NLP Sentiment Analysis - Final Project

Mehul Gupta, Satyakin Kohli

Introduction

Our project is about creating an **NLP sentiment analysis system** for viewer comments on **YouTube videos**. The objective is to classify each comment as positive, negative or neutral. In a recent update rolled out by YouTube, the dislike count on videos was made private such that it is now only visible to the creator. Comparing the like vs dislike count used to be the obvious way to assess the public's opinion of any given video and hence its success. However, with the new update, this is no longer possible. A potential alternative is to study the proportion of positive and negative comments but neither YouTube nor the user who is commenting label the comment as positive, negative or neutral. This is where our system comes in to carry out this **multiclass classification problem** and hence facilitate an understanding of audience feedback.

Project Idea Details

We decided to do a NLP project since real world implementations like Apple's Siri, email spam detection etc really fascinated us as consumers. Upon researching the various NLP applications, we came across sentiment analysis. It seemed extremely exciting and relevant to us and we chose to go ahead with it. **The reasons behind opting for YouTube** as the platform to train as well as deploy (purpose) our model on were-

i) One of the largest social media with an abundance of comments and reviews on all kinds of videos, thus allowing us choice.
ii) Offering an alternate way to understand consumer perception after the new update removing dislike count (as discussed in the intro).

Since YouTube is a vast platform with diverse types of videos, the content of comments across different types of videos sometimes tends to differ a lot. We do not intend to cater to every possible comment. We went to different kinds of Youtube videos and manually scanned comments to assess what we could train our model on and what would it be able to correctly classify. Often the comments are very contextual and include a lot of references from the video itself. In some types of videos, it was extremely probable that comments would contain multiple subjects. Now if a certain type of emotion was detected, it would be very difficult to ascertain to whom it was targeted and we would not get a clear picture of the commenter's sentiment towards the video/creator.

Having kept all this in mind, we have chosen to focus primarily on song videos. The model has also been trained accordingly. As a result, the model would be most accurate when deployed on comments for such song videos.

Dataset Specification

The dataset we chose to train our model are comments on **Justin Bieber's 'Baby' song** uploaded on YouTube in February 2010. The rationale behind choosing this specific video is the fact that the song was subject to a lot of mixed reviews. In fact, as per records, it is the 4th most disliked video on YouTube with over 12 million dislikes (List of most-disliked YouTube videos). Yet, it has 19 million likes as well. Such a proportion of likes to dislikes is very rare. Usually, the interactions are very skewed towards either of the two. As a result, we believe it was only logical that the comments on this video would include a roughly **even blend of positive and negative comments** which would better train our machine learning model.

Cleaning

The dataset was obtained using **YouTube's API** and required cleaning before it could be used to train the model. Timestamps in the form of hyperlinks often feature in audience comments and they had to be omitted. We used regex to delete parts of the comment which began with the 'https' keyword to make sure that our comments didn't have any hyperlinks. Moreover, emojis as well as other unicode characters were also often present and had to be eliminated from the comments. This problem was handled by encoding the comments in the ascii format and then decoding them in the utf-8 format. A considerable percentage of the comments were in languages other than English and because they would not add much value to the model, we chose to take a **large dataset of over 20,000 comments.**

Assigning Polarity

Since our ultimate goal is classification, our machine learning problem is a supervised one. We need labels- Positive, Neutral or Negative - for the training comments that we have scraped in order to be able to train our model. It was obviously not realistic to assign polarity to all the comments ourselves so we had to use an existing tool for this. This is where we had to make use of lexical methods of sentiment analysis. This implies a rule based approach wherein a huge dictionary of words is created and each word is assigned a corresponding polarity (can be negative or positive). To then assign polarity to a string of words, all the individual polarities are compounded to arrive at a final figure which indicates the string's polarity.

Two common python libraries performing this sentiment analysis task are **Textblob and VADER**. Whilst Textblob performs more number of NLP tasks like translation, noun-phrase extraction etc, **Vader seemed to be the better fit for our training comments**. Firstly, VADER claims to be attuned to social media sentiments. That is suitable for our system. Moreover, even before knowing this, we opted for a brute force approach of applying both VADER and Textblob to random comments and VADER performed better.

The SentimentIntensityAnalyzerModel was imported and an instance of it was created which was used to retrieve the vader polarity scores for each comment in our dataset. It yields a dictionary of 4 elements for each comment. One element each is to denote the score for the three classification labels i.e positive, neutral and negative. The last one is the compound score and we only consider that to determine our polarity. The compound score isn't taken directly either. It has float values with a lot of precision but this amount of complexity is not needed for us. We reassign all positive values to 1 and all negative values to -1. At this point, all comments with -1 polarity are recognized as negative, those with 0 polarity as neutral and those with 1 polarity as positive.

We also randomly sample 25% of the comments before proceeding. After some test runs, we understood that **18000+ comments were too much** and they did not add much to the performance. **We obtained similar results with roughly 5000 comments**. This also greatly **reduced the cost in terms of time taken training the model** and even allowed us to check for uniformity in the model's performance across different samples.

Further Cleaning

There is yet more cleaning required for the comments before they are used to train the model. To start with, the comments and polarity data was read into a pandas dataframe. All the comments are then **converted to lowercase** to maintain consistency and to ensure that specific words are considered the same irrespective of the case. This is done using a basic lower() pre-defined function in python. After this we **omitted all numeric characters, special characters, punctuation etc by using regex**, to only retain alphabetical lowercase characters.

Next, we used **regex to avoid spam words with duplicating characters** such as "zzzzzzzzzzzzzz", "aaaaaaaaaaaaa" and "niceeeeeeeee" to have a maximum of 2 repeated characters and they become "zz", "aa" and "nicee" respectively.

To convert words such as "worked", "works", "working" to "work", we used the **SnowballStemmer to incorporate a technique called stemming**. Stemming is the process of reducing an inflected word to its root form. Here, a constant problem we faced was that stemming is not a perfect science. It works on a fixed set of rules and thus, it sometimes changes words to the root forms which aren't actual English words. We saw this happening in our dataset when words like "disliked" were changed to "dislik" and "beautiful" to "beauti".

Moving on, words like "is" which don't add much value in training our model were removed from the comments using nltk's '**stopwords**' library. We believed some words such as "not", "aren't", "couldn't" etc. which were important to show negation of statements were important, and deleted them from the list of words to be removed from the comments.

Vectorising

Once we had the corpus of words, we needed to change this list of words into something meaningful for the computer. While we humans can understand words, make sense of them and the context they are used in, computers have no property which allows them to do so. Thus, we needed to convert these words into some kind of numbers which a machine could work with. This process of **transforming words into a matrix of numbers** is called vectorisation.

Out of the couple of vectorisation algorithms out there (such as Count Vectorisation) we chose the **Term Frequency - Inverse Document Frequency (or TF-IDF) algorithm**. We deemed this vectorisation algorithm to be better than the simple Count Vectorisation algorithm becauses the TF-IDF algorithm utilizes the logic that words that (a) appear too many times in the comments and (b) words that appear too little in the comments are both statistically insignificant. As a result,we can say that TFIDF also takes into account the **importance of words** whilst Count Vectorisation only accounts for the frequency of each word.

NOTE: for simplicity, we will call each element of the corpus a ‘sentence’ (i.e. corpus = [sentence1, sentence2, sentence3, …..])

This is done by taking the set of unique words appearing in all the sentences and assigning them a TF-IDF score $tf - idf_{t,d}$ against all the sentences.

$$\text{tf-idf(t,d)} = \text{tf(t,d)} \times \text{idf(t)}.$$

The term frequency $tf_{t,d}$ is the number of times a particular word appears in that sentence. The document frequency df_t is the number of sentences which contain that word. n is the number of sentences (or the length of the list corpus). The inverse document frequency idf_t is the score which is assigned to that particular word. It is calculated as follows:

$$\text{idf}(t) = \ln\left(\frac{1+n}{1+\text{df}(t)}\right) + 1,$$

After getting the value of the matrix through this formula, the values are normalized (so that they lie between 0 and 1) through a process called Euclidean normalization, which is something we have abstracted away for now.

For our implementation, an instance of the Tfidf vectorizer class is created and then the fit_transform() method is used to convert the words in corpus to a sparse matrix with the relevant TF-IDF scores corresponding to specific words in specific comments. The ‘fit’ part of fit.transform() is responsible for carrying out all the computations and calculations. In this case, it is the calculation of the TF-IDF scores The ‘transform’ is responsible for the actual transformation. The .fit_transform() method is a combination of the .fit() and .transform() method. These methods could have just as well been used sequentially too.

Train-Test Split

The sparse matrix can be converted back to a 2d array now and the features (X) are ready for machine learning. At this point we perform a training-test split using sklearn’s function to eventually cross validate. We opt for a test size of 10% because even after sampling we have a large dataset and there will be enough number of test comments to form a clear picture of the performance of the model.

Machine Learning Models

We decided to employ 3 different multiclass classification models and compare results across these algorithms on the test set. These 3 algorithms we chose are **Naive Bayes, Random Forest and K Nearest Neighbors (KNN)**. The aim was to try diverse algorithms which would supposedly yield different levels of result in theory and then check if that was actually the case.

The models work in very different ways from each other and thus, given the size of the training set, the type and nature of data and the number of features, they were expected to perform differently from each other.

How the models work:

- Naive Bayes - It is a probabilistic classifier. It works on the principle of conditional probability as per the **Bayes Theorem.** For categorical variables, the following formula is used

$$P(y|x_1, ..., x_n) = \frac{P(x_1|y)P(x_2|y)...P(x_n|y)P(y)}{P(x_1)P(x_2)...P(x_n)}$$

  x1 …. xn are the features of the model and y represents the output classes.

  The **assumption** made to arrive at this formula is that all the **features are independent** and values of one feature do not affect the other features in any way. Moreover, all features are given equal importance/weight. For a given datapoint, the LHS of the above equation gets calculated for all outcome classes. This is possible because having trained the model, all values on the RHS are made available to us and the formula can be applied. Whichever value of y has the highest conditional probability given the features, is deemed as the class for the given datapoint.

  Since we have continuous data as our features we make use of the Gaussian distribution and modify the formula to get

$$f(x) = \frac{1}{\sqrt{2\pi}\sigma_a} e^{-\frac{(x-\mu_a)^2}{2\sigma_a^2}}$$

where μ_a is the *sample mean*: $\mu_a = \frac{1}{|D_a|} \sum_{x \in D_a} x.a$
σ_a is the *sample standard deviation*, and
σ_a^2 the *sample variance*: $\sigma_a^2 = \frac{1}{|D_a|-1} \sum_{x \in D_a} (x.a - \mu_a)^2$

The metrics like sample mean and standard deviation etc. are calculated based on values provided in the training set. The probabilities are calculated based on the distribution and the rest of the process remains the same as above.

Citation for formulae - here and here

- Random forest classification - It is an **ensemble method** since it is a set of multiple decision trees. The number of trees to consider is one of the most important parameters while creating a random forest model. Each individual tree makes its classification decision and the **majority vote** is counted and the class with highest votes is selected. It would be required to understand how a decision tree works and that knowledge can be extrapolated to random forests.

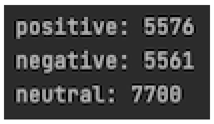
  In decision trees, the aim is to segregate the data into different classes by continuously creating yes/no questions for the features present in the data.There are decision nodes and leaf nodes. At each decision node there is a split of data, thereby creating two subtrees. This is done repeatedly until we are left with nodes belonging to a particular class. When this happens, there is no further splitting and that node becomes a leaf node. A decision tree model is trained by identifying such attribute splits in the training data. Once the model has been trained and the tree has been built, a new data point is classified by starting from the root node and traversing the tree based on how the features answer the yes/no questions at the decision nodes.

- K Nearest Neighbors - The K Nearest Neighbor algorithm works on the principle of **distance of data points**. Many types of distances may be considered - Euclidean, Minkowski etc. All the training data points are plotted on the feature space with their respective classes. For a new data point that needs to be classified, it's k nearest neighbors' classifications are considered. This means that the **classification for the new point would be decided by checking the majority class amongst the k data points lying closest to it**. K or the number of neighbors is one of the most important parameters for this method.

Performance Metrics

To analyze the performance of our model, we used 2 metrics - **accuracy and the f1 score.** Accuracy simply calculates the fraction of comments which were labeled correctly by the machine learning model (positive as positive, negative as negative and neutral vs neutral) out of the total predictions. The f1 score on the other hand is a bit more complex, it uses precision and recall to give a final score.

While we calculated both, **we preferred the accuracy score** more because accuracy is the better measure when the **ratio of positive, negative and neutral labels is approximately the same**. This was indeed the case for our dataset. We found that the ratio of positive, negative and neutral labels for our training data was as follows:



```
positive: 5576
negative: 5561
neutral: 7700
```

The context of the data also sometimes matters to help decide which metric to opt for. The perceived importance of false positives and false negatives in the classification process influences the decision. If even a few instances of either of them would be very costly, then more priority needs to be given to them for evaluating performance. F1 score is a metric which takes both of these situations into account by providing a balance between the two cases. True negatives are neglected for evaluating performance. **In our case, true negatives are important and a few false positives or false negatives aren't super costly.** Therefore, accuracy is a useful metric for our system.

Observations

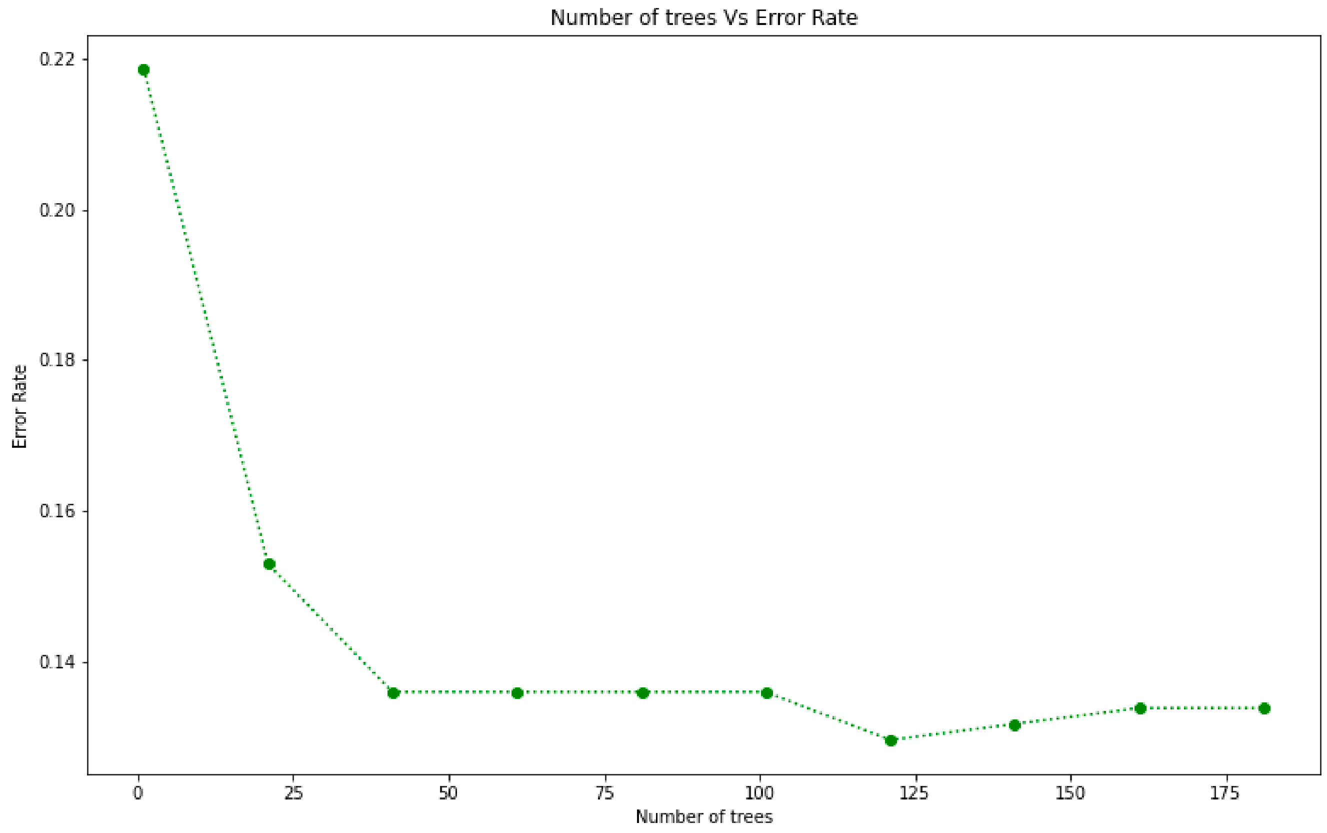
To get the best accuracy results from our models, we conducted **parameter tuning** (on the Random Forest and KNN models) - i.e. finding the value of the parameter which gives us the lowest error value.

After training our model on the training data, we ran the Random Forest classifier on the test data with different values of the number-of-decision-trees parameter (starting from x, going to y in steps of k). Our classifier predicted the labels that should be given to the test data, and these predictions were cross checked with the labels that were originally given to it. We were able to ascertain an error rate associated with these different values of the parameter. **The error rate was calculated as equal to (1 - accuracy)**. Then, the value of the parameter with the **lowest error rate** was chosen and fed as the value to be used for the final classification that our model outputs as that would give us the highest accuracy on the predictions. The same was done for the **KNN model and its n-neighbors parameter.**

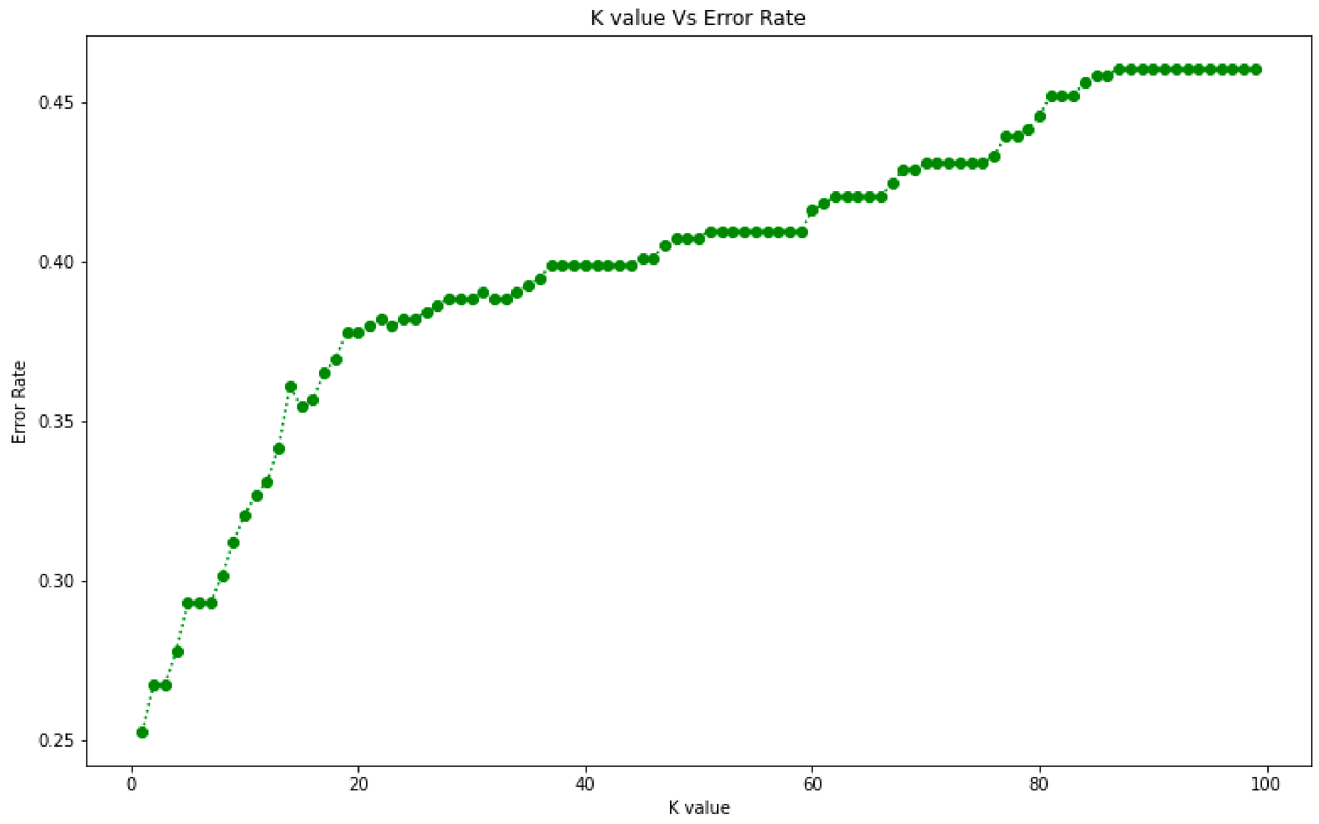
Please also note that the generated confusion matrices for all three classifiers are being displayed through our code and can be seen in the Google Colab notebook.

In the Random Forest model, we saw that with the number-of-decision-trees parameter equal to 120, we received a maximum accuracy of 86.83%. In the KNN model, with the n-neighbors parameter equal to 1, we received an accuracy of 74.73%. Additionally, the Naive Bayes model gave us an accuracy of 44.79.%.

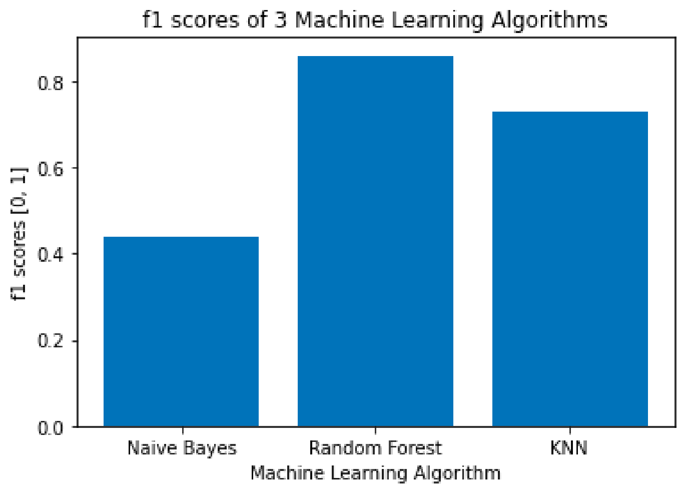
(continued below …)

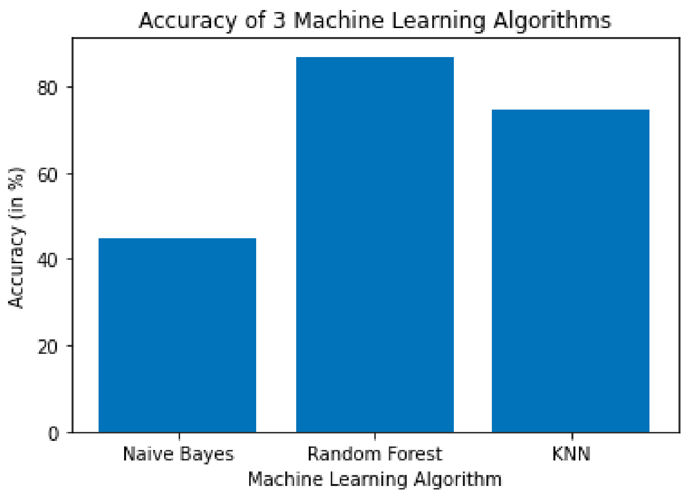


```
optimal value of parameter: 120
```



```
optimal value of parameter: 1
```





Conclusions

Random Forest produced the best results, followed by KNN and Naive Bayes performing the most poorly out of the 3.

The **Random Forest is known for its high accuracy results** and this was the case here as well. The only potential downside could be high training time but that wasn't applicable to us either since we did not train the model on too many examples (not a huge accuracy boost even when we tried to increase training examples). Because Random Forest is an ensemble technique, the possibility of **overfitting our data becomes minimal**.. As we saw in the graph earlier, when the value of the number-of-decision-trees parameter was small (from about 1 to 20), we witnessed significant overfitting of data which resulted in high error rates. As the number of trees increased from 35 to 100, the error rate remained low but constant. When the number of trees went up to 120, we achieved the lowest error rate and our training model fit our data optimally. This low error rate was achieved at a certain value of the parameter due to the working behind the Random Forest classifier. The RF classifier avoids the problem of overfitting as it **averages out the results of multiple decision trees** (120, in this case). Outliers are managed by taking a majority vote on the predictions made by the decision trees, and predictions made on them are rejected if they lose the vote. This way incorrect predictions are kept to a minimum. As the value of the number of trees went up over 120, the error rate rose ever so slightly and then remained constant - this was because there was no significant improvement over the result achieved at 120 decision trees.

The KNN classifier worked better than the Naive Bayes algorithm but wasn't wary of the problems caused by outliers. Since KNN is an algorithm that works on finding the Euclidean distance between two points, any outliers caused it to modify its results significantly. After vectorizing through the TF-IDF, we had a multitude of '0' values in our vector because of the presence of so many features (words). As a result, **the number of outliers also shot up. Overall, however, KNN performed decently.**

Naive Bayes **performed underwhelmingly** despite generally being a popular choice for sentiment analysis. It does not require a large training dataset so we know the problem was not on that front. According to us, the problem could lie in the fact that we had to use Gaussian Naive Bayes because, having used the TF-IDF vectorizer, we had continuous variables containing float values. **If the distribution of the TF-IDF scores wasn't Gaussian**, it was only natural that the accuracy could be substandard. If we had used the count vectorizer, it could have been better for the Naive Bayes algorithm but it made sense to use the TF-IDF vectorizer keeping in mind its advantages and the two other classification algorithms that we employed.

Limitations

We also took note of the limitations of our machine learning models for the task at hand. These were as following:

1. Due to the **removal of the dislike count** over the dislike button, many of the comments were **targeted specifically towards this event and perhaps not towards a dislike of the song/song video**. According to us, the Youtube API does not provide a way to fetch comments only from a certain time period. Whatever the case, the presence of the word "dislike" in a comment suggested a negative review of the comment regardless of its subject. Obviously, the presence of the phrase "not dislike" did the opposite.

2. There were also a small portion of the **comments not in English** which we decided to keep in our dataset. VADER (or TextBlob) were unable to correctly identify the language the comments were written in, and thus it became risky to remove comments.

We felt these problems **misrepresented our data** at times and our algorithm could have performed better had there been a solution to these problems.