# Machine Learning - CS7CS4

# Final Assignment

Brendan O'Connell – 16319077

## Contents



## Question 1

### Introduction

The aim of this report is to evaluate how feasible it is to use Steam review texts and their metadata (voted up or early access) to predict the values of these metadata. The goals of this report are: to preprocess the text by removing useless/nonsense characters and translating all reviews to English, to find the input features that are most relevant for analysis, to optimise different machine learning models which learn about the data and to analyse the results of the machine learning models. By the end of the report, I will have demonstrated and explained how feasible these Steam reviews are in terms of being used as predictors for upvote/downvote and if the review was dealing with an early access game release.

## Text Processing

The most obvious notes about the raw data were that many of the reviews are not in English, many of the reviews contain emojis (and other characters) which may cause difficulty in the context of natural language processing, and also that many of the reviews that were in English were using very colloquial language (e.g., “git gud” which is common slang for “get good [at the game]”). From here, I made the decision to separate the dataset into 2 different datasets: native English and translated English.

```
en - 1997
fr - 71
ru - 874
pt - 215
nl - 37
so - 104
zh-cn - 120
uk - 25
tr - 267
```

I found a Python package (lang_detect) that can detect languages, and I discovered that there were 41 unique languages across the whole dataset (not including the roughly 100 reviews which could not be matched to a language). English was by far the most frequently occurring language (a few of the results are shown to the left), making up almost 40% of the dataset. I feel this justified my decision to use English as the target language for translations. I used the same package discussed here to find all (roughly 2000) native English reviews and separate them into their own dataset then I saved them.

I found a Python package (google_trans_new) that uses Google Translate to translate any text you give to it into a target language. Running this code over the roughly 3000 non-English reviews was lengthy, as the package would detect when I had made their ‘fair usage’ number of requests and not allow me to translate any more text. I had to use my VPN to get around this issue, so I had to run the code in 4 separate chunks in order to translate all reviews. I collected all 5000 reviews (now in English) into a dataset and saved them.

The next step I took was to remove all of the non-ASCII characters which were still in the datasets. This was to reduce the number of features available to learn on that were not English words (things like emojis, stray accented characters or characters from other alphabets that were not translated).

## Input/Output features

Now that I had the datasets in a format that I was happy with, the next step was to figure out how to turn these reviews and their metadata into a format that could allow for machine learning to be applied to them. The bag-of-words method is the method that I have chosen to represent these reviews. The bag of words method is a simple natural language processing technique that can be easily applied in order to facilitate machine learning. One disadvantage of the bag of words approach is that if the document being analysed is large, the number of features can grow quite rapidly leading to very high dimension input features. Another disadvantage of the bag of words approach is that the order of the words in a sentence is lost in this approach. While there are some disadvantages, this approach can still lead to accurate results when used correctly.

A common method for reducing the number of features in natural language processing is removing what are called ‘stopwords’. Stopwords are words that are very common and often add no extra meaning or sentiment to a sentence. In English, these are words like “is”, “the”, “and” etc… Another good way to reduce the number of features is stemming. Stemming is the process of removing common endings of verbs or nouns to make them into one feature e.g., “likes”, “liked”, “liking” would all become “lik” in certain stemming methods. A similar, but more powerful technique called lemmatization is also used. Lemmatization performs similarly to stemming however, it attempts to reduce feature numbers even greater by linking input words to their ‘root’ word (the word that best describes their meaning) e.g., “likes”, “liking”, “likelihood” would all become “like” in lemmatization.

Another powerful method of feature selection in natural language processing is TF-IDF (term frequency – inverse document frequency). This method allows for the frequency of a term in a document across a whole set of documents to be taken into account when the features are being created. In the Sci-kit Learn TF-IDF vectorizer, it allows for a maximum number of features to be given as an argument when vectorizing the whole set of documents. This means that the number of features can be set to a certain limit, therefore each of the input vectors only include the most frequent and therefore assumed to be the most important features.

I attempted to use Sci-kit Learn's count vectorizer, which simply makes feature vectors the length of the number of features it finds across all documents and counts every time one of the features is used in a document. This method produced over 16,000 features for the translated dataset. This meant that some of my models were taking a very long time to make their predictions. In order to reduce features, as discussed above, I used Sci-kit Learn's TF-IDF vectorizer with a set maximum number of features, to which I attached NLTK's lemmatizer and I also removed all English stopwords.

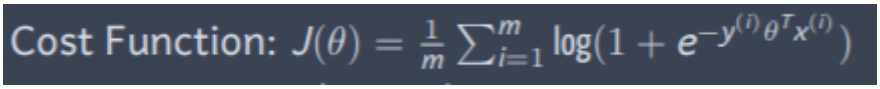
```
vect = TfidfVectorizer(tokenizer=LemmaTokenizer(), max_features=1500, stop_words='english')
```

The dataset, as it was given, is unshuffled in the 'up voted' category. This means that if I pull the first 1500 samples from the dataset, they are all reviews which have up voted the game they are reviewing. Using an unshuffled dataset in this binary classification problem can lead to extremely poor results, I will show this in the results and analysis section. As such, I will be using shuffled data for most of the analysis.

## Machine Learning Models

I have chosen 3 different machine learning approaches in order to tackle the issue of classifying the review texts: logistic regression, k nearest neighbours and random forest. These are all drastically different approaches to this problem.

Logistic regression is a form of regression that uses a log in its cost function (shown below, taken from lecture notes) for training the model parameters. As with other regression methods, the cost function can be altered by adding penalties of different kinds (L1 or L2), in order to stop the model parameter values from being too large. Logistic regression is well suited to binary classification problems because its main strength is being able to easily produce a confidence in a prediction, rather than just spewing out a prediction. For it to be used as a binary classifier, all that is needed is to check if the probability calculated for a given testing point to be in class A is greater than 0.5, if it is, then label that point as class A, if not, label it as class B. Logistic regression is a parametric method, which means that no matter how much data you give the model (as long as all the data is the same shape), the number of parameters produced at the end of training will always be the same.




The k nearest neighbours (kNN) approach is a very simple method for predicting the class of a given testing point. kNN makes its decisions by finding out the class of the closest k neighbouring points to any testing point and then using majority voting to work out what class the testing point should be. Odd values of k are favoured as they will never result in a 'hung vote' between the neighbouring points. The kNN method technically requires no training, as it can only perform predictions at runtime when presented with data. This, however, does not mean that the kNN method is always fast, as sometimes these runtime calculations (especially for input vectors of high dimensions, large datasets, or high values for k) can be very lengthy. kNN is a non-parametric model, instance-based

model, which means that the size of the model changes with the amount of data it is given. There are only 2 features of kNN that require tuning, the value of k and the distance formula. The formula used to calculate the distance between points comes in two common types: uniform and Gaussian.

The random forest approach is a complex way of implementing decision trees in classification. During training the random forest model creates different decision tress for each of the classes it is trying to predict. It uses sub-samples of the data, and then averages across those sub-samples in order to create more accurate decision trees. When making predictions, the random forest will calculate its confidence in each of the decision trees and then classify the testing point with the decision tree that had the highest level of confidence. This approach works very well for data where the classes are highly uncorrelated. I believe that the review dataset is uncorrelated, as positive reviews should use different words to negative reviews. This should allow this approach to achieve a good fit. It is worth noting that the size of each of the decision trees can be limited in Sci-kit Learn's implementation of the random forest classifier, however, I will be using it stochastically i.e., I will allow the classifier to learn from every piece of training data I supply to it.

## Results and Analysis

### Key Performance Indicators (KPIs)

The main KPIs I will be using for rating the performance of each model are accuracy, recall (true positive rate) and ROC AUC score.

The accuracy of a model is the ratio of correct predictions to total predictions. For a balanced dataset in binary classification i.e., especially for the 'up voted' dataset, accuracy will be a very good indicator of the performance of the model. However, accuracy can be misleading for unbalanced datasets, because, if the model constantly predicts the more common class, it can 'accurately' predict the class a lot of the time.

The recall of a model is the ratio of correctly predicted positive classifications to all positive classifications. This metric is more powerful than accuracy when used for unbalanced datasets, as it will harshly penalise a model that always predicts negative outcomes (as is the case with the 'early access' dataset).

### Voted Up Dataset

#### *Native English*

This dataset is well balanced as can be shown by these results:

```
Number of positive reviews: 936
Number of negative reviews: 1061
```

#### Common Class Predictor – Baseline

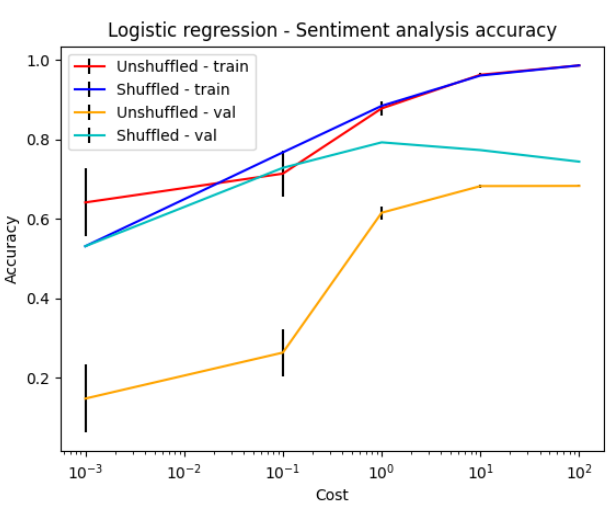
The baseline model for this would be a model which always guesses that a review is negative because negative reviews are more numerous. This model would be expected to achieve an accuracy of 53.13%. The results for an actual baseline model are shown below and the results match the prediction.

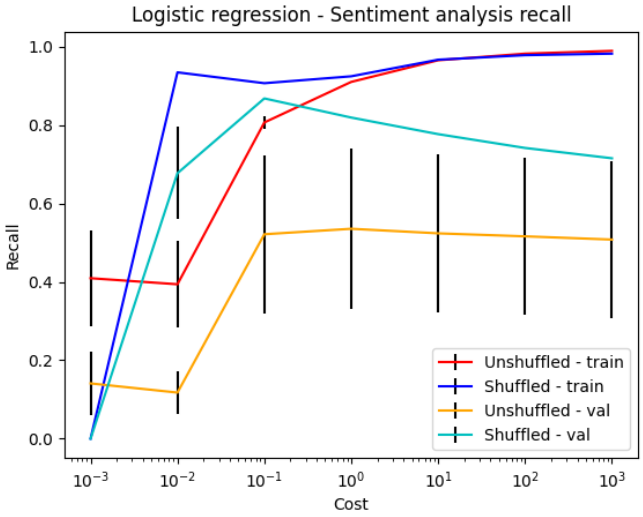
```
BASELINE
Training accuracy - 0.5312974185797369 +/- 6.337335595968481e-06
Validation accuracy - 0.5313045112781956 +/- 0.00010141048737131084
Training recall - 0.0 +/- 0.0
Validation recall - 0.0 +/- 0.0
```

## Logistic regression

For logistic regression, the only hyperparameter that needs to be tuned is the C value.  applied an L2 penalty using this C value. First, I tested a wide range of C values to narrow down the search for the best-fit hyperparameter.

```
costs = [0.001, 0.01, 0.1, 1, 10, 100, 1000]
```




Shown above here are the graphs generated when k-fold cross-validation (k equal to 5 for all cases of k-fold cross-validation) was used to analyse a wide range of C values.

The above graphs show very clearly that for models where C is greater than 10, the training data thinks it is very accurate (approaching 1) without much variance but the validation data shows that the model is not as accurate as it appears, this is a sign of over-fitting. For models where the C value is less than 0.1 the recall is extremely low. This is probably because the model makes very few positive predictions in these cases. Therefore, the range of C values that Iinvestigated further werein the range of 0.1 – 10.

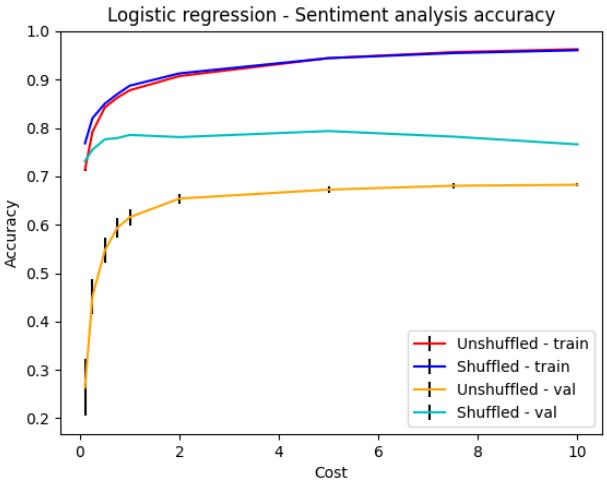
Another thing to note is that the unshuffled data performs worse than the shuffled data in all cases and with much higher variance in its results.

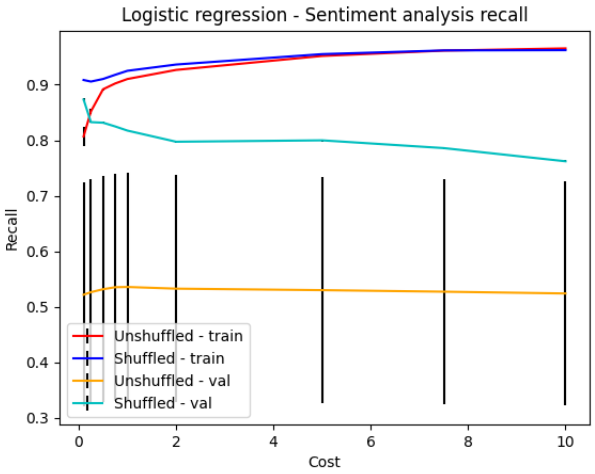
```
UNSHUFFLED
LOGISTIC REG
COST - 1
Training accuracy - 0.8780629826105425 +/- 0.0001872910361030637 8
Validation accuracy - 0.6155726817042606 +/- 0.016800965911018144
Training recall - 0.9102514863374985 +/- 0.0008923725531379582
Validation recall - 0.5355263157894737 +/- 0.20504847645429364
```

```
SHUFFLED
LOGISTIC REG
COST - 1
Training accuracy - 0.8854537175853036 +/- 2.213205476360623e-05
Validation accuracy - 0.7846829573934837 +/- 0.0005069617904410154
Training recall - 0.9245519814567007 +/- 5.971149116083842e-05
Validation recall - 0.8192634035739351 +/- 0.0002585774840792624
```

Shown below here are the graphs which are generated when k-fold cross-validation is performed on C values in the range:

```
costs = [0.1, 0.25, 0.5, 0.75, 1, 2, 5, 7.5, 10]
```



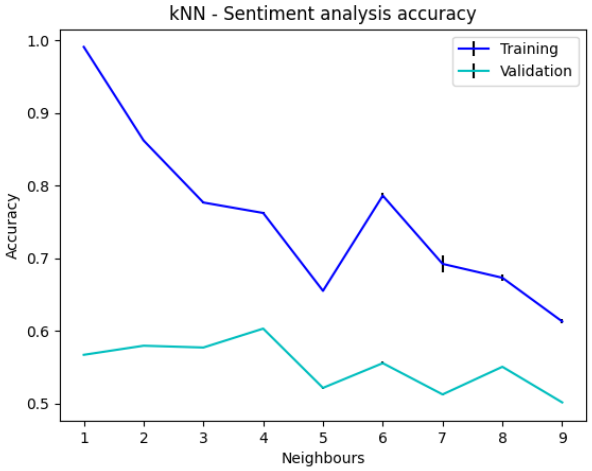

These graphs indicate that for the shuffled data, any C value close to 1 will give a good combination of accuracy and recall. Therefore, I chose a C value equal to 1 as the best-fit hyperparameter for this model. The other obvious thing shown by these graphs is that the unshuffled data will be worse in all cases when compared with the shuffled data. This is due to how this balanced set of data is split during the k-fold cross validation. If very large values of k (e.g., k = 100) were to be used, I would expect that the average of the recall would stay roughly the same while the variance would greatly decrease.

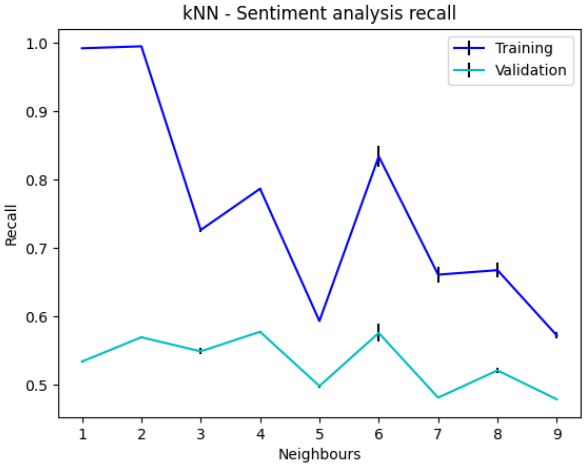
```
SHUFFLED
LOGISTIC REG
COST - 1
Training accuracy - 0.8874557504958844 +/- 4.390304640154381e-06
Validation accuracy - 0.7856842105263159 +/- 0.0001420295004428366
Training recall - 0.9250406224402552 +/- 8.968894231973698e-06
Validation recall - 0.8173265879344243 +/- 0.0005364132147267018
```

## k-Nearest Neighbours

For the k-nearest neighbours approach, the only hyperparameter that requires tuning is k, the number of neighbours. As a result of the size of the input features (1500), this training required a lot of time for my laptop to run. As mentioned in the earlier section about kNN, input features with high dimensions are a potential weakness for kNN. This is because, as the dimensionality of the input vector grows, so too does the number of runtime calculations needed in order to work out which vectors are close to each other. I ran cross-validation for this model using a range of k values from 1-9.

```
neighbours = [1,2,3,4,5,6,7,8,9]
```




Shown above are the results of the cross-validation. These graphs indicate that for all values of k, the performance of kNN is poor for this problem. This is likely due to the sparse nature of the input vectors. If the vectors are sparse, then many of them may appear to be geometrically close together without actually being related. Apart from the poor performance overall, despite what is usually the case (odd numbers being preferred for k in kNN), these graphs indicate quite clearly that the best choice for k in this case is 4. This value for k gives the best balance of a good accuracy and a good recall score for the validation set of data.

```
kNN
Neighbours - 4
Training accuracy - 0.762270778360239 +/- 0.00122154644650788
Validation accuracy - 0.6029248120300752 +/- 0.0004238304784517688
Training recall - 0.7868927103810673 +/- 0.001649727350865423
Validation recall - 0.5776373831485868 +/- 0.0006901583516639749
```

## Random Forest

The random forest approach has no hyperparameters that need to be tuned in this case. Any hyperparameters that are used in making predictions for the random forest can be set to their maximum value without any extra computational power for this problem. As such, no cross-validation is needed to figure out the best performing model. I used the k-fold technique to find the accuracy and recall for the random forest model.

```
RANDOM FOREST
Training accuracy - 0.9922382627627051 +/- 7.222828307235112e-07
Validation accuracy - 0.7551265664160401 +/- 0.0003724889981846845
Training recall - 0.9896284683575656 +/- 1.7264053456293932e-05
Validation recall - 0.7860447057221768 +/- 0.0011542253957695153
```

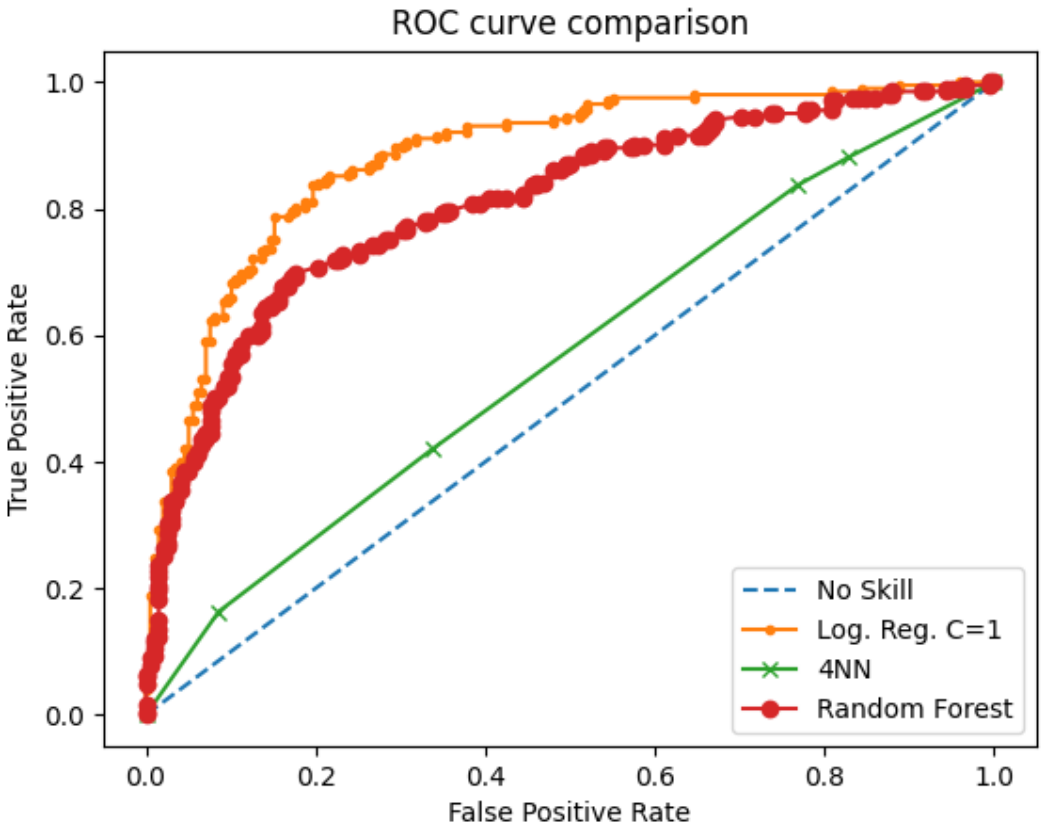
The above results show that the random forest approach works very well for this problem. It has an accuracy and recall comparable to the best performing logistic regression model.

## Compare All

A good comparison for all of the best fit models is the ROC (receiver operating characteristic) curve. This curve plots the false positive rate against the true positive rate of each classifier for different levels of confidence. The closer a classifier gets to the top-left corner, the better the classifier is. If the classifier crosses the no skill line, then it is effectively worse than a complete guess as a classifier. A simple numerical way of showing how well a classifier performs on the ROC curve is representing the area under the curve (AUC). The closer it is to 1, the better the classifier performs in the ROC. As long as the ROC AUC is greater than 0.5, it is better than the random guesser.

As the results to the right demonstrate, the logistic regression classifier is the best performing in this metric, the random forest is not far behind, and the kNN classifier is not much better than the no skill classifier. These findings are in line with the rest of the results analysed during cross-validation.

```
No Skill: ROC AUC=0.500
4-NN: ROC AUC=0.564
Logistic Reg ROC AUC=0.882
Random Forest ROC AUC=0.809
```




*All reviews translated into English*

This dataset is well balanced as can be shown by these results:

```
Number of positive reviews: 2494
Number of negative reviews: 2500
```

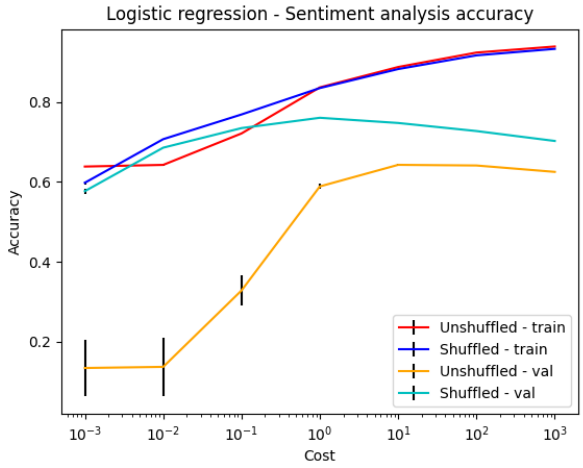
### Common Class Predictor - Baseline

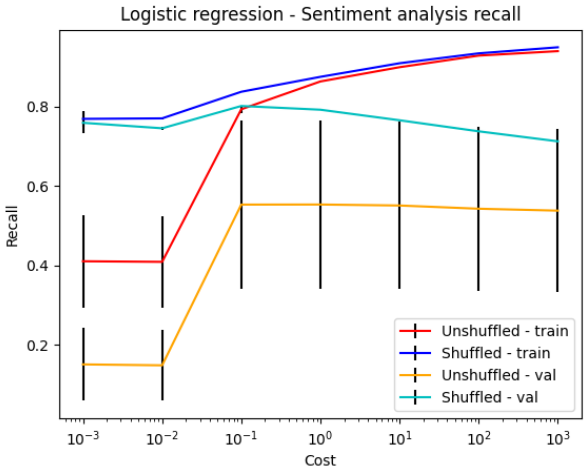
The baseline model for this would be a model which always guesses that a review is negative, because negative reviews are slightly more numerous. This model would be expected to achieve an accuracy of 50.006%. The results for an actual baseline model are shown below and the results match the prediction very closely. The results of the recall show that for one of the folds that was tested, positive reviews were more numerous, therefore the recall reflects those predictions, and this explains the small deviation from the predicted accuracy of 50.006%.

```
BASELINE
Training accuracy - 0.5037545806131538 +/- 6.153211302427625e-06
Validation accuracy - 0.4867855831783688 +/- 0.00014383020773912154
Training recall - 0.2 +/- 0.16000000000000006
Validation recall - 0.2 +/- 0.16000000000000006
```

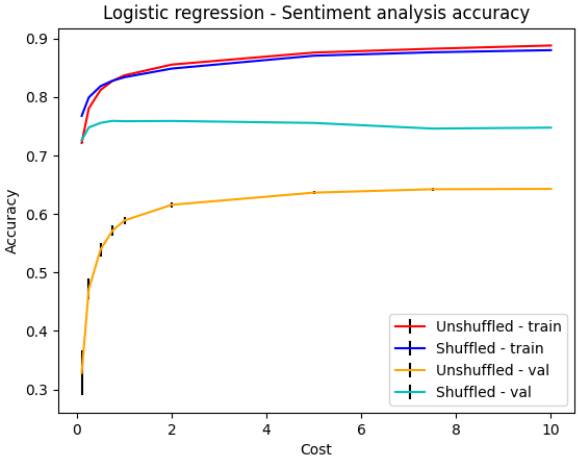
## Logistic Regression

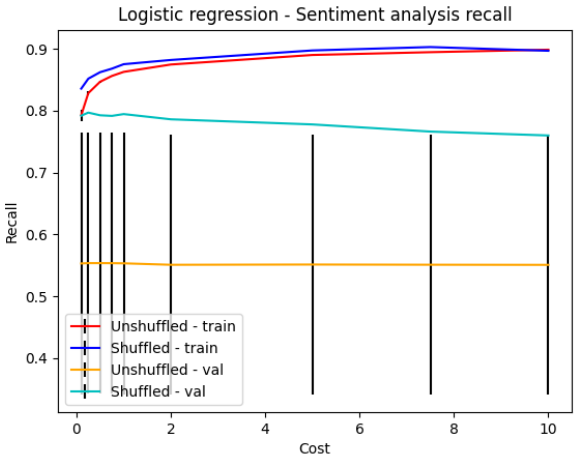
I performed k-fold cross validation in order to find an optimal value for C in this dataset. I started with a wide range of C. I considered both shuffled and unshuffled data to prove again, that the shuffled data will always perform better.




The above graphs show that in all cases the unshuffled data performs worse and with far greater variance than the shuffled data. The best range of values to further investigate for good C values is 0.1 – 10.



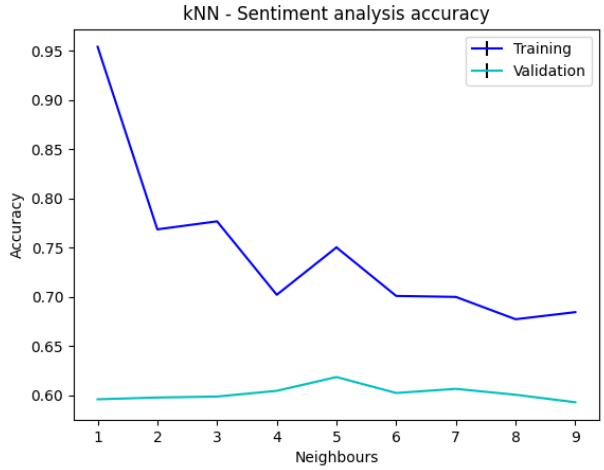



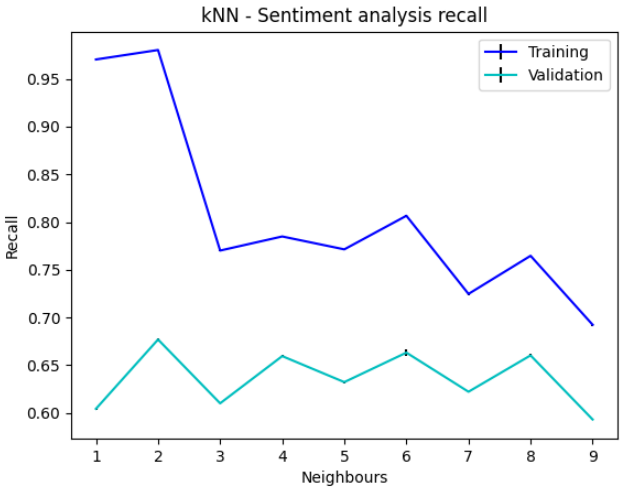

The above graphs show a similar story to the dataset that was just native English, except for the accuracy and recall scores are slightly worse in all cases. It seems that a C value of 1 will give the best fit in this case as well.

```
SHUFFLED
LOGISTIC REG
COST - 1
Training accuracy - 0.8339005964663035 +/- 2.122702137997969e-06
Validation accuracy - 0.7587076054009921 +/- 0.00015094304260679917
Training recall - 0.875248400563077 +/- 5.903847420002526e-06
Validation recall - 0.7946020964349982 +/- 0.0004643713768402761 5
```

## k-Nearest Neighbours

I performed k-fold cross-validation in order to find the best fit value for the k nearest neighbours hyperparameter. I varied k in the range of 1-9. As mentioned the previous time I ran this cross-validation, it took a long time for the code to finish because of the size of the input vectors and the number of them.




In the same way as with the native English dataset the performance was poor overall, only beating the baseline model by a small margin. There is no clear best value for k to be chosen from these graphs, but a simple model that has a good balance of recall and accuracy is created when k is equal to 2.

```
kNN
Neighbours - 2
Training accuracy - 0.7683720641793232 +/- 2.8176656492325263e-05
Validation accuracy - 0.5975205666588432 +/- 0.0005796119913049684
Training recall - 0.98025874675225 +/- 1.0007945610310585e-05
Validation recall - 0.6769003988916028 +/- 0.001721861492573287
```

## Random Forest

The random forest model was run again without cross-validation as no hyperparameters need to be tuned. The below results show that the model performs very well for this problem because it is far better than the baseline model.

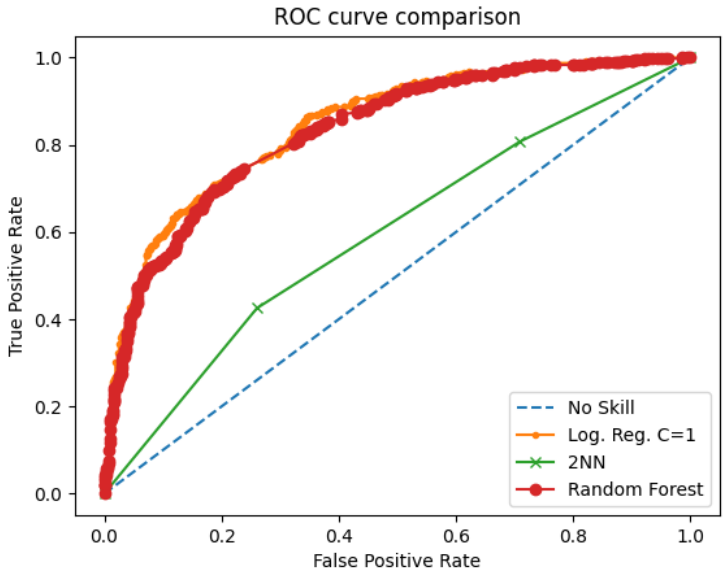
```
RANDOM FOREST
Training accuracy - 0.9616539192509155 +/- 1.8444367421588548e-06
Validation accuracy - 0.7382859813721537 +/- 5.391687924281389e-05
Training recall - 0.9779965177486627 +/- 2.3011701259371395e-06
Validation recall - 0.7817554485636207 +/- 0.00013449821964586927
```

## Compare All

The ROC AUC shows that the kNN classifier is operating slightly better than the no skill classifier, the logistic regression classifier is operating the best, and the random forest classifier is operating slightly less efficiently than logistic regression.

```
No Skill: ROC AUC=0.500
2-NN: ROC AUC=0.595
Logistic Reg ROC AUC=0.847
Random Forest ROC AUC=0.833
```

These results are only marginally worse than the results from the native English only dataset.




## Early Access Dataset

### *Native English*

This dataset is unbalanced as can be shown by these results:

```
Number of early access reviews: 247
Number of non early access reviews: 1750
```

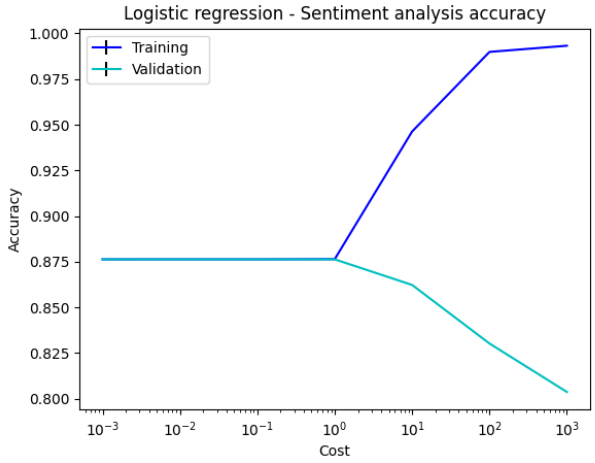
### Common Class Predictor – Baseline

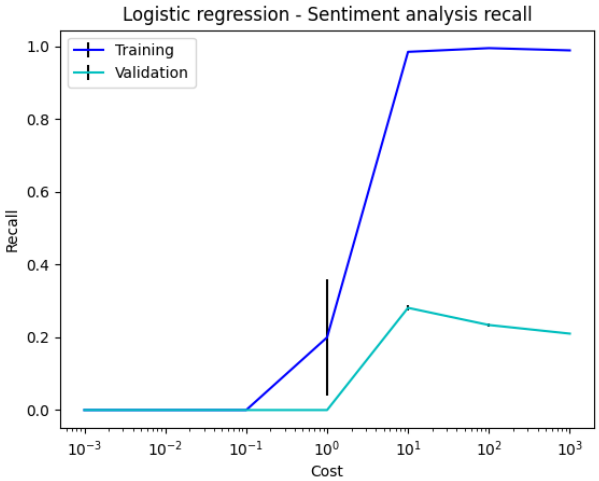
The baseline model for this would be a model which always guesses that a review is not an early access game, because these are more numerous. This model would be expected to achieve an accuracy of 87.63%. The results for an actual baseline model are shown below and the results match the prediction.

```
BASELINE
Training accuracy - 0.8763143190102218 +/- 1.0942709929709176e-05
Validation accuracy - 0.8763120300751879 +/- 0.00017518972242636655
Training recall - 0.0 +/- 0.0
Validation recall - 0.0 +/- 0.0
```

## Logistic Regression

I performed k-fold cross validation on this dataset across a wide range of C values in order to determine a best-fit value for C.

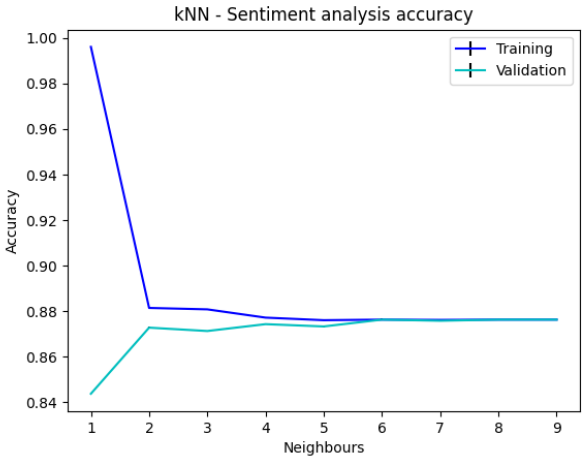


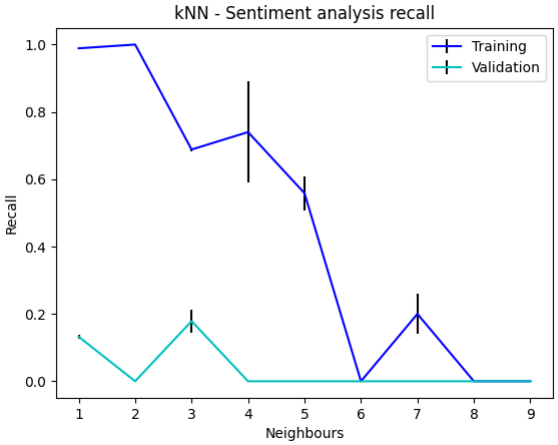




The above graphs show that for small values of C, the classifier will always choose the most common class, and this will result in it achieving quite a high accuracy (roughly 88%). This means that it is a useless predictor for these values of C, as it performs no better than the baseline. However, for larger values of C, the model begins to attempt to make predictions about which reviews are early access, and which are not, however, it is only correct around 28% of the time, which makes it a worse predictor than when it was simply predicting the common class all of the time. There is no best fit value for C in this case.

## k-Nearest Neighbours

I performed k-fold cross-validation on this dataset for a wide range of k values.




The graphs shown above show similar behaviour to the logistic regression model on this problem. That is to say that when the model predicts a constant class (in this case the negative class), it is more accurate than when the model attempts to make educated guesses about which of the reviews might be describing an early access version of a game. For all values of k, the model never got a prediction correct a significant enough number of times in order to beat out the constant class predictor. As such, there is no best fit value for k in this problem.

### Random Forest

I used the random forest approach here. As shown by the results below, it performed as well as any of the other classifiers, in that its performance is comparable to a model that always predicts the common class. This behaviour in the accuracy makes the recall score earned by the model quite interesting as it suggests that the model did make a fair number of guesses, but still only managed to be as good as the baseline.

```
RANDOM FOREST
Training accuracy - 0.9957436620446817 +/- 6.232462758454552e-08
Validation accuracy - 0.8728032581453634 +/- 0.0002812428565147198
Training recall - 0.9979682343234323 +/- 6.198755047784505e-06
Validation recall - 0.27999999999999997 +/- 0.03137777777777778
```

### *All reviews translated into English*

This dataset is unbalanced as can be shown by these results:

```
Number of early access reviews: 527
Number of non early access reviews: 4467
```

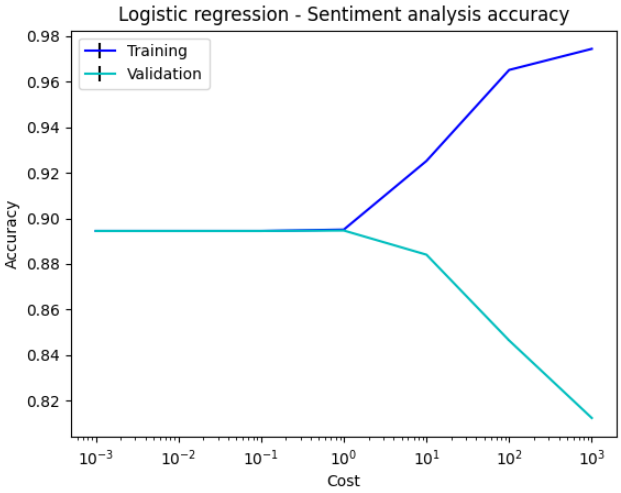
### Common Class Predictor – Baseline

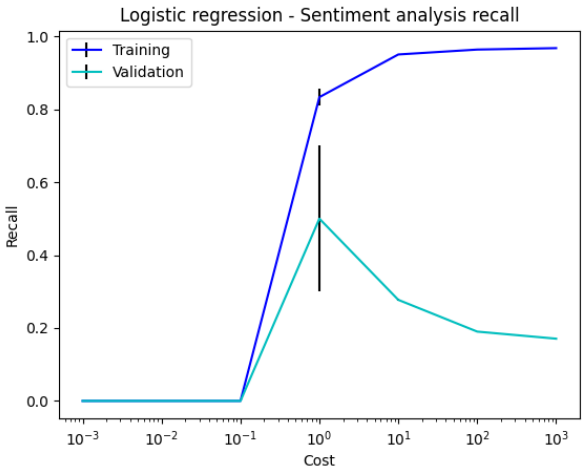
The baseline model for this would be a model which always guesses that a review is not an early access game, because these are more numerous. This model would be expected to achieve an accuracy of 89.44%. The results for an actual baseline model are shown below and the results match the prediction.

```
BASELINE
Training accuracy - 0.8944733594671016 +/- 3.4753743334004083e-06
Validation accuracy - 0.8944732307457759 +/- 5.557839717057863e-05
Training recall - 0.0 +/- 0.0
Validation recall - 0.0 +/- 0.0
```

### Logistic Regression

I performed k-fold cross-validation using a wide range of C values in order to determine a best fit value for C.

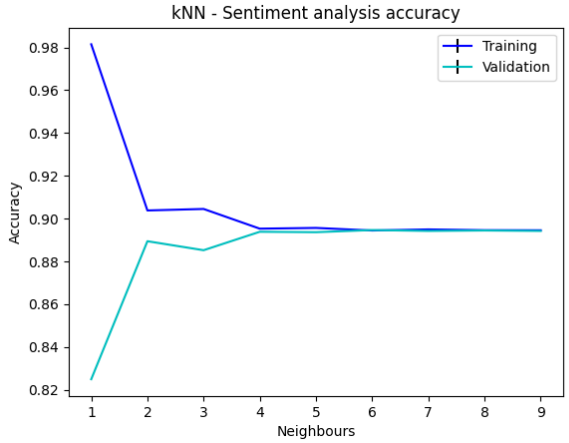


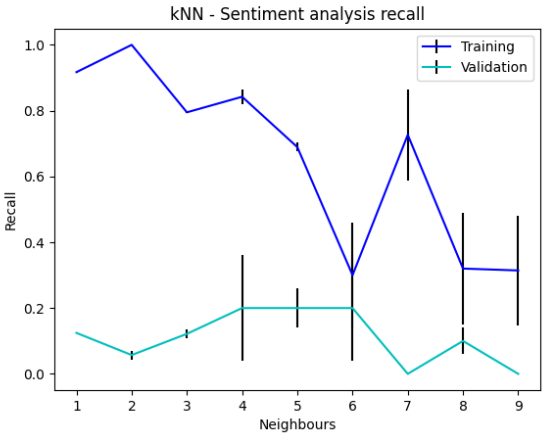




These models performed very similarly to the native English models, in that while the classifier was always choosing the most common class, the predictor was very accurate (but useless). When the models began to attempt to make predictions about whether the review was for an early access game or not, it got its predictions wrong 50% ± 20% of the time. There is no best fit value for C in this case.

### k-Nearest Neighbours

I performed k-fold cross-validation on this dataset for a wide range of k values.




The issues with being able to predict early access from a review's text persist. Yet again, the model is most accurate when it is predicting a constant class. As soon as the model attempts to make its own predictions, it wrong far more often than it is right. This makes this model a poor classifier for this problem. As such, there is no best fit value for k.

### Random Forest

This classifier performed comparably to the common class predictor. It is not an improvement over the random classifier. This model may be discarded.

```
Training accuracy - 0.9882859329918153 +/- 1.8569550398995474e-07
Validation accuracy - 0.8904686249375627 +/- 6.637702403918031e-06
Training recall - 0.9968265995317992 +/- 9.492035388529724e-06
Validation recall - 0.3404373404373404 +/- 0.01904901378760852
```

## Conclusion

The aim of this report was to evaluate how feasible it is to use Steam review texts and their metadata (voted up or early access) to predict the values of these metadata. I separated the review text into those that were written natively in English and then I translated all of the reviews into English because I believed that there may be a drop off in accuracy of sentiment when text is translated from the language in which it is natively written. I used several different machine learning methods to analyse the sentiment of these texts both in regard to the polarity of the review and in regard to whether the review was concerning an early access version of a game or a normal release of a game.

It turns out that my assumption that text being translated into English would not preserve the same amount of sentiment as the purely native English reviews was only correct to a very small degree. For all models, solely native English reviews performed better than the translated texts but usually only by a margin of a few percentages. I believe that all texts being translated into English (despite being slightly less accurate) would make the sentiment analysis far more powerful as much more text could be analysed.

In conclusion, I believe that evaluating the text of these reviews can reveal enough sentiment to make a feasible attempt to classify the polarity of the review. This conclusion is backed up by the fact that all 3 of the machine learning models I trained performed better than the baseline model.

The results for the logistic regression model which used only native English reviews were overwhelmingly positive. Even the random forest model showed very positive results.

I do not believe that analysing the review texts can reveal enough sentiment to predict whether the game being reviewed was an early access version or not. This conclusion can be backed up by the fact that there were no machine learning models that performed much better than the baseline model for this task. I believe that this was possibly a weakness of the approach I took but could also be a result of the fact that not many people reviewing early access games acknowledge that in their review.

# Question 2

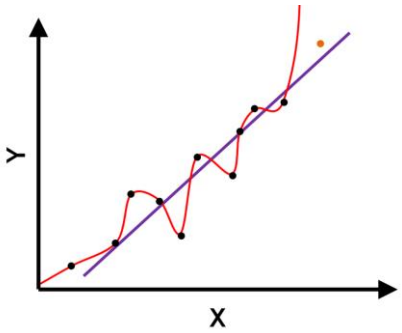
## Part (i)

**What are under-fitting and over-fitting, give an example of each.**

Under-fitting and over-fitting are terms used when referring to how well a machine learning model performs on its training set of data compared to how it performs on any validation or testing data.

Under-fitting is what occurs when the machine learning model being analysed has not 'learned' enough about its training dataset to create meaningful accurate predictions for the training dataset and any validation data passed to it. For example, if a machine learning model were being used to calculate the steering angle of a car's wheel when a car is driving around a track, a model that has under-fitted may simply calculate the average steering angle from the training dataset and apply that as a constant. This would show that the model has not 'learned' how to make predictions accurately, even for the data it trained on. This approach would not provide a satisfactory route around the track it trained on, or any racing track (barring a large circular track).

Over-fitting is what occurs when the machine learning model being analysed has fitted too well to the training data supplied to it. If a model fits too strongly to its training data, then it will not generalise well when making predictions for a validation dataset. Over-fitting often manifests as the model learning about the noise in the data which will provide misleading results. For example, look at the figure below:




This figure shows data (black dots) being modelled by a suitable linear regression (blue/purple line). The figure also shows a model (red line) which appears to have over-fitted the data. As described above, the red line has tried to learn about the noise of the system, which will lead to the model not generalising to other data very well.

## Part (ii)

**Give pseudo-code implementing k-fold cross-validation.**

This program divides a set of data into a given number of folds, and loops though all combinations of training data (number of folds – 1 / number of folds) and testing data (1 / number of folds). For each training and testing set, metrics are collected and measured. At the end of the loop, the metrics are averaged, variance is found, and the results are presented.

X = input features

y = output features

param_arr = array of hyperparameters to be cross validated

metrics_mean = metrics_variance = empty array

for param in param_arr

num_folds = number of folds (often 5 or 10)

metrics = empty array

for training_range, testing_range split(X) into num_folds

model = instantiate model you are testing using the param from the main loop

train the model to fit X[training_range] to y[training_range]

predictions = use the trained model to predict y values for X[testing_range]

metrics.append(calculate metrics using predictions versus y[testing_range])

metrics_mean.append( average of the metrics across every combination of the folds )

metrics_variance.append( variance of the metrics across every combination of the folds )

plot a graph with the hyperparameter being validated on the x-axis, the value of the metric being measured (e.g. accuracy, precision, recall, etc…) on the y-axis, the x and y values coming from the metrics_mean array and the errorbars for these values coming from the metrics_variance array.

## Part (iii)

**Why does k-fold cross-validation provide a way to select a model hyperparameter so as to strike a balance between over/under-fitting.**

K-fold cross-validation provides a way for hyperparameters to be given a wide range of values and then compared using any metric of choice. This allows for a wide range of models to be analysed, where the only difference between them is the hyperparameter being isolated. As mentioned in part(i), models which are under-fitting and models which are over-fitting can be identified easily. When a model that is not performing well is spotted, the value for the hyperparameter at that point can be disregarded, narrowing the search for a best-fit hyperparameter.

If a wide range of values for the hyperparameter does not supply sufficient information for analysis, a smaller and more specific range can be used to find appropriate values. Being able to vary the range of analysis will provide the person selecting the hyperparameters with all of the information needed to make a reasonable decision for the best-fit value of the hyperparameter.

Another advantage of k-fold cross-validation is that it can give variances of performance metrics. This in turn, gives the person selecting the hyperparameter information about the consistency of the models being analysed. If a model performs seemingly well in a metric but with a very high variance of that performance, it indicates that the model may not be performing as consistently as it may have appeared at first and the model may actually give a poor fit.

## Part (iv)

**Discuss three pros and cons of a logistic regression classifier vs a kNN classifier.**

Pros logistic reg vs kNN:

- Logistic regression is faster to make predictions, especially for many-dimensional data. For kNN, as the number of dimensions grow, so too does the amount of computation power needed to calculate the distance between 2 features. This results in massive runtimes for kNN

- Logistic regression model coefficients can easily indicate direction and sensitivity of certain features towards the outcome.
- Logistic regression is more scalable than kNN as it requires fewer runtime calculations after training and the size of the model never changes (parametric model).

Pros kNN vs logistic reg:

- kNN can learn non-linear decision boundaries very easily.
- kNN is very easy to implement in multi-class problems.
- kNN is a non-parametric model so no assumptions need to be made about the shape of the data before kNN is applied.

## Part (v)

**Give two examples of situations when a kNN classifier would give inaccurate predictions. Explain your reasoning.**

High dimensionality

A kNN classifier does not work well with data that has very high dimensions. A good example of a machine learning dataset that often has many dimensions is text analysis using the bag-of-words approach, especially for large document sizes. The input vectors for this kind of a problem are a sparse array with many dimensions. When considering sparse arrays of high dimensions, the distance between any 2 features can be very small by complete chance, thus giving poor prediction results.

Unbalanced datasets

A kNN classifier can make poor predictions when it is presented with an unbalanced dataset ( a dataset where one of the classes appears a hundred times more often than the others). The reason that the kNN classifier performs poorly with this kind of dataset is because the model relies on 'majority voting' but if one class is severely outnumbered by the majority class, then all votes will lead to the model predicting the more common class. A good example of an unbalanced dataset would be data tracking online purchases' legitimacy (purchases authorised by the card holder). In this dataset, there would exist far more data concerning legitimate purchases than fraudulent ones. The system would be expected to detect all fraudulent purchases, so the cost of a false negative is quite high. If a kNN classifier were used to classify this dataset, there is a good chance that the model would be very poor at making the correct predictions, as it would always predict that the purchase is legitimate.

## Appendix 1 – Text processing code

```
import json_lines
from langdetect import detect, detect_langs
import pandas as pd
import numpy as np
from google_trans_new import google_translator

def find_english(X, y, z):
    count = 0
    no_lang_count = 0
    X_en = []
    for i in range(len(X)):
        try:
            if(detect_langs(X[i])[0].lang == "en"):
                count += 1
                X_en.append([X[i], y[i], z[i]])
        except:
            no_lang_count += 1
    return X_en, count, no_lang_count

def find_non_empty(X, y, z):
    empty_count = 0
    X_non_empty = []
    for i in range(len(X)):
        if X[i] == "":
            empty_count += 1
        else:
            X_non_empty.append([X[i], y[i], z[i]])
    return X_non_empty, len(X) - empty_count, empty_count

# <OPEN DATASET>
X = []; y = []; z = []
with open ( 'final_steam_reviews.jl' , 'rb' ) as f:
    for item in json_lines.reader( f ):
        X.append( item['text'] )
        y.append( item['voted_up'] )
        z.append( item ['early_access'] )
# <\OPEN DATASET>

# <SIMPLE LANGUAGE DETECTION>
no_lang_count = 0
langs = []
lang_freq = []
for i in range(len(X)):
    try:
        lang = detect_langs(X[i])[0].lang
        if(lang in langs):
            lang_freq[langs.index(lang)] += 1
```

```
        else:
            langs.append(lang)
            lang_freq.append(1)
    except:
        no_lang_count += 1

for i in range(len(langs)):
    print(f"{langs[i]} - {lang_freq[i]}")

print(f"No language detected - {no_lang_count}")

# <\SIMPLE LANGUAGE DETECTION>

# <TRANSLATE ALL REVIEWS TO ENGLISH AND SAVE>

X_non_empty, non_empty_count, empty_count = find_non_empty(X, y, z)
print(f"Non-empty count: {non_empty_count}")
print(f"Empty count: {empty_count}")

translator = google_translator()
X_en = []

translated_texts = []
y_trans = []
z_trans = []

start_index = 0     # variable to allow for translation to be run in chunks
df1 = pd.read_csv('translated_reviews.csv', comment='#', columns=['index','rev
iews', 'upvoted', 'early_access'])
for i in range(len(X_non_empty) - start_index):
    curr_index = i + start_index
    translate_text = translator.translate(X_non_empty[curr_index][0],lang_tgt=
'en')
    translated_texts.append(translate_text)
    y_trans.append(y[curr_index])
    z_trans.append(z[curr_index])
    data = {'trans_reviews': translated_texts, 'upvoted': y_trans, 'early_acce
ss': z_trans}
    df = pd.DataFrame(data)
    df.to_csv('translated_reviews_all.csv', index=False, header=False)
    print(curr_index)

# <\TRANSLATE ALL REVIEWS TO ENGLISH AND SAVE>

# <FIND AND SAVE ALL NON-EMPTY ENGLISH REVIEWS>

X_non_empty, non_empty_count, empty_count = find_non_empty(X)
print(f"Non-empty count: {non_empty_count}")
```

```
print(f"Empty count: {empty_count}")

X_en, count, no_lang_count = find_english(X_non_empty)
print(f"English count: {count}")
print(f"No language count: {no_lang_count}")

df = pd.DataFrame(np.array(X_en), columns=['en_reviews', 'upvoted', 'early_acc
ess'])
df.to_csv('english_reviews.csv', index=False, header=False)

# <\FIND AND SAVE ALL NON-EMPTY ENGLISH REVIEWS>

# <REMOVE ALL NON-ASCII CHARS FROM ENGLISH REVIEWS AND SAVE>

df_col_names = ['review', 'upvote', 'early_access']
df = pd.read_csv('translated_reviews_all.csv', comment='#', names=df_col_names
)

for index, row in df.iterrows():
    row['review'] = row['review'].encode('ascii',errors='ignore').decode('asci
i')

df.to_csv('translated_reviews_all_ascii.csv', index=False, header=False)

# <\REMOVE ALL NON-ASCII CHARS FROM ENGLISH REVIEWS AND SAVE>
```

## Appendix 2 – Machine Learning Code

```
from sklearn.feature_extraction.text import CountVectorizer
import pandas as pd
import numpy as np
import random
import matplotlib.pyplot as plt
import statistics

from nltk import word_tokenize
from nltk.stem import WordNetLemmatizer
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import KFold
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.svm import LinearSVC
from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import roc_curve
from sklearn.metrics import roc_auc_score
from sklearn.metrics import accuracy_score, recall_score

class LemmaTokenizer:
    def __init__(self):
        self.wnl = WordNetLemmatizer()
    def __call__(self, doc):
        return [self.wnl.lemmatize(t) for t in word_tokenize(doc)]

def softmax(y):
    for i in range(len(y)):
        if y[i] > 0:
            y[i] = 1
        else:
            y[i] = -1
    return y

def confusion_matrix(y_pred, y):
    tp = tn = fp = fn = 0
    for idx, val in enumerate(y_pred):
        if val == -1 and y[idx] == -1:
            tn += 1
        elif val == 1 and y[idx] == 1:
            tp += 1
        elif val == -1 and y[idx] == 1:
            fn += 1
        elif val == 1 and y[idx] == -1:
            fp += 1
    return tp, tn, fp, fn
```

```
def accuracy_calc(conf_tuple):
    tp = conf_tuple[0]
    tn = conf_tuple[1]
    fp = conf_tuple[2]
    fn = conf_tuple[3]
    return (tp+tn)/(tp+tn+fp+fn)

def calc_shuffle_order(length):
    shuffle_order = np.arange(length)
    random.shuffle(shuffle_order)
    return shuffle_order

def shuffle2(X, y, shuffle_order):
    X_shuff = []
    y_shuff = []
    for i in shuffle_order:
        X_shuff.append(X[i])
        y_shuff.append(y[i])
    return np.array(X_shuff), np.array(y_shuff)

def unshuffle1(X_shuff, shuffle_order):
    X = []
    for i in range(len(shuffle_order)):
        X.append(X_shuff[np.where(shuffle_order == i)[0][0]])
    return np.array(X)


vect = TfidfVectorizer(tokenizer=LemmaTokenizer(), max_features=1500, stop_wor
ds='english')

df_col_names = ['text', 'upvote', 'early_access']
df = pd.read_csv('en_ascii_reviews.csv', comment='#', names=df_col_names)
# df = pd.read_csv('translated_reviews_all_ascii.csv', comment='#', names=df_c
ol_names)

transform = vect.fit_transform(df.text)
X_orig = np.array(transform.toarray())

y = []
for index, row in df.iterrows():
    if(row['upvote'] == True):
    # if(row['early_access'] == True):
        y.append(1)
    else:
        y.append(-1)
y_orig = np.array(y)

print(f"All Features - {vect.get_feature_names()}")
```

```
print(f"Num input features - {len(X_orig[0])}\nNum rows - {len(X_orig)}")
print(f"Num output rows - {len(y_orig)}")

# <BEST-FIT MODEL COMPARISONS>

lr_model = LogisticRegression(penalty='l2',C=1, max_iter=1000)
knn_model = KNeighborsClassifier(n_neighbors=4)
rf_model = RandomForestClassifier()

X_train, X_test, y_train, y_test = train_test_split(X_orig, y_orig, test_size=
0.2, shuffle=True)

lr_model.fit(X_train, y_train)
knn_model.fit(X_train, y_train)
rf_model.fit(X_train, y_train)

ns_probs = [0 for _ in range(len(y_test))]
lr_probs = lr_model.predict_proba(X_test)
lr_probs = lr_probs[:, 1]
knn_probs = knn_model.predict_proba(X_test)
knn_probs = knn_probs[:, 1]
rf_probs = rf_model.predict_proba(X_test)
rf_probs = rf_probs[:, 1]

ns_auc = roc_auc_score(y_test, ns_probs)
lr_auc = roc_auc_score(y_test, lr_probs)
knn_auc = roc_auc_score(y_test, knn_probs)
rf_auc = roc_auc_score(y_test, rf_probs)

print('No Skill: ROC AUC=%.3f' % (ns_auc))
print('4-NN: ROC AUC=%.3f' % (knn_auc))
print('Logistic Reg ROC AUC=%.3f' % (lr_auc))
print('Random Forest ROC AUC=%.3f' % (rf_auc))

ns_fpr, ns_tpr, _ = roc_curve(y_test, ns_probs)
knn_fpr, knn_tpr, _ = roc_curve(y_test, knn_probs)
lr_fpr, lr_tpr, _ = roc_curve(y_test, lr_probs)
rf_fpr, rf_tpr, _ = roc_curve(y_test, rf_probs)

train_acc_lr = accuracy_score(lr_model.predict(X_train), y_train)
test_acc_lr = accuracy_score(lr_model.predict(X_test), y_test)
train_acc_knn = accuracy_score(knn_model.predict(X_train), y_train)
test_acc_knn = accuracy_score(knn_model.predict(X_test), y_test)
train_acc_rf = accuracy_score(rf_model.predict(X_train), y_train)
test_acc_rf = accuracy_score(rf_model.predict(X_test), y_test)

print(f"Log Reg: training accuracy - {train_acc_lr}")
print(f"Log Reg: validation accuracy - {test_acc_lr}")
```

```
print(f"4NN: training accuracy - {train_acc_knn}")
print(f"4NN: validation accuracy - {test_acc_knn}")
print(f"Random forest: training accuracy - {train_acc_rf}")
print(f"Random forest: validation accuracy - {test_acc_rf}")

plt.plot(ns_fpr, ns_tpr, linestyle='--', label='No Skill')
plt.plot(lr_fpr, lr_tpr, marker='.', label='Log. Reg. C=1')
plt.plot(knn_fpr, knn_tpr, marker='x', label='4NN')
plt.plot(rf_fpr, rf_tpr, marker='o', label='Random Forest')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.legend()
plt.title('ROC curve comparison')
plt.show()

# <\BEST-FIT MODEL COMPARISONS>

# <LOGISTIC REGRESSION CROSS-VAL>

# costs = [0.1, 0.25, 0.5, 0.75, 1, 2, 5, 7.5, 10]
costs = [0.001, 0.01, 0.1, 1, 10, 100, 1000]

logit_shuff_train_metrics = []
logit_shuff_test_metrics = []
logit_shuff_train_variances = []
logit_shuff_test_variances = []
logit_unshuff_train_metrics = []
logit_unshuff_test_metrics = []
logit_unshuff_train_variances = []
logit_unshuff_test_variances = []

for cost in costs:
    for shuffle in [True]:
        if(not shuffle):
            print("UNSHUFFLED")
            X = X_orig
            y = y_orig
        else:
            X, y = shuffle2(X_orig, y_orig, calc_shuffle_order(len(X_orig)))
            print("SHUFFLED")

        kf = KFold(n_splits=5)
        temp_train_metrics = []
        temp_test_metrics = []
        for train, test in kf.split(X):
            logit_model = LogisticRegression(penalty='l2',C=cost, max_iter=100
0)
            logit_model.fit(X[train],y[train])
```

```
            y_hat_train = logit_model.predict(X[train])
            y_hat_test = logit_model.predict(X[test])

            y_hat_train_logit_acc = accuracy_score(y_hat_train, y[train])
            y_hat_test_logit_acc = accuracy_score(y_hat_test, y[test])

            y_hat_train_logit_rec = recall_score(y_hat_train, y[train])
            y_hat_test_logit_rec = recall_score(y_hat_test, y[test])

            temp_train_metrics.append([y_hat_train_logit_acc, y_hat_train_logi
t_rec])
            temp_test_metrics.append([y_hat_test_logit_acc, y_hat_test_logit_r
ec])


        logit_train_metrics = [np.mean([row[0] for row in temp_train_metrics])
, np.mean([row[1] for row in temp_train_metrics])]
        logit_test_metrics = [np.mean([row[0] for row in temp_test_metrics]),
np.mean([row[1] for row in temp_test_metrics])]
        logit_train_variance = [np.var([row[0] for row in temp_train_metrics])
, np.var([row[1] for row in temp_train_metrics])]
        logit_test_variance = [np.var([row[0] for row in temp_test_metrics]),
np.var([row[1] for row in temp_test_metrics])]

        # print(f"Some model coefs - {model.coef_[0]}")
        print("LOGISTIC REG")
        print(f"COST - {cost}")
        print(f"Training accuracy - {logit_train_metrics[0]} +/- {logit_train_
variance[0]}")
        print(f"Validation accuracy - {logit_test_metrics[0]} +/- {logit_test_
variance[0]}")
        print(f"Training recall - {logit_train_metrics[1]} +/- {logit_train_va
riance[1]}")
        print(f"Validation recall - {logit_test_metrics[1]} +/- {logit_test_va
riance[1]}\n")

        if(not shuffle):
            logit_unshuff_train_metrics.append(logit_train_metrics)
            logit_unshuff_test_metrics.append(logit_test_metrics)
            logit_unshuff_train_variances.append(logit_train_variance)
            logit_unshuff_test_variances.append(logit_test_variance)
        else:
            logit_shuff_train_metrics.append(logit_train_metrics)
            logit_shuff_test_metrics.append(logit_test_metrics)
            logit_shuff_train_variances.append(logit_train_variance)
            logit_shuff_test_variances.append(logit_test_variance)

plt.figure(1)
```

```
# plt.errorbar(costs, [row[0] for row in logit_unshuff_train_metrics], yerr=[r
ow[0] for row in logit_unshuff_train_variances], color='r', ecolor='k', label=
"Unshuffled - train")
plt.errorbar(costs, [row[0] for row in logit_shuff_train_metrics], yerr=[row[0
] for row in logit_shuff_train_variances], color='b', ecolor='k', label="Train
ing")
# plt.errorbar(costs, [row[0] for row in logit_unshuff_test_metrics], yerr=[ro
w[0] for row in logit_unshuff_test_variances], color='orange', ecolor='k', lab
el="Unshuffled - val")
plt.errorbar(costs, [row[0] for row in logit_shuff_test_metrics], yerr=[row[0]
 for row in logit_shuff_test_variances], color='c', ecolor='k', label="Validat
ion")
plt.legend()
plt.xlabel("Cost")
plt.ylabel("Accuracy")
plt.title("Logistic regression - Sentiment analysis accuracy")
plt.xscale("log")
plt.savefig('logistic_acc_cv_early_acc_all.png')


plt.figure(2)
# plt.errorbar(costs, [row[1] for row in logit_unshuff_train_metrics], yerr=[r
ow[1] for row in logit_unshuff_train_variances], color='r', ecolor='k', label=
"Unshuffled - train")
plt.errorbar(costs, [row[1] for row in logit_shuff_train_metrics], yerr=[row[1
] for row in logit_shuff_train_variances], color='b', ecolor='k', label="Train
ing")
# plt.errorbar(costs, [row[1] for row in logit_unshuff_test_metrics], yerr=[ro
w[1] for row in logit_unshuff_test_variances], color='orange', ecolor='k', lab
el="Unshuffled - val")
plt.errorbar(costs, [row[1] for row in logit_shuff_test_metrics], yerr=[row[1]
 for row in logit_shuff_test_variances], color='c', ecolor='k', label="Validat
ion")
plt.legend()
plt.xlabel("Cost")
plt.ylabel("Recall")
plt.title("Logistic regression - Sentiment analysis recall")
plt.xscale("log")
plt.savefig('logistic_rec_cv_early_acc_all.png')
plt.show()

# <\LOGISTIC REGRESSION CROSS-VAL>

# <KNN CROSS-VAL>

knn_unshuff_train_metrics = []
knn_unshuff_test_metrics = []
knn_unshuff_train_variances = []
knn_unshuff_test_variances = []
```

```
knn_shuff_train_metrics = []
knn_shuff_test_metrics = []
knn_shuff_train_variances = []
knn_shuff_test_variances = []

neighbours = [1,2,3,4,5,6,7,8,9]
for neighbour in neighbours:
    for shuffle in [True]:
        if(not shuffle):
            print("UNSHUFFLED")
            X = X_orig
            y = y_orig
        else:
            X, y = shuffle2(X_orig, y_orig, calc_shuffle_order(len(X_orig)))
            print("SHUFFLED")

        kf = KFold(n_splits=5)
        temp_train_metrics = []
        temp_test_metrics = []
        for train, test in kf.split(X):
            knn_model = KNeighborsClassifier(n_neighbors=neighbour)
            knn_model.fit(X[train],y[train])

            y_hat_train = knn_model.predict(X[train])
            y_hat_test = knn_model.predict(X[test])

            y_hat_train_logit_acc = accuracy_score(y_hat_train, y[train])
            y_hat_test_logit_acc = accuracy_score(y_hat_test, y[test])

            y_hat_train_logit_rec = recall_score(y_hat_train, y[train])
            y_hat_test_logit_rec = recall_score(y_hat_test, y[test])

            temp_train_metrics.append([y_hat_train_logit_acc, y_hat_train_logi
t_rec])
            temp_test_metrics.append([y_hat_test_logit_acc, y_hat_test_logit_r
ec])


        knn_train_metrics = [np.mean([row[0] for row in temp_train_metrics]),
np.mean([row[1] for row in temp_train_metrics])]
        knn_test_metrics = [np.mean([row[0] for row in temp_test_metrics]), np
.mean([row[1] for row in temp_test_metrics])]
        knn_train_variance = [np.var([row[0] for row in temp_train_metrics]),
np.var([row[1] for row in temp_train_metrics])]
        knn_test_variance = [np.var([row[0] for row in temp_test_metrics]), np
.var([row[1] for row in temp_test_metrics])]

        print("kNN")
        print(f"Neighbours - {neighbour}")
```

```
        print(f"Training accuracy - {knn_train_metrics[0]} +/- {knn_train_vari
ance[0]}")
        print(f"Validation accuracy - {knn_test_metrics[0]} +/- {knn_test_vari
ance[0]}")
        print(f"Training recall - {knn_train_metrics[1]} +/- {knn_train_varian
ce[1]}")
        print(f"Validation recall - {knn_test_metrics[1]} +/- {knn_test_varian
ce[1]}\n")

        if(not shuffle):
            knn_unshuff_train_metrics.append(knn_train_metrics)
            knn_unshuff_test_metrics.append(knn_test_metrics)
            knn_unshuff_train_variances.append(knn_train_variance)
            knn_unshuff_test_variances.append(knn_test_variance)
        else:
            knn_shuff_train_metrics.append(knn_train_metrics)
            knn_shuff_test_metrics.append(knn_test_metrics)
            knn_shuff_train_variances.append(knn_train_variance)
            knn_shuff_test_variances.append(knn_test_variance)

plt.figure(1)
# plt.errorbar(neighbours, [row[0] for row in knn_unshuff_train_metrics], yerr
=[row[0] for row in knn_unshuff_train_variances], color='r', ecolor='k', label
="Unshuffled - train")
plt.errorbar(neighbours, [row[0] for row in knn_shuff_train_metrics], yerr=[ro
w[0] for row in knn_shuff_train_variances], color='b', ecolor='k', label="Trai
ning")
# plt.errorbar(neighbours, [row[0] for row in knn_unshuff_test_metrics], yerr=
[row[0] for row in knn_unshuff_test_variances], color='orange', ecolor='k', la
bel="Unshuffled - val")
plt.errorbar(neighbours, [row[0] for row in knn_shuff_test_metrics], yerr=[row
[0] for row in knn_shuff_test_variances], color='c', ecolor='k', label="Valida
tion")
plt.legend()
plt.xlabel("Neighbours")
plt.ylabel("Accuracy")
plt.title("kNN - Sentiment analysis accuracy")
plt.savefig('knn_acc_cv_early_acc_all.png')


plt.figure(2)
# plt.errorbar(neighbours, [row[1] for row in knn_unshuff_train_metrics], yerr
=[row[1] for row in knn_unshuff_train_variances], color='r', ecolor='k', label
="Unshuffled - train")
plt.errorbar(neighbours, [row[1] for row in knn_shuff_train_metrics], yerr=[ro
w[1] for row in knn_shuff_train_variances], color='b', ecolor='k', label="Trai
ning")
```

```
# plt.errorbar(neighbours, [row[1] for row in knn_unshuff_test_metrics], yerr=
[row[1] for row in knn_unshuff_test_variances], color='orange', ecolor='k', la
bel="Unshuffled - val")
plt.errorbar(neighbours, [row[1] for row in knn_shuff_test_metrics], yerr=[row
[1] for row in knn_shuff_test_variances], color='c', ecolor='k', label="Valida
tion")
plt.legend()
plt.xlabel("Neighbours")
plt.ylabel("Recall")
plt.title("kNN - Sentiment analysis recall")
plt.savefig('knn_rec_cv_early_acc_all.png')
plt.show()

# <\KNN CROSS-VAL>

# <RANDOM FOREST K-FOLD ANALYSIS>

X, y = shuffle2(X_orig, y_orig, calc_shuffle_order(len(X_orig)))
print("SHUFFLED")

kf = KFold(n_splits=5)
temp_train_metrics = []
temp_test_metrics = []
for train, test in kf.split(X):
    rf_model = RandomForestClassifier(max_depth=None, random_state=0)
    rf_model.fit(X[train],y[train])

    y_hat_train = rf_model.predict(X[train])
    y_hat_test = rf_model.predict(X[test])

    y_hat_train_rf_acc = accuracy_score(y_hat_train, y[train])
    y_hat_test_rf_acc = accuracy_score(y_hat_test, y[test])

    y_hat_train_rf_rec = recall_score(y_hat_train, y[train])
    y_hat_test_rf_rec = recall_score(y_hat_test, y[test])

    temp_train_metrics.append([y_hat_train_rf_acc, y_hat_train_rf_rec])
    temp_test_metrics.append([y_hat_test_rf_acc, y_hat_test_rf_rec])


rf_train_metrics = [np.mean([row[0] for row in temp_train_metrics]), np.mean([
row[1] for row in temp_train_metrics])]
rf_test_metrics = [np.mean([row[0] for row in temp_test_metrics]), np.mean([ro
w[1] for row in temp_test_metrics])]
rf_train_variance = [np.var([row[0] for row in temp_train_metrics]), np.var([r
ow[1] for row in temp_train_metrics])]
rf_test_variance = [np.var([row[0] for row in temp_test_metrics]), np.var([row
[1] for row in temp_test_metrics])]
```

```
print("RANDOM FOREST")
print(f"Training accuracy - {rf_train_metrics[0]} +/- {rf_train_variance[0]}")
print(f"Validation accuracy - {rf_test_metrics[0]} +/- {rf_test_variance[0]}")
print(f"Training recall - {rf_train_metrics[1]} +/- {rf_train_variance[1]}")
print(f"Validation recall - {rf_test_metrics[1]} +/- {rf_test_variance[1]}\n")

# <\RANDOM FOREST K-FOLD ANALYSIS>

# <BASELINE K-FOLD ANALYSIS>

X, y = shuffle2(X_orig, y_orig, calc_shuffle_order(len(X_orig)))
print("UNSHUFFLED")
kf = KFold(n_splits=5)
temp_train_metrics = []
temp_test_metrics = []
for train, test in kf.split(y):
    train_common_class = statistics.mode(y[train])

    y_hat_train = np.ones(len(y[train]), dtype=np.int32) * train_common_class
    y_hat_test = np.ones(len(y[test]), dtype=np.int32) * train_common_class

    y_hat_train_baseline_acc = accuracy_score(y[train], y_hat_train)
    y_hat_test_baseline_acc = accuracy_score(y[test], y_hat_test)

    y_hat_train_baseline_rec = recall_score(y[train], y_hat_train)
    y_hat_test_baseline_rec = recall_score(y[test], y_hat_test)

    temp_train_metrics.append([y_hat_train_baseline_acc, y_hat_train_baseline_
rec])
    temp_test_metrics.append([y_hat_test_baseline_acc, y_hat_test_baseline_rec
])

baseline_train_metrics = [np.mean([row[0] for row in temp_train_metrics]), np.
mean([row[1] for row in temp_train_metrics])]
baseline_test_metrics = [np.mean([row[0] for row in temp_test_metrics]), np.me
an([row[1] for row in temp_test_metrics])]
baseline_train_variance = [np.var([row[0] for row in temp_train_metrics]), np.
var([row[1] for row in temp_train_metrics])]
baseline_test_variance = [np.var([row[0] for row in temp_test_metrics]), np.va
r([row[1] for row in temp_test_metrics])]
print("BASELINE")
print(f"Training accuracy - {baseline_train_metrics[0]} +/- {baseline_train_va
riance[0]}")
print(f"Validation accuracy - {baseline_test_metrics[0]} +/- {baseline_test_va
riance[0]}")
print(f"Training recall - {baseline_train_metrics[1]} +/- {baseline_train_vari
ance[1]}")
```

```
print(f"Validation recall - {baseline_test_metrics[1]} +/- {baseline_test_vari
ance[1]}\n")

# <\BASELINE K-FOLD ANALYSIS>
```