = super(StemmedCountVectorizer, self).build_analyzer()
        return lambda doc: ([stemmer.stem(w) for w in analyzer(doc)])


stemmed_count_vect = StemmedCountVectorizer(stop_words='english')
text_mnb_stemmed = Pipeline([('vect', stemmed_count_vect),
                    ('tfidf', TfidfTransformer()),
                   ('mnb', MultinomialNB(fit_prior=False)),])
```

ACCURACY 96.84%

Finally the ***testSet_categories.csv*** is created in the test_prediction.ipynb using the ‘beat the benchmark’ model.

Conclusions

I really like that I had the chance to experiment with Text Classification and NLP techniques. The thing that I didn’t expect the most, was that really simple models like KNN and Naive Bayes, perform so well compared to really sophisticated models, like Neural Networks with Gradient Descent, SVM etc. The last thing I found during the last day of experimentation was the StemmedCountVectorizer, which I would like to experiment further on my own, because it seems to outperform the simple Bag of Words.