Q1
The data was first processed to remove as many non-English reviews as possible, this was done for convenience purposes, as there was a lot of gibberish on top of there being different languages. The benefit of only having English is that it allows me to know if the other processing methods are having the desired effect.

The tokenization, removal of stop words and the adjustment of max_dand min_df are done in the same loop. Tokenization makes each word in the review its own feature. Stop words are any word that has no significant use in terms of sentiment analysis, only the english stop words are removed, as we no longer need to consider other languages.

Min_df has a default value of 1, this means that it will remove words that appear in less than one document.
Max_df has a default value of 1.0, meaning that it will remove words that appear in more than 100% of documents. In both cases, the default value does not remove any words from the documents. We will be choosing the highest performing values through cross-validation.
To measure the success of our model, we will be using the ROC (Receiver Operating Characteristics) and AUC (Area Under the Curve). The calculations of ROC compare the true positives rates to false negatives rates, where they are on the Y and X axis respectively.

AUC is used to measure the ability for the model to separate good reviews from bad reviews. And AUC of 0.5 means that it has no ability to separate the two from another, this would be a kin to blindly selecting an answer. A value of 1 would be a perfect score, and 0 would be getting the separations wrong every time.

Logistic regression

For our classification model, we will be using a Logistic regression with an L1 penalty, this is also known as Lasso Regression. The cost function on the Lasso regression reduces a feature's coefficient to 0, this is of benefit particularly for datasets with a large amount of features, which is the case for us.
Now to choose our highest performing value for max_df. The value for min_df is set to default. The mean value of the 3 runs for those specific values AUC's are calculated in the table below. The reason I am able to get meaningful results with each run, is because on the train_test_split function random_state is set to active, so every run mixes the data randomly instead of working on the same data.

Mean calculation of max_df

runs/ values	max_df= 1.0	max_df= 0.9	max_df= 0.8	max_df= 0.7	max_df= 0.6	max_df= 0.5	max_df= 0.4	max_df= 0.3	max_df= 0.2	max_df= 0.1
1st	0.753	0.739	0.734	0.801	0.783	0.766	0.742	0.742	0.784	0.755
2nd	0.819	0.801	0.832	0.852	0.800	0.803	0.834	0.843	0.831	0.817
3rd	0.797	0.866	0.812	0.845	0.833	0.821	0.810	0.809	0.812	0.837
mean	**0.789**	**0.802**	**0.792**	**0.832**	**0.805**	**0.796**	**0.795**	**0.798**	**0.809**	**0.803**

From the above calculations we can see that the highest performing value for max_df is 0.7, with a performance of 83.2% accuracy.
Now repeating the same process for min_df, while having max_df set to the default.

Mean calculation of min_df

Runs	Min_ df= 0.01	min_ df= 0.02	min_ df= 0.04	min_ df= 0.08	min_ df= 0.16	min_ df= 0.32	min_ df= 0.64	min_ df= 0.80	min_ df= 0.90	min_ df= 0.99
1st	0.762	0.767	0.705	0.636	0.568	0.563	ERR	ERR	ERR	ERR
2nd	0.765	0.739	0.718	0.621	0.543	0.496	ERR	ERR	ERR	ERR
3rd	0.759	0.724	0.697	0.636	0.551	0.545	ERR	ERR	ERR	ERR
mean	**0.762**	**0.743**	**0.706**	**0.631**	**0.554**	**0.534**	**ERR**	**ERR**	**ERR**	**ERR**

For some reason my for loop could not iterate through the last 4 values due to a difficult to fix error, but nonetheless it does not negatively impact our judgment because we can already see a trend that the lower the value for min_df leads to the best performance, it's time to combine the two optimal values for min_df and max_df to see how the model performs.

The result of the 3 iterations are as follows:

AUC = 0.788, AUC = 0.755, AUC = 0.771 ; **mean AUC = 0.771**

This implies that the default value for min_df works best as that was the one value that I didn't test for, changing it back to the default gives us the following results:

AUC = 0.834, AUC = 0.817, AUC = 0.828 ; **mean AUC = 0.826.**

As expected, the accuracy score is much higher, at 82.6%. This is due to the Lasso regressions cost model, useless features get removed anyway so there is no need for me to remove them myself. That concludes testing for whether a review was positive or not using Lasso Regression. Now we will use LR to try to predict whether a review was for an early_access game or not.

To achieve this in the code, the target variable is changed from df_eng['voted_up'] to df_eng['early_access']. Keeping the same optimal values for max_df and min_df (0.7 and 'default' respectively) the results for predicting whether a review is for early access or not are as follows:

AUC = 0.578, AUC = 0.605, AUC = 0.595 ; **mean AUC = 0.593**

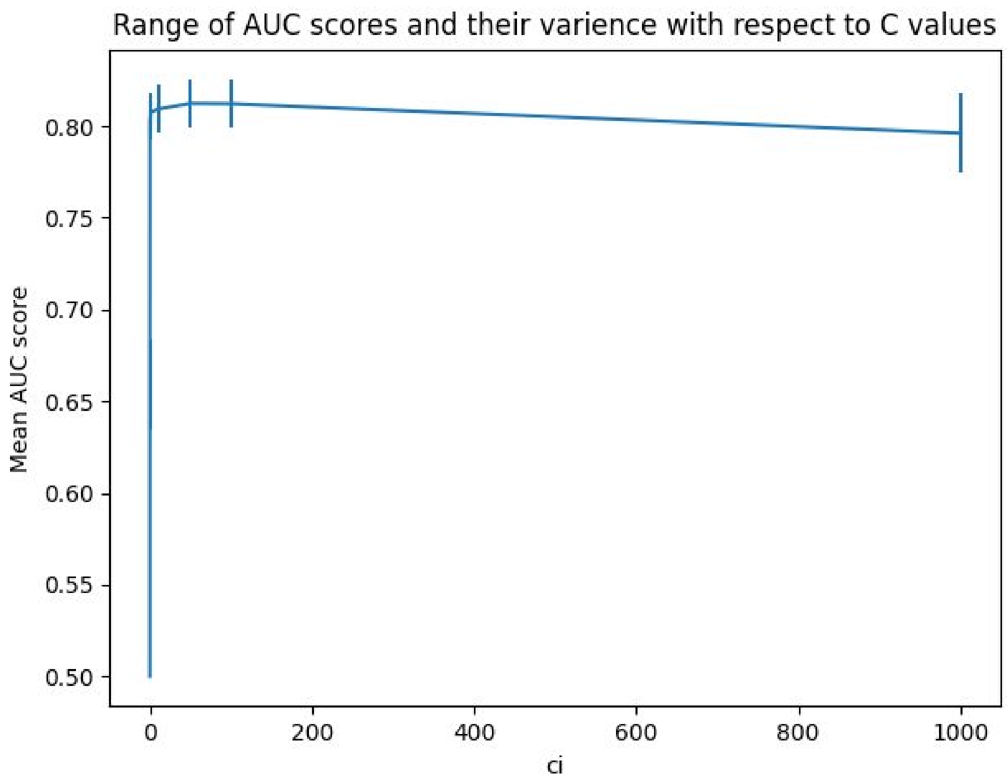
With an accuracy of only 59.3%, we can see that the model struggles to classify early access status, as this must be much harder than classifying sentiment.

The C hyperparameter was also tuned using cross validation. This is the adjustment of the penalty value of the Lasso regression. Here is the range of C values:
C Range = [0.001, 0.01, 0.1, 1.0, 10.0, 50.0, 100.0, 1000.0]
Code Snippet

```
for ci in ci_range: #iterate through the min and max values by turn, manually
    tokenizer = WhitespaceTokenizer().tokenize
    vectorizer = TfidfVectorizer(tokenizer=tokenizer,
stop_words=nltk.corpus.stopwords.words('english'), max_df = 0.7, min_df = 1,
norm=None)
    X_fit = vectorizer.fit_transform(df_eng['text'])
    df_fit =
pd.DataFrame(X_fit.toarray(),columns=vectorizer.get_feature_names())

    temp = []

    for i in range(5):
        X_train, X_test, y_train, y_test = train_test_split(df_fit, target,
test_size=0.2, random_state=i )

        # Run logistic regression
        logisticReg = LogisticRegression(C = ci, penalty= 'l1',
solver='liblinear')
        logisticReg.fit(X_train, y_train)
```

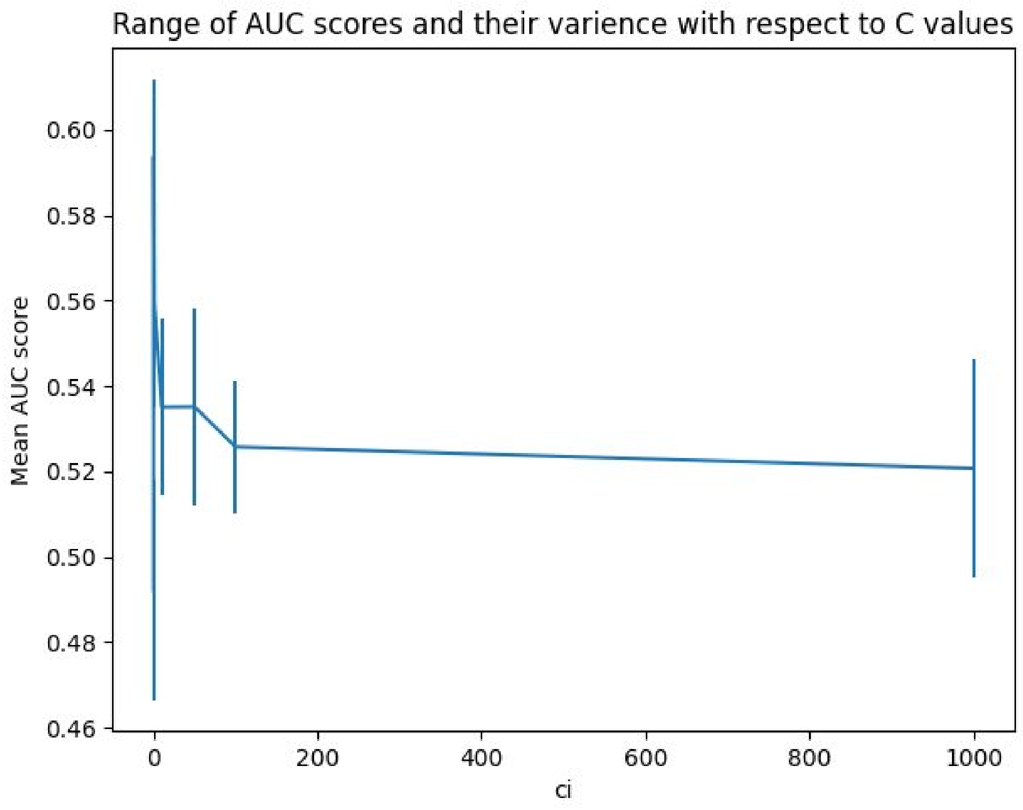
If you look at the above code, you will see that I approached cross validation by iterating through different train_test_splits as the index i passed into the “random state” acts as a seed for randomly splitting the data, allowing for meaningful data to be returned through fitting the logistic regression model.

For Predicting Sentiment



From the above graph we can see that C = 100, 50 and 10 are all more or less the same in terms of performance and would be preferable as they are the highest in AUC scores with more or less the same variance. Personally, I will choose C = 50

For Predicting Early Access



Above, we can see that the model still struggles to get good predictions for early access classification. The lower values of C all provide too much variance. Whereas, again, the C values of 10, 50, and 100 might be the good choices but I choose C = 50 due to its variants achieving the highest AUC scores.

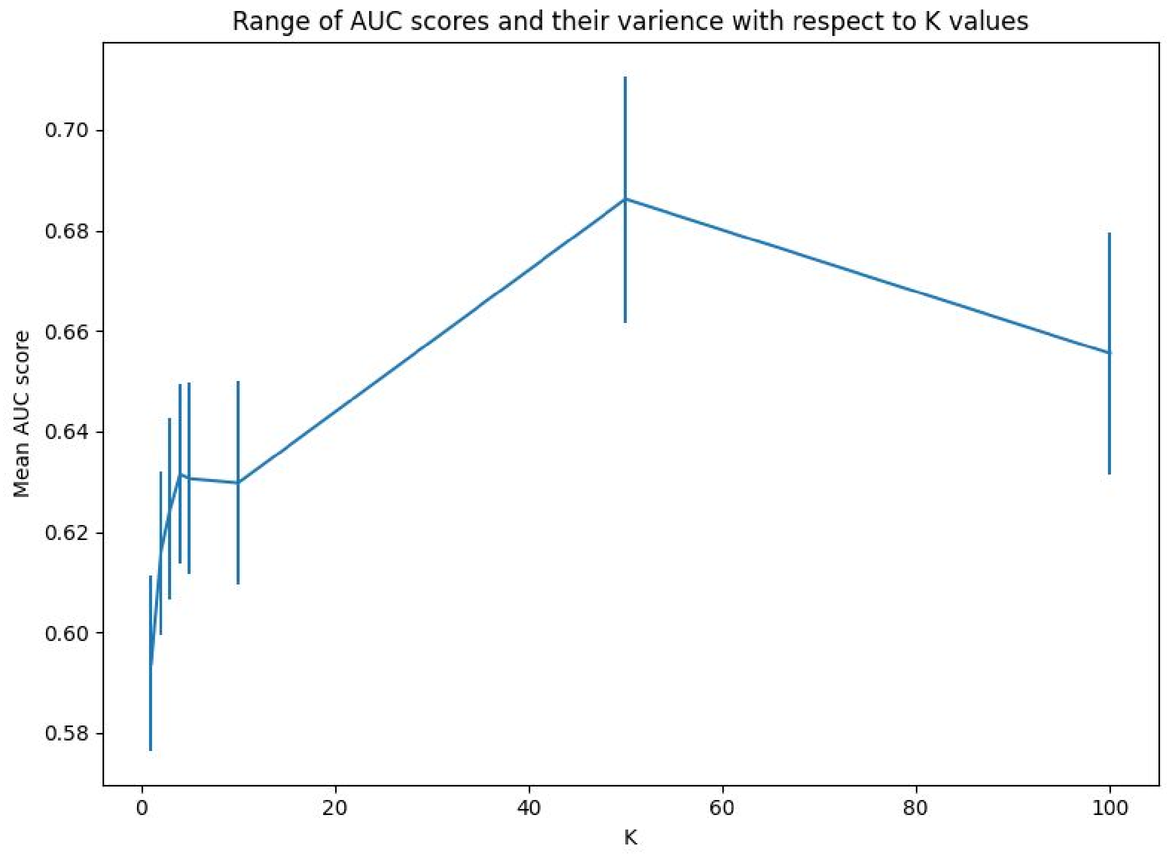
K Nearest Neighbours

For my next model I decided to use the K Nearest Neighbor classification (KNN). I kept the same min and max_df values as the last model.

KNN is an instance based model where it does not have a loss function that can be minimised during training. KNN is a non-parametric method of classification and regression, so we cannot get those useful metrics out of our model once it is deployed. We will be using the default metric, minkowski with p=2, which is the equivalent to the standard Euclidean metric.

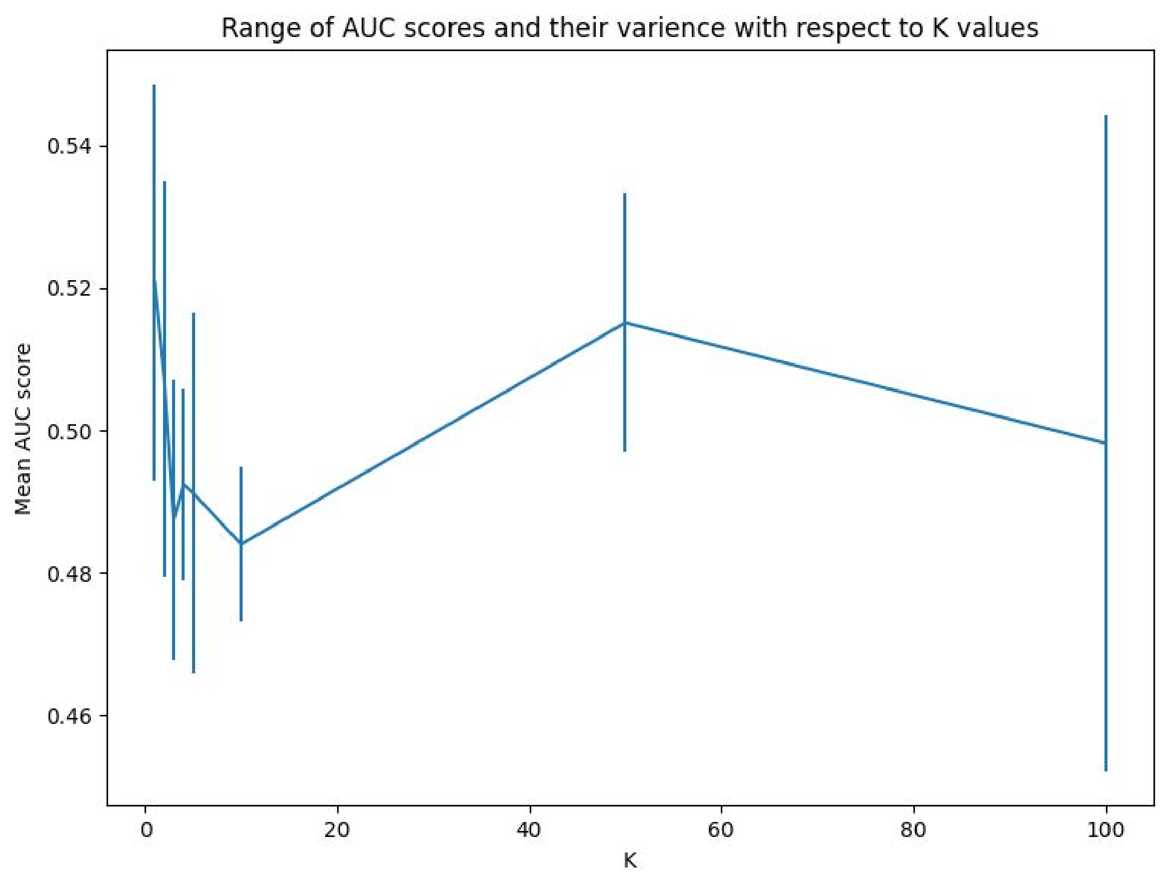
The hyperparameter K in the KNN algorithm will be tuned via cross-validation, looping through a similar setup to the above snippet of code, these are the values of K that will be fitted:
neighbours = [1, 2, 3, 4, 5, 10, 50, 100]
The results of which are displayed below

<u>**For Predicting Sentiment**</u>



When comparing the above results of the sentiment analysis to the results of the Logistic regression's sentiment analysis, they are not as good, the best K value performance here is by K=50, that gives us a potential peak of around 70% accuracy, which is not great. Perhaps the Logistic regression's abilities to reduce coefficients down to zero is superior to KNN's distance and neighbours algorithm.

For Predicting Early Access

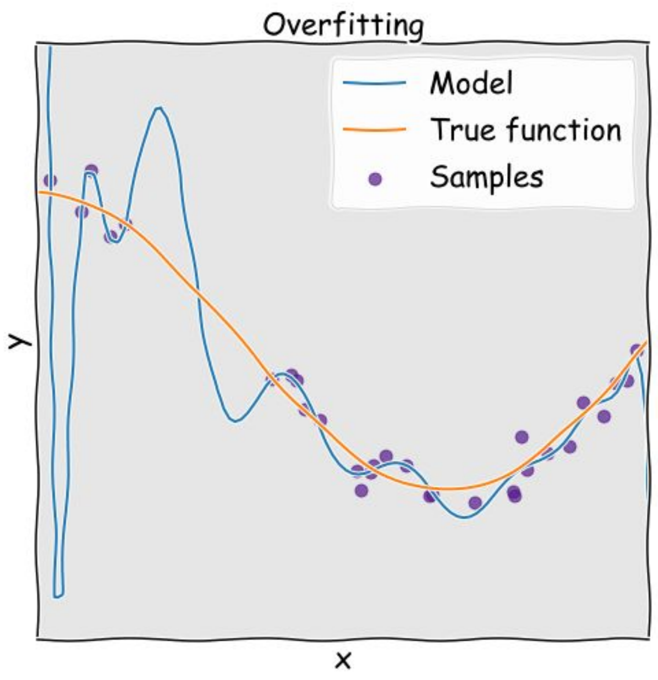


Again, from the above graph we see disappointing AUC scores, with many of the earlier values struggling to predict past the ability of random choice to predict.
K=10 is actually returning good results if my goal was the opposite, having a tight variance and good prediction if you only compare it with the other K values. But for optimal performance I would choose K=50, the same choice as the sentiment analysis K value.

In conclusion, the Logistic Regression performs better than the KNN classifier. From what I can gather it is the result of the superior method of the LR with l1 penalty, reducing the coefficients down to zero.

Q2(i)
While adding more features does improve the fit of a model, adding too many leads to over-fitting and having too few features leads to under-fitting.

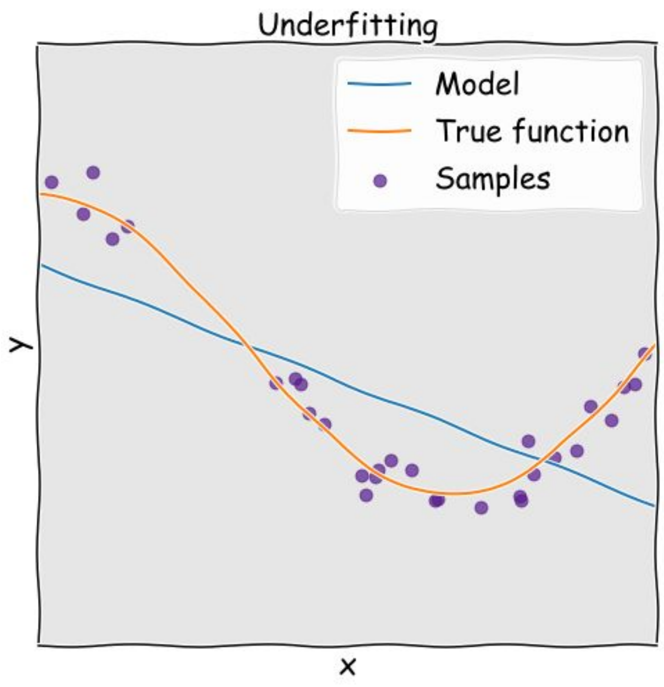
The appearance of a model to be under or over-fitted onto the data is up to the eye of the beholder but people usually agree with each other when describing something as over or under-fitted. As more parameters are added, data that we perceive as being noise becomes over represented by the model which would lead to inaccurate predictions. An example of which is best represented by a diagram:



As you can see, the hypothetical “true function” exists and data was gathered from something that closely resembles the true function but the data is noisy. If we add too many parameters to our hypothetical model, we see the model overfitting. We know it is overfitting because an ideal fir would be very close to the true function.

Having more data doesn’t improve the model’s ability to generalise, it instead improves the performance only in the region where you have data.

A model is considered to be underfitted if the fit of the model looks like a line. The model no longer is able to capture genuine characteristics of the data and therefore it is not only bad at generalising past the data but it also performs poorly on the training data as well. As seen in the example diagram below:



Having more data has no effect on the models ability to generalise.

[Diagrams by Keith T. Butler : https://keeeto.github.io/blog/bias_variance/]

(ii)

Randomly shuffle data (you may not want to, it’s a choice)
Divide the data into equal parts (in this case 10)

```
for i = 1 : 10
        i is set as the test set
        The rest (k-1) is set to training
        Fit the model onto the training set and evaluate it on the test set
        Calculate the prediction error
End for loop
Compute average accuracy of the k amount of runs
Show error rate of each iteration
```

(iii)

When running the K-Fold cross validation, it is useful to have an array of hyperparameter values that you would wish to test. You can loop through each of these values in an outer loop and on the inner loop you would train the model using each of those splits for that one particular value of hyperparameter. You can then append all of those mean square error metrics of each split into a temp array and get the mean of those values into a separate array, giving you a fairly unbiased metric of mean squared error for each value of hyperparameter.

You would also calculate the standard error, which shows you the standard deviation of each of your measurements in the temp array, which is what you will be using when choosing the value of your hyperparameter.

Graphing out the mean squared error and standard error together against the values of the hyperparameters shows you which hyperparameter to use, using the simplest model possible means that you are not overfitting the data, this means you should use the smallest value of hyperparameter without trading too much standard deviation in the mean squared errors of your model. So, to simplify that statement, once standard error stops dropping significantly, use that value of hyperparameter.

(iv)

For this answer I will be giving three pros and cons of Logistic regression and then comparing those directly to KNN within the same bullet point.

Pros:

- Logistic regression has a convex loss function, it uses a logistic loss function, this means that we can minimise this by reaching the global minimum vertex and complete our optimisation. Here is the formula for this function below, notice that if y = 1 and the prediction is 0, the cost is extremely high, whereas if the prediction is 1 then there is no punishment. Vice versa for y=0.
  $Cost(h\theta(x), y) \ = \ -ylog(h\theta(x)) \ - \ (1-y)log(h\theta(x))$
  And on the other hand, KNN does not have a loss function that can be minimised during training.

- Logistic is easy, fast and simple to implement, it does not have any hyperparameters to tune, which makes training the model much easier and less time consuming. Whereas in KNN, we need to define what is the notion of “near” for our neighbours in general, in the Scikit-Learn library there is a parameter p that needs to be chosen in order to specify what distance metric to use, by default it is Euclidean, or p=2. We also need to specify what value of k to use, this needs to be chosen via cross-validation
- θ parameters in Logistic regression help explain the direction (whether positive or negative) and how significance of independent variable over dependant variables. In contrast, KNN is a non-parametric method of classification and regression, so we cannot get those useful metrics out of our model once it is deployed.

<u>Cons:</u>

- Logistic regression cannot be applied to non-linear classification problems, for example instead of having a line on the decision boundary, there are times when the decision boundary is a circle and LR cannot handle that. Whereas KNN supports non-linear classification problems.
- Both Logistic Regression and KNN suffer in the presence of collinearity and outliers in their data, these need to be removed in order for us to get useful reading from the coefficients in the LR model. And in KNN collinear variables carry extra weight on the distance function than is desired.
- Proper feature selection is needed for both Logistic Regression and KNN. Features that are not predictive will make the prediction worse in both cases, unlike in Neural networks which, the model just learns to ignore them.

(v)

1. KNN cannot extrapolate on the data provided, It can only make classifications on the data available, that is because KNN is an exception to the general workflow of machine learning algorithms, where it only does template matching and interpolation. Because of this, the only way to test a KNN model is if you already know the target values and to keep them secret from the model so that you can compare them together.
   Example: trying to predict past the data
2. KNN cannot give predictions on big data as the cost of computation is too high, it only works on small to medium sized data, so modern machines would not be able to compute a prediction within a reasonable amount of time past, say, around 50,000 data points. The reason for this is that KNN needs to compute the distance of one point with the rest of the data points, and so on for the next and the next until the last data point is taken into consideration.

Appendix:

```
import os
import requests
import json_lines
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from googletrans import Translator
from langdetect import detect
from sklearn import metrics
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.linear_model import LogisticRegression
from nltk.tokenize import WhitespaceTokenizer, word_tokenize
from nltk.stem.porter import PorterStemmer
from sklearn.model_selection import train_test_split, KFold
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import mean_squared_error, confusion_matrix,
classification_report, f1_score, roc_curve, auc
from sklearn.neighbors import KNeighborsClassifier

import nltk
nltk.download('punkt')
nltk.download('stopwords')

X=[]; y=[]; z=[]
with open ("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"])
fields = ['text', 'voted_up', 'early_access']
obj = {}
obj[fields[0]] = X
obj[fields[1]] = y
obj[fields[2]] = z

df =
pd.DataFrame(columns=['text','text_translate','voted_up','early_access'])
df['text'] = pd.Series(X)
df['voted_up'] = pd.Series(y)
df['early_access'] = pd.Series(z)

#remove non-english
i = 0
X_ENG = {}
y_ENG = {}
```

```
z_ENG = {}
Obj_ENG = {}
for idx, text in enumerate(X):
    try:
        det = detect(text)
        if det == 'en':
            X_ENG[idx] = text
            y_ENG[idx] = y[idx]
            z_ENG[idx] = z[idx]
    except:
        print('IS GIBBERISH')
Obj_ENG[fields[0]] = X_ENG
Obj_ENG[fields[1]] = y_ENG
Obj_ENG[fields[2]] = z_ENG

df_eng = pd.DataFrame(columns=['text','voted_up','early_access'])
df_eng['text'] = pd.Series(X_ENG)
df_eng['voted_up'] = pd.Series(y_ENG)
df_eng['early_access'] = pd.Series(z_ENG)

# GET TARGET
target = df_eng['early_access']
print(target)
print(df_eng['text'])

#Logisteic Regression part in Q1

max_df_values = [1.0, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1]
# max_df default is 1.0
# so will remove words that appear in more than 100% of docs
min_df_values = [0.01, 0.02, 0.04, 0.08, 0.16, 0.32, 0.64, 0.80, 0.90, 0.99]
# min_df, default 1, will remove words that appear in less than 1 doc
# in both cases default removes none
#iterate through the min and max values by turn, manually

ci_range= [0.001, 0.01, 0.1, 1.0, 10.0, 50.0, 100.0, 1000.0]
mean_range = []
std_range = []

for ci in ci_range: #iterate through the min and max values by turn, manually
    tokenizer = WhitespaceTokenizer().tokenize
    vectorizer = TfidfVectorizer(tokenizer=tokenizer,
stop_words=nltk.corpus.stopwords.words('english'), max_df = 0.7, min_df = 1,
norm=None)
    X_fit = vectorizer.fit_transform(df_eng['text'])
    df_fit =
pd.DataFrame(X_fit.toarray(),columns=vectorizer.get_feature_names())
```

```
    temp = []

    for i in range(5):
        X_train, X_test, y_train, y_test = train_test_split(df_fit, target, 
test_size=0.2, random_state=i )

        # Run logistic regression
        logisticReg = LogisticRegression(C = ci, penalty= 'l1', 
solver='liblinear')
        logisticReg.fit(X_train, y_train)

        predictions = logisticReg.predict(X_test)
        #print(confusion_matrix(y_test, predictions))
        #print(classification_report(y_test, predictions))

        scores = logisticReg.predict_proba(X_test)
        fpr, tpr, _= roc_curve(y_test, scores[:, 1])
        print('AUC = {}'.format(auc(fpr, tpr)))
        temp.append(format(auc(fpr, tpr)))
    mean_range.append(np.array(temp).astype(np.float).mean())
    std_range.append(np.array(temp).astype(np.float).std())
plt.errorbar(ci_range, mean_range, yerr=std_range)
plt.xlabel('ci'); plt.ylabel('Mean AUC score')
plt.title('Range of AUC scores and their varience with respect to C values')
plt.show()


#Q1 KNN part
neighbours = [1, 2, 3, 4, 5, 10, 50, 100]
for neigh in neighbours: #iterate through the K value in KNN
    tokenizer = WhitespaceTokenizer().tokenize
    vectorizer = TfidfVectorizer(tokenizer=tokenizer, 
stop_words=nltk.corpus.stopwords.words('english'), max_df = 0.7, min_df = 1, 
norm=None)
    X_fit = vectorizer.fit_transform(df_eng['text'])
    df_fit = 
pd.DataFrame(X_fit.toarray(),columns=vectorizer.get_feature_names())

    temp = []

    for i in range(5):
        X_train, X_test, y_train, y_test = train_test_split(df_fit, target, 
test_size=0.2, random_state=i )

        # Run logistic regression
        model = KNeighborsClassifier(n_neighbors= neigh, weights= 'uniform')
        model.fit(X_train, y_train)
        predictions = model.predict(X_test)
        #print(confusion_matrix(y_test, predictions))
        #print(classification_report(y_test, predictions))

        scores = model.predict_proba(X_test)
        fpr, tpr, _= roc_curve(y_test, scores[:, 1])
        print('AUC = {}'.format(auc(fpr, tpr)))
        temp.append(format(auc(fpr, tpr)))
    mean_range.append(np.array(temp).astype(np.float).mean())
    std_range.append(np.array(temp).astype(np.float).std())
plt.errorbar(neighbours, mean_range, yerr=std_range)
plt.xlabel('K'); plt.ylabel('Mean AUC score')
plt.title('Range of AUC scores and their varience with respect to K values')
plt.show()
```