Machine Learning Project Report

Why So Harsh?

Team Name: YourWish

IMT2020039 Anshul Jindal

IMT2020535 Shreeya Venneti

October 29, 2022

Task

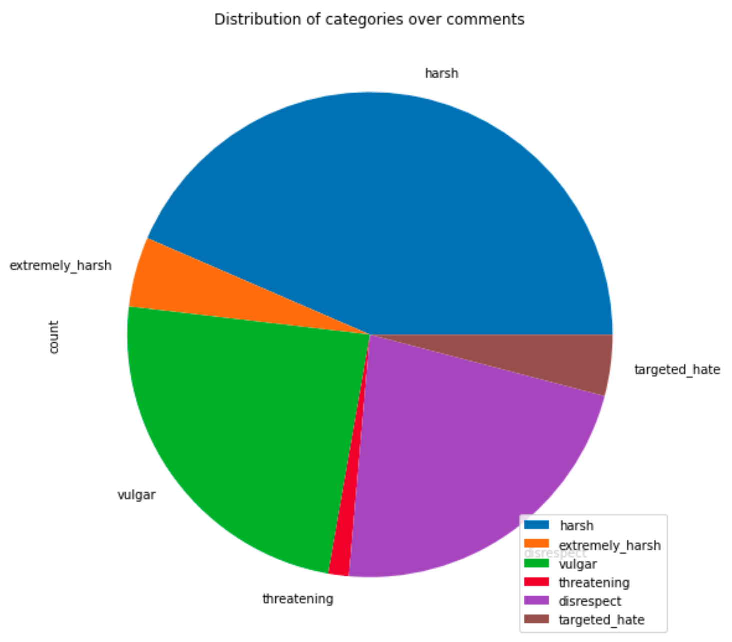
Given a comment, our task is to classify the comment into “harsh”, “extremely harsh”, “vulgar”, “threatening”, “disrespect”, “targeted hate” categories. A training dataset of about 89K comments is given to us, on which appropriate NLP based pre-processing techniques need to be deployed on the text and various classification models are to be fitted & eventually tested on the testing data.

Link to Project’s GitHub Repository ↗

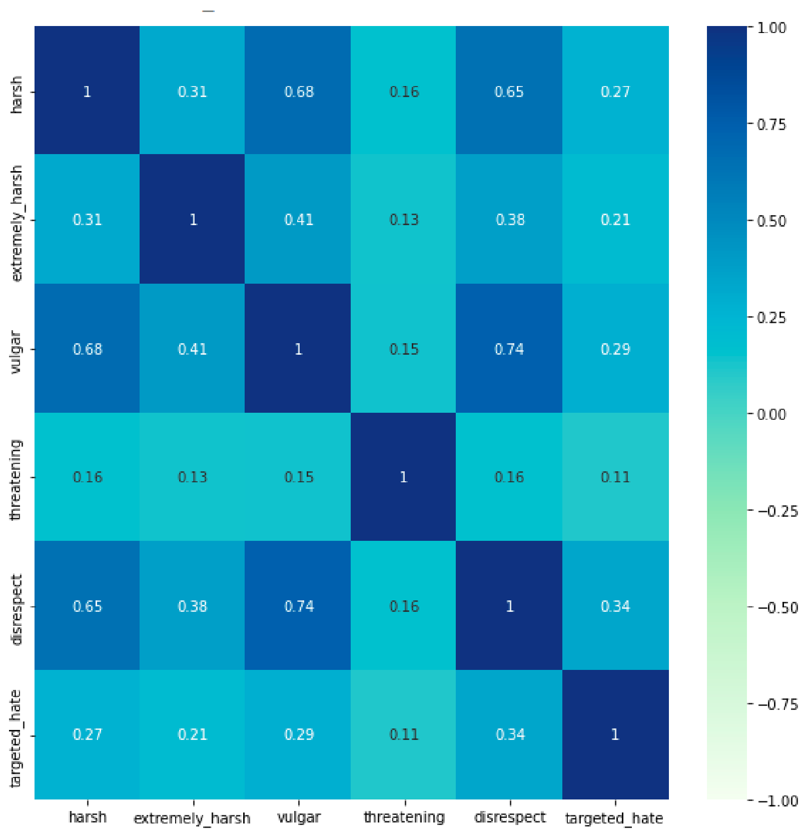
1. Data Pre-Processing and EDA

PIE-CHART & CORRELATION MATRIX

Created a pie-chart using matplotlib to show category wise distribution. The pie chart, (as shown) shows that mostly comments come under: “harsh”, “vulgar” and “disrespectful” categories and thus we can expect a similar distribution in the test data as well.

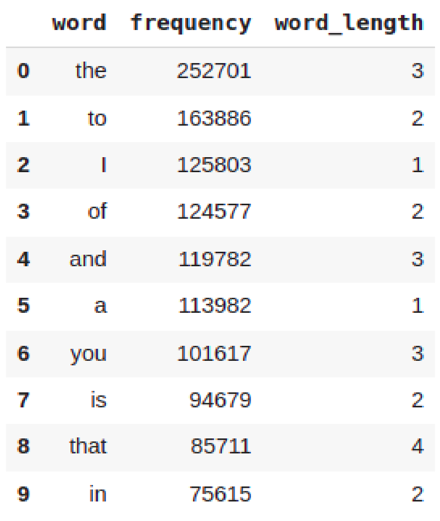


Also plotted a heatmap using seaborn library to check for correlation between categories if any. The categories as shown in the heatmap below, have very negligible correlation between them and thus cannot be linked to each other.



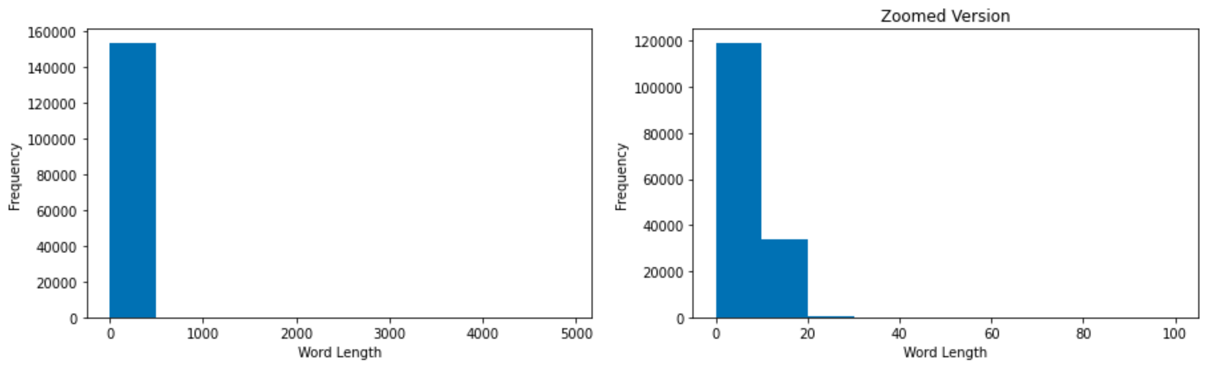
WORD-BASED ANALYSIS

Extracted a data frame consisting of words, their lengths and their frequency of occurrence. Checked if extremely long words are real english words or not. Words having length greater than 40 or 50 (if any) and if they mostly don't make

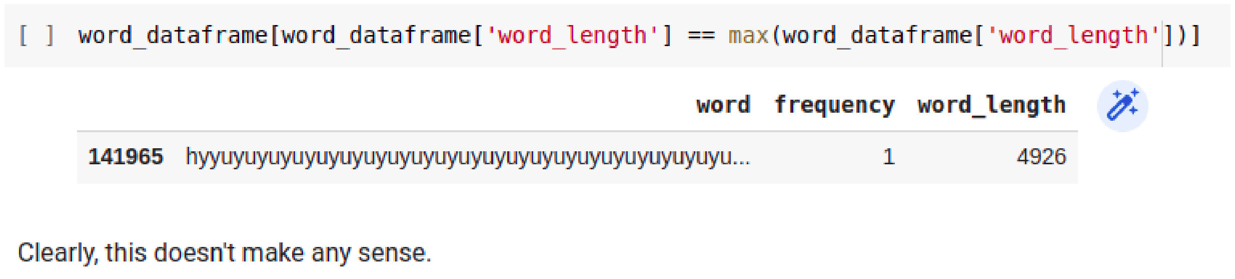


	word	frequency	word_length
0	the	252701	3
1	to	163886	2
2	I	125803	1
3	of	124577	2
4	and	119782	3
5	a	113982	1
6	you	101617	3
7	is	94679	2
8	that	85711	4
9	in	75615	2

sense, then they can be removed. Above is an image of the first 10 rows of the word dataframe extracted:



As shown in histograms above, we have words having word lengths ranging from 0 to 5000 (approx). There are very few words having word length greater than 40. There is very less probability that words of length greater than 40 would make any sense. Further, will they have any contribution towards sentiment analysis, chances are very very slim. Hence, we can remove all the words of length greater than 40. This will speed up the Lemmatization process also. We checked the word with maximum length to verify this claim, and the word as shown is not a real english word and can be removed.



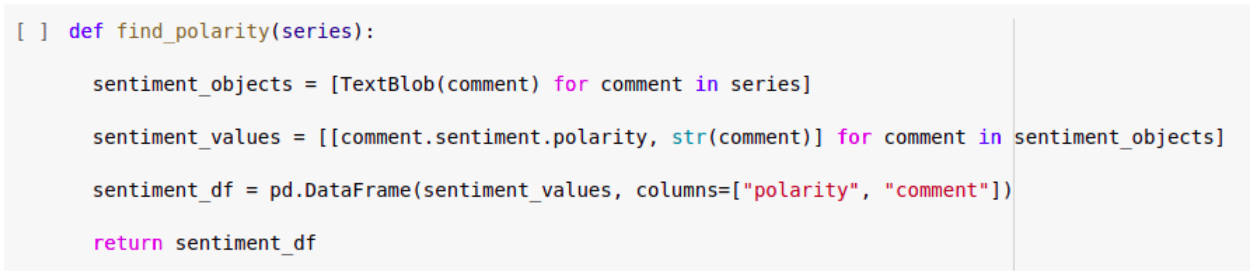
```
[ ] word_dataframe[word_dataframe['word_length'] == max(word_dataframe['word_length'])]
```

	word	frequency	word_length
141965	hyyuyuyuyuyuyuyuyuyuyuyuyuyuyuyuyuyuyuyuyuyuyuyu...	1	4926

Clearly, this doesn't make any sense.

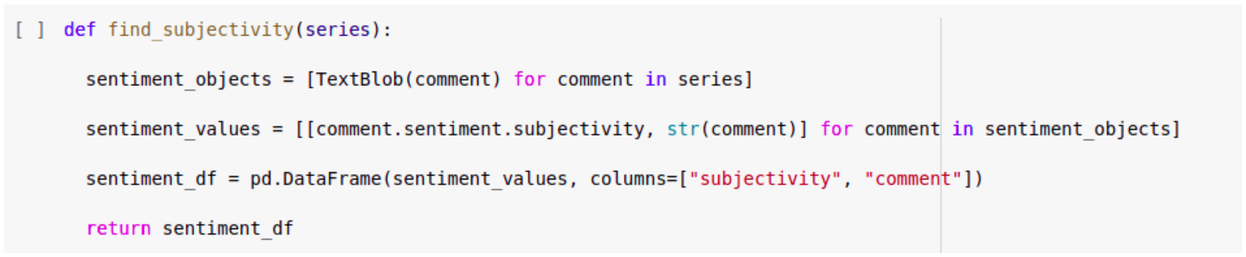
SENTIMENT ANALYSIS

Wrote methods to estimate polarity and subjectivity of the text using the **TextBlob** library to get to know the general sentiment of the commenters. Plotted a histogram to check the distributions of polarity and subjectivity. The **find_polarity** and **find_subjectivity** methods return dataframes containing comments and their polarity values, and comments and their subjectivity values respectively.



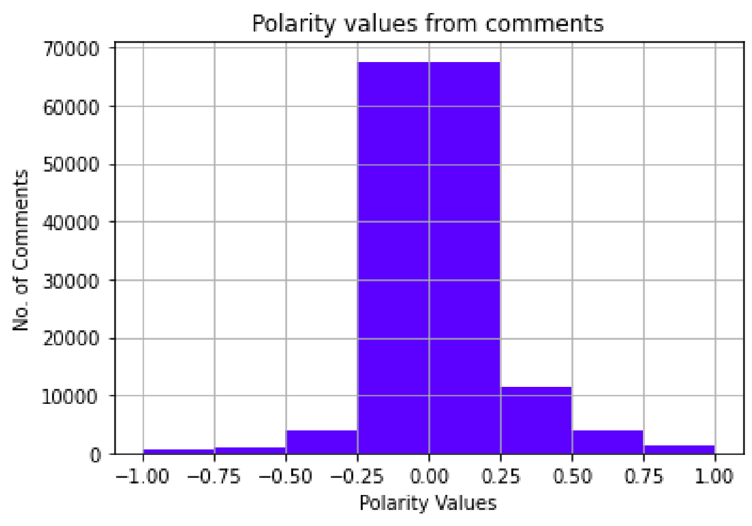
```
[ ] def find_polarity(series):

      sentiment_objects = [TextBlob(comment) for comment in series]

      sentiment_values = [[comment.sentiment.polarity, str(comment)] for comment in sentiment_objects]

      sentiment_df = pd.DataFrame(sentiment_values, columns=["polarity", "comment"])

      return sentiment_df
```

find_polarity method

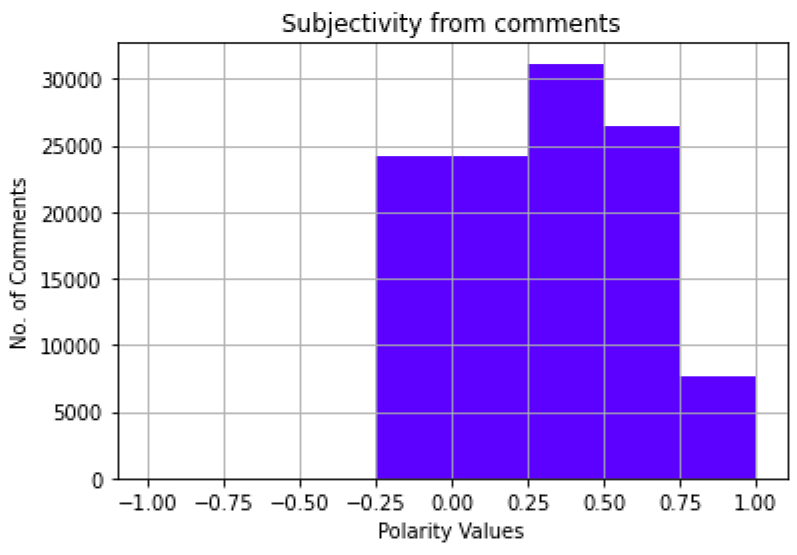


```
[ ] def find_subjectivity(series):

      sentiment_objects = [TextBlob(comment) for comment in series]

      sentiment_values = [[comment.sentiment.subjectivity, str(comment)] for comment in sentiment_objects]

      sentiment_df = pd.DataFrame(sentiment_values, columns=["subjectivity", "comment"])

      return sentiment_df
```

find_subjectivity method



Histogram showing polarity distribution



Histogram showing subjectivity distribution

The polarity histogram shows that most commenters (about 80,000) have a neutral sentiment polarity or no sentiment.

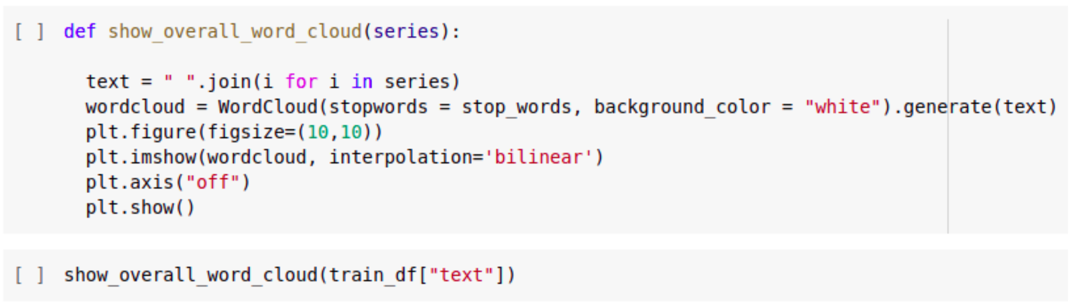
The rest of the commenters about 9,000 have made polar comments, which is the subject of our analysis. Thus we can expect similar results upon fitting our classification model.

The subjectivity histogram shows that the comments are highly biased towards subjectivity, i.e every comment is subjective and is purely based on the commenter's opinion & experience and is not objective/fact-based.

This also indicates presence of strong emotions/feelings of the commenter.

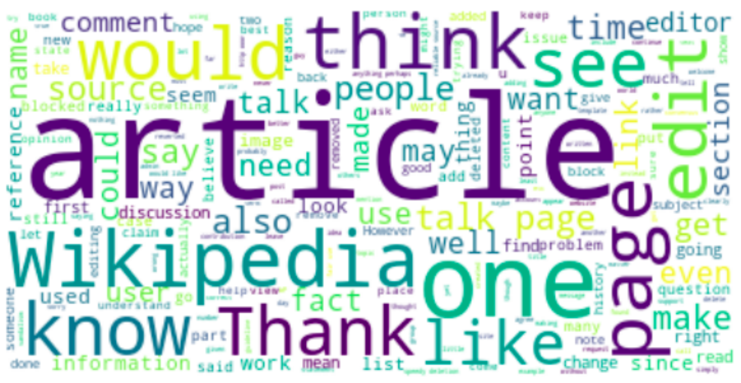
WORD CLOUD

Wrote a method **show_overall_word_cloud()** that displays a word cloud to get a visual representation of the top frequently used words by the commenters. The larger the size of the word in the word cloud, the higher the frequency of usage by the commenter.



```
[ ] def show_overall_word_cloud(series):

      text = " ".join(i for i in series)
      wordcloud = WordCloud(stopwords = stop_words, background_color = "white").generate(text)
      plt.figure(figsize=(10,10))
      plt.imshow(wordcloud, interpolation='bilinear')
      plt.axis("off")
      plt.show()
```

```
[ ] show_overall_word_cloud(train_df["text"])
```



The words "one", "article", "page", "think", etc seem to used very frequently. Some of these words are of no use in case of sentiment analysis.

For example the word "article" can't contribute to sentiment analysis. Hence, we can manually remove these words for out text data.

TEXT EXPANSION

Contractions are words or combinations of words that are shortened by dropping letters and replacing them by an apostrophe. Wrote a method **Expand_text()** to expand the contraction to the full english word and wrote the method **Expand_Dataset()** to perform this for the whole text column.

```
def Expand_text(string):

 string = contractions.fix(string)

 return string
```

```
def Expand_Dataset(dataframe, column_name):

  dataframe[column_name] = dataframe[column_name].apply(Expand_text)

  return dataframe
```

```
# Expanding Text
train_df = Expand_Dataset(train_df, "text")
test_df = Expand_Dataset(test_df, "text")
```

TOKENIZATION AND LEMMATIZATION

Performed tokenization on the entire text column of the training dataset by splitting the text into individual words using the **WordNetLemmatizer()** of the **nltk library**. Wrote a method **tokenize_and_lemmatize_text()** that splits one entry of the text column into words (tokenizes) and lemmatizes the word. This process is repeated for the entire column using the **.apply()** method inside the **lemmatize_dataset()** method.

TF-IDF VECTORIZATION

After performing tokenization and lemmatization, in order to extract features from the lemmatized words generated from the text so far, we then performed TF-IDF vectorization using **TfidfVectorizer()** of sklearn on the words to consider every **word as a feature** with a relevant value associated to it. This concludes the pre-processing and EDA on the dataset.

We also performed **character vectorization**, which performs vectorization on individual **characters by taking characters** as features. We then merged the two vectorized matrices horizontally to form a combined feature matrix, this is the final matrix to be used for model fitting.

2. Data Sampling

RANDOM OVERSAMPLING

There is a high bias in the dataset, i.e there are more number of predictions/labels from one class than the other. At times the ML algorithm proceeds to ignore the minority class completely which is the most crucial/determining class.

The approach is to randomly resample the training dataset, in case of the random oversampling method, we duplicate examples from the minority class to compensate for the number of labels from majority class.

We used the **RandomOverSampler()** from **imblearn.over_sampling**, to implement this. Upon using this, the examples from the minority class are duplicated and adding them to the training dataset.

Whereas, SMOTE works by selecting examples that are close in the feature space, drawing a line between the examples in the feature space and drawing a new sample at a point along that line.

Specifically, a random example from the minority class is first chosen. Then k of the nearest neighbors for that example are found (typically k=5). A randomly selected neighbor is chosen and a synthetic example is created at a randomly selected point between the two examples in feature space.

RANDOM UNDERSAMPLING

In case of random undersampling, we randomly pick training samples from the dataset whose label is from the majority class and delete these samples. This makes the dataset more balanced thus making it suitable for model fitting.

We used the RandomUnderSampler() from imblearn.over_sampling which performs the task of deletion of random majority class samples.

We saw that only oversampling increased the accuracy for some models whereas decreased the accuracy for others. Whereas undersampling decreased the accuracy for every model. This might have to do with the fact that undersampling might be removing some important training examples which play a major role in predictions' accuracy.

PIPELINE

This is essentially a mix of both - oversampling and undersampling. We used the Pipeline from imblearn.pipeline to perform this.

However, it turned out that using pipeline had almost to no effect on the accuracy.

3. Model Fitting

MODELS IMPLEMENTED:

- Logistic Unbalanced
- Logistic Balanced
- Random Forest Classifier
- Ridge Classifier
- Gradient Boosting Classifier
- Adaboost Classifier
- XG Boost Classifier
- Perceptron
- RBF Kernel SVM
- Decision Tree Classifier
- Voting Classifier

UNBALANCED & BALANCED LOGISTIC REGRESSION

We implemented both balanced and unbalanced logistic regression. In the unbalanced case, all the class labels are assigned equal weights, this model gave us an accuracy of 0.98116 on Kaggle. In the balanced case, every class label is assigned weight inversely proportional to it’s frequency, which gave us an accuracy 0.98295 on Kaggle.

RANDOM FOREST CLASSIFIER

The random forest classifier is based on **ensemble learning**, which is a process of combining multiple classifiers to solve a complex problem that improves the performance of the model.

Random forest involves a number of decision trees on various subsets of the datasets, each of which gives out a prediction, from which the majority vote of the decisions are taken and the final output is predicted.

We used RandomForestClassifier from sklearn.ensemble, the criterion was set to gini. Gini impurity is a metric used to determine how the dataset features should split nodes to form the tree. If a node has uniform class distribution then it has the highest impurity. Minimum impurity occurs when all the samples belong to the same class. The feature with smallest gini impurity is used to split the node.

We used the **RandomForestClassifier** module from sklearn.ensemble. Upon implementing this classifier, we got an accuracy of 0.96953 on Kaggle.

RIDGE CLASSIFIER

The ridge classifier is essentially a modification the linear regression model. Here, a penalty term or a regularization term is added to the cost function. The cost function is L2 norm (as in linear regression), unlike Logistic's cross entropy loss. It converts the target variable into -1 and 1. Based on the "regression" output, if the output value is less than zero, the sample is assigned a label of -1, otherwise the predicted label is equal to 1.

We used the **Ridge** module from **sklearn.linear_model**. Alpha value was set to 30. Alpha indicates the regularization strength, i.e higher the coefficient of lambda, lower is the overfitting. Upon implementing this model, we got an accuracy of 0.98314 on Kaggle.

GRADIENT BOOSTING CLASSIFIER

The gradient boosting classifier is based on **ensemble learning**, which is a process of combining multiple classifiers to solve a complex problem that improves the performance of the model.

This is based on the concept of boosting, where predictions are done in a sequential manner where each subsequent predictor learns from the errors of the previous predictors.

Here, we are slowly inching in the right direction towards better prediction (This is done by identifying negative gradient and moving in the opposite direction to reduce the loss, similar to gradient descent). Thus, by no. of classifiers, we arrive at a predictive value very close to the observed value. Here, all classifiers are weighed equally and their predictive capacity is restricted with learning rate to increase accuracy.

Gradient boosting thus combines various “weak predictors” (decision trees) to give out a final additive model. We used **GradientBoostingClassifier** module from **sklearn.ensemble**, upon implementing this model, it however ran indefinitely thus gave us poor results.

ADA BOOST CLASSIFIER

The ada boost classifier is based on **ensemble learning**, which is a process of combining multiple classifiers to solve a complex problem that improves the performance of the model.

This is based on the concept of boosting, where predictions are done in a sequential manner where each subsequent predictor learns from the errors of the previous predictors.

In contrary to the random forest, here each classifier has different weights assigned to it based on the classifier’s performance (more weight is assigned to the classifier when accuracy is more and vice-verse), weights are also assigned to the observations at the end of every round, in such a way that wrongly predicted observations have increased weight resulting in their probability of being picked more often in the next classifier’s sample.

Thus, in contrast to random forest where every classifier/tree is assigned equal weights, in adaboost, every classifier has a different weightage on the final prediction.

We used the **AdaBoostClassifier** module from **sklearn.ensemble** and upon implementing the model, it gave us a low accuracy of 0.54134 on Kaggle.

XGBOOST CLASSIFIER

This is a modification of the gradient boosting classifier. It is much faster and efficient than the original gradient boosting classifier. Upon implementing this model, we got an accuracy of 0.96703 on Kaggle.

PERCEPTRON

Implemented the perceptron model. This is a simple linear model which assigns weights to the features and passes this linear combination of features and learned weights into an activation function (like sigmoid) and obtains the class label. This technique gives us a 100% training accuracy if the data is linearly separable, otherwise it needs to be converged by putting a limit on the number of iterations using the pocket algorithm.

Since it is a linear model, high accuracy wasn't quite expected and the results were in accordance with what was expected. We used the **Perceptron** module from **sklearn.linear_model** and upon implementing the model, it gave us a low accuracy of 0.58991 on Kaggle.

RBF KERNEL SVM

This is one type of kernel svm. Kernel SVM involves projection/conversion of our dataset's feature space into another dimension using a kernel function where in this new dimension, the data points become linearly seperable on which svm perceptron algorithm can be applied in the usual way.

The RBF kernel is one flavour of the kernel svm model where the kernel function is the gaussian function and the space is split into infinitely many dimensions. In this particular space, the weights are learnt and the line fitting is performed. Then this line is then brought back our dimension using the inverse of the kernel function and thus the line now becomes a curve. We have essentially performed curve-fitting.

We used the SVC module from sklearn.svm and upon implementing the model (which ran for 8+ hrs), it gave us an accuracy of 0.94357 on Kaggle.

DECISION TREE CLASSIFIER

This is a simple model where internal nodes of the tree are features, branches or edges are the decisions and the leaf nodes are the final results. Due to it's simplicity, accuracy wasn't expected to be high. We used the **DecisionTreeClassifier** module from **sklearn.tree** and upon implementation of the model, it gave us an accuracy of 0.73080 on Kaggle.

VOTING CLASSIFIER

This classifier worked the best. This classifier aggregates the findings of each classifier passed and predicts the output class based on the highest majority of voting. The idea is instead of creating separate dedicated models and finding the accuracy for each them, we create a single model which trains by these models and predicts output based on their combined majority of voting for each output class.

We implemented the soft voting classifier, where the output class is the prediction based on the average of probability given to that class. We gave Logistic regression, Random forest, SGD classifier and XG Boost classifiers in our voting classifier. Upon implementing it, we got our highest accuracy of 0.98576 putting us at the 4th place o leaderboard on Kaggle.

A COMPARISON OF CLASSIFIER ACCURACIES

S.No	Classifier	Accuracy
1	Logistic Regression	0.98295
2	Random Forest	0.96953
3	Ridge Classifier	0.98314
4	AdaBoost Classifier	0.54314
5	XGBoost Classifier	0.96703
6	Perceptron	0.58991
7	RBF Kernel SVM	0.94357
8	Decision Tree	0.73080
9	**Voting Classifier**	**0.98576**

FUTURE SCOPE OF OUR PROJECT

More powerful algorithms based on Neural Networks like MLP, LSTM which demand resource usage can implemented and tested for accuracy. Overfiting needs to be properly handled since these models are prone to overfit.

With greater access to resources, flags like limit on the number of features (max_features flag) can be removed to make the model run at it's full capacity. This can also improve predictions' accuracy.

THANK YOU!