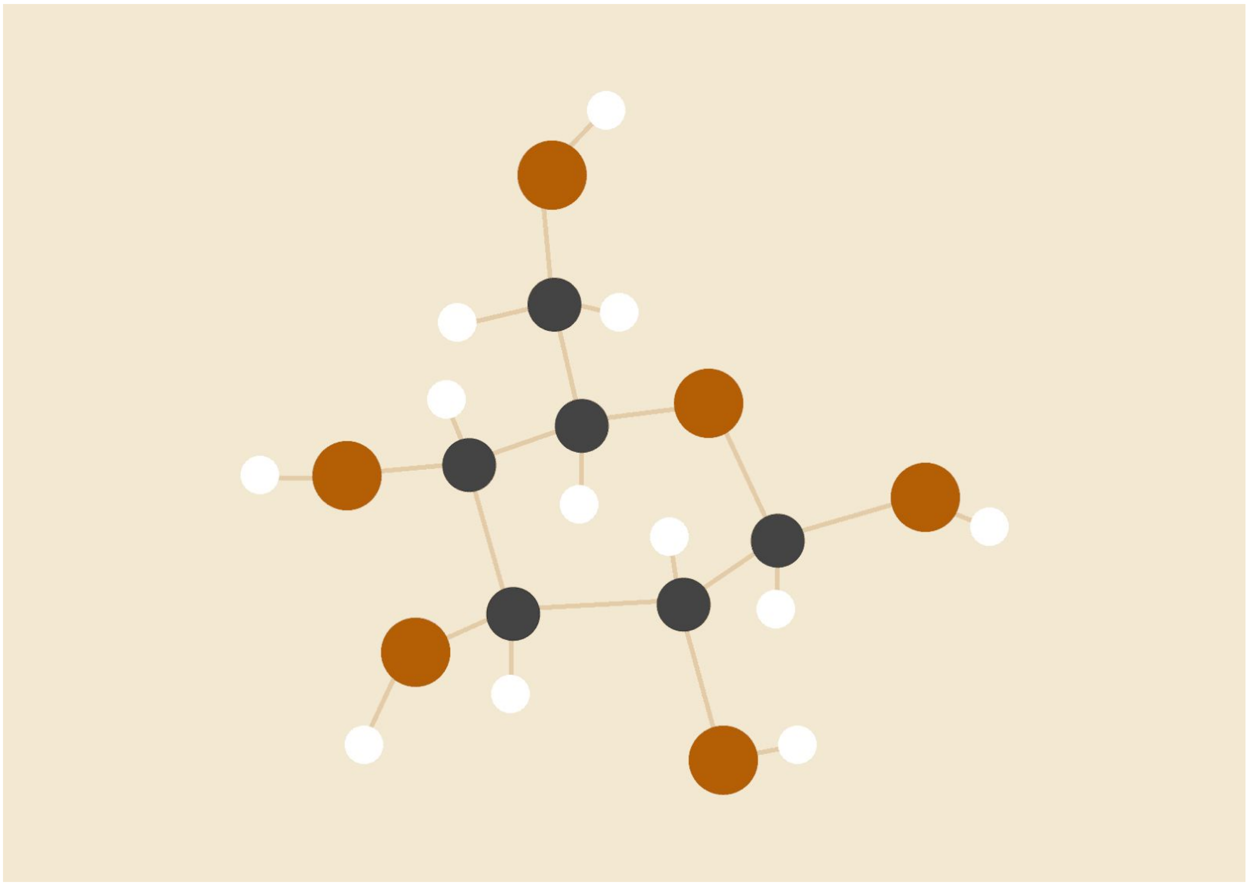
EE514 ASSIGNMENT

Data Analysis and Machine Learning



Niall Lyons

7.07.2020

Student Number: 13493628
EE514

Table Of Contents

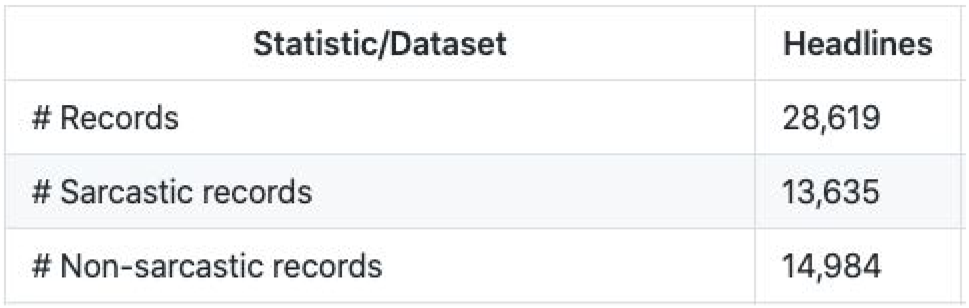
DATA SET DESCRIPTION

In the machine learning and the natural language processing space, sarcasm detection is a relevant and widely researched area. Many researchers work on this problem by scraping tweets collected using hashtags, labels and replies to tweets. Each of these datasets, although large and contain plenty of data, can be quite noisy. With tweets, comes misspellings, informal language and hate speech resulting in noisy datasets. In this report I will use a dataset specifically designed to overcome the disadvantages of Twitter datasets. It is a Headlines dataset for sarcasm detection that is collected from two news websites. These two websites, namely TheOnion and the HuffPost specialise in producing sarcastic and non sarcastic headlines respectively. The advantage of this dataset over previous datasets is that the headlines are written by professionals with no spelling mistakes using formal language. TheOnion's sole purpose is to create sarcastic headlines resulting in high quality data with less noise.

Each record in the dataset has three attributes:

- is_sarcastic: 1 if the record is sarcastic, otherwise 0
- Headline: the headline of the news article
- Article_link: link to the original news article

The dataset statistics can be seen in the below table. This shows us that we have a balanced dataset, with a very small margin of bias toward non sarcastic records. However, this is not a cause of concern, as it is so small.



Statistic/Dataset	Headlines
# Records	28,619
# Sarcastic records	13,635
# Non-sarcastic records	14,984

Fig 1. Dataset Statistics

RELATED WORK

The main work that has been done on this dataset is using a Hybrid Neural Network.[7] Mainly CNN and LSTM architectures have been found to improve the baseline by 5%. Given the scale and popularity of this area I feel that there is a lot more to come in this area. NLP researchers are keen to understand how sarcasm can be detected and LSTM and BLSTM architectures are finding those answers.

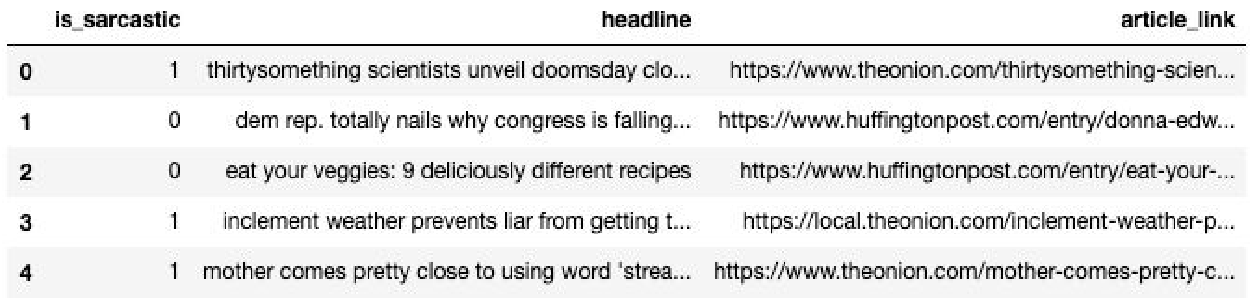
Pre Processing and Cleaning

Throughout this project the approach to cleaning the data was simple, which included a sequence of three steps aiming at producing high quality data to produce the best quality models. These steps included:

1. Inspection: Detect what data was given and how it could be adjusted and enhanced
2. Cleaning: Fix anomalies in the data
3. Splitting of data: Splitting data to training and test sets
4. Feature Extraction: Using TF-IDF and CountVectorizer
5. Verifying: Inspect results of data after cleaning to ensure correctness

INSPECTION

The data is in JSON format and was easily loaded by parsing the file. I was then able to pass the parsed file to a pandas dataframe to inspect what data was given. I started by looking at the head of the data, which can be seen in figure two. As explained in the description it has 3 attributes. I then wanted to find out if we had any missing data in each attribute by using `df.isna().sum()` and found that all values were present. I looked through some of the links aligned to each headline to understand what headline represented and how “sarcastic” was personified. Although interesting, for the case of this project it is not needed.



	is_sarcastic	headline	article_link
0	1	thirtysomething scientists unveil doomsday clo...	https://www.theonion.com/thirtysomething-scien...
1	0	dem rep. totally nails why congress is falling...	https://www.huffingtonpost.com/entry/donna-edw...
2	0	eat your veggies: 9 deliciously different recipes	https://www.huffingtonpost.com/entry/eat-your-...
3	1	inclement weather prevents liar from getting t...	https://local.theonion.com/inclement-weather-p...
4	1	mother comes pretty close to using word 'strea...	https://www.theonion.com/mother-comes-pretty-c...

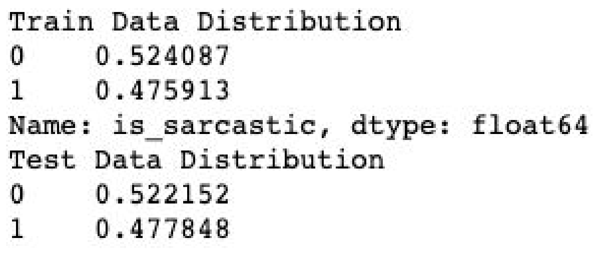
Fig 2. Head of data

CLEANING

The first thing to do was to delete the article_link column from the dataframe as it is obselete. Although each headline is written by professionals it is not noise free. As we are dealing with words, the first method I wrote was to make all words lowercase. As with sentences comes different punctuations, which would hamper our models. Each headline was cleaned of all punctuations and anomalies. As we now have a clean sentence I wanted to clean them up some more. I researched both lemmatization and stemming and came to the conclusion that lemmatization in this case is better as it performs morphological analysis of the words. Lemmatization is the process of grouping together the different inflected forms of a word so they can be analysed as a single item, bringing a context to the words. [1] I now have data that can be understood, however I still have useless words such as 'the', 'is', 'a' that give no context. These are called stop words and have little input to our models on taking up processing time. These words are removed from all headlines. The data is now cleaned and can be split up into train and test sets.

SPLITTING THE DATA

Before splitting the data into train and test sets, I shuffled the data to ensure that there is absolutely no bias. The data is split 75% training and 25% testing. From reading different NLP papers, it seems that this ratio split is becoming the norm with a good distribution seen in the figure below. From here the training data was stored and used for data exploration while the test data was put away to be used in later stages.



```
Train Data Distribution
0    0.524087
1    0.475913
Name: is_sarcastic, dtype: float64
Test Data Distribution
0    0.522152
1    0.477848
```

Fig 3. Data Distribution

FEATURE EXTRACTION

This element of the project is the area which will determine how good your model really is. There are many different ways to approach text related problems. For example, HashingVectorizer, Bag of Words, TFidfVectorizer, TFidfTransformer and CountVectorizer. For the purpose of this project I focused on two, TFidfVectorizer and CountVectorizer. These two are the state of the art in NLP. I started with CountVectorizer as it is the simpler of the two, here I got an idea of the data I was dealing with as a starting point. This method returns a sparse vector with the number of times a word appears in a document and uses this as its weight. However, this is quite a simple method and can run into problems resulting in skewed models. A much better approach is TFidfVectorizer which tokenises documents, learns the vocabulary and inverse document weights. TFidf stands for "Term Frequency – Inverse Document" will define how important a word is in a document, while also looking at how relevant it is to the overall document. A score for each word is assigned, the mathematics behind the TFidf scores can be found in [2]. To ensure I was happy with how TFidfVectorizer was running I tuned the hyperparameters on a trial and error basis to ensure the data would produce the best models. This trial and error stage happened when validating the model and determining its accuracy. These set up for TFidfVectorizer and CountVectorizer included,

- Encoding set to utf-8, as this is the default feature used to decode
- Decode_error is set to 'ignore' , meaning that if a byte sequence is given to analyze that contains characters not of the given encoding it is ignored.
- Stop_words was not set as these have already been removed
- Strip_accents set to ascii, as this is the fastest mapping, and accents are removed from the data
- Analyser set to 'word' , meaning that the features analysed are of word form
- Max_df set to 0.98, this will now build a vocabulary, not using terms that have a document frequency higher than the threshold. Trial and error was used in order to find this threshold
- Min_df set to 0.15, again this was found using trial and error. This will build a vocabulary while ignoring terms with a document frequency of lower than the threshold set.

VERIFICATION

To ensure data was to a high standard and ready to be trained, I inspected the data looking at each data frame and was happy with how it looked. I made one method that brought together the whole pre processing stage as shown:

```
def get_data():
    input_file = './Sarcasm_Headlines_Dataset.json'
    data = list(parseJson(input_file))

    df = pd.DataFrame(data)
    df = df[['is_sarcastic','headline']]
    df = shuffle(df)

    df['headline'] = df.headline.apply(to_lower)
    df['headline'] = df.headline.apply(punctuation_stp_words_lemmatize)
    tfidf_vectorizer = TfidfVectorizer(analyzer ='word', encoding= 'utf-8',
decode_error = 'ignore', strip_accents='ascii')
    X = tfidf_vectorizer.fit_transform(df['headline'])
    X = pd.DataFrame(X.toarray(),
columns=tfidf_vectorizer.get_feature_names())
    y = df['is_sarcastic']
    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.25, random_state=42)

    return X_train, X_test, y_train, y_test
```

EXPLORATORY DATA ANALYSIS

In this section I came familiar with the data that I had and was able to visualise in my mind my end goal. Exploratory data analysis gives you an opportunity to become accustomed to your data and find what is useful and non useful. I began by analysing how many sarcastic and non sarcastic headlines I had. This showed me I had a balanced dataset which I was happy with.

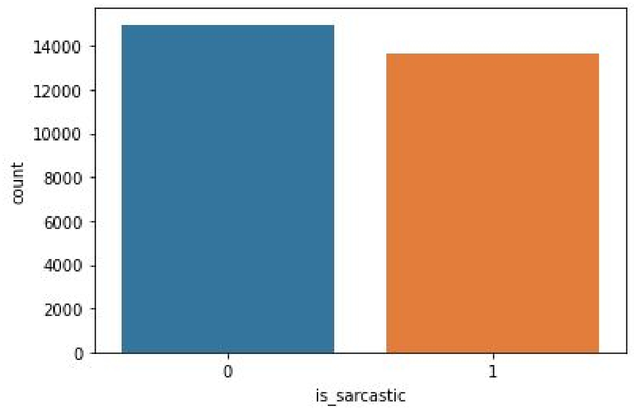


Fig. 4 Total Headlines Count

From here I thought about what differences could there be between both. I looked at the number of words in each headline as well as the number of characters, to get a feeler for any trends in the classifications. I used a boxplot to gain a better understanding. A box plot is a standardised way of showing the distribution of data based across min, first quartile, median, third quartile and maximum. It's a good indication of any outliers in your data and if it is symmetrical. In figure 5, the minimum which is calculated by Q1 -1.5*IQR and maximum calculated by Q3 + 1.5*IQR are slightly higher for the sarcastic headlines in each case. Again we can see the median of sarcastic is slightly higher also in this case. Interestingly, it can also be seen that there are more outliers in the sarcastic headlines with one especially standing out above the rest. In figure 6, it can be seen slightly the same trends however with even more outliers in the sarcastic headlines. The sarcastic headlines also seem to have a slightly larger interquartile range, however not large enough to say the data is skewed.

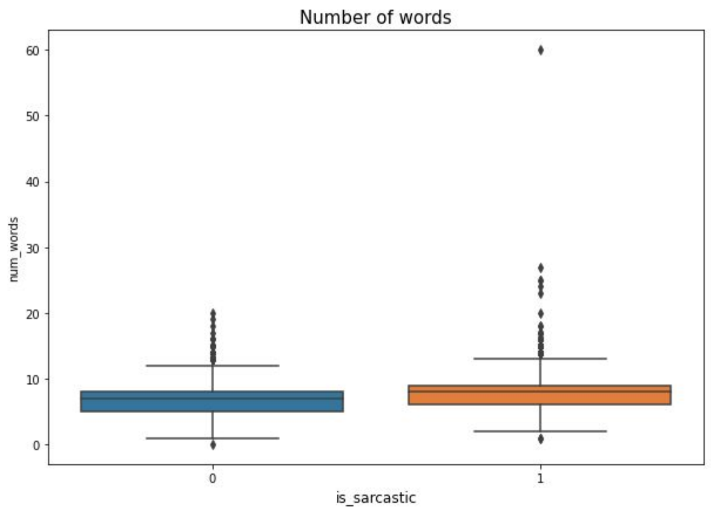


Fig. 5 Words in headlines

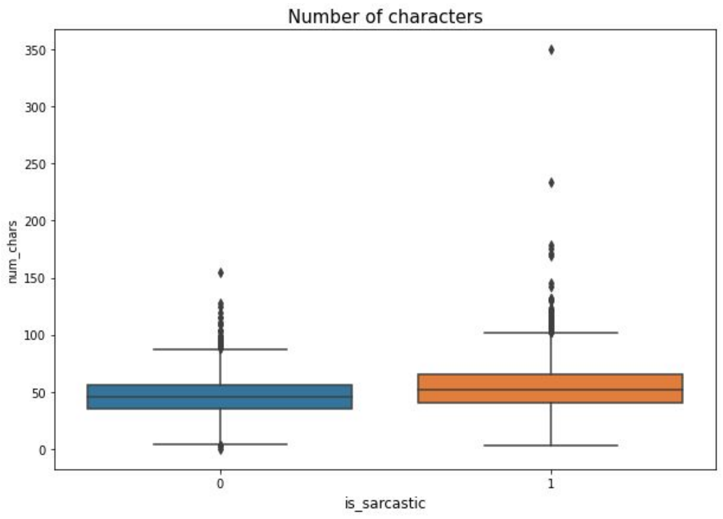


Fig. 6 Characters in headlines

From observing this information I wanted to then look at the average word lengths in each class. It can be seen in figure 7, that the majority of the words in both classes are of length four to six characters with sarcastic headlines having a slightly larger kernel density.

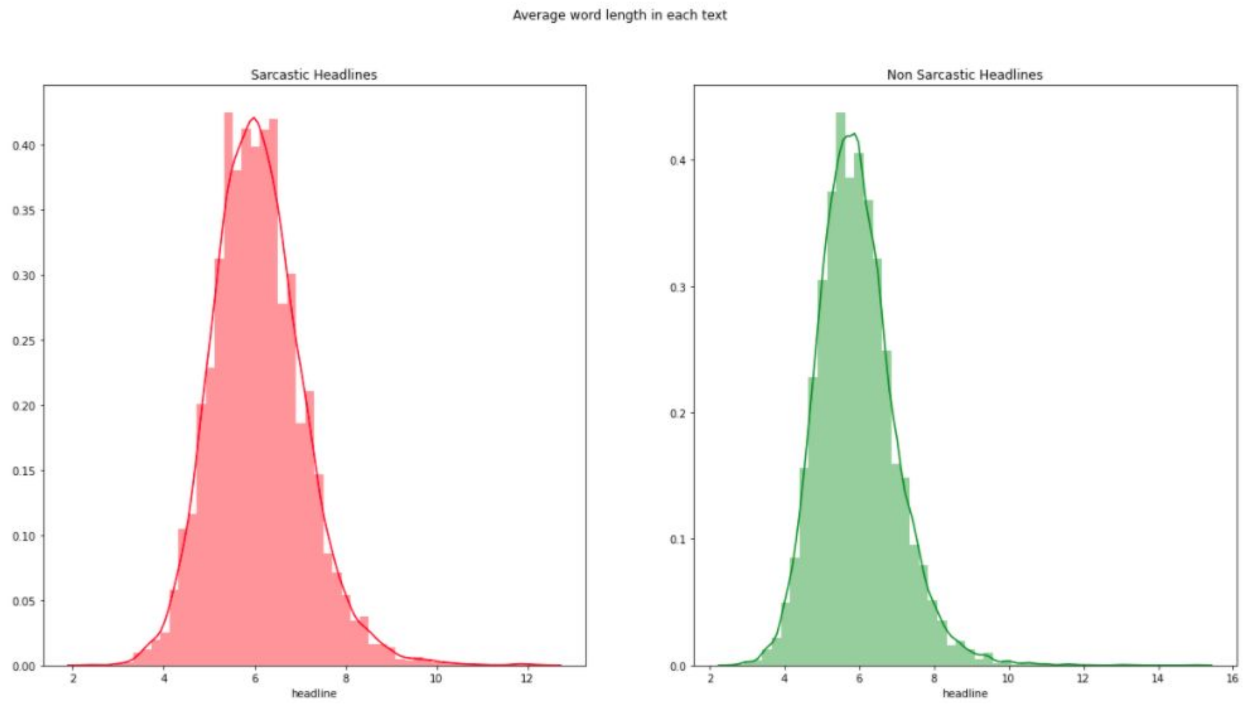


Fig. 7 Average word length in headlines

From here I wanted to see how badly stop words affected the data and in figure 8, it can be clearly seen that in both classes the frequency of stop words is high. Meaning that keeping them would severely affect the performance of the models. Code for this part of the project is in the notebook, ML_classification_pipeline.

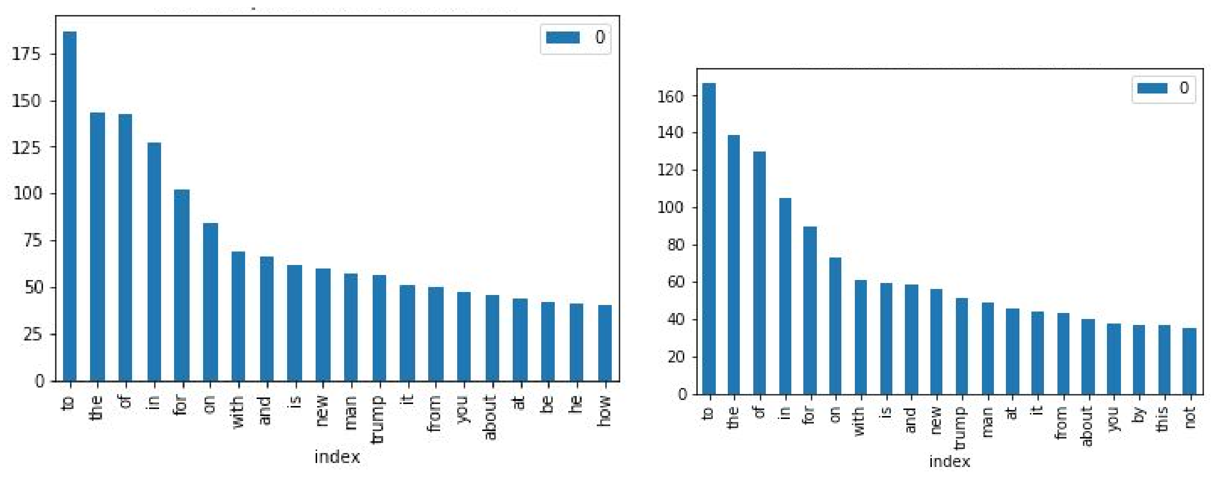
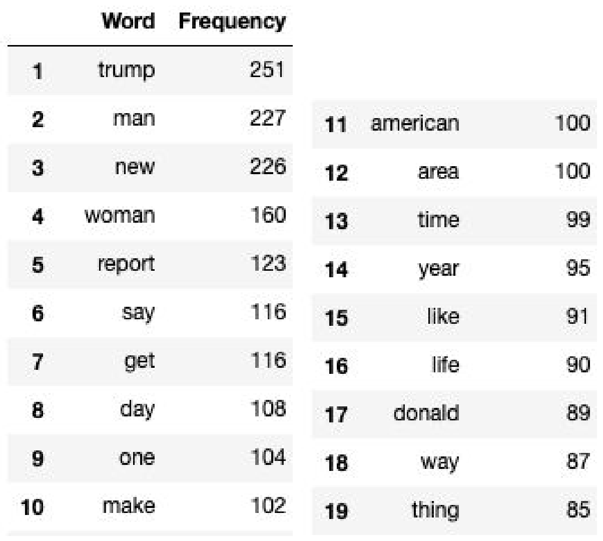


Fig. 8 Non-sarcastic (Left), Sarcastic (Right)

I then removed the stop words and applied TFidfVectorizer and plotted the top 20 most frequent words. I split the data into sarcastic, non sarcastic and both classes using data frames to examine how different the word frequencies were in each. Figure 9, shows all of the headlines, where figure 10 and 11 show sarcastic and non-sarcastic words respectively.



	Word	Frequency
1	trump	251
2	man	227
3	new	226
4	woman	160
5	report	123
6	say	116
7	get	116
8	day	108
9	one	104
10	make	102
11	american	100
12	area	100
13	time	99
14	year	95
15	like	91
16	life	90
17	donald	89
18	way	87
19	thing	85

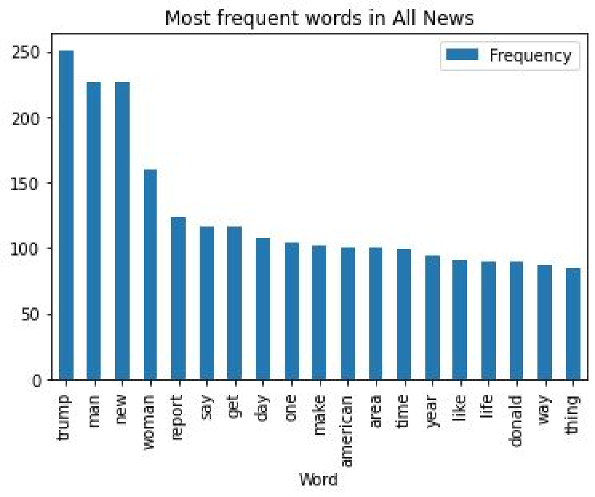
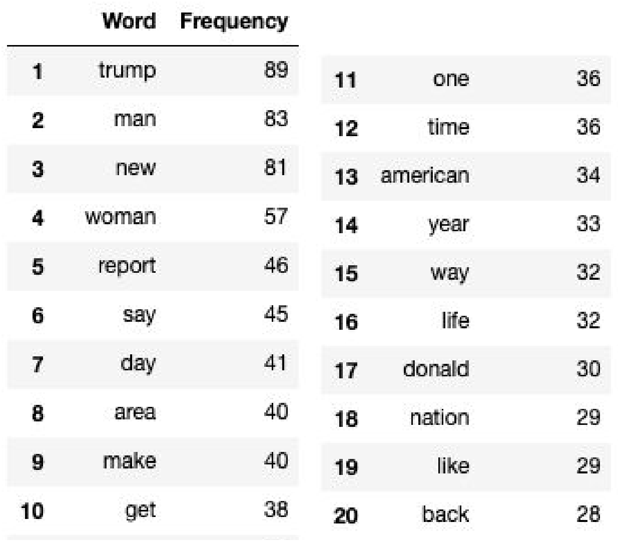


Fig.9 Charts of Top 20 Words of all Headlines (TFidf)



	Word	Frequency
1	trump	89
2	man	83
3	new	81
4	woman	57
5	report	46
6	say	45
7	day	41
8	area	40
9	make	40
10	get	38
11	one	36
12	time	36
13	american	34
14	year	33
15	way	32
16	life	32
17	donald	30
18	nation	29
19	like	29
20	back	28

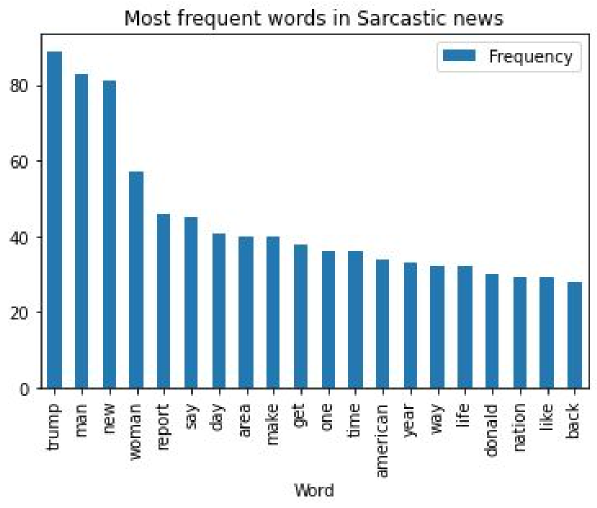
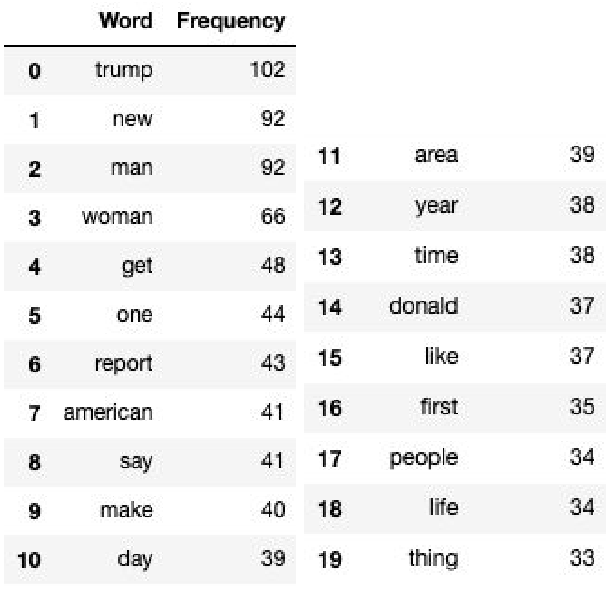


Fig. 10 Charts of Top 20 Words Sarcastic Headlines (TFidf)



	Word	Frequency
0	trump	102
1	new	92
2	man	92
3	woman	66
4	get	48
5	one	44
6	report	43
7	american	41
8	say	41
9	make	40
10	day	39
11	area	39
12	year	38
13	time	38
14	donald	37
15	like	37
16	first	35
17	people	34
18	life	34
19	thing	33

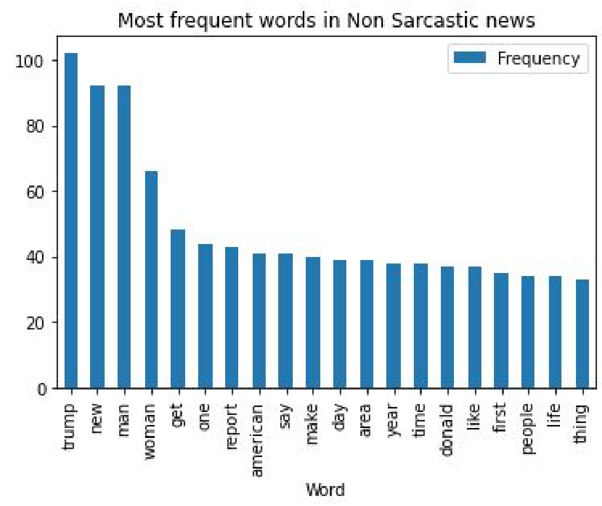
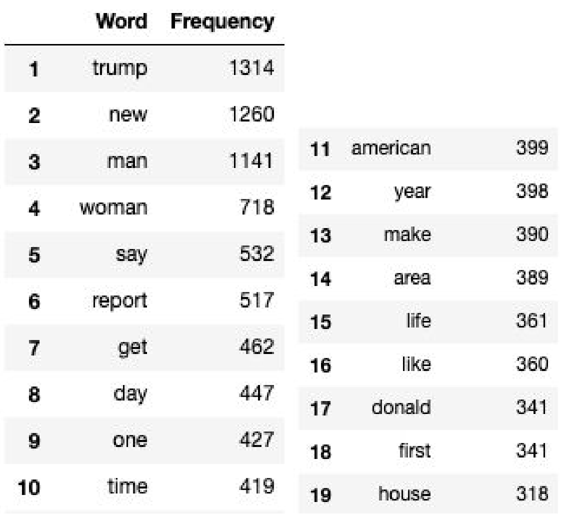


Fig. 11 Charts of Top 20 Words Non Sarcastic Headlines (TFidf)

To compare and contrast both TfidfVectorizer and CountVectorizer I re-applied the method using CountVectorizer. Figure 12, shows all of the headlines, where figure 13 and 14 show sarcastic and non-sarcastic words respectively. Code for this part of the project is in the notebook, CountVectorizer_Analysis.



	Word	Frequency
1	trump	1314
2	new	1260
3	man	1141
4	woman	718
5	say	532
6	report	517
7	get	462
8	day	447
9	one	427
10	time	419
11	american	399
12	year	398
13	make	390
14	area	389
15	life	361
16	like	360
17	donald	341
18	first	341
19	house	318

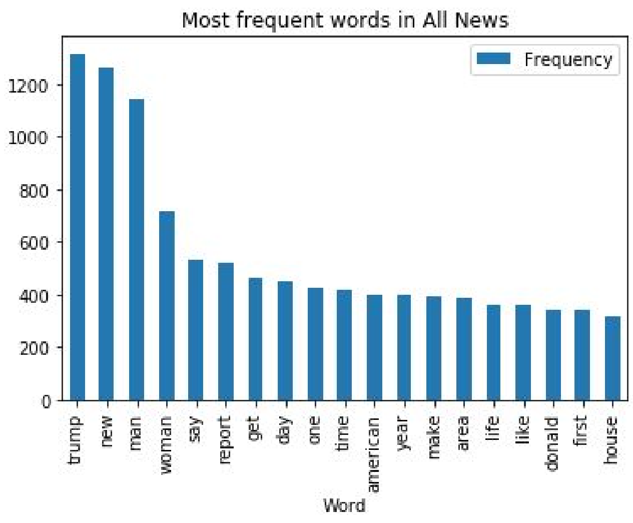
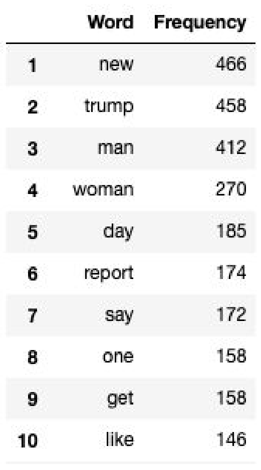
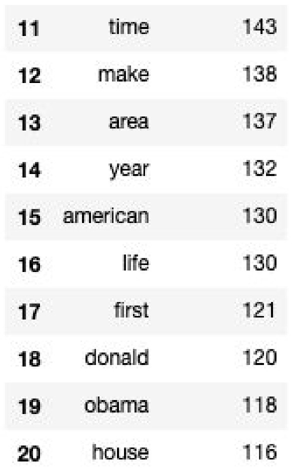


Fig. 12 Charts of Top 20 Words Headlines (CountVectorizer)



	Word	Frequency
1	new	466
2	trump	458
3	man	412
4	woman	270
5	day	185
6	report	174
7	say	172
8	one	158
9	get	158
10	like	146



11	time	143
12	make	138
13	area	137
14	year	132
15	american	130
16	life	130
17	first	121
18	donald	120
19	obama	118
20	house	116

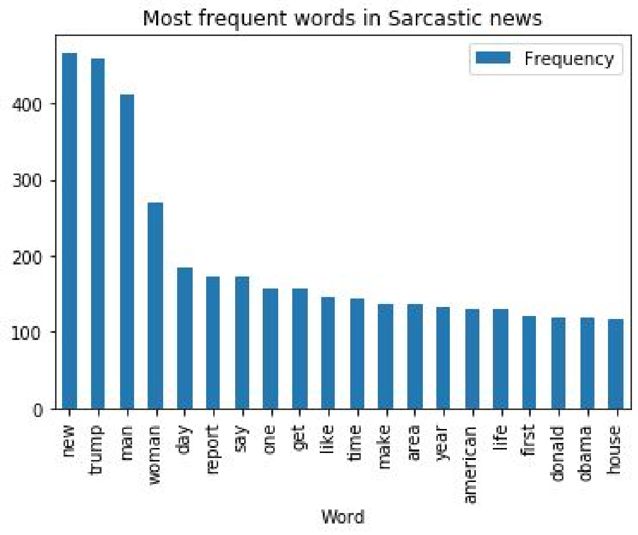
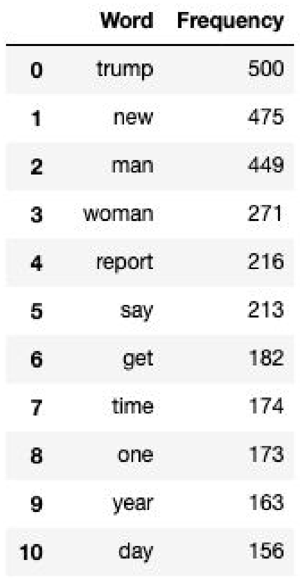
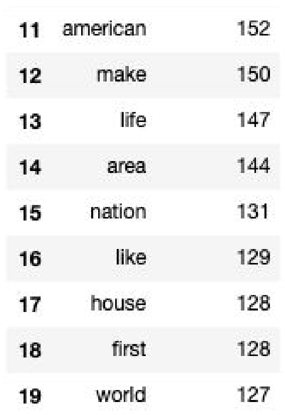


Fig. 13 Charts of Top 20 Words Sarcastic Headlines (CountVectorizer)



	Word	Frequency
0	trump	500
1	new	475
2	man	449
3	woman	271
4	report	216
5	say	213
6	get	182
7	time	174
8	one	173
9	year	163
10	day	156



11	american	152
12	make	150
13	life	147
14	area	144
15	nation	131
16	like	129
17	house	128
18	first	128
19	world	127

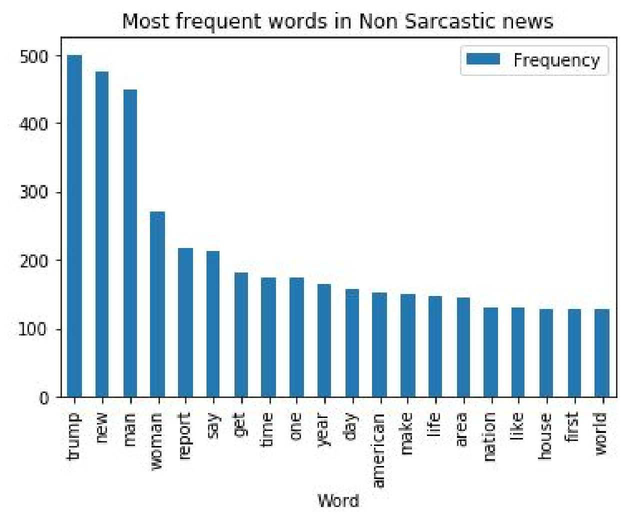


Fig. 14 Charts of Top 20 Words Non Sarcastic Headlines (CountVectorizer)

From generating these plots it can be seen that the same vocabulary is used when using TF-idf and CountVectorizer, however in some cases the weights of certain words vary. This proves the point that TFidf is a better approach and from this I was then able to move forward with the TFidf dataset.

In conclusion, I gained valuable insights into my data and learned alot about both classes and how they were shaped. TF-idf will be used for training, however I will also experiment with CountVectorizer on the test set. The exploration stage of a machine learning process is vital to ensure that the right data is passed to the model for training.

SUPERVISED CLASSIFICATION

In this section I look at the different types of supervised classification methods that can be used to produce different models. The training and accuracies of the models are looked at in detail in this section. I looked at a number different models using both a holdout validation approach and the k-fold cross validation approach. I picked an array of linear and non linear classification algorithms namely (Classification_Models Notebook):

1. Logistic Regression
2. K-Nearest Neighbour
3. Decision Tree Classifier
4. Gaussian Naive Bayes
5. Multinomial Naive Baye
6. Support Vector Classifier

I then went on to investigate ensemble boosting and bagging methods (Ensemble_Models Notebook). Namely,

1. AdaBoost Classifier
2. GradientBoosting Classifier
3. Random Forest Classifier
4. ExtraTrees Classifier

For all classification models I compared both holdout validation and K-Fold validation methods. This means that in the holdout validation method I used the test set on the trained model to validate the performance of the model. The data is split 75:25 training and test respectively. This technique can be useful, however it can lead to model overfitting and underfitting. However, this problem is rectified using K-fold cross validation. Here, the data is divided into k-folds and trained on k-1 folds with one fold held for testing. This method is usually preferred as you can train on multiple trains and test splits, which in turn gives you a better outlook on how the model will perform on unseen data. However, this is computationally more expensive to run and can take a lot of power and time to run on large datasets.

Throughout the course of this analysis I will be using confusion matrices, classification reports, precision recall curves, AUC scores and ROC curves to analyse each model.

A confusion matrix is shown in figure 15. Each row in a confusion matrix represents an actual class, while each column represents a predicted class.

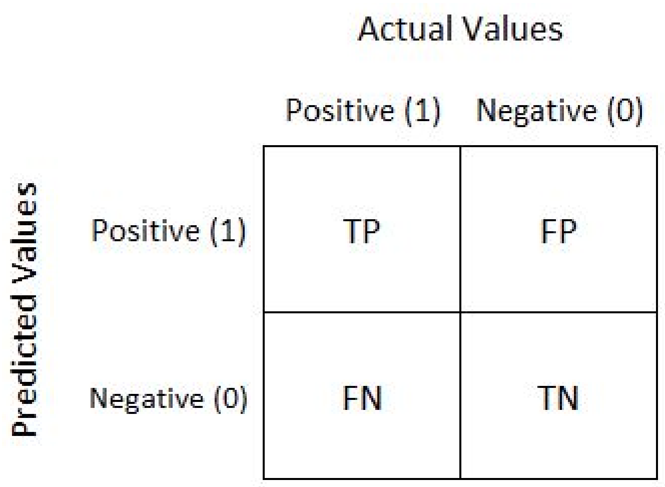


Fig.15 Confusion Matrix

This matrix is extremely useful for accessing a classification model. True Positives, False Positives, True negatives and False Negatives. These represent:

- True Positives (TP): Predicted positive and it’s true. In our case the model predicted a sarcasm headline is sarcastic and that is true.
- False Positives (FP): Predicted positive and it’s false. In our case the model predicted a sarcasm headline is sarcastic but it is actually not.
- True negatives (TN): Predicted negative and it’s true. In our case the model predicted a sarcasm headline is not sarcastic and it is actually not.
- False Negatives (FN): Predicted negative and it’s false. In our case the model predicted a sarcasm headline is not sarcastic but it actually is.

From the confusion matrix, we move onto a classification report, which measures the quality of predictions made by the model. Looking at how many are true and how many are false. More specifically, True Positives, False Positives, True negatives and False Negatives are used to compute the metrics. Figure 16, illustrates the difference between precision and recall as shown in the classification report.

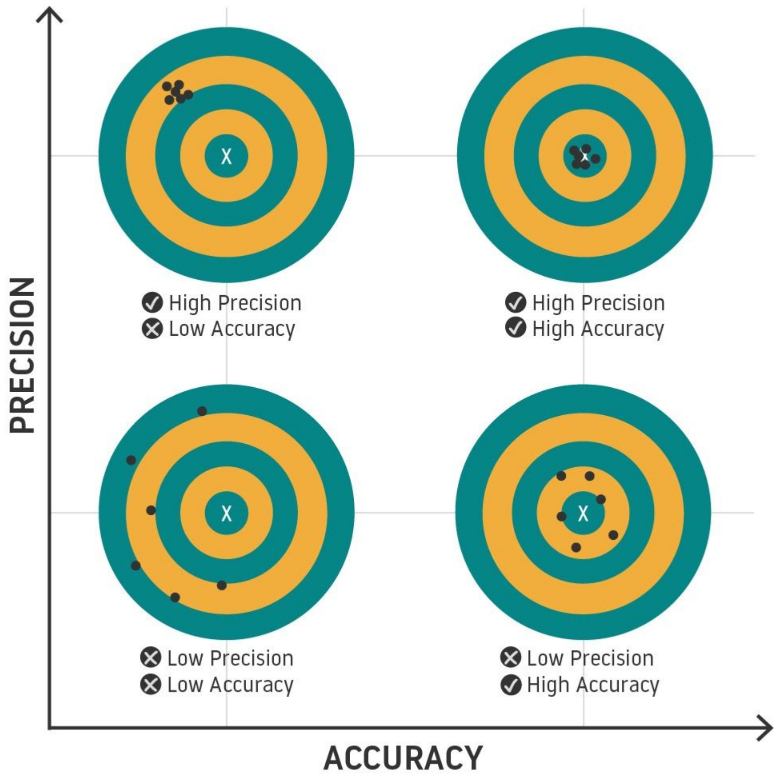


Fig. 16 Precision and accuracy

From this image it is shown that it's imperative that a model has high accuracy and high precision. Precision is the accuracy of positive predictions the classifier makes. It is defined as:

$$\text{precision} = \frac{TP}{TP + FP}$$

Where TP is the number of true positives and FP is the number of false positives. Precision is widely used beside recall this is the ratio of positive instances that are correctly detected by the classifier. It is defined as:

$$\text{recall} = \frac{TP}{TP + FN}$$

Where FN is the number of false negatives. The accuracy of a model is also given in the classification report, this is a performance measure that is a ratio between corrected predicted cases and overall total cases. Having a high accuracy does not mean your model is very good, it depends on the shape of your data and also if FP and FN outcomes are close together. Accuracy is defined as:

$$\text{Accuracy} = \frac{tp + tn}{tp + tn + fp + fn}$$

The F1 score of a model is also outlined in the report also. This is a combination of both precision and recall and is useful to compare classifiers. It is the harmonic mean of precision and recall, meaning that it gives more weight to lower values. It is defined as:

$$F_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{FN+FP}{2}}$$

To understand these metrics even deeper I also developed a precision recall curve. This shows the trade off between precision and recall for different thresholds. The higher the area under the curve the higher the recall and precision where high precision relates to a low false positive rate, and high recall relates to a low false negative rate. The higher the score the higher the accuracy. Figure 17, shows a perfect precision recall curve. However, if you were to get this after training your model there is a good chance something is wrong!

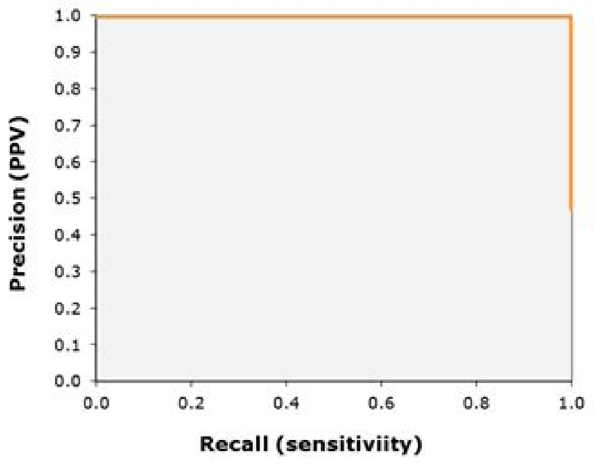


Fig. 17 Ideal Precision Recall Curve

Another metric I used to calculate the models is a receiver operating characteristic (ROC) curve. Here the true positive rate (TPR) is plotted against the false positive rate (FPR), which is the amount of negative observations that are incorrectly classified. In the ROC curve there is a trade off, the higher the TPR the more false positives the classifier produces. From this curve, we can also compare classifiers using the metric area under curve (AUC). The best AUC score will be 1.0.

In real world machine learning projects, to determine whether to use a precision recall (PR) curve or an ROC curve. Most of the time ROC curves are used, however if we want to specifically look at false positives instead of false negatives the PR curve would be used. Figure 18 shows different ROC curves and their corresponding AUC score.

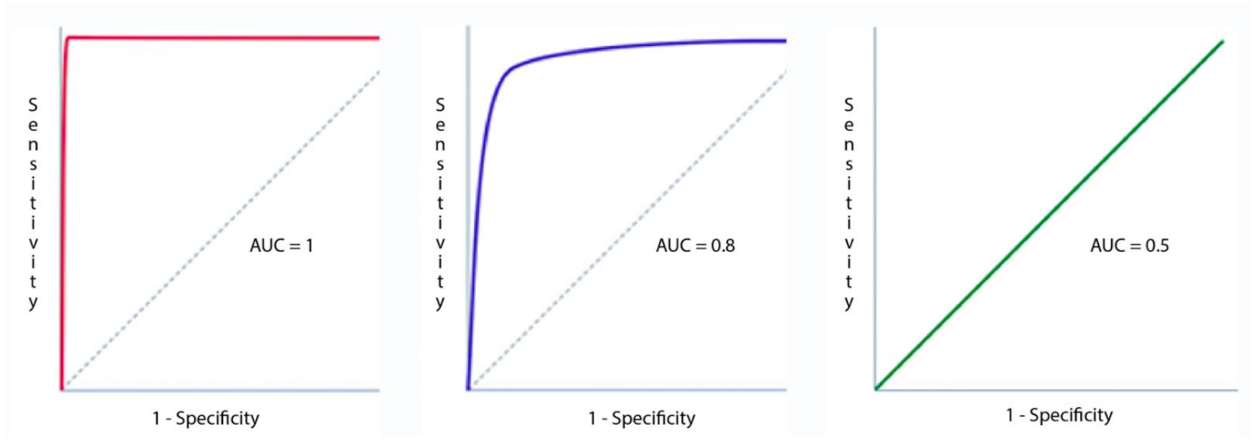


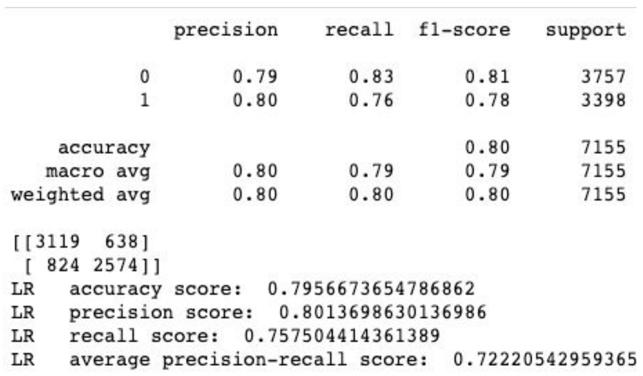
Fig. 18 ROC Curves and corresponding AUC scores

CLASSIFICATION ALGORITHMS

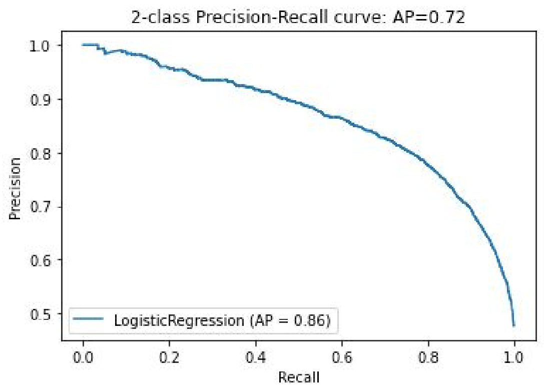
```
LR_model.fit(X_train, y_train)
GNB_model.fit(X_train, y_train)
NB_model.fit(X_train, y_train)
SVM_model.fit(X_train, y_train)
```

HOLDOUT VALIDATION

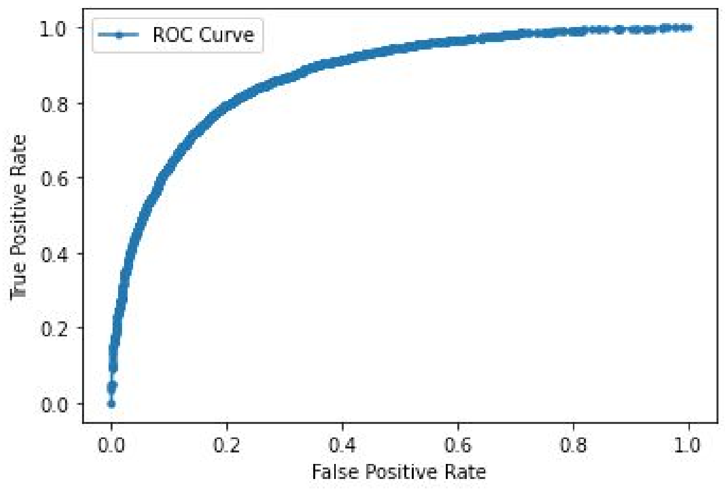
LOGISTIC REGRESSION



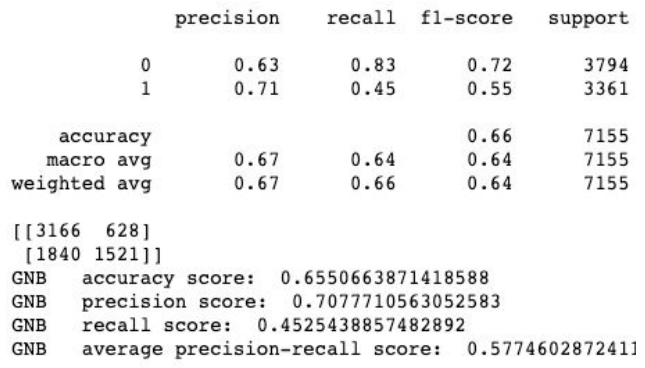
```
              precision    recall  f1-score   support

           0       0.79      0.83      0.81      3757
           1       0.80      0.76      0.78      3398

    accuracy                           0.80      7155
   macro avg       0.80      0.79      0.79      7155
weighted avg       0.80      0.80      0.80      7155

[[3119  638]
 [ 824 2574]]
LR   accuracy score:  0.7956673654786862
LR   precision score:  0.8013698630136986
LR   recall score:  0.757504414361389
LR   average precision-recall score:  0.72220542959365
```



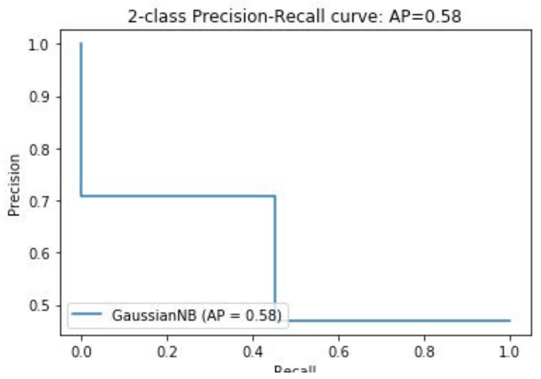
```
AUC:   0.8739055352512078
```

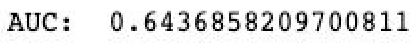


GAUSSIAN NB

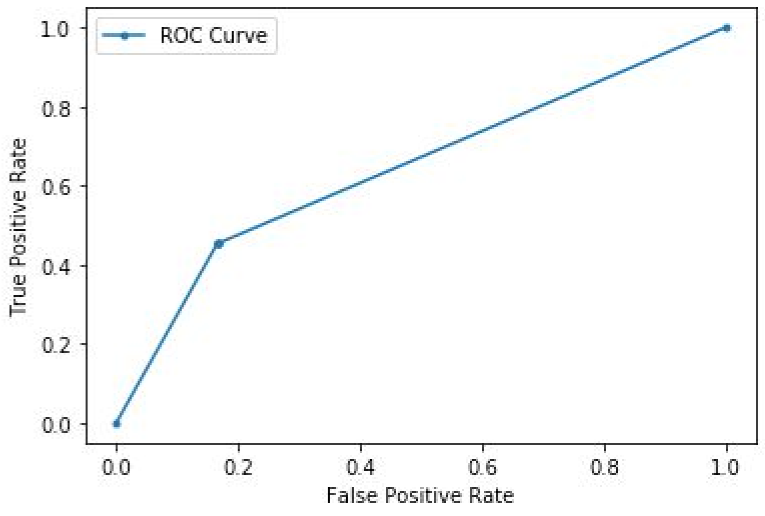


```
              precision    recall  f1-score   support

           0       0.63      0.83      0.72      3794
           1       0.71      0.45      0.55      3361

    accuracy                           0.66      7155
   macro avg       0.67      0.64      0.64      7155
weighted avg       0.67      0.66      0.64      7155

[[3166  628]
 [1840 1521]]
GNB   accuracy score:  0.6550663871418588
GNB   precision score:  0.7077710563052583
GNB   recall score:  0.4525438857482892
GNB   average precision-recall score:  0.577460287241
```

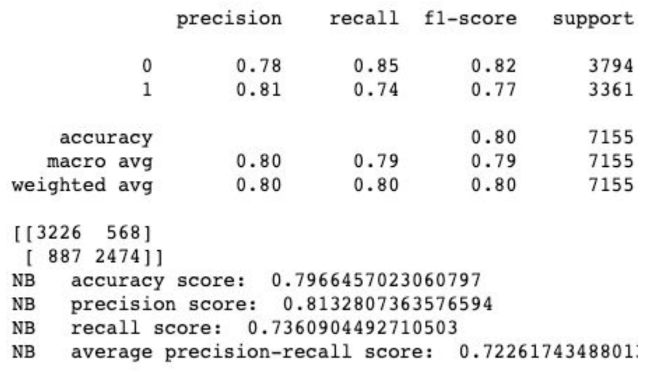




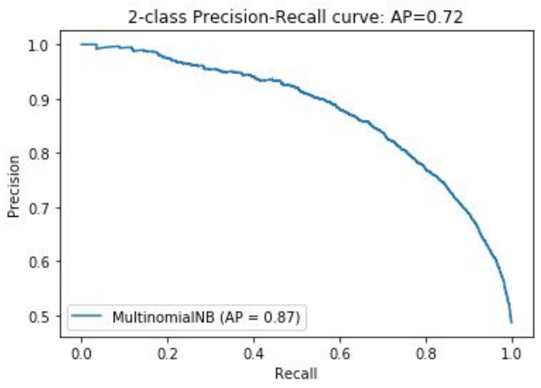
```
AUC:   0.6436858209700811
```



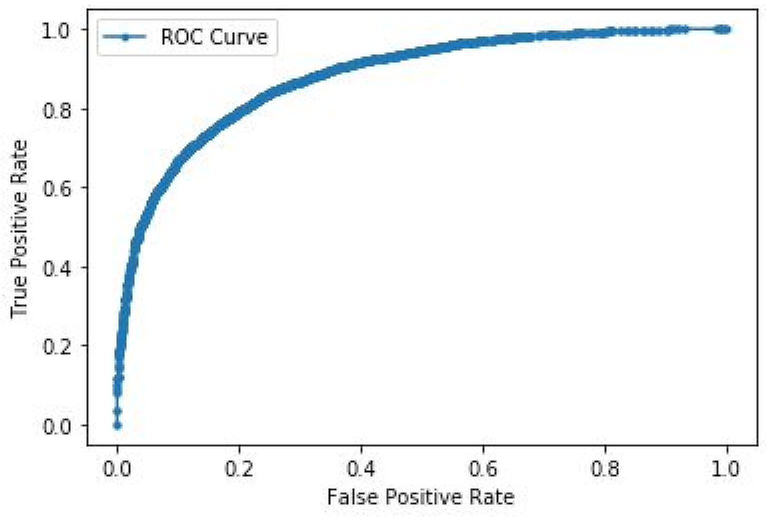
MULTINOMIAL NB



```
              precision    recall  f1-score   support

           0       0.78      0.85      0.82      3794
           1       0.81      0.74      0.77      3361

    accuracy                           0.80      7155
   macro avg       0.80      0.79      0.79      7155
weighted avg       0.80      0.80      0.80      7155

[[3226  568]
 [ 887 2474]]
NB   accuracy score:  0.7966457023060797
NB   precision score:  0.8132807363576594
NB   recall score:  0.7360904492710503
NB   average precision-recall score:  0.722617434880
```

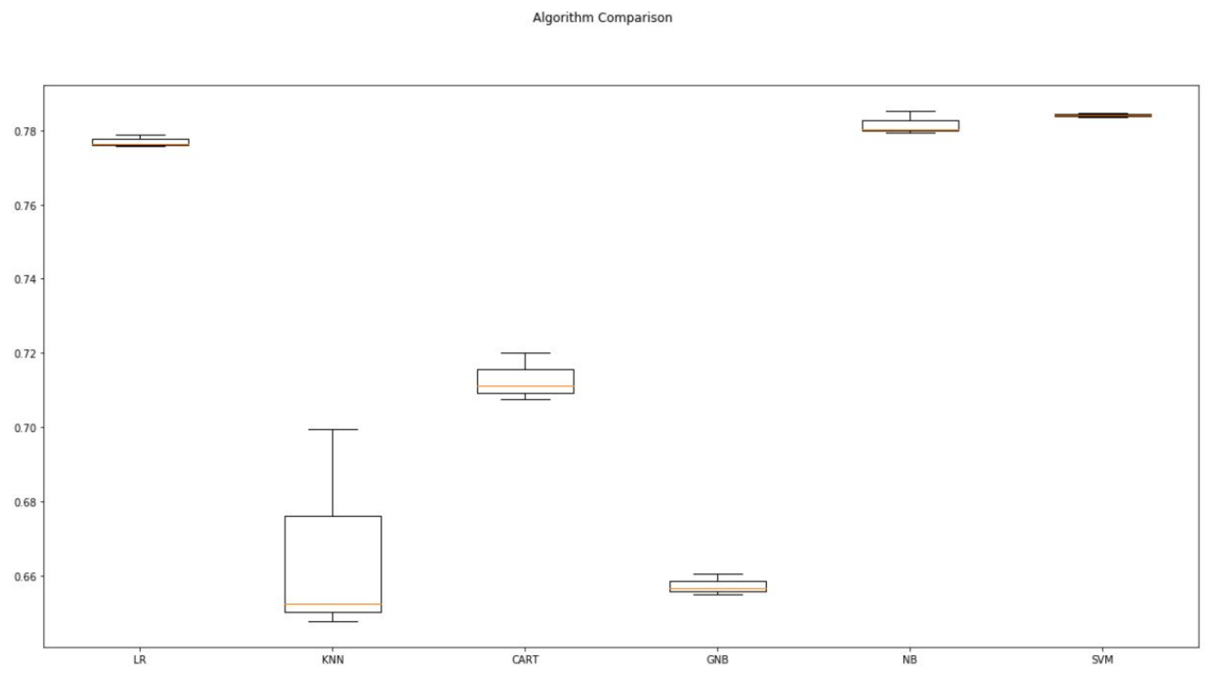


```
AUC:  0.8812989770565876
```

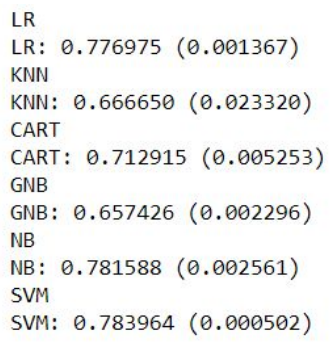


K-FOLD VALIDATION

In K-Fold validation, looking at the distribution of accuracy values calculated across all folds is vital to ensure you understand the outcome of your model. The below plot shows tight distributions across Logistic regression , Naives Baye and SVC which is good as it suggests low variance. However, Gaussian Naive Baye being performing so poor is slightly surprising.



As the attributes are not varied very much, apart from KNN there is no need to standardize the data in any way. By training the models and using cross_val_score with k is 3 the resulting mean of all results and std of results were:



```
LR
LR: 0.776975 (0.001367)
KNN
KNN: 0.666650 (0.023320)
CART
CART: 0.712915 (0.005253)
GNB
GNB: 0.657426 (0.002296)
NB
NB: 0.781588 (0.002561)
SVM
SVM: 0.783964 (0.000502)
```

SVC model performs the best over 3 folds with a very low STD. However, very closely behind are logistic regression and Naive Bayes. Both are computationally more efficient. The snippet of code to produce this outcome:

```
models = []
models.append(('LR', LogisticRegression()))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC()))
results = []
names = []
for name, model in models:
kfold = KFold(n_splits=num_folds, random_state=seed)
cv_results = cross_val_score(model, X_train, Y_train, cv=kfold,
scoring=scoring) results.append(cv_results)
names.append(name)
msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
print(msg)
```

CLASSIFICATION MODELS K-FOLD COMPARISON TABLE

K-Fold Validation (K=3)		
Classifier	Accuracy(%)	STD
Logistic Regression	78%	0.001367
KNN	67%	0.023320
Decision Tree Classifier	71%	0.005253
Gaussian NB	66%	0.002296
Multinomial NB	78%	0.002561
Support Vector Classifier	78%	0.000502

CLASSIFICATION MODELS HOLDOUT VALIDATION COMPARISON TABLE

Holdout Validation		
Classifier	Accuracy(%)	Precision
Logistic Regression	80	0.81
KNN	67	0.82
Gaussian NB	66	0.71
Multinomial NB	81	0.81

ENSEMBLE METHODS

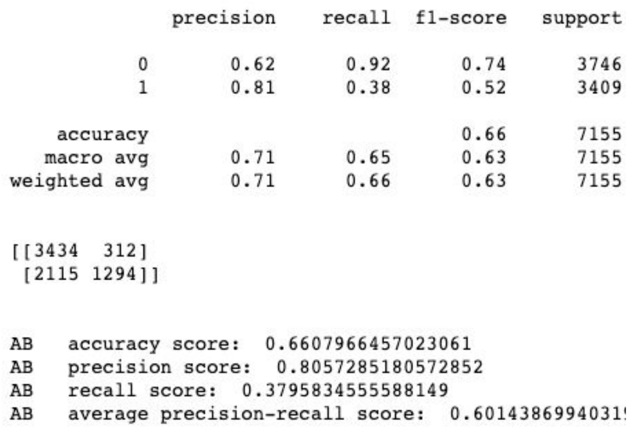
The main cause of error is machine learning models is noise, boas and variance. Ensemble methods try to minimise these factors. Their traits include improving the accuracy and overall performance of the models. As I am not overly familiar with these methods I wanted to try them out.

The below classifiers and analyse specifically use holdout validation to analyse the performance of the model.

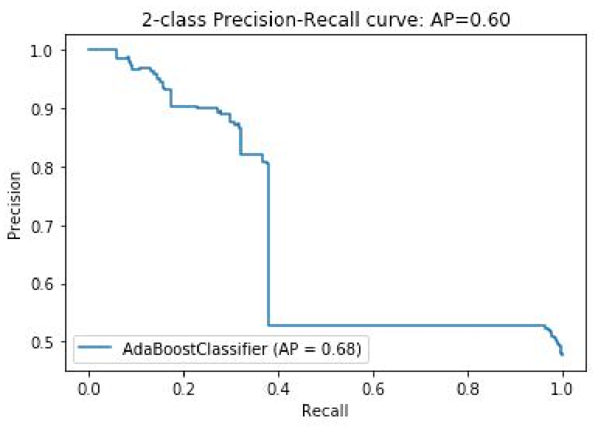
HOLDOUT VALIDATION

ADABOOST CLASSIFIER

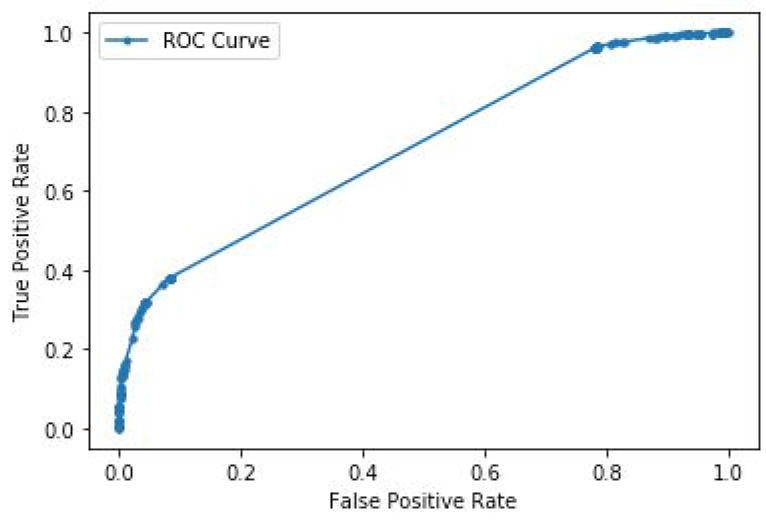
An AdaBoost classifier is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases. [3]



```
              precision    recall  f1-score   support

           0       0.62      0.92      0.74      3746
           1       0.81      0.38      0.52      3409

    accuracy                           0.66      7155
   macro avg       0.71      0.65      0.63      7155
weighted avg       0.71      0.66      0.63      7155


[[3434  312]
 [2115 1294]]


AB   accuracy score:  0.6607966457023061
AB   precision score:  0.8057285180572852
AB   recall score:  0.3795834555588149
AB   average precision-recall score:  0.6014386994031
```

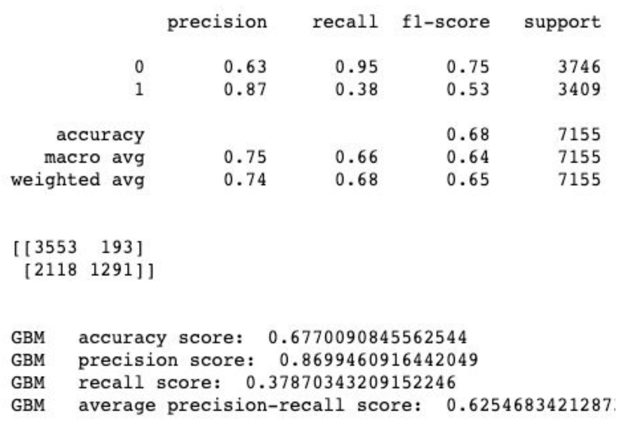


```
AUC:  0.7074883591485557
```

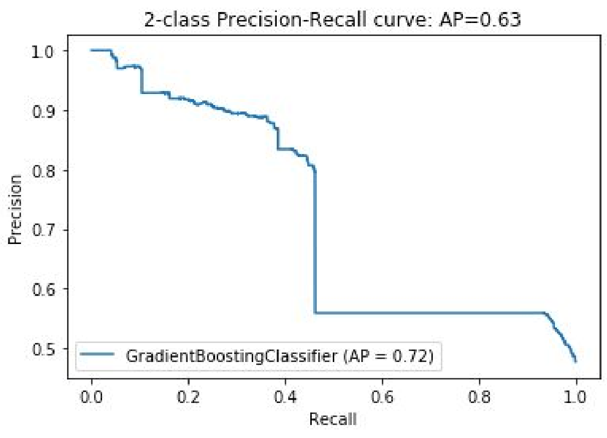


GRADIENT BOOSTING CLASSIFIER

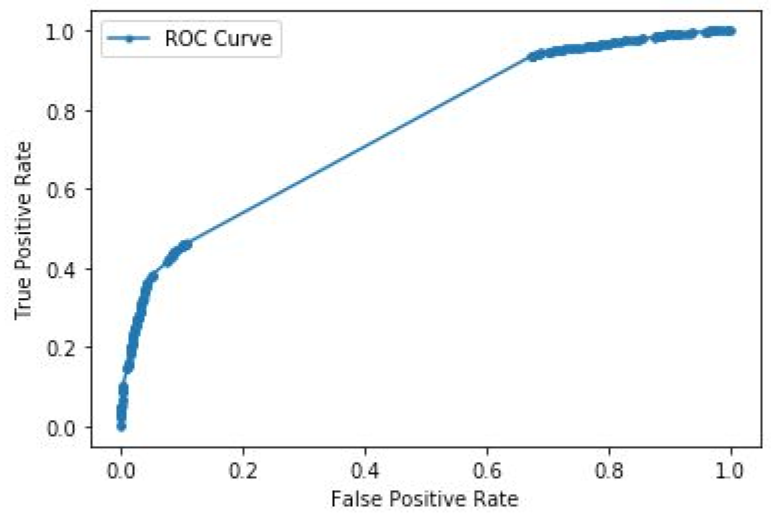
GB builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage n_classes_ regression trees are fit on the negative gradient of the binomial or multinomial deviance loss function. Binary classification is a special case where only a single regression tree is induced. [4]



```
              precision    recall  f1-score   support

           0       0.63      0.95      0.75      3746
           1       0.87      0.38      0.53      3409

    accuracy                           0.68      7155
   macro avg       0.75      0.66      0.64      7155
weighted avg       0.74      0.68      0.65      7155


[[3553  193]
 [2118 1291]]


GBM   accuracy score:  0.6770090845562544
GBM   precision score:  0.8699460916442049
GBM   recall score:  0.37870343209152246
GBM   average precision-recall score:  0.6254683421287
```

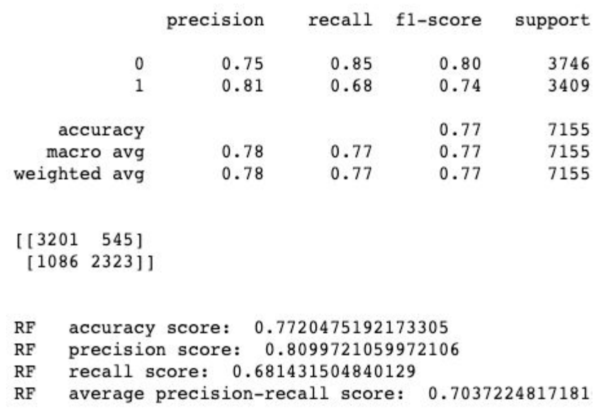


```
AUC:  0.7502858236034542
```

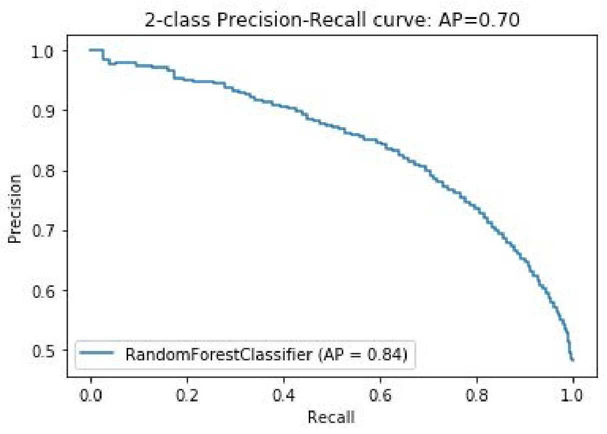


RANDOM FORESTS CLASSIFIER

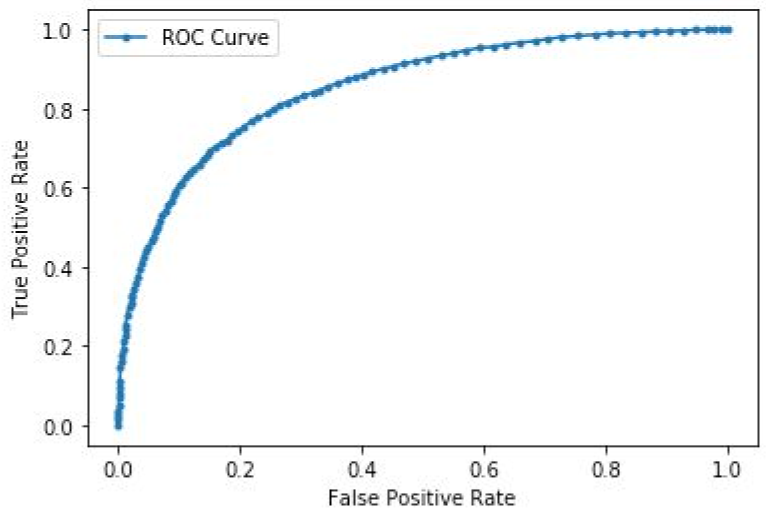
A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is controlled with the max_samples parameter if bootstrap=True (default), otherwise the whole dataset is used to build each tree. [5]



```
              precision    recall  f1-score   support

           0       0.75      0.85      0.80      3746
           1       0.81      0.68      0.74      3409

    accuracy                           0.77      7155
   macro avg       0.78      0.77      0.77      7155
weighted avg       0.78      0.77      0.77      7155


[[3201  545]
 [1086 2323]]


RF   accuracy score:  0.7720475192173305
RF   precision score:  0.8099721059972106
RF   recall score:  0.681431504840129
RF   average precision-recall score:  0.7037224817181
```

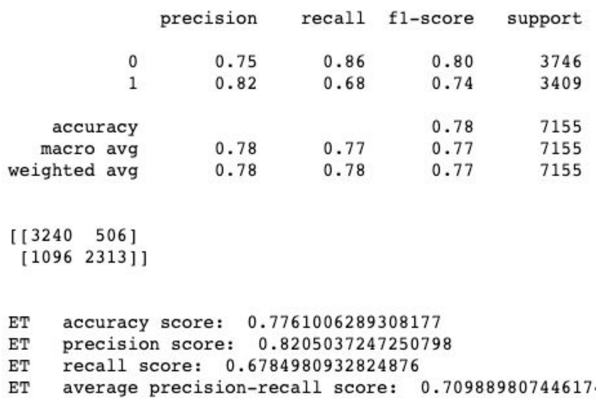


```
AUC:   0.854005375363133
```

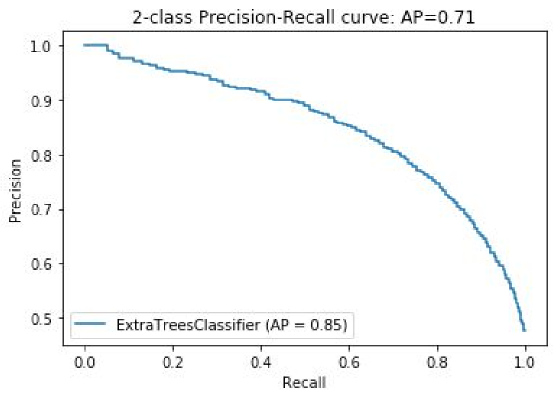


EXTRA TREES CLASSIFIER

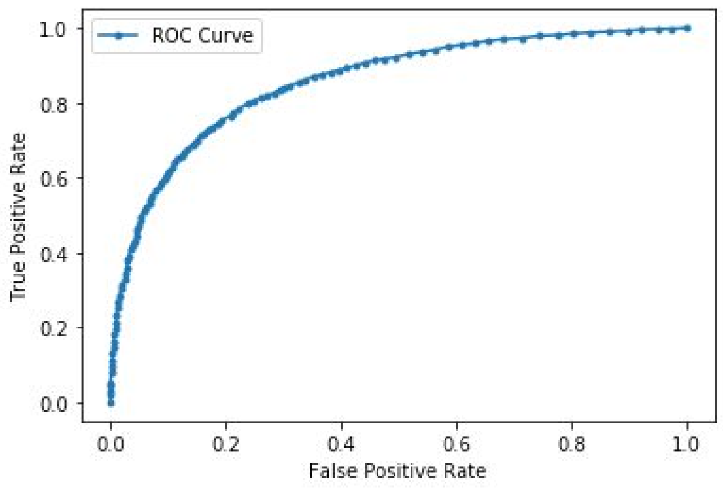
This class implements a meta estimator that fits a number of randomized decision trees (a.k.a. extra-trees) on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. [6]



```
              precision    recall  f1-score   support

           0       0.75      0.86      0.80      3746
           1       0.82      0.68      0.74      3409

    accuracy                           0.78      7155
   macro avg       0.78      0.77      0.77      7155
weighted avg       0.78      0.78      0.77      7155


[[3240  506]
 [1096 2313]]


ET   accuracy score:  0.7761006289308177
ET   precision score:  0.8205037247250798
ET   recall score:  0.6784980932824876
ET   average precision-recall score:  0.70988980744617
```

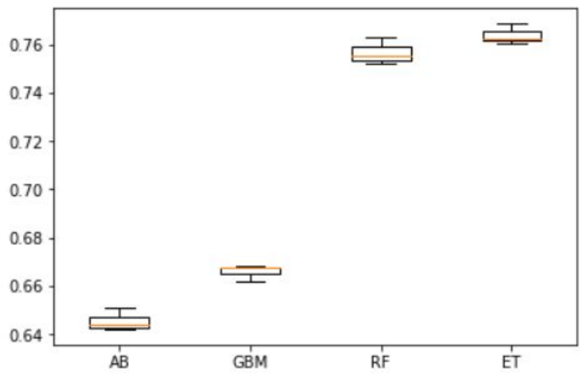


```
AUC:  0.8577697896823787
```

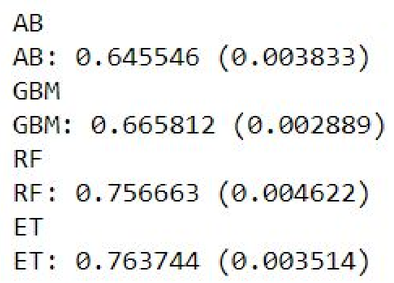


K-FOLD VALIDATION

In this box plot I compare the accuracies of the different ensemble methods. Just like above as the attributes are not varied very much, there is no need to standardize the data in any way.



By training the models and using cross_val_score with k = 3 the resulting mean of all results and std of results were:



```
AB
AB: 0.645546 (0.003833)
GBM
GBM: 0.665812 (0.002889)
RF
RF: 0.756663 (0.004622)
ET
ET: 0.763744 (0.003514)
```

The code was to produce these results:

```
ensembles = []
ensembles.append(('AB', AdaBoostClassifier()))
ensembles.append(('GBM', GradientBoostingClassifier()))
ensembles.append(('RF', RandomForestClassifier()))
ensembles.append(('ET', ExtraTreesClassifier()))

results = []
names = []

for name, model in ensembles:
kfold = KFold(n_splits=num_folds, random_state=seed)
cv_results = cross_val_score(model, X_train, Y_train, cv=kfold,
scoring=scoring)
results.append(cv_results)
names.append(name) msg = "%s: %
```

ENSEMBLE MODELS HOLDOUT VALIDATION COMPARISON TABLE

Holdout Validation				
Classifier	Accuracy	Precision	AUC	Misclassified Headlines
AdaBoost Classifier	66%	0.80	0.70	9731
Gradient Boost Classifier	68%	0.87	0.75	9159
Random Forest Classifier	77%	0.81	0.86	6583
ET Classifier	78%	0.82	0.85	6297

ENSEMBLE MODELS K-FOLD COMPARISON TABLE

K-Fold Cross validation (K=3)		
Classifier	Accuracy (%)	STD
AdaBoost Classifier	65%	0.003833
Gradient Boost Classifier	67%	0.002889
Random Forest Classifier	76%	0.004622
Extra Trees Classifier	76%	0.003514

Interesting from the above results the Extra trees classifier has performed the best over both validation techniques with a precision of 0.82 and AUC of 0.85. When starting the ensemble methods I had thought that Adaboost would have been the most significant algorithm for this problem but I was proven wrong.

MODEL SELECTION

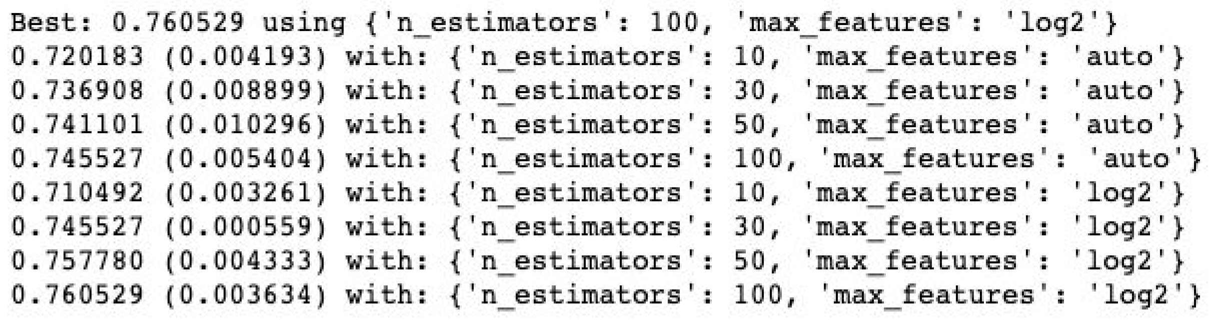
At this point I have engineered a lot of models, using different validation techniques. Different data has been passed and I am now happy that I decided on TFidfVectorizeras it yielded the best results. As of now the model hyperparameters were chosen through trial and error and defaults. Hyperparameters are set by the developer and can have a large impact on the outcome of a model and will in most cases make the model perform to a higher standard. In order to get the best hyperparameters for a model we can use GridSearchCV which searches for the best hyperparameters from values that are passed to fine tune the model and raise the robustness and performance. For example, n_estimators in RandomForestClassifier are often tuned using GridSearchCV. However, this method of fine tuning is computationally expensive and takes time to run. I found and used RandomSearchCV, which is computationally less expensive and yields the same results in most cases. From the code snippet below it was very important to use the hyperparameters of RandomizedSearchCV to my advantage. For example setting n_jobs to -1 , utilities all processors and runs jobs in parallel, without it tuned models would have taken a long time to run.

```
grid = RandomizedSearchCV(model, param_distributions=param_grid,
scoring=scoring, cv=2, n_jobs=-1,verbose=10)
```

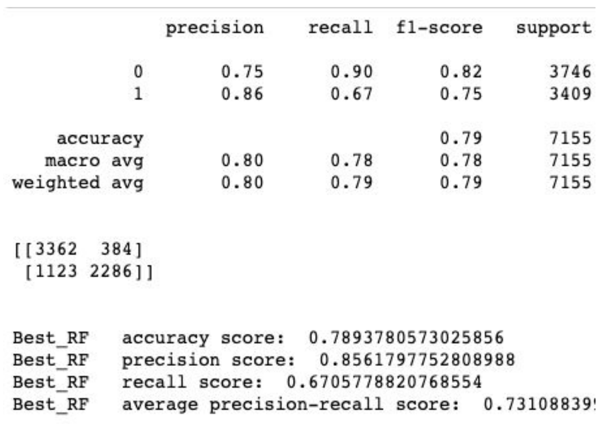
I have chosen three different models to fine tune and look into.

RANDOMFORESTCLASSIFER

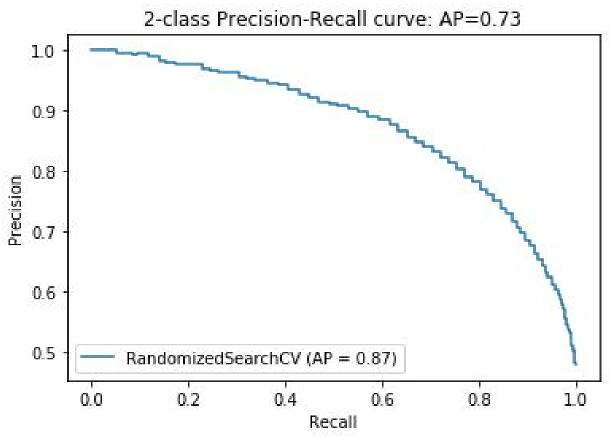
The tuned model using RandomSearchCV yielded the following results:



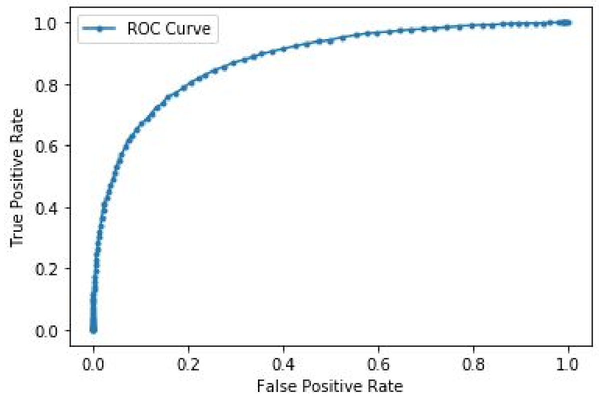
```
Best: 0.760529 using {'n_estimators': 100, 'max_features': 'log2'}
0.720183 (0.004193) with: {'n_estimators': 10, 'max_features': 'auto'}
0.736908 (0.008899) with: {'n_estimators': 30, 'max_features': 'auto'}
0.741101 (0.010296) with: {'n_estimators': 50, 'max_features': 'auto'}
0.745527 (0.005404) with: {'n_estimators': 100, 'max_features': 'auto'}
0.710492 (0.003261) with: {'n_estimators': 10, 'max_features': 'log2'}
0.745527 (0.000559) with: {'n_estimators': 30, 'max_features': 'log2'}
0.757780 (0.004333) with: {'n_estimators': 50, 'max_features': 'log2'}
0.760529 (0.003634) with: {'n_estimators': 100, 'max_features': 'log2'}
```



```
              precision    recall  f1-score   support

           0       0.75      0.90      0.82      3746
           1       0.86      0.67      0.75      3409

    accuracy                           0.79      7155
   macro avg       0.80      0.78      0.78      7155
weighted avg       0.80      0.79      0.79      7155


[[3362  384]
 [1123 2286]]


Best_RF    accuracy score:  0.7893780573025856
Best_RF    precision score:  0.8561797752808988
Best_RF    recall score:  0.6705778820768554
Best_RF    average precision-recall score:  0.73108839
```



```
AUC:   0.8811124160677031
```

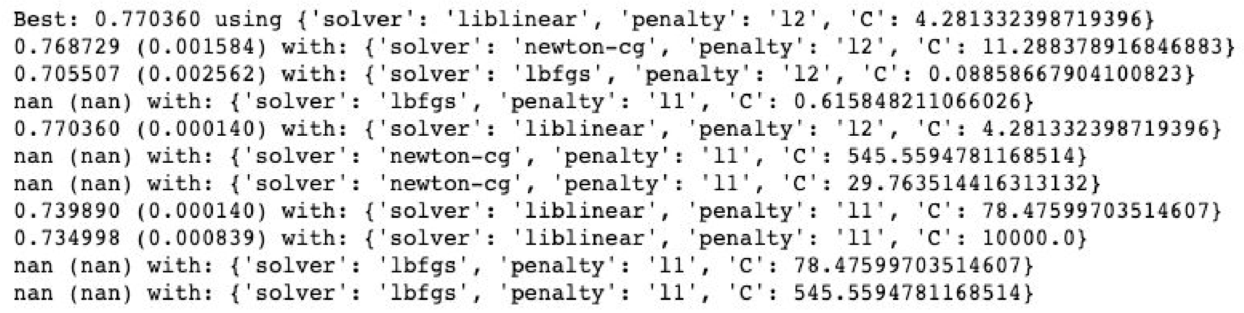


LOGISTIC REGRESSION

Across both k-fold and holdout validation Logistic regression has been performing well.I wanted to try it with many hyperparameters so I set it to:

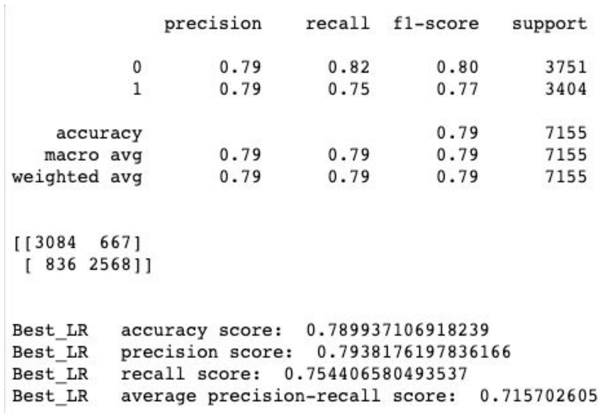
```
penalty = ['l1', 'l2']
c = np.logspace(-4, 4, 20)
solver = ['liblinear', 'lbfgs', 'newton-cg']
param_grid = dict(C=c, penalty=penalty,solver=solver)

model = LogisticRegression()
grid = RandomizedSearchCV(model, param_distributions=param_grid,
scoring=scoring, cv=2, n_jobs=-1,verbose=10)
grid_result = grid.fit(X_train, y_train)
```

This yielded the below results. I really wanted to look into the different solvers used as I thought liblinear would fit well for this and I wanted to gain a deeper understanding of l1 and l2 penalties.

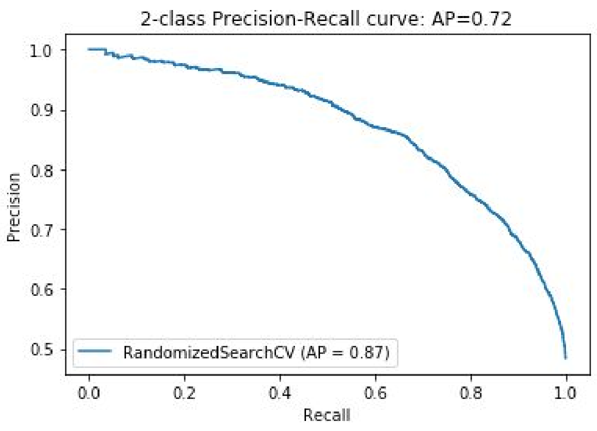


```
Best: 0.770360 using {'solver': 'liblinear', 'penalty': 'l2', 'C': 4.281332398719396}
0.768729 (0.001584) with: {'solver': 'newton-cg', 'penalty': 'l2', 'C': 11.288378916846883}
0.705507 (0.002562) with: {'solver': 'lbfgs', 'penalty': 'l2', 'C': 0.08858667904100823}
nan (nan) with: {'solver': 'lbfgs', 'penalty': 'l1', 'C': 0.615848211066026}
0.770360 (0.000140) with: {'solver': 'liblinear', 'penalty': 'l2', 'C': 4.281332398719396}
nan (nan) with: {'solver': 'newton-cg', 'penalty': 'l1', 'C': 545.5594781168514}
nan (nan) with: {'solver': 'newton-cg', 'penalty': 'l1', 'C': 29.763514416313132}
0.739890 (0.000140) with: {'solver': 'liblinear', 'penalty': 'l1', 'C': 78.47599703514607}
0.734998 (0.000839) with: {'solver': 'liblinear', 'penalty': 'l1', 'C': 10000.0}
nan (nan) with: {'solver': 'lbfgs', 'penalty': 'l1', 'C': 78.47599703514607}
nan (nan) with: {'solver': 'lbfgs', 'penalty': 'l1', 'C': 545.5594781168514}
```

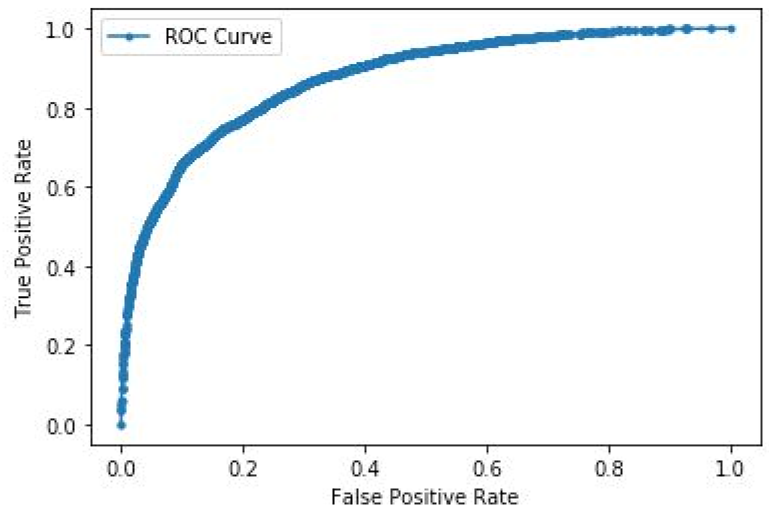
When evaluated the tuned logistic regression model



```
              precision    recall  f1-score   support

           0       0.79      0.82      0.80      3751
           1       0.79      0.75      0.77      3404

    accuracy                           0.79      7155
   macro avg       0.79      0.79      0.79      7155
weighted avg       0.79      0.79      0.79      7155

[[3084  667]
 [ 836 2568]]

Best_LR   accuracy score:  0.789937106918239
Best_LR   precision score:  0.7938176197836166
Best_LR   recall score:  0.754406580493537
Best_LR   average precision-recall score:  0.715702605
```



```
AUC:   0.8745138390044676
```

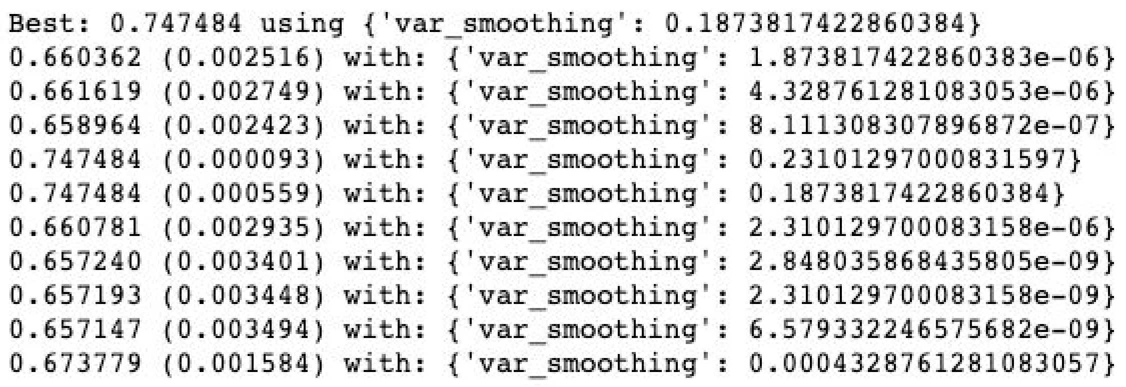


GAUSSIAN NB

A classifier not entirely used in industry, but I thought it would be good to research it and look into it. The code was set up as below,

```
nb_classifier = GaussianNB()
var_smoothing = np.logspace(0,-9, num=100)
param_grid = dict(var_smoothing=var_smoothing)
gs_NB = RandomizedSearchCV(nb_classifier,
                     param_distributions=param_grid,
                     cv=2,
                     n_jobs=-1,
                     verbose=10,
                     scoring='accuracy')

gs_NB.fit(X_train, y_train);
```

This yielded the below results. By changing the var_smothing, this is the portion of the largest variance of all features that is added to variances for calculation stability.



```
Best: 0.747484 using {'var_smoothing': 0.1873817422860384}
0.660362 (0.002516) with: {'var_smoothing': 1.873817422860383e-06}
0.661619 (0.002749) with: {'var_smoothing': 4.328761281083053e-06}
0.658964 (0.002423) with: {'var_smoothing': 8.111308307896872e-07}
0.747484 (0.000093) with: {'var_smoothing': 0.23101297000831597}
0.747484 (0.000559) with: {'var_smoothing': 0.1873817422860384}
0.660781 (0.002935) with: {'var_smoothing': 2.310129700083158e-06}
0.657240 (0.003401) with: {'var_smoothing': 2.848035868435805e-09}
0.657193 (0.003448) with: {'var_smoothing': 2.310129700083158e-09}
0.657147 (0.003494) with: {'var_smoothing': 6.579332246575682e-09}
0.673779 (0.001584) with: {'var_smoothing': 0.0004328761281083057}
```

TUNED MODEL COMPARISON TABLE

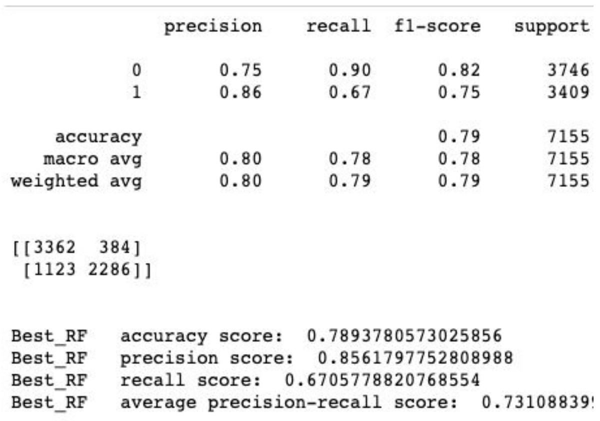
Tuned Models			
Classifier	Accuracy	Precision	AUC
Random Forest Classifier	79%	0.86	0.88
Logistic Regression	79%	0.80	0.87
Gaussian NB	75%	0.85	0.86

It can be seen that the random forest classifier has a 2% increase in accuracy, a 0.05 increase in precision and 0.02 increase in AUC score. This means that by fine tuning the model is now performing better. With more time and more parameters passed, and more computational power all hyperparameters could be accessed and a much more accurate and robust model can be developed.

One model I really wanted to fine tune is the SVC, however I don’t have the computational power at the moment and it is taking a very long time to do a thorough evaluation and analysis unfortunately.

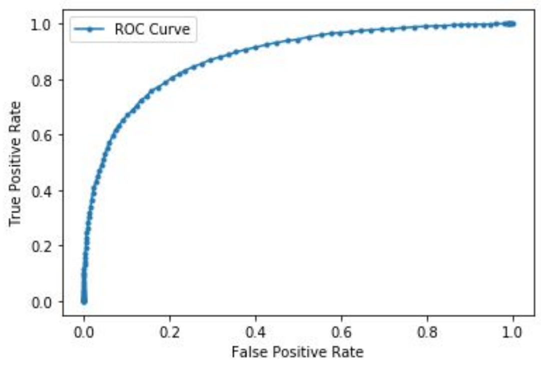
MODEL EVALUATION

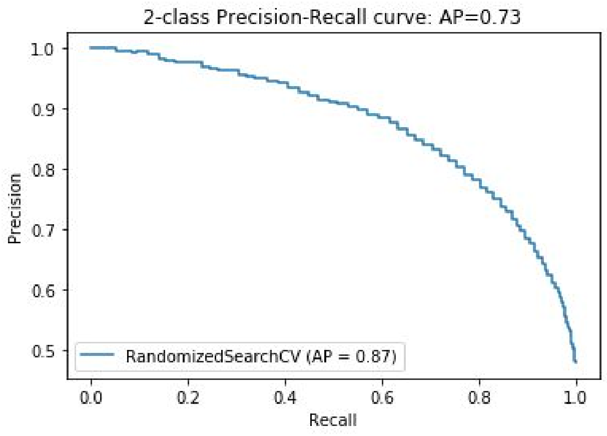
The final model was saved with pickle and loaded to test against the holdout validation set. The testing dataset was set up the same way as all other TFidfVectorizer sets. The below are the statistics for the final model.



```
              precision    recall  f1-score   support

           0       0.75      0.90      0.82      3746
           1       0.86      0.67      0.75      3409

    accuracy                           0.79      7155
   macro avg       0.80      0.78      0.78      7155
weighted avg       0.80      0.79      0.79      7155


[[3362  384]
 [1123 2286]]


Best_RF   accuracy score:  0.7893780573025856
Best_RF   precision score:  0.8561797752808988
Best_RF   recall score:  0.6705778820768554
Best_RF   average precision-recall score:  0.73108839
```

```
AUC:  0.8811124160677031
```





The out of sample error here is 0.211 which was achieved by calculating 1 - accuracy. This metric helps you understand the fit of the model. As the out of sample error gets closer to 1, our main priority is not to overfit the data. Thus, with the out of sample error increasing and degrees of freedom increasing as previously stated overfitting can occur.

The ROC curve of the final model is good for a college project, but at a production level the AUC score would need to be much closer to 1, resulting in a lot more accurate predictions.

Overall I am happy with the final model. However, I am aware that the Naives Baye model had the highest accuracy of 80%. There are no real hyperparameters to tune in Naives bayes and that is the reason I left it as it was and explored different classifiers to enhance my learning.

DISCUSSION AND CONCLUSIONS

Doing this project encouraged me deeper to understand the fundamentals of machine learning and the process of developing an end to end project. If this was a project for production I would have continued evaluating and fine tuning each model I created to ensure that performance and robustness were of high quality. The logistics regression and SVC would have been of significant importance as I feel with some shifting of hyperparameters a lot could be done with it.

Something that I would like to do in the future is to use deep learning on this problem to enhance the performance of models while also trying new things with this very interesting dataset. In particular using Long Short Term Memory and Bidirectional LSTM networks would really enhance the performance of these models on the dataset and with some good engineering bring accuracies up to the low nineties at least.

I learned a lot doing this project, and can definitely bring the knowledge I gained into some of my job interviews. I think what I learned the most is that developing a end to end machine learning project is a long but enjoyable task with many roadblocks that you must adapt and overcome.

I found this project, and EE514 a very knowledgeable experience and I look forward to more machine learning projects in the future.

REFERENCES

[1] NLTK, http://www.nltk.org/api/nltk.stem.html?highlight=lemmatizer

[2] Toward Data Science, https://towardsdatascience.com/tf-idf-explained-and-python-sklearn-implementation-b020c5e83275

[3] Sklearn, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html

[4] Sklearn, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html

[5] Sklearn, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[6] Sklearn, https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.ExtraTreesClassifier.html

[7] Cornell University, https://arxiv.org/abs/1908.07414, 2019