CS4061: Final Assignment

Conor McCauley - 17323203

January 23, 2021

Question 1

Pre-processing Data

Language Detection and Translation

Upon looking at the dataset it became clear that a large portion of the reviews were not in English. I used the `spaCy` library to try to detect the language of each of the reviews:

```
nlp = spacy.load('en_core_web_sm')
nlp.add_pipe(LanguageDetector(), name='ld')
# for each review
nlp(review)._.language['language']
```

The results from the above code suggested that less than 40% of the reviews were in English, 17% were in Russian, 5% were in Turkish, etc. Due to these results it seemed that it would be worth translating all of the reviews into English before processing the data in order to build a better predictive model. I used the `google_trans_new` library to communicate with *Google Translate*'s API and translate each of the reviews:

```
trans = google_translator()
# for each review
trans.translate(review, lang_tgt='en')
```

Both the original and translated datasets can be found in Appendix B.

TF-IDF and Lemmatisation

The next step was to apply pre-processing to the dataset. I chose to use the TF-IDF method on the dataset and used the `TfidfVectorizer` function to handle both vectorisation and transformation at once.

Due to the fact that processing the dataset and training models took less than a second or two to complete I was able to quickly test a number of vectorisation and transformation parameters on a number of machine learning models before moving on to proper cross-validation. Using the default settings for a multinomial Naive Bayes classifier, a random forest classifier, a linear SVC classifier, a KNN classifier and an SGD classifier I discovered that, in every case, using an n-gram range of $(1-2)$, sub-linear term-frequency scaling and stripping character accents resulted in predictions that were more accurate by $1-3\%$, regardless of whether or not the reviews were translated.

I also experimented with using the raw data, using stop-words and using a lemmatiser and discovered that, in every case, a lemmatiser resulted in predictions that were much more accurate. I used the `WordNetLemmatizer` from the `nltk` library to carry out the lemmatisation process (the code was inspired by this *Stack Overflow* post).

This testing was done for both (i) the review polarity, and (ii) whether or not a game was 'early access' and the increased predictive accuracy was always found to occur for (i) but the accuracy of (ii) rarely changed by much - this effect will be discussed in detail later on.

All of the pre-processing was done using the following code:

```
X = TfidfVectorizer(
    sublinear_tf=True,
    strip_accents='unicode',
    tokenizer=Lemmatizer(),
    ngram_range=(1, 2)
).fit_transform(X)
```

(i) Review Polarity

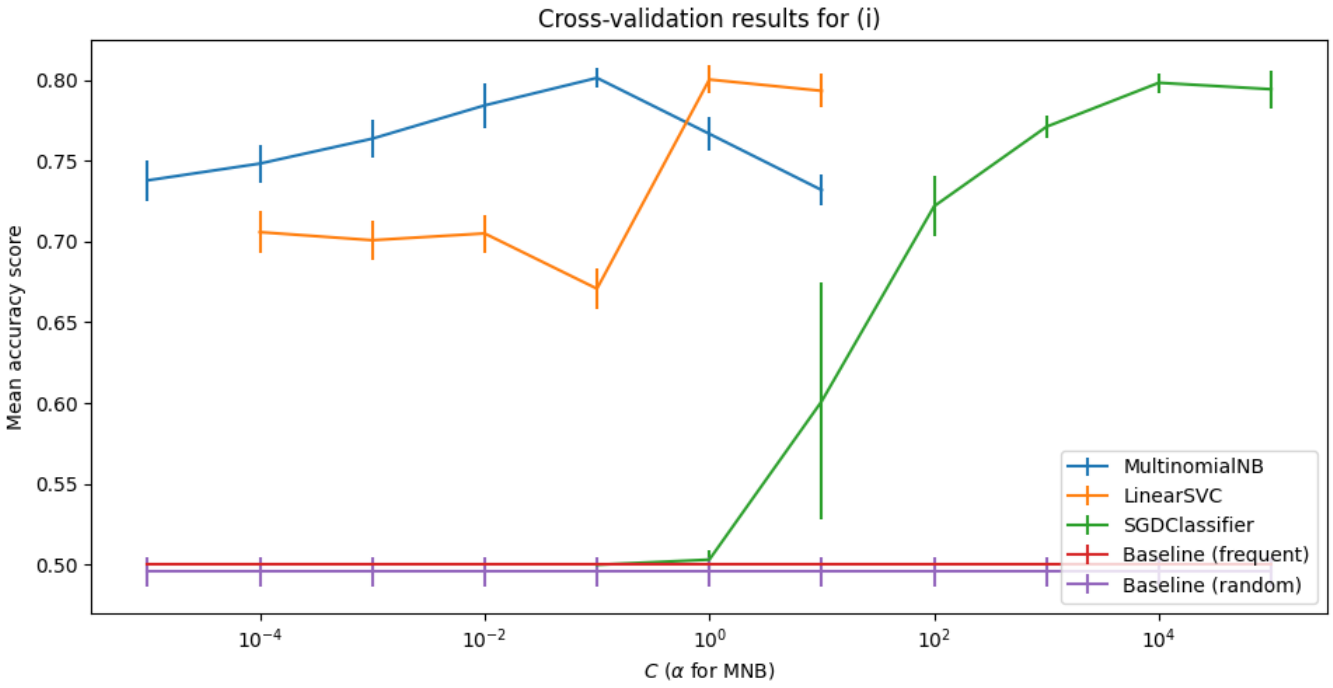
Cross-validation

Having done some minor testing while choosing TF-IDF parameters I discovered that both the random forest and KNN classifiers were always outperformed by the other three classifiers that were tested: the multinomial Naive Bayes classifier, the linear SVC classifier and the SGD classifier. As such, I chose to perform full cross-validation on just those three classifiers.

I also considered performing cross-validation on an MLP classifier, however, given that it took over 45 minutes to train a single model and produced results that were actually less accurate than the other classifiers, I chose to abandon that approach altogether.

5-fold cross-validation was used to measure the predictive accuracy of the trained models while two baseline classifiers were used to compare the relative accuracy of the trained models: one that always predicted the most frequent class and one that predicted a random class. For the Naive Bayes classifier I varied the α parameter and for the other two classifiers I varied the C parameter (for SGD the given value was actually $\alpha = 1/C$).

The models were evaluated using their mean accuracy score and were tested on a very wide range of parameter values (every power of 10 from 10^{-5} to 10^{5}). The mean scores as well as the standard deviations are plotted below (note: only the relevant parameter ranges are plotted for each model in order to prevent the graph from getting cluttered):



From the above graph it is quite clear that each of the models vastly outperform both baselines and all have a single parameter value that is clearly the optimal choice:

- the NB classifier has an accuracy of 80.1% with an SD of 0.6% when $\alpha = 0.1$
- the SVC classifier has an accuracy of 80.0% with an SD of 0.9% when $C = 1$
- the SGD classifier has an accuracy of 79.7% with an SD of 0.4% when $C = 10000$

Given each of these high results any of the classifiers could be chosen, however, given the slightly higher accuracy and very low standard deviation of the Naive Bayes classifier I ultimately decided to select it as the optimal model for evaluation.

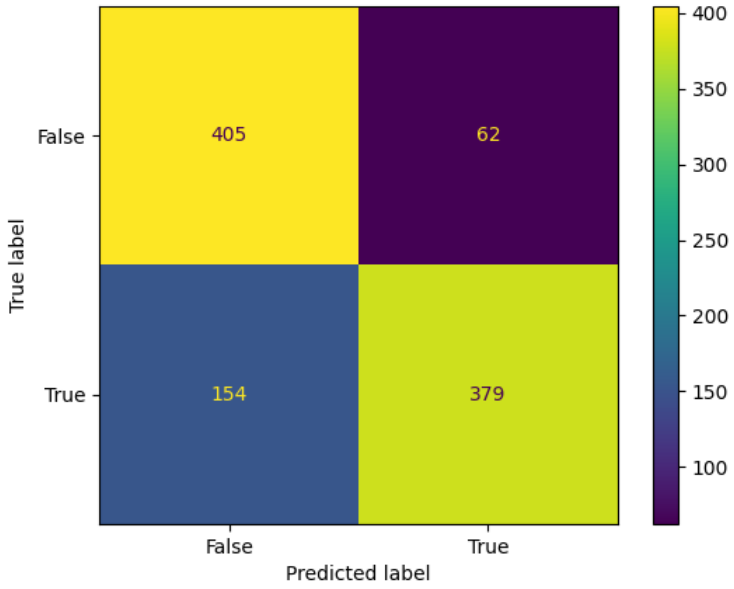
Model Evaluation

The trained model's classification report indicated an accuracy of 82% for the test data and an accuracy of $98 - 99\%$ for the training data which, unfortunately, indicates that we are most likely over-fitting our dataset quite a lot. Retroactively calculating the training and test accuracy of each of our cross-validation models indicates that, in each case, over-fitting seems to be occurring to a similar degree.

The following precision and recall values were reported for the model which, despite the apparent over-fitting, do indicate the strength of the model's results:

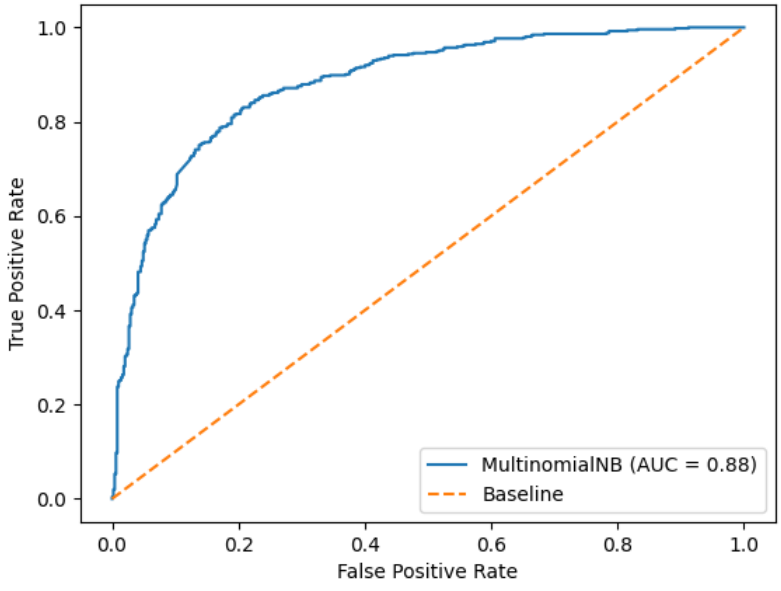
	Precision	Recall
False	0.79	0.84
True	0.84	0.79

The following confusion matrix was produced for the model which indicates a similar result:



The above matrix does reveal that the model does incorrectly predicts the false label a good deal more than it incorrectly predicts the true label - leading to a slightly higher false negative rate.

Finally, we can evaluate our model using ROC curves to visualise the true and false positive rates of the model - the following curve was produced:



Given the data from the aforementioned classification report these plotted values are to be expected: the model vastly outperforms the baseline - maintaining a TPR of around 80%.

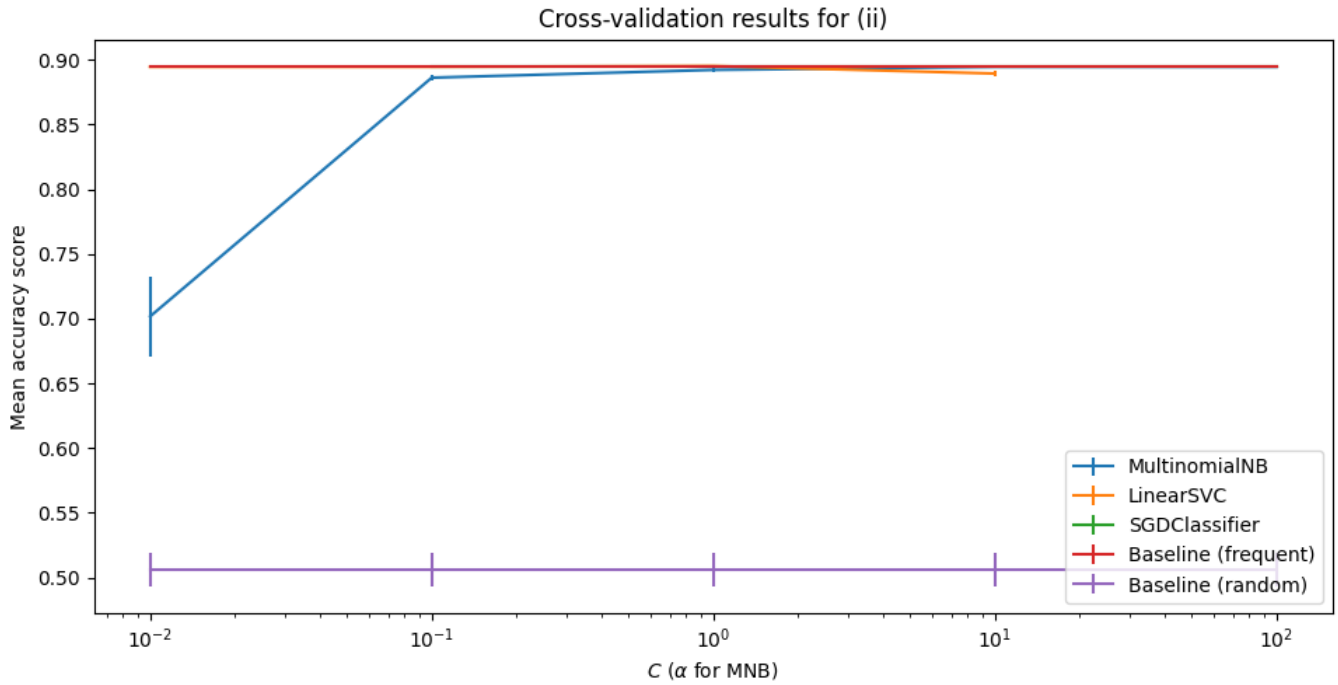
In conclusion, it would seem that it is very possible to accurately predict the polarity of the reviews in our dataset although it does appear that all of the evaluated models lead to a moderately high degree of over-fitting.

(ii) Early Access Games

Cross-validation

As previously mentioned, modifying the TF-IDF parameters produced few changes in the predictive accuracy of the trained models when attempting to discern whether or not a review was for an 'early access' game. Nonetheless, as was the case in (i) the models that appeared to perform consistently better (albeit only slightly better) were still the Naive Bayes, linear SVC and SGD classifiers. Similarly, the MLP classifier took an inordinate amount of time to run while producing results that were no better than the other classifiers. The same two baseline classifiers were used: one that predicted the most frequent label and one that predicted a random label.

As in (i), 5-fold cross-validation was used with a slightly smaller range of parameter values (every power of 10 from 10^{-2} to 10^{2}) while the models were evaluated using their mean accuracy score. The results for the relevant parameter ranges are plotted below:



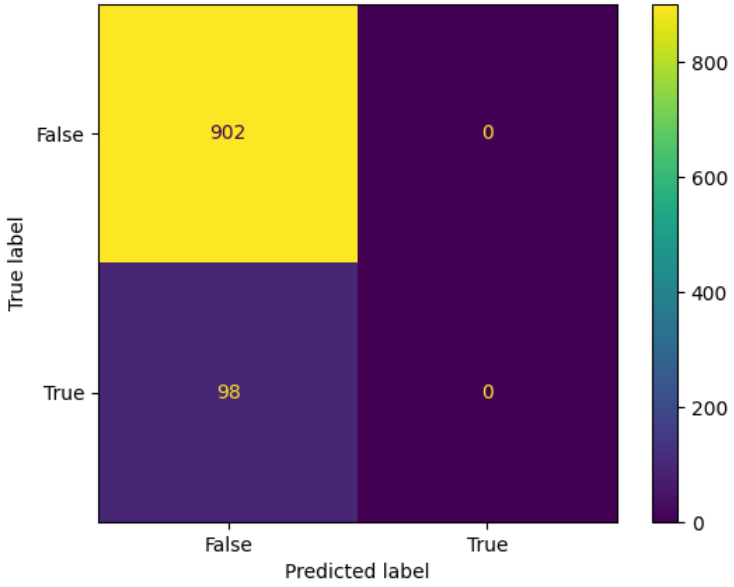
From the above graph it is very clear that the optimal trained models do not perform any better than the 'most frequent' baseline classifier - meaning that each of the trained models is simply predicting a constant label. These results (an accuracy of 89.46% with a negligible standard deviation) occur regardless of whether or not the reviews have been translated. In fact, when I tried to manually predict the 'early access' label for a selection of reviews myself I was unable to perform any better which would indicate that this label cannot be predicted reliably given only the review text (even the literal occurrence of the phrase *'early access'* in the review did not actually help). Nonetheless, in order to evaluate the 'optimal' model I chose to use the linear SVC classifier with $C = 1.0$ so as to avoid meaninglessly evaluating the results of a baseline classifier.

Model Evaluation

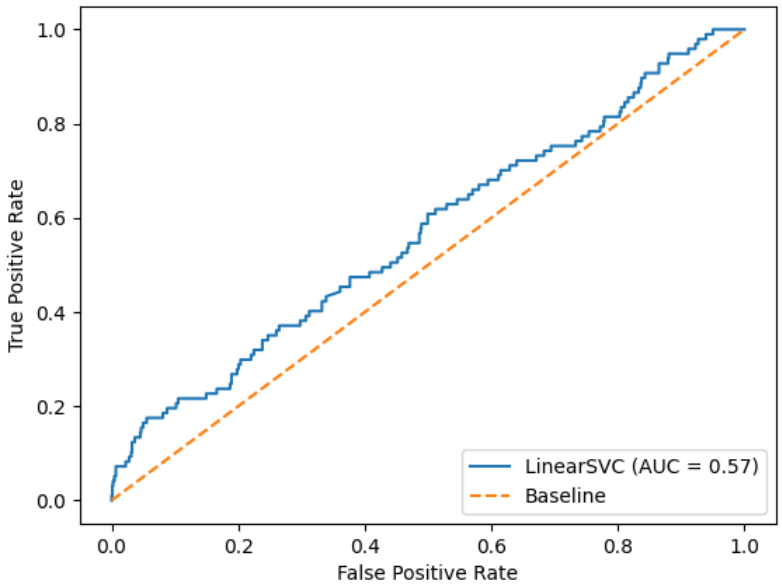
The trained model's classification report indicated an accuracy of 90% for the test data and an accuracy of 94% for the training data which, unlike (i), does not indicate that we are over-fitting our dataset to a meaningful degree. As is expected given the previous cross-validation results, our model reports the following precision and recall values:

	Precision	Recall
False	0.9	1.0
True	1.0	0.01

These values do not tell us anything of note given that we were already aware of the constant predictions of our model - this is further illustrated in the following confusion matrix which shows that the model always predicts the false label:



The following ROC curve was produced by the code which indicates a TPR which is almost identical to that of a baseline:

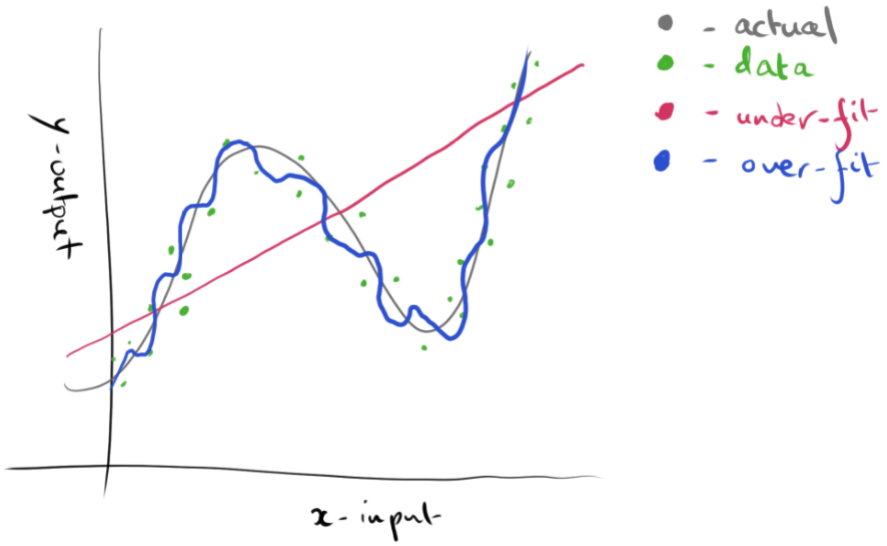


In conclusion, it would seem that it is not possible to determine whether or not a review is for an early access game given only the review text.

Question 2

(i) Under-fitting is when a model cannot accurately predict values/labels for both the data it's been trained on and data it has not seen. In contrast, over-fitting is when a model has begun to factor in noise and other unimportant details from the data it's been trained on which then leads to poorer predictions on unseen data.

As an example, let's assume that there is an input upon which the output is roughly determined by some polynomial function (i.e. a cubic curve) - assume that our input features are dispersed roughly along this curve. A model that was under-fitting might only produce a predictions determined by some simple linear curve while a model that was over-fitting would make predictions that factor in slight discrepancies between the true underlying function and noise that is found in the sample data. A simple hand-drawn visualisation is included below to help with the explanation:



(ii) The following pseudo-code performs k-fold cross-validation on some given ML model and dataset - it can make predictions and return an array of k generic scores (accuracy, F1, etc.):

```
function kfold_cv(k, model, data):
    # shuffle the order of both input/output
    shuffle(data)
    X, Y = data.x, data.y
    # init results
    results = []
    # sub-section length
    L = data.length / k
    # run cross-val k times
    for i in range(k):
        # test set is the i-th sub-section of data
        X_test = X[L*i:L*(i+1)]
        Y_test = Y[L*i:L*(i+1)]
        # training set is the rest
        X_train = X[:L*i] + X[L*(i+1):]
        Y_train = Y[:L*i] + X[L*(i+1):]
        # fit some model to the training data
        fitted_model = model.fit(X_train, Y_train)
        # make predictions and log the score
        score = fitted_model.predict_and_score(X_test, Y_test)
        results.add(score)
    # return the resulting array of 'scores'
    # these results can be averaged, etc.
    return results
```

Note: shuffling is initially performed in order to add an extra layer of randomness to improve the reliability of the results.

(iii) As discussed in (i) under-fitting occurs when a model fails to make predictions on neither the training data nor the test data and over-fitting occurs when a model incorporates noise/etc. from the training data which results in poorer predictions on test data. k-fold cross-validation helps detect the occurrence of both effects via the following: by splitting the dataset into numerous subsets and separately training models for each of them we are able to determine the quality of our models for many (as many as we choose!) different combinations of training data and unseen data despite only

having a single overall dataset. Each trained model can be tested on both the $k-1$ training subsets and the 1 test subset - these two sets of results can be compared and, if the mean training and test results differ to a significant degree then under-fitting (training < test) or over-fitting (test < training) can be easily detected.
(iv) A kNN classifier does not need to be trained on a dataset since it simply considers the k nearest training points to a given input when determining the classification label - this eliminates the need for an extensive training period. Similarly, a kNN classifier can have additional data added to it without any extra training/computation since future predictions will simply factor the new data into the distance calculations without any issue. In contrast, a logistic regression (LR) classifier, while requiring training and additional computation, will be able to give probabalistic predictions for each of the dataset's output labels while a kNN classifier will only be able to provide a single predicted result without any additional data. Another advantage of LR classifiers is that they are capable of scaling well to very large datasets without too much additional computation - kNN classifiers need to consider a lot more data points when making predicting classes which can cause problem when datasets become quite large. Furthermore, due to the fact that kNN classifiers need to calculate distances between test points and training points, an increase in the number of features in a dataset (i.e. the dimensionality of the dataset) can cause an exponential increase in the number of calculations that need to be made during each prediction - this computational issue does not occur in an LR classifier! Another disadvantage of kNN classifiers which we saw in previous weekly assignments was their inability to make predictions given inputs that did not contain a large number of close training data points: since predictions are determined based on the closest data points in the training set an absence of said points can lead to highly inaccurate predictions.
(v) As discussed in (iv) a clear instance in which a kNN classifier would give incorrect predictions is when, given some input, there are few training points that are close to that input which will lead to an inaccurate 'snapping' of the predicted output to the nearest training points which could be very far away from the ideal predicted output. Another case in which kNN classifiers can lead to inaccurate predictions is when the number of features (or dimensions) is quite large relative to the number of training points - this is due to the fact that an increase in input dimensions causes an increase in the complexity of the calculated distances which leads to predictions that are less representative of the training data. This effect can be expressed in simpler terms: the size of the sample space increases with each added dimension despite the number of training points staying the same. This effect can be rectified by increasing the number of training points to essentially 'fill in the gaps'.

Appendix A: Code

1. Language Detection

```
import json_lines
import spacy
from collections import defaultdict
from spacy_langdetect import LanguageDetector

nlp = spacy.load('en_core_web_sm')
nlp.add_pipe(LanguageDetector(), name='ld')
results = defaultdict(int)

with open('dataset.jl', 'rb') as f:
    for line in json_lines.reader(f):
        results[nlp(line['text'])._.language['language']] += 1

for k, v in results.items():
    print(k, v / 5000)
```

2. Review Translation

```
from google_trans_new import google_translator
import csv
import json_lines

with open('dataset.jl', 'rb') as f1:
    with open('translated.csv', 'w+', encoding='UTF-8', newline='') as f2:
        i = 0
        trans = google_translator()
        writer = csv.writer(f2, delimiter=',', quotechar='\"')
        for line in json_lines.reader(f1):
            raw = line['text']
            # don't translate very long reviews
            if len(raw) >= 5000: result = raw
            else: result = trans.translate(raw, lang_tgt='en')
            writer.writerow([result, line['voted_up'], line['early_access']])
            if not i % 10: print(i)
            i += 1
```

3. Main Code

```
from sklearn.dummy import DummyClassifier
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import classification_report, plot_confusion_matrix, plot_roc_curve
from sklearn.model_selection import cross_val_score, train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.neural_network import MLPClassifier
from sklearn.svm import LinearSVC
import csv
import json_lines
import matplotlib.pyplot as plt
import nltk
import numpy as np

# only needs to be run once
def download_nltk():
    nltk.download('punkt')
    nltk.download('stopwords')
    nltk.download('wordnet')
download_nltk()

# read raw data
def read_original_data():
    X, Y1, Y2 = [], [], []
    with open('dataset.jl', 'rb') as f:
        for line in json_lines.reader(f):
            X.append(line['text'])
            Y1.append(line['voted_up'])
            Y2.append(line['early_access'])
    return X, Y1, Y2

# read translated data
def read_translated_data():
    X, Y1, Y2 = [], [], []
    with open('translated.csv', encoding='UTF-8', newline='') as f:
        reader = csv.reader(f, delimiter=',', quotechar='\"')
        for row in reader:
            X.append(row[0])
            Y1.append(row[1])
            Y2.append(row[2])
    return X, Y1, Y2

# lemmatize all the entries in the dataset
# https://stackoverflow.com/questions/47423854/sklearn-adding-lemmatizer-to-countvectorizer
```

```
class Lemmatizer(object):
    def __init__(self):
        self.lem = nltk.stem.WordNetLemmatizer()
    def __call__(self, docs):
        return [self.lem.lemmatize(term) for term in nltk.word_tokenize(docs)]

# run cross-validation for all models
def cross_validation(X, Y, is_part_i=True):
    # different hyper-parameter ranges for each question
    if is_part_i:
        C_rng_bl = [1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1e0, 1e1, 1e2, 1e3, 1e4, 1e5]
        alpha_rng = [1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1e0, 1e1]
        C_rng_svc = [1e-4, 1e-3, 1e-2, 1e-1, 1e0, 1e1]
        C_rng_sgd = [1e-1, 1e0, 1e1, 1e2, 1e3, 1e4, 1e5]
    else:
        C_rng_bl = [1e-2, 1e-1, 1e0, 1e1, 1e2]
        alpha_rng = [1e-2, 1e-1, 1e0, 1e1, 1e2]
        C_rng_svc = [1e-2, 1e-1, 1e0, 1e1]
        C_rng_sgd = [1e-1, 1e0, 1e1, 1e2]

    # lemmatize and vectorize data
    X = TfidfVectorizer(
        sublinear_tf=True,
        strip_accents='unicode',
        tokenizer=Lemmatizer(),
        ngram_range=(1, 2)
    ).fit_transform(X)

    print('=== Baselines ===')
    model = DummyClassifier(strategy='most_frequent')
    scores = cross_val_score(model, X, Y, cv=5, scoring='accuracy')
    blf_means = [scores.mean()] * len(C_rng_bl)
    blf_stds = [scores.std()] * len(C_rng_bl)
    print('Frequent >> (%.4f, %.4f)' % (blf_means[0], blf_stds[0]))
    model = DummyClassifier(strategy='uniform')
    scores = cross_val_score(model, X, Y, cv=5, scoring='accuracy')
    blr_means = [scores.mean()] * len(C_rng_bl)
    blr_stds = [scores.std()] * len(C_rng_bl)
    print('Random >> (%.4f, %.4f)' % (blr_means[0], blr_stds[0]))

    print('=== MultinomialNB ===')
    mnb_means, mnb_stds = [], []
    for alpha in alpha_rng:
        model = MultinomialNB(alpha=alpha).fit(X, Y)
        scores = cross_val_score(model, X, Y, cv=5, scoring='accuracy', n_jobs=-1)
        mnb_means.append(scores.mean())
        mnb_stds.append(scores.std())
        print('a = %f >> (%.4f, %.4f)' % (alpha, scores.mean(), scores.std()))

    print('=== LinearSVC ===')
    svc_means, svc_stds = [], []
    for C in C_rng_svc:
        model = LinearSVC(loss='hinge', penalty='l2', C=C, max_iter=10000).fit(X, Y)
        scores = cross_val_score(model, X, Y, cv=5, scoring='accuracy', n_jobs=-1)
        svc_means.append(scores.mean())
        svc_stds.append(scores.std())
        print('C = %f >> (%.4f, %.4f)' % (C, scores.mean(), scores.std()))

    print('=== SGDClassifier ===')
    sgd_means, sgd_stds = [], []
    for C in C_rng_sgd:
        model = SGDClassifier(loss='log', penalty='l2', alpha=1/C).fit(X, Y)
        scores = cross_val_score(model, X, Y, cv=5, scoring='accuracy', n_jobs=-1)
        sgd_means.append(scores.mean())
        sgd_stds.append(scores.std())
```

```
        print('C = %f >> (%.4f, %.4f)' % (C, scores.mean(), scores.std()))

    # plot results
    plt.errorbar(alpha_rng, mnb_means, yerr=mnb_stds)
    plt.errorbar(C_rng_svc, svc_means, yerr=svc_stds)
    plt.errorbar(C_rng_sgd, sgd_means, yerr=sgd_stds)
    plt.errorbar(C_rng_bl, blf_means, yerr=blf_stds)
    plt.errorbar(C_rng_bl, blr_means, yerr=blr_stds)
    plt.xticks(C_rng_bl)
    plt.xlabel('$C$ ($\\alpha$ for MNB)')
    plt.ylabel('Mean accuracy score')
    plt.xscale('log')
    plt.legend(
        [
            'MultinomialNB',
            'LinearSVC',
            'SGDClassifier',
            'Baseline (frequent)',
            'Baseline (random)'
        ], loc='lower right'
    )
    if is_part_i: plt.title('Cross-validation results for (i)')
    else: plt.title('Cross-validation results for (ii)')
    plt.show()

# train the optimal model and display results
def optimal_model(X, Y, is_part_i=True):
    # lemmatize and vectorize data
    X = TfidfVectorizer(
        sublinear_tf=True,
        strip_accents='unicode',
        tokenizer=Lemmatizer(),
        ngram_range=(1, 2)
    ).fit_transform(X)

    # train models
    Xtr, Xte, Ytr, Yte = train_test_split(X, Y, test_size=0.2)
    if is_part_i:
        model = MultinomialNB(alpha=1e-1).fit(Xtr, Ytr)
    else:
        model = LinearSVC(loss='hinge', penalty='l2', C=1.0, max_iter=10000).fit(Xtr, Ytr)

    # classification reports
    print('=== Test Report ===')
    Ypr1 = model.predict(Xte)
    print(classification_report(Yte, Ypr1))
    print('=== Train Report ===')
    Ypr2 = model.predict(Xtr)
    print(classification_report(Ytr, Ypr2))

    # confusion matrices
    plot_confusion_matrix(model, Xte, Yte)
    plt.show()
    plot_confusion_matrix(model, Xtr, Ytr)
    plt.show()

    # roc curves
    fig, ax = plt.subplots()
    plot_roc_curve(model, Xte, Yte, ax=ax)
    ax.plot([0, 1], [0, 1], linestyle='--', label='Baseline')
    ax.legend(loc='lower right')
    plt.show()

def main():
    X, Y1, Y2 = read_translated_data()
```

```
    print('>>> Review Polarity (cross-val) <<<')
    cross_validation(X, Y1, is_part_i=True)
    print('>>> Early Access Game (cross-val) <<<')
    cross_validation(X, Y2, is_part_i=False)
    print('>>> Review Polarity (optimal model) <<<')
    optimal_model(X, Y1, is_part_i=True)
    print('>>> Early Access Game (optimal model) <<<')
    optimal_model(X, Y2, is_part_i=False)

main()
```

Appendix B: Datasets

Original Dataset
Translated Dataset