EECS 445 Project 1 Report
Uniquename: boyangx
October 5, 2019

1. Introduction

The objective of this project is to solve the task of finding the most critically-acclaimed movies by training various Support Vector Machines (SVMs) to classify the sentiment of a review

2. Feature Extraction

a. Implement extract_dictionary (see code appendix)
b. Implement generate_feature_matrix (see code appendix)
c. The number of unique words, denoted by d, is 10619
   The average number of non-zero features per rating in the training data is 68.168

3. Hyperparameter and Model Selection

3.1 Hyperparameter selection for linear-kernel SVM

a.

Implemented cv_performance with a helper function performance (see code appendix)
Maintaining class proportions across folds is beneficial because we want draw the training dataset and the test data set from the same distribution so that training data is a representative sample of the test data.

b.

Implemented select_param_linear with a helper function select_classifier (see code appendix)

c.

Best C values for different metrics with performances

Performance Measures	C	Performance
Accuracy	0.1	0.8109999999999999
F1-Score	0.1	0.8125587612890968
AUROC	0.1	0.8724399999999999
Precision	0.1	0.8084184094655127
Sensitivity	0.001	0.9099999999999999
Specificity	0.1	0.8039999999999999

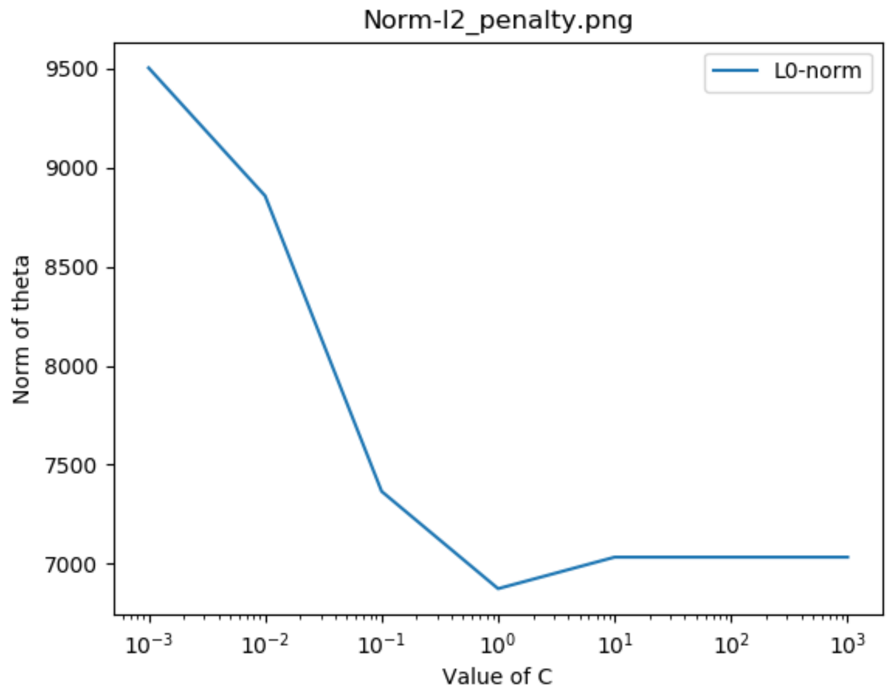
Generally, for all performance measures except for sensitivity, as C increases, cv_performance increases to peak and then decreases. For sensitivity, the cv_performance reaches the highest when C = 0.001 and then decreases. If I have to train a final model, I would optimize accuracy measure when choosing C since accuracy is the most direct measure to address the importance of identifying both positive and negative classes.

d.

Performances of SVM created with C = 0.1 for different metrics

Performance Measures	Performance
Accuracy	0.828
F1-Score	0.8313725490196078
AUROC	0.901088
Precision	0.8153846153846154
Sensitivity	0.848
Specificity	0.808

e.



Implemented plot_weight (see code appendix)
We can see from the above figure that L0-norm of theta decreases as C increases but gradually converges when C reaches the value of 10.

f.

Positive/Negative coefficients with corresponding words

Positive Coefficient	Word	Negative Coefficient	Word
0.5264765929951933	great	-0.5003295720357759	stupid
0.4816236395618302	hope	-0.4728701857334847	boring
0.4577006461614173	love	-0.45261704023663785	not
0.41676940676838137	well	-0.4292173682944536	nothing

3.2 Hyperparameter selection for quadratic-kernel SVM

a. Implemented select_param_quadratic (see code appendix)
   i. Grid Search (see code appendix)
   ii. Random Search (see code appendix)

b.

AUROC results for two different methods

Tuning Scheme	C	R	AUROC
Grid Search	10.0	10.0	0.87446
Random Search	28.09538854212485	7.139888817587492	0.87944

In general, for unchanged C value, the performance typically increases as r value increases; for unchanged r value, the performance typically increases to peak and then decreases as C value increases. The use of random search is better than the grid search since random search will cover more distinct values of C and r. Also, in most cases where C and r do not contribute equally to the performance, random search is more likely to generate closer results to the optimal hyper parameter. However, the random search may be less useful than the grid search when deterministic nature is emphasized.

3.3 Learning non-linear classifiers with a linear-kernel SVM

a.

The quadratic kernel is $K(\bar{x},\bar{x}') = (\bar{x}\cdot\bar{x}'+r)^2$. Suppose $\bar{x},\bar{x}' \in \mathbb{R}^d$, we can expand it as

$$\begin{aligned}
K(\bar{x},\bar{x}') =& (\sum_{i=1}^{d} x_i x_i' + r)^2 = (\sum_{i=1}^{d} x_i x_i')^2 + 2(\sum_{i=1}^{d} x_i x_i')r + r^2 \\
=& \sum_{i=1}^{d} (x_i x_i')^2 + 2\sum_{i=1}^{d-1}\sum_{j=i+1}^{d} (x_i x_i')(x_j x_j') + 2(\sum_{i=1}^{d} x_i x_i')r + r^2 \\
=& \sum_{i=1}^{d} x_i^2 x_i'^2 + 2\sum_{i=1}^{d-1}\sum_{j=i+1}^{d} (x_i x_j)(x_i' x_j') + 2r(\sum_{i=1}^{d} x_i x_i') + r^2 \\
=& \phi(\bar{x})\phi(\bar{x}')
\end{aligned}$$

Therefore $\phi(x) = [(x_i^2)_{i=1..d}, (\sqrt{2}x_i x_j)_{i=1..d-1, j=i+1..d}, (\sqrt{2r}x_i)_{i=1..d}, r]^T$

b.

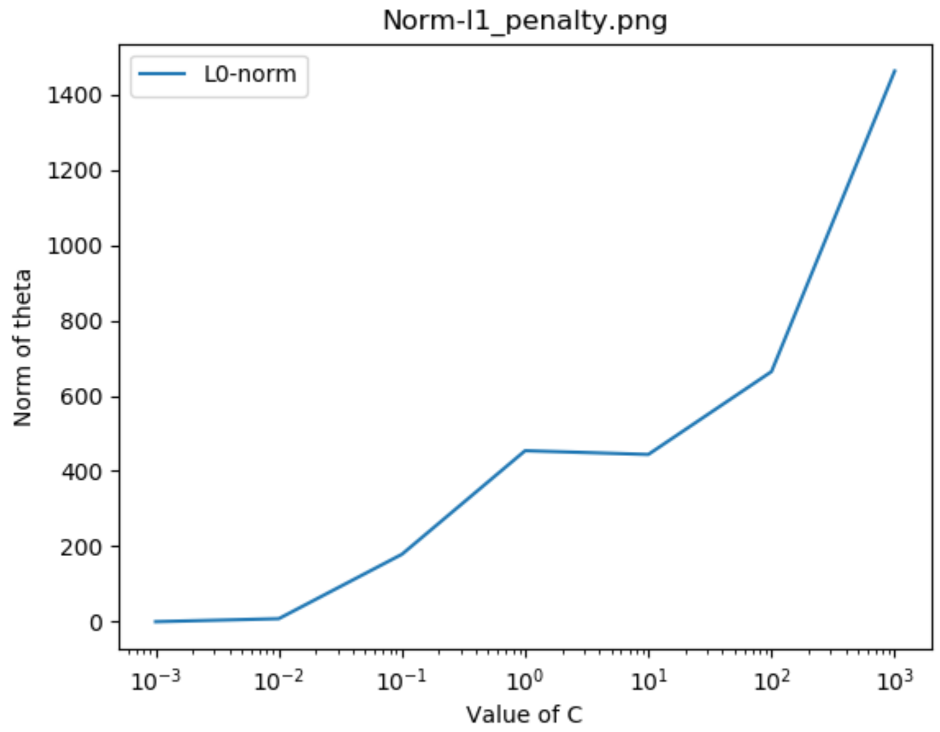
Pros: we can access weights foe each feature if using explicit feature mapping

Cons: If the features of the original data is of order d and the transformed data would have features of order d square, making it expensive to compute

3.4 Linear-kernel SVM with L1 penalty and squared hinge loss

a. C = 0.1 and performance = 0.85694

b.



c.

Primarily, the L0-norm value of the learned parameter significantly dropped under L1 penalty and it increases as the value of C increases. Also, when value of C is small, weights are zero.

d.

If Squared Hinge Loss is used instead of the Hinge Loss, the penalty of the misclassified data points will be larger so that more data points would become support vectors and the margin would be larger. Thus, the optimal solution will have a smaller L0-norm under Squared Hinge Loss

4. Asymmetric cost functions and class imbalance

4.1 Arbitrary class weights

a.

If W_n is much greater than W_p, the misclassified negative data points would bear a much larger penalty than the misclassified positive ones. This would result in a model which emphasizes more upon classifying negative data points. Referring to the weighted SVM formula, data points with higher weights now would have a lower slack variable and negative data points are more likely to be correctly classified.

b.

Performance of the modified SVM for different metrics

Performance Measures	Performance
Accuracy	0.678
F1-Score	0.5515320334261838
AUROC	0.8669279999999999
Precision	0.908256880733945
Sensitivity	0.396
Specificity	0.96

c.

Compared to result in 3.1d, performances of F1-Score and Sensitivity are the most affected ones. As stated in 4.1a, our model is now emphasizing more on classifying negative data points. Therefore, performance measure for classifying positive data points such as sensitivity would decrease. Performance of F1-Score decreases since it is a function of sensitivity and precision, and it will be closer to the smaller one of the two. In this case, sensitivity is smaller than precision.

4.2 Imbalanced data

a.

Performance of the SVM (C = 0.01)

Class Weights	Performance Measures	Performance
W_n= 1, W_p= 1	Accuracy	0.8
W_n= 1, W_p= 1	F1-Score	0.8883928571428571
W_n= 1, W_p= 1	AUROC	0.8469
W_n= 1, W_p= 1	Precision	0.8024193548387096
W_n= 1, W_p= 1	Sensitivity	0.995
W_n= 1, W_p= 1	Specificity	0.02

b.

In the given imbalanced data set, there are more positive data points than negative ones. This has caused sensitivity to increase significantly and specificity to decrease significantly. Because of this imbalance, the classifier tends to classifier almost all the data as positive. In fact, 99.5% of the positive data are correctly classified as positive, this is shown by the true positive rate (sensitivity). Only 2% of the data are correctly classified as negative, this is shown by the true negative rate (specificity).

4.3 Choosing appropriate class weights

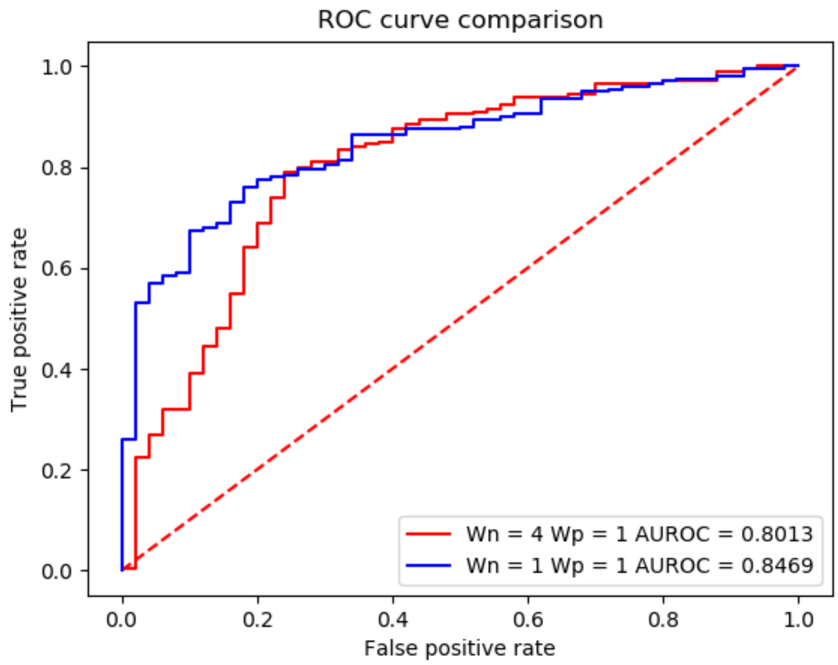
a.

Since our data set is imbalanced, therefore, we want to choose a metric that could balance the classifications of both negative and positive classes, the F1-score is more informative in such situation since it maintains a balance between sensitivity and precision, which helps mitigating the situation in 4.2. If we keep the positive weight as 1 and in order to counter the class size imbalance, negative weights should be at least 4 (i.e. Wn should be reasonably greater than Wp) since the ratio between the class sizes is 4:1. Negative weights found using Cross Validation with respect to f1-score across a reasonably large range of weights. However, since the minor class in this case is the negative class, we want to pick the weight within the range but would have the best performance for the f1-score, which is a metric focuses on classifying the negative data points. Eventually, the found Wn is 4 and the corresponding performance is 0.8963855421686747.

b.

Class Weights	Performance Measures	Performance
W_n= 4, W_p= 1	Accuracy	0.828
W_n= 4, W_p= 1	F1-Score	0.8963855421686747
W_n= 4, W_p= 1	AUROC	0.8013
W_n= 4, W_p= 1	Precision	0.8651162790697674
W_n= 4, W_p= 1	Sensitivity	0.93
W_n= 4, W_p= 1	Specificity	0.42

4.4 The ROC curve



5. Challenge

In this part, I implemented the following approaches to train the model:

1. Linear kernel SVM with L1 penalty
2. Linear kernel SVM with L2 penalty and one-vs-one method
3. Linear kernel SVM with L2 penalty and one-vs-rest method
4. Quadratic kernel SVM with L2 penalty and one-vs-one method
5. Feature engineering: using the number of times a word appears in reviews as feature and Linear kernel SVM with L1 penalty
6. Feature engineering: consider rating as part of the feature selection (i.e. extreme ratings such as 1 and 5 are doubly weighted) and Linear kernel SVM with L1 penalty
7. Feature engineering: Do not leave out the punctuations when extracting features form the review text and Linear kernel SVM with L1 penalty

This time, we have three classes and we want to equally classify each class; Therefore, we would use accuracy as the metric. The procedure of choosing C for the linear SVCs is as follows:

- Initialize C as 10 ** I, I as -3 and step size as 2
- run cv_performance
- If the new performance is better than the best so far, step size remains as 2
- If the step size is positive, divide it by -2, if the step size is negative, divide it by 2
- Increment I by the step size accordingly

The convergence criteria would be when the absolute difference of the new cv_performance and the best cv_performance is less than 0.001. For quadratic SVCs, grid search is used with C and r that lie in the same ranges as the previous parts. When calculating the cv_performance, I am now using 10-fold instead of 5-fold.

From cross validation performances I generated for each approach, I found out that using linear kernel SVM with L2 penalty would generate the highest accuracy cv_performance whereas the quadratic kernel SVMs generated a lower cv_performance than linear kernel SVM with L1 penalty. As for the multiclass methods, one-vs-one and one-vs-rest are producing the same cv_performance for Linear kernel SVM with the L2 penelty. I modified the original feature matrix generation function to generate_feature_matrix_multi, which takes into account the number of occurrences of a word in reviews (i.e. by setting Count to True), the existences of extreme ratings (i.e. by setting Rating to True), and the existences of original punctuations (i.e. by setting Punc to True). However, the improve in performance is neglectable with Count, meaning that this feature engineering method would not improve the final result of the trained model by much. One possible reason for this could be that the majority of the repeated words within the review are most likely to be common words such as "we", "I", "feel", etc. These words would not contribute to the classification as they do not show emotional inclination. As for rating and punctuations, there was seen an increase in both of the cv_performances but the improved performances are still slightly lower than the Linear CVM with L2 penelty, meaning that these two feature engineering methods could potentially generate better models, possible reasons for this could be: extreme rating if assumed that audience are reviewing honestly could reflect more on the actual movie quality; punctuations like "!" "?" "…" sometimes do play an important role in showing emotional inclinations.

In conclusion, I chose to use approach 2 with C being roughly 0.03 and the cv_performance is 0.633.

```python
# EECS 445 - Fall 2019
# Project 1 - project1.py

import pandas as pd
import numpy as np
import itertools
import string
import warnings
import random

warnings.filterwarnings("ignore", category = FutureWarning, module = "sklearn")

from sklearn.svm import SVC, LinearSVC
from sklearn.model_selection import StratifiedKFold, GridSearchCV
from sklearn import metrics
from matplotlib import pyplot as plt

from helper import *


def select_classifier(penalty='l2', c=1.0, degree=1, r=0.0, class_weight='balanced'):
    """
    Return a linear svm classifier based on the given
    penalty function and regularization parameter c.
    """
    if penalty == 'l1':
        return LinearSVC(penalty = 'l1', dual = False, C = c, class_weight = 'balanced')
    elif penalty == 'l2':
        if degree == 1:
            return SVC(kernel = 'linear', C = c, degree = 1, class_weight = class_weight)
        else:
            return SVC(kernel = 'poly', C = c, degree = degree, coef0 = r, class_weight = class_weight)


def extract_dictionary(df):
    """
    Reads a panda dataframe, and returns a dictionary of distinct words
    mapping from each distinct word to its index (ordered by when it was found).
    Input:
        df: dataframe/output of load_data()
    Returns:
        a dictionary of distinct words that maps each distinct word
        to a unique index corresponding to when it was first found while
        iterating over all words in each review in the dataframe df
    """
    word_dict = {}
    k = 0
    for entry in df['reviewText']:
        entry = entry.lower()
        for char in entry:
            if char in string.punctuation:
                entry = entry.replace(char, ' ')
        for word in entry.split():
            if word not in word_dict.keys():
                word_dict[word] = k
                k += 1
    return word_dict


def generate_feature_matrix(df, word_dict):
    """
    Reads a dataframe and the dictionary of unique words
    to generate a matrix of {1, 0} feature vectors for each review.
    Use the word_dict to find the correct index to set to 1 for each place
    in the feature vector. The resulting feature matrix should be of
    dimension (number of reviews, number of words).
    Input:
        df: dataframe that has the ratings and labels
        word_list: dictionary of words mapping to indices
```

```python
        word_list: dictionary of words mapping to indices
    Returns:
        a feature matrix of dimension (number of reviews, number of words)
    """
    number_of_reviews = df.shape[0]
    number_of_words = len(word_dict)
    feature_matrix = np.zeros((number_of_reviews, number_of_words))
    for i in range(number_of_reviews):
        entry = df['reviewText'][i]
        entry = entry.lower()
        for char in entry:
            if char in string.punctuation:
                entry = entry.replace(char, ' ')
        for word in word_dict.keys():
            if word in entry.split():
                feature_matrix[i][word_dict[word]] = 1
    return feature_matrix

def generate_feature_matrix_multi(df, word_dict, Count = False, Rating = False, Punc = False):

    number_of_reviews = df.shape[0]
    number_of_words = len(word_dict)
    feature_matrix = np.zeros((number_of_reviews, number_of_words))
    for i in range(number_of_reviews):
        entry = df['reviewText'][i]
        rating = df['rating'][i]
        entry = entry.lower()
        if Punc == False:
            for char in entry:
                if char in string.punctuation:
                    entry = entry.replace(char, ' ')
        for word in word_dict.keys():
            if word in entry.split():
                if Count:
                    feature_matrix[i][word_dict[word]] += 1
                elif Rating:
                    if rating == 1 or rating == 5:
                        feature_matrix[i][word_dict[word]] = 2
                else:
                    feature_matrix[i][word_dict[word]] = 1
    return feature_matrix


def cv_performance(clf, X, y, k=5, metric="accuracy"):
    """
    Splits the data X and the labels y into k-folds and runs k-fold
    cross-validation: for each fold i in 1...k, trains a classifier on
    all the data except the ith fold, and tests on the ith fold.
    Calculates the k-fold cross-validation performance metric for classifier
    clf by averaging the performance across folds.
    Input:
        clf: an instance of SVC()
        X: (n,d) array of feature vectors, where n is the number of examples
           and d is the number of features
        y: (n,) array of binary labels {1,-1}
        k: an int specifying the number of folds (default=5)
        metric: string specifying the performance metric (default='accuracy'
             other options: 'f1-score', 'auroc', 'precision', 'sensitivity',
             and 'specificity')
    Returns:
        average 'test' performance across the k folds as np.float64
    """
    scores = np.zeros(k)
    skf = StratifiedKFold(n_splits = k, shuffle = False)
    i = 0
    for train_index, test_index in skf.split(X,y):
        X_train, y_train = X[train_index], y[train_index]
        clf.fit(X_train, y_train)
        X_test, y_test = X[test_index], y[test_index]
        # Put the performance of the model on each fold in the scores array
        if metric == "AUROC":
```

```python
            y_pred = clf.decision_function(X_test)
        else:
            y_pred = clf.predict(X_test)
        scores[i] = performance(y_test, y_pred, metric)
        i += 1

    #And return the average performance across all fold splits.
    return np.float64(np.array(scores).mean())


def select_param_linear(X, y, k=5, metric="accuracy", C_range = [], penalty='l2'):
    """
    Sweeps different settings for the hyperparameter of a linear-kernel SVM,
    calculating the k-fold CV performance for each setting on X, y.
    Input:
        X: (n,d) array of feature vectors, where n is the number of examples
        and d is the number of features
        y: (n,) array of binary labels {1,-1}
        k: int specifying the number of folds (default=5)
        metric: string specifying the performance metric (default='accuracy',
             other options: 'f1-score', 'auroc', 'precision', 'sensitivity',
             and 'specificity')
        C_range: an array with C values to be searched over
    Returns:
        The parameter value for a linear-kernel SVM that maximizes the
        average 5-fold CV performance.
    """
    best_c = 0.0
    best_performance = 0
    for c in C_range:
        clf = select_classifier(penalty = penalty, c = c, degree = 1, r = 0.0, class_weight = 'balanced')
        new_performance = cv_performance(clf, X, y, k, metric)
        if new_performance > best_performance:
            best_performance = new_performance
            best_c = c
    return best_c, best_performance


def plot_weight(X,y,penalty,C_range):
    """
    Takes as input the training data X and labels y and plots the L0-norm
    (number of nonzero elements) of the coefficients learned by a classifier
    as a function of the C-values of the classifier.
    """

    print("Plotting the number of nonzero entries of the parameter vector as a function of C")
    norm0 = []
    for c in C_range:
        norm = 0
        clf = select_classifier(penalty = penalty, c = c, degree = 1, r = 0.0, class_weight = 'balanced')
        clf.fit(X,y)
        for i in clf.coef_[0]:
            if i != 0:
                norm += 1
        norm0.append(norm)
    #This code will plot your L0-norm as a function of c
    plt.plot(C_range, norm0)
    plt.xscale('log')
    plt.legend(['L0-norm'])
    plt.xlabel("Value of C")
    plt.ylabel("Norm of theta")
    plt.title('Norm-'+penalty+'_penalty.png')
    plt.savefig('Norm-'+penalty+'_penalty.png')
    plt.close()


def select_param_quadratic(X, y, k=5, metric="accuracy", param_range=[]):
    """
        Sweeps different settings for the hyperparameters of an quadratic-kernel SVM,
```

```python
        calculating the k-fold CV performance for each setting on X, y.
        Input:
            X: (n,d) array of feature vectors, where n is the number of examples
               and d is the number of features
            y: (n,) array of binary labels {1,-1}
            k: an int specifying the number of folds (default=5)
            metric: string specifying the performance metric (default='accuracy'
                     other options: 'f1-score', 'auroc', 'precision', 'sensitivity',
                     and 'specificity')
            parameter_values: a (num_param, 2)-sized array containing the
                parameter values to search over. The first column should
                represent the values for C, and the second column should
                represent the values for r. Each row of this array thus
                represents a pair of parameters to be tried together.
        Returns:
            The parameter value(s) for a quadratic-kernel SVM that maximize
            the average 5-fold CV performance
    """
    best_c, best_r, best_performance = 0.0, 0.0, 0.0
    for c, r in param_range:
        clf = select_classifier(penalty = 'l2', c = c, degree = 2, r = r,
        class_weight = 'balanced')
        new_performance = cv_performance(clf, X, y, k, metric)
        if new_performance > best_performance:
            best_performance = new_performance
            best_c = c
            best_r = r
    return best_c, best_r, best_performance

def performance(y_true, y_pred, metric="accuracy"):
    """
    Calculates the performance metric as evaluated on the true labels
    y_true versus the predicted labels y_pred.
    Input:
        y_true: (n,) array containing known labels
        y_pred: (n,) array containing predicted scores
        metric: string specifying the performance metric (default='accuracy'
                 other options: 'f1-score', 'auroc', 'precision', 'sensitivity',
                 and 'specificity')
    Returns:
        the performance as an np.float64
    """
    if metric == "accuracy":
        return metrics.accuracy_score(y_true, y_pred)
    elif metric == "f1-score":
        return metrics.f1_score(y_true, y_pred)
    elif metric == "precision":
        return metrics.precision_score(y_true, y_pred)
    elif metric == "AUROC":
        return metrics.roc_auc_score(y_true, y_pred)
    elif metric == "sensitivity":
        m = metrics.confusion_matrix(y_true, y_pred, labels = [-1,1])
        return m[1,1] / (m[1,1] + m[1,0])
    elif metric == "specificity":
        m = metrics.confusion_matrix(y_true, y_pred, labels = [-1,1])
        return m[0,0] / (m[0,0] + m[0,1])

#@jit(target="cuda")
def main():
    # Read binary data
    # NOTE: READING IN THE DATA WILL NOT WORK UNTIL YOU HAVE FINISHED
    #       IMPLEMENTING generate_feature_matrix AND extract_dictionary
    X_train, Y_train, X_test, Y_test, dictionary_binary = get_split_binary_data()
    IMB_features, IMB_labels = get_imbalanced_data(dictionary_binary)
    IMB_test_features, IMB_test_labels = get_imbalanced_test(dictionary_binary)

    # Q2
    print(len(dictionary_binary))
    num_feature = 0
    for i in X_train:
        for j in i:
            if j == 1:
```

```
                num_feature += 1
    avg_feature = num_feature / X_train.shape[0]
    print(avg_feature)

    # Q3.1(c)
    C_range = [1e-3, 1e-2, 1e-1, 1, 1e1, 1e2, 1e3]
    print("Accuracy", select_param_linear(X_train, Y_train, 5, "accuracy", C_range, penalty = 'l2'))
    print("F1-Score", select_param_linear(X_train, Y_train, 5, "f1-score", C_range, penalty = 'l2'))
    print("AUROC", select_param_linear(X_train, Y_train, 5, "AUROC", C_range, penalty = 'l2'))
    print("Precision", select_param_linear(X_train, Y_train, 5, "precision", C_range, penalty = 'l2'))
    print("Sensitivity", select_param_linear(X_train, Y_train, 5, "sensitivity", C_range, penalty = 'l2'))
    print("Specificity", select_param_linear(X_train, Y_train, 5, "specificity", C_range, penalty = 'l2'))

    # Q3.1(d)
    clf = SVC(kernel='linear', C = 1e-1)
    clf.fit(X_train, Y_train)
    Y_pred = clf.predict(X_test)
    Y_pred_auroc = clf.decision_function(X_test)
    print("Accuracy", performance(Y_test, Y_pred, metric = "accuracy"))
    print("F1-Score", performance(Y_test, Y_pred, metric = "f1-score"))
    print("AUROC", performance(Y_test, Y_pred_auroc, metric = "AUROC"))
    print("Precision", performance(Y_test, Y_pred, metric = "precision"))
    print("Sensitivity", performance(Y_test, Y_pred, metric = "sensitivity"))
    print("Specificity", performance(Y_test, Y_pred, metric = "specificity"))

    # Q3.1(e)
    plot_weight(X_train, Y_train, 'l2', C_range)

    # Q3.1(f)
    clf = select_classifier(penalty = 'l2', c = 0.1, degree = 1, r = 0.0, class_weight = 'balanced')
    clf.fit(X_train, Y_train)
    arg = clf.coef_[0].argsort()
    neg_ind4 = arg[:4]
    pos_ind4 = arg[:-5:-1]
    neg_words = []
    pos_words = []
    for idx in neg_ind4:
        for word, index in dictionary_binary.items():
            if index == idx:
                neg_words.append(word)
    print("Most negative words")
    for i in range(4):
        print(clf.coef_[0,neg_ind4[i]], neg_words[i])
    for idx in pos_ind4:
        for word, index in dictionary_binary.items():
            if index == idx:
                pos_words.append(word)
    print("Most positive words")
    for i in range(4):
        print(clf.coef_[0,pos_ind4[i]], pos_words[i])

    # Q3.2(a)
    # (i)
    grid = []
    c_range = [1e-3, 1e-2, 1e-1, 1, 1e1, 1e2, 1e3]
    r_range = [1e-3, 1e-2, 1e-1, 1, 1e1, 1e2, 1e3]
    for i in c_range:
        for j in r_range:
            grid.append([i,j])
    print(select_param_quadratic(X_train, Y_train, k = 5, metric = "AUROC", param_range = grid))
    #ii)
    param_random = np.zeros([25,2])
    for i in range(25):
        [c,r] = [pow(10,np.random.uniform(-3,3)), pow(10,np.random.uniform(-3,3))]
```

```
        param_random[i] = [c,r]
    print(select_param_quadratic(X_train, Y_train, k = 5, metric = "AUROC", param_range = param_random))

    # 3.4(a)
    print(select_param_linear(X_train, Y_train, k = 5, metric = "AUROC", C_range = C_range, penalty='l1'))

    # 3.4(b)
    plot_weight(X_train, Y_train, 'l1', C_range)

    # 4.1(b)
    clf = select_classifier(penalty = 'l2', c = 0.01, degree = 1, class_weight = {-1: 10, 1: 1})
    clf.fit(X_train, Y_train)
    Y_pred = clf.predict(X_test)
    Y_pred_auc = clf.decision_function(X_test)
    print("Accuracy", performance(Y_test, Y_pred, metric="accuracy"))
    print("F1-Score", performance(Y_test, Y_pred, metric="f1-score"))
    print("AUROC", performance(Y_test, Y_pred_auc, metric="AUROC"))
    print("Precision", performance(Y_test, Y_pred, metric="precision"))
    print("Sensitivity", performance(Y_test, Y_pred, metric="sensitivity"))
    print("Specificity", performance(Y_test, Y_pred, metric="specificity"))

    # 4.2(a)
    clf = select_classifier(penalty = 'l2', c = 0.01, degree = 1, class_weight = {-1: 1, 1: 1})
    clf.fit(IMB_features, IMB_labels)
    Labels_pred = clf.predict(IMB_test_features)
    Labels_pred_auc = clf.decision_function(IMB_test_features)
    print("Accuracy", performance(IMB_test_labels, Labels_pred, metric="accuracy"))
    print("F1-Score", performance(IMB_test_labels, Labels_pred, metric="f1-score"))
    print("AUROC", performance(IMB_test_labels, Labels_pred_auc, metric="AUROC"))
    print("Precision", performance(IMB_test_labels, Labels_pred, metric="precision"))
    print("Sensitivity", performance(IMB_test_labels, Labels_pred, metric="sensitivity"))
    print("Specificity", performance(IMB_test_labels, Labels_pred, metric="specificity"))

    # 4.3(a)
    best_wn, best_perf = 0.0, 0.0
    for i in range(25):
        Wn = 2**random.uniform(2, 5) # at least 4
        clf = select_classifier(penalty = 'l2', c = 0.1, degree = 1, class_weight = {-1: Wn, 1: 1})
        new_perf = cv_performance(clf, IMB_features, IMB_labels, 5, metric = "f1-score" )
        spec_perf = cv_performance(clf, IMB_features, IMB_labels, 5, metric = "specificity" )
        print(Wn, new_perf, spec_perf)
        if new_perf > best_perf:
            best_perf = new_perf
            best_wn = Wn
    print("f1-score (random search)", best_wn, best_perf)


    # 4.3(b)
    clf = select_classifier(penalty = 'l2', c = 0.1, degree = 1, class_weight = {-1: best_wn, 1: 1})
    clf.fit(IMB_features, IMB_labels)
    Labels_pred = clf.predict(IMB_test_features)
    Labels_pred_auc = clf.decision_function(IMB_test_features)
    print("Accuracy", performance(IMB_test_labels, Labels_pred, metric="accuracy"))
    print("F1-Score", performance(IMB_test_labels, Labels_pred, metric="f1-score"))
    print("AUROC", performance(IMB_test_labels, Labels_pred_auc, metric="AUROC"))
    print("Precision", performance(IMB_test_labels, Labels_pred, metric="precision"))
    print("Sensitivity", performance(IMB_test_labels, Labels_pred, metric="sensitivity"))
    print("Specificity", performance(IMB_test_labels, Labels_pred, metric="specificity"))

    # 4.4
```

```
    # customized Wn Wp
    fpr, tpr, thresholds = metrics.roc_curve(IMB_test_labels, Labels_pred_auc)
    # balanced
    clf_bal = select_classifier(penalty = 'l2', c = 0.01, degree = 1, class_weight = {-1: 1, 1: 1})
    clf_bal.fit(IMB_features, IMB_labels)
    Labels_pred_auc_bal = clf_bal.decision_function(IMB_test_features)
    fpr_bal, tpr_bal, thresholds_bal = metrics.roc_curve(IMB_test_labels, Labels_pred_auc_bal)
    plt.figure()
    plt.plot([0,1], [0,1], 'r--')
    plt.plot(fpr, tpr, color = 'red', label = 'Wn = 4 Wp = 1 AUROC = 0.8013')
    plt.plot(fpr_bal, tpr_bal, color = 'blue', label = 'Wn = 1 Wp = 1 AUROC = 0.8469')
    plt.legend(loc = "lower right")
    plt.xlabel('False positive rate')
    plt.ylabel('True positive rate')
    plt.title('ROC curve comparison')
    plt.savefig('ROC curve comparison.png')
    plt.close()


    # Read multiclass data
    # TODO: Question 5: Apply a classifier to heldout features, and then use
    # generate_challenge_labels to print the predicted labels
    multiclass_features, multiclass_labels, multiclass_dictionary = get_multiclass_training_data()
    heldout_features = get_heldout_reviews(multiclass_dictionary)

    # Approach 1 Linear kernel SVM with l1 penalty
    best_c, best_perf = 0.0, 0.0
    I = -3
    i = 2
    threshold = 1
    while abs(threshold) > 0.001:
        c = 10 ** I
        clf = LinearSVC(penalty = 'l1', dual = False, C = c, class_weight = 'balanced', max_iter = 100000)
        new_perf = cv_performance(clf, multiclass_features, multiclass_labels, 10)
        threshold = new_perf - best_perf
        print(c, new_perf)
        if new_perf > best_perf:
            best_perf = new_perf
            best_c = c
            i = 2
            I = I + i
        else:
            if i > 0:
                i = i/-2
            else:
                i = i/2
            I = I + i
    print("Linear-l1 optimal c and performance:", best_c, best_perf)

    # Approach 2 Linear kernel SVN with l2 penalty and ovo method
    best_c, best_perf = 0.0, 0.0
    I = -3
    i = 2
    threshold = 1
    while abs(threshold) > 0.001:
        c = 10 ** I
        clf = SVC(kernel = 'linear', C = c, degree = 1, class_weight = 'balanced', decision_function_shape='ovo')
        new_perf = cv_performance(clf, multiclass_features, multiclass_labels, 10)
        threshold = new_perf - best_perf
        print(c, new_perf)
        if new_perf > best_perf:
            best_perf = new_perf
            best_c = c
            i = 2
            I = I + i
        else:
            if i > 0:
```

```
                    i = i/-2
                else:
                    i = i/2
                I = I + i
        print("Linear-l2(ovo) optimal c and performance:", best_c, best_perf)

        # Approach 3 Linear kernel SVN with l2 penalty and ovr method
        best_c, best_perf = 0.0, 0.0
        I = -3
        i = 2
        threshold = 1
        while abs(threshold) > 0.001:
            c = 10 ** I
            clf = SVC(kernel = 'linear', C = c, degree = 1, class_weight = 'balanced')
            new_perf = cv_performance(clf, multiclass_features, multiclass_labels, 10)
            threshold = new_perf - best_perf
            print(c, new_perf)
            if new_perf > best_perf:
                best_perf = new_perf
                best_c = c
                i = 2
                I = I + i
            else:
                if i > 0:
                    i = i/-2
                else:
                    i = i/2
                I = I + i
        print("Linear-l2(ovr) optimal c and performance:", best_c, best_perf)

        # Approach 4 quadratic ovo
        c_range = [1e-3, 1e-2, 1e-1, 1, 1e1, 1e2, 1e3]
        r_range = [1e-3, 1e-2, 1e-1, 1, 1e1, 1e2, 1e3]
        best_c, best_r, best_perf = 0.0, 0.0, 0.0
        for c in c_range:
            for r in r_range:
                clf = SVC(kernel = 'poly', C = c, coef0 = r, degree = 2, class_weight = 'balanced', decision_function_shape='ovo')
                new_perf = cv_performance(clf, multiclass_features, multiclass_labels, 10)
                if new_perf > best_perf:
                    best_perf = new_perf
                    best_c = c
                    best_r = r
        print("Quadratic(ovo) optimal c, r and performance:", best_c, best_r, best_perf)

        # Approach 5 feature engineering
        # Using the number of times a word occurs in a review as a feature
        multiclass_features, multiclass_labels, multiclass_dictionary = get_multiclass_training_data(Count = True)
        # L1-linear
        best_c, best_perf = 0.0, 0.0
        I = -3
        i = 2
        threshold = 1
        while abs(threshold) > 0.001:
            c = 10 ** I
            clf = LinearSVC(penalty = 'l1', dual = False, C = c, class_weight = 'balanced')
            new_perf = cv_performance(clf, multiclass_features, multiclass_labels, 10)
            threshold = new_perf - best_perf
            print(c, new_perf)
            if new_perf > best_perf:
                best_perf = new_perf
                best_c = c
                i = 2
                I = I + i
            else:
                if i > 0:
                    i = i/-2
                else:
                    i = i/2
                I = I + i
```

```python
    print("Linear-l1 with count optimal c and performance:", best_c, best_perf)

    # Approach 6 feature engineering
    # Consider rating when generating features
    multiclass_features, multiclass_labels, multiclass_dictionary = get_multiclass_training_data(Rating = True)
    # L1-linear
    best_c, best_perf = 0.0, 0.0
    I = -3
    i = 2
    threshold = 1
    while abs(threshold) > 0.001:
        c = 10 ** I
        clf = LinearSVC(penalty = 'l1', dual = False, C = c, class_weight = 'balanced')
        new_perf = cv_performance(clf, multiclass_features, multiclass_labels, 10)
        threshold = new_perf - best_perf
        print(c, new_perf)
        if new_perf > best_perf:
            best_perf = new_perf
            best_c = c
            i = 2
            I = I + i
        else:
            if i > 0:
                i = i/-2
            else:
                i = i/2
            I = I + i
    print("Linear-l1 with rating optimal c and performance:", best_c, best_perf)

    # Approach 7 feature engineering
    # Do consider punctuations
    multiclass_features, multiclass_labels, multiclass_dictionary = get_multiclass_training_data(Punc = True)
    # L1-linear
    best_c, best_perf = 0.0, 0.0
    I = -3
    i = 2
    threshold = 1
    while abs(threshold) > 0.001:
        c = 10 ** I
        clf = LinearSVC(penalty = 'l1', dual = False, C = c, class_weight = 'balanced')
        new_perf = cv_performance(clf, multiclass_features, multiclass_labels, 10)
        threshold = new_perf - best_perf
        print(c, new_perf)
        if new_perf > best_perf:
            best_perf = new_perf
            best_c = c
            i = 2
            I = I + i
        else:
            if i > 0:
                i = i/-2
            else:
                i = i/2
            I = I + i
    print("Linear-l1 with punctuation optimal c and performance:", best_c, best_perf)

    # Conclusion
    multiclass_features, multiclass_labels, multiclass_dictionary = get_multiclass_training_data()
    heldout_features = get_heldout_reviews(multiclass_dictionary)
    clf = SVC(kernel = 'linear', C = 0.03, degree = 1, class_weight = 'balanced', decision_function_shape='ovo')
    clf.fit(multiclass_features, multiclass_labels)
    y_pred = clf.predict(heldout_features)
    generate_challenge_labels(y_pred, "boyangx")

if __name__ == '__main__':
    main()
```