Project 1 Report Name: Qiming Ying uniqname: yingqm

For code, please refer to appendix

2. Feature Extension

a) The number of unique words is 8483

b) The value of d is 8483

The average number of non-zero features per rating in the training data is 74.415

3.1 Hyperparameter Selection for a Linear-Kernel SVM

b) Since not maintaining class proportions may lead to classification bias. If each part of the data is not proportional. They tend to weight each instance equally which means overrepresented classes get too much weight, leading to classification bias.

d)

Performance Measure	C	Performance
accuracy	0.1	0.8719999999999999
f1-score	0.1	0.8721513941411461
AUROC	0.01	0.9399200000000001
precision	0.1	0.8730277135145557
sensitivity	0.001	0.898
specificity	0.1	0.8719999999999999

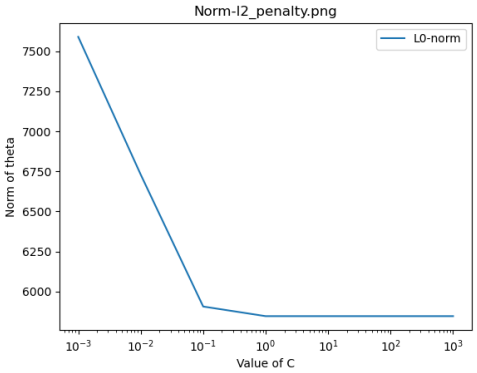
For accuracy, f1-score, precision and specificity, C with the best performance is 0.1 For AUROC, C with the best performance is 0.01. For sensitivity, C with the best performance is 0.001

I would choose accuracy as the performance measure. Since it is a balanced dataset, and the best way to tell the effectiveness of a model is to check the proportion of data that is correctly classified, which is the definition of accuracy performance measure.

e) I would use C=0.1.

Performance Measure	Performance
accuracy	0.89
f1-score	0.8906560636182903
AUROC	0.9417119999999998
precision	0.8853754940711462
sensitivity	0.896
specificity	0.884

g)



h)

Positive coefficient	Word	Negative coefficient	Word
0.6014089060331154	great	-0.4511754287602237	not
0.5273428235727176	excellent	-0.3224441819723553	worst
0.500723332804897	amazing	-0.27569270875905316	rude
0.4264157945565001	delicious	-0.2538363977269925	cold

i) I heard from my friends that it is a great restaurant but actually it is not amazing at all, even though it has an excellent environment, the food here is not very delicious.

3.2 Hyperparameter Selection for a Quadratic-Kernel SVM

b)

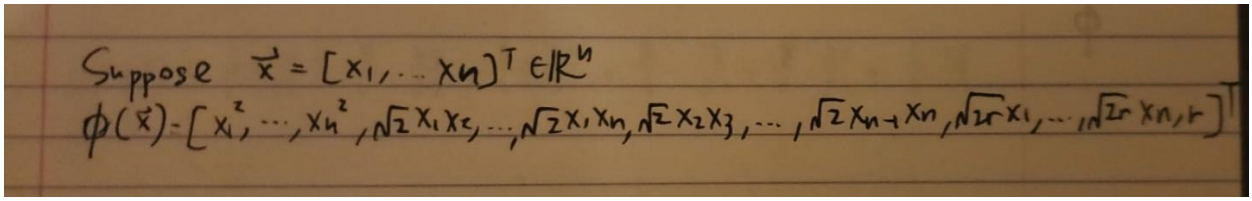
Tuning Scheme	C	r	AUROC
Grid Search	0.001	1000.0	0.9444799999999999
Random Search	0.0013289448722869186	659.8711072054074	0.94492

Grid search's advantage is exhaustive search, will find the absolute best way to tune the hyperparameters based on the training set. Its disadvantage is time-consuming.

Random search's advantage is to reduce chance of overfitting and faster than grid search. Its disadvantage is there is a lot of potential for variance.

3.3 Learning Non-linear Classifiers with a Linear-Kernel SVM

a)



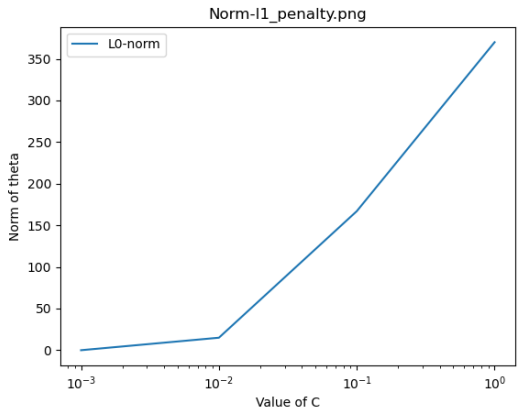
b) Pros: More straightforward, easy to understand

Cons: The transformed dimension may be much larger than original dimension, leading to increase computation

3.4 Linear-Kernel SVM with L1 Penalty and Squared Hinge Loss

a) C=0.1, performance=0.92452

b)



c) We find out that as for increasing C value, the norm of theta has different trends in terms of l1 penalty and l2 penalty, as l1-penalty's curve going up while l2-penalty's curve going down.

As C increases, the penalty on misclassification gets higher, which means that the model tends to be more complex with more non-zero features coefficient in order to avoid penalties.

For L1-norm, since the gradient of l1-norm does not change, thus increasing C does not influence the optimization problem, leading to norm of theta increases.

For L2-norm, since the gradient of l2-norm increases as the theta increases, we need to decrease theta to minimize the whole function. Therefore, the sparsity of theta in norm-l1 goes up and the sparsity of theta in norm -l2 goes down.

d) Squared hinge loss function will impose more penalty for the misclassification, since it is the square of (1-z) when (1-z) is greater than 1, and impose less penalty for not misclassified, but still causing hinge loss points, since at that time (1-z) is between 0 and 1 so the square of 1-z is less than 1-z. For well classified points, there is no difference between hinge loss function and squared hinge loss function since the penalty are all zero.

4.1 Arbitrary class weigh

a) The modification leads to the different penalty on misclassified points that is predicted as positive or negative. If Wn is much greater than Wp, the model will well classify points with -1 label, since a misclassified -1 label point leads to great penalty.

c)

Performance Measures	Performance
Accuracy	0.828
F1-score	0.8036529680365296
AUROC	0.937168
Precision	0.9361702127659575
Sensitivity	0.704
Specificity	0.952

d) Sensitivity is affected most by the new class weight since with the much higher penalty on negative points over positive points, the correctly classified positive point will be decrease in order to maximize the correctly classified negative points, leading to the huge decrease on sensitivity.

4.2 Imbalanced data

a)

Class Weights	Performance measures	Performance
Wn = 1, Wp = 1	Accuracy	0.852
Wn = 1, Wp = 1	F1-score	0.9153318077803204
Wn = 1, Wp = 1	AUROC	0.9174000000000001
Wn = 1, Wp = 1	Precision	0.8438818565400844
Wn = 1, Wp = 1	Sensitivity	1.0
Wn = 1, Wp = 1	Specificity	0.26

b) Since the data has 800 positive and 200 negative data points, still setting class weight to be 1:1 leads to the bias of classification. As shown in the chart, the sensitivity (true positive rate) gets improved and specificity (true negative rate) get decreased, because the penalty of predicting a negative label as positive is lower than it should be.

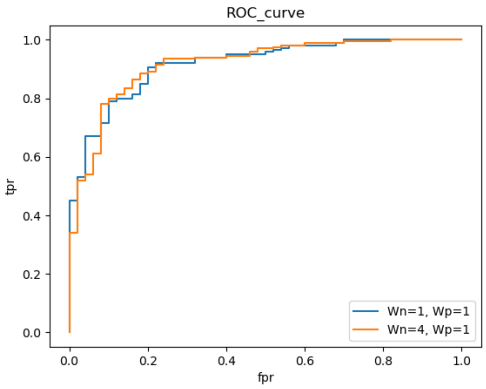
4.3 Choosing appropriate class weights

a) The appropriate class weight should be Wn=4, Wp=1 since the rate of positive points over negative points is 4:1. I would choose AUROC as the measure strategy. Since in this case, we want to make sure that the classification is not biased, and the auroc and represents the trade off between positive true positive rate and true negative rate, which means that it can represent if the classification is good for both positive and negative points.

b)

Class Weights	Performance Measures	Performance
Wn = 0.8 Wp = 0.2	accuracy	0.892
Wn = 0.8 Wp = 0.2	F1-score	0.9319899244332494
Wn = 0.8 Wp = 0.2	AUROC	0.9172999999999999
Wn = 0.8 Wp = 0.2	precision	0.9390862944162437
Wn = 0.8 Wp = 0.2	sensitivity	0.925
Wn = 0.8 Wp = 0.2	specificity	0.76

4.4 The ROC Curve



5. Challenge

1) I used tf-idf to score each word in a review text. We combine the reviews from dataset.csv and heldout.csv, generating a list of all strings. Then we use TfidfVectorizer in sklearn to generate a matrix of tf-idf scores. Then split the matrix into two parts, one is the data in dataset.csv, one is the data in heldout.csv. We train the model using data in dataset.csv, and then predict the reviews in heldout.csv.

The reason I prefer tf-idf score is that it represents the importance of a word in a corpus. Thus, it can to some extent, indicate information about what is meaningful in terms of predicting labels and what is not.

I also considered using N-grams, since multiple words together may be more meaningful. However, N-grams greatly increase the length of dictionary, lower down the efficiency since N-words appear together twice in a text is unusual. Thus, I decided not to use N-grams.

For hyperparameter selection, I used K-folds to find the relatively optimal hyperparameter value 1 in a range of [0.001, 0.1, 1, 10, 100, 100]. When C is 1, the model has the best performance.

For the algorithm selection, I chose linear kernel since it is much faster than quadratic kernel. Also, using quadratic kernel does not significantly increase the performance. Therefore, I chose LinearSVC().

I chose ovr, since for ovr, if there are n classes, I just need to calculates n times, while for ovo I need calculate n(n-1)/2 times. Therefore, ovr is much faster than ovo.

3) Facial Recognition

a) Problem: training a large dataset of human's faces, each human has many different pics of their faces. Given a pic of face belongs to a specific person in the dataset but is different from all the pics in the dataset. Use machine learning to recognize the person whom the face is belong to.

b) link: https://www.kaggle.com/kpvisionlab/tufts-face-database

c) First normalize all the pics with the same size, same pixels. For each pic, list the gray level of all pixels into a vector, and then use SVM to classify. For a new pic, just use the same strategy in previous question to predict the owner of the pic. Also, we can use similar schemes to predict the emotion of one face.

Appendix:

```
# EECS 445 - Fall 2020
# Project 1 - project1.py

import pandas as pd
import numpy as np
import itertools
import string
import re

from sklearn.svm import SVC, LinearSVC
from sklearn.model_selection import StratifiedKFold
from sklearn import metrics
from matplotlib import pyplot as plt
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split


from helper import *

def extract_dictionary(df):
    """
    Reads a pandas 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
    """
    count = 0
    word_dict = {}
    for i in df['reviewText']:
        review = i
        for ch in review:
            if ch in string.punctuation:
                review = review.replace(ch, " ")
        words = review.split()
        for word in words:
            if word.lower() not in word_dict:
                word_dict[word.lower()] = count
                count = count + 1
    # TODO: Implement this function
    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 (# of reviews, # of words in dictionary).
    Input:
        df: dataframe that has the ratings and labels
        word_dict: dictionary of words mapping to indices
    Returns:
        a feature matrix of dimension (# of reviews, # of words in dictionary)
    """
    number_of_reviews = df.shape[0]
    number_of_words = len(word_dict)
    feature_matrix = np.zeros((number_of_reviews, number_of_words))
    row = 0
    for i in df['reviewText']:
        review = i
        for ch in review:
            if ch in string.punctuation:
                review = review.replace(ch, " ")
        words = review.split()
        for word in words:
            if word.lower() in word_dict:
                feature_matrix[row][word_dict[word.lower()]] = 1
        row = row + 1
    # TODO: Implement this function
    return feature_matrix

def challenge_feature_matrix(df, word_dict):
    # print(len(word_dict))
    stop_words = ['a', 'an', 'the', 'this', 'that', 'with', 'i', 'went', 'go', 'for', 'of'
, 'you', 'he', 'she', 'they', 'are', 'is']
    vectorizer = TfidfVectorizer(stop_words=stop_words)
    temp = np.array(df.values.tolist())
    corpus = temp[:, 1]

    print(len(corpus))
    new_corpus = []
    corpus = [review.lower() for review in corpus]
    # for i in string.punctuation:
    for review in corpus:
        temp = review
        for ch in temp:
            if ch in string.punctuation:
                temp = temp.replace(ch, " ")
        new_corpus.append(temp)
    print(len(new_corpus))
    feature_matrix = vectorizer.fit_transform(new_corpus)
```

```
    # for row in feature_matrix:
    #     for entry in row:
    #         if entry != 0:
    #             for
    # print(vectorizer.vocabulary_)
    return feature_matrix

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 == "auroc":
        return np.float64(metrics.roc_auc_score(y_true, y_pred, labels=[1, -1]))
    TP, FN, FP, TN = metrics.confusion_matrix(y_true, y_pred, labels=[1, -1]).ravel()
    if metric == "f1-score":
        return np.float64(2*TP/(2*TP+FP+FN))
    if metric == "accuracy":
        return np.float64((TP+TN)/(TP+TN+FP+FN))
    if metric == "precision":
        # return np.float64(metrics.precision_score(y_true, y_pred))
        return np.float64(TP/(TP+FP))
    if metric == "sensitivity":
        return np.float64(TP/(TP+FN))
    if metric == "specificity":
        return np.float64(TN/(TN+FP))

    # TODO: Implement this function
    # This is an optional but very useful function to implement.
    # See the sklearn.metrics documentation for pointers on how to implement
    # the requested metrics.

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
    """

    # TODO: Implement this function
    #HINT: You may find the StratifiedKFold from sklearn.model_selection
    #to be useful

    skf = StratifiedKFold(n_splits = k, shuffle = False)


    #Put the performance of the model on each fold in the scores array
    scores = []
    for train_index, test_index in skf.split(X, y):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        clf.fit(X_train, y_train)
        y_pred = []
        if metric == "auroc":
            y_pred = clf.decision_function(X_test)
        if metric != "auroc":
            y_pred = clf.predict(X_test)
        scores.append(performance(y_test, y_pred, metric=metric))


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

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.
    """
    # TODO: Optionally implement this helper function if you would like to
    # instantiate your SVM classifiers in a single function. You will need
    # to use the above parameters throughout the assignment.
    if penalty == 'l1':
        return LinearSVC(penalty=penalty, dual=False, C=c, class_weight='balanced')
    elif degree == 2:
        return SVC(kernel="poly", C=c, degree=2, coef0=r, class_weight=class_weight)
```

```
    elif degree == 1:
        # return LinearSVC(penalty=penalty, C=c, class_weight=class_weight)
        return SVC(kernel="linear", C=c, degree=1, class_weight=class_weight)

    # return LinearSVC(penalty=penalty, C=c, class_weight=class_weight)

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_val=0.0
    best_performance = 0.0
    # TODO: Implement this function
    #HINT: You should be using your cv_performance function here
    #to evaluate the performance of each SVM
    for c in C_range:
        clf = select_classifier(penalty = penalty, c = c, degree = 1)
        current_performance = cv_performance(clf=clf, X=X, y=y, k=k, metric=metric)
        # print(current_performance)
        if current_performance > best_performance:
            best_performance = current_performance
            best_C_val = c

    print("Best c value is " + str(best_C_val))
    print("Corresponding performance is " + str(best_performance))
    print()
    return best_C_val


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:
        clf = select_classifier(penalty=penalty, c=c, degree=1)
        clf.fit(X, y)
        # current_norm = clf.decision_function(X)
        # print(clf.intercept_)
        theta0 = 0
        for element in np.array(clf.coef_[0]):
            if element != 0:
                theta0 = theta0 + 1
        norm0.append(theta0)
    # TODO: Implement this part of the function
    #Here, for each value of c in C_range, you should
    #append to norm0 the L0-norm of the theta vector that is learned
    #when fitting an L2- or L1-penalty, degree=1 SVM to the data (X, y)

    # for c in C_range:


    #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,
        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')
            param_range: 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 values for a quadratic-kernel SVM that maximize
            the average 5-fold CV performance as a pair (C,r)
    """
    best_C_val,best_r_val = 0.0, 0.0
    best_performance = 0.0
    # TODO: Implement this function
    #HINT: You should be using your cv_performance function here
    #to evaluate the performance of each SVM
    for element in param_range:
        clf = select_classifier(degree=2, c = element[0], r=element[1])
        current_performance = cv_performance(clf=clf, X=X, y=y, k=k, metric=metric)
        # print(current_performance)
        if current_performance > best_performance:
            best_performance = current_performance
            best_C_val = element[0]
            best_r_val = element[1]
    # return best_C_val
    # TODO: Implement this function
    # Hint: This will be very similar to select_param_linear, except
    # the type of SVM model you are using will be different...
    print("auroc is " + str(best_performance))
    return best_C_val,best_r_val

def challenge():
    # 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_trainin
g_data_challenge()
    heldout_features = get_heldout_reviews(multiclass_dictionary)
    print(type(multiclass_features))
    print(type(heldout_features))
    print(multiclass_features.drop(['label'], axis=1))
    print(heldout_features)
    text = pd.concat([multiclass_features.drop(['label'], axis=1), heldout_features])
    print(text)
    feature_matrix = challenge_feature_matrix(text, multiclass_dictionary)
    multiclass_features = feature_matrix[:2250]
    print(multiclass_features.shape)
    heldout_features = feature_matrix[-1500:]
    print(multiclass_labels.shape)
    # print(multiclass_features)
    # print(multiclass_labels)
    # print(multiclass_dictionary)
    # print(heldout_features)
    # X_train, X_val, y_train, y_val = train_test_split(multiclass_features, multiclass_la
bels, train_size = 0.75)
    C_range = [1e-3, 1e-2, 1e-1, 1e0, 1e1, 1e2, 1e3]
```

```
    best_performance = 0
    best_c = 0
    for c in C_range:
        clf = LinearSVC(penalty='l2', dual=False, C=c)
        current_performance = cv_performance(clf, multiclass_features, multiclass_labels)
        if current_performance > best_performance:
            best_performance = current_performance
            best_c = c
    clf = LinearSVC(penalty='l2', dual=False, C=best_c)
    print("Best C is " + str(best_c))
    print(best_performance)
    clf.fit(multiclass_features, multiclass_labels)
    y_pred = clf.predict(heldout_features)
    generate_challenge_labels(y_pred, 'yingqm')
    # print(performance(y_val, y_pred))
    # clf = SVC(kernel='linear', penalty='l2', C=best_c)
    # clf.fit(multiclass_features, multiclass_labels)
    # y_pred = clf.predict(heldout_features)



def main():
    # Read binary data
    # NOTE: READING IN THE DATA WILL NOT WORK UNTIL YOU HAVE FINISHED
    #       IMPLEMENTING generate_feature_matrix AND extract_dictionary

    challenge()

    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)
    # print(X_train)
    # print(dictionary_binary)
    # TODO: Questions 2, 3, 4
    C_range = [1e-3, 1e-2, 1e-1, 1, 1e1, 1e2, 1e3]
    print("The value of d after extracting training data:")
    print(X_train.shape[1])
    print("• The average number of non-zero features per rating in the training data:")
    print(np.sum(X_train) / X_train.shape[0])
    print("==========================================")
    metric_list = ["accuracy", "f1-
score", "auroc", "precision", "sensitivity", "specificity"]
    C_dict = {}
    for metric in metric_list:
        print("For " + str(metric) + ":")
        C_dict[metric] = select_param_linear(X=X_train, y=Y_train, k=5, metric=metric, C_r
ange=C_range)
    # 3.1.d
    print("==========================================")
```

```
    print("3.1.e")
    clf = select_classifier(penalty='l2', c=0.1, class_weight='balanced')
    clf.fit(X_train, Y_train)
    for metric in metric_list:
        if metric == "auroc":
            y_pred = clf.decision_function(X_test)
        if metric != "auroc":
            y_pred = clf.predict(X_test)
        print("For " + metric + ", performance is " + str(performance(Y_test, y_pred, metr
ic=metric)))

    print("=============================================")


    print("=============================================")
    plot_weight(X_train, Y_train, penalty='l2', C_range=C_range)

    print("=============================================")
    clf = select_classifier(penalty='l2', c=0.1, degree=1, class_weight='balanced')
    clf.fit(X_train, Y_train)
    temp = np.array(clf.coef_[0])
    print(temp)
    print("Top 4 words with positive coefficients:")
    # print(type(np.argmax(temp)))
    # print(word_dict)
    for i in range(4):
        for key, value in dictionary_binary.items():
            if value == np.argmax(temp):
            # print(value)
                print(key + " " + str(temp.max()))
                temp[value] = 0
                break

    print()
    print("Top 4 words with negative coefficients:")
    for i in range(4):
        for key, value in dictionary_binary.items():
            if value == np.argmin(temp):
              # print(value)
                print(key + " " + str(temp.min()))
                temp[value] = 0
                break
    # 3.2
    print("=============================================")
    r_range = [1e-3, 1e-2, 1e-1, 1, 1e1, 1e2, 1e3]
    parameter = np.zeros((49,2))
    count = 0
    for c in C_range:
        for r in r_range:
```

```
            parameter[count] = np.array([c, r])
            count = count + 1
    best_auroc_performance = 0
    print("Tuning Scheme: Grid Search")
    result = select_param_quadratic(X=X_train, y=Y_train, k=5, metric="auroc", param_range
=parameter)

    print("C is " + str(result[0]))
    print("r is " + str(result[1]))
    print()

    np.random.seed(42)
    parameter = np.full(shape=(25,2), fill_value=10)
    parameter = parameter ** np.random.uniform(-3, 3, size=(25,2))
    print("Tuning Scheme: Random Search")
    result = select_param_quadratic(X=X_train, y=Y_train, k=5, metric="auroc", param_range
=parameter)

    print("C is " + str(result[0]))
    print("r is " + str(result[1]))
    print()

    # 3.4.a
    print("============================================")

    C_range = [1e-3, 1e-2, 1e-1, 1]
    best_auroc_performance = 0
    best_C_val = 0
    for c in C_range:
        clf = LinearSVC(penalty='l1', dual=False, C=c, class_weight='balanced')
        clf.fit(X_train, Y_train)
        current_performance = cv_performance(clf, X_train, Y_train, metric="auroc")
        if current_performance > best_auroc_performance:
            best_auroc_performance = current_performance
            best_C_val = c
    print("Best C value is " + str(best_C_val))
    print("Best performance is " + str(best_auroc_performance))
    print()
    print("============================================")
    plot_weight(X_train, Y_train, penalty='l1', C_range=C_range)

    # 4.1.b
    print("============================================")

    clf = select_classifier(penalty='l2', c=0.01, degree=1, class_weight={-1:10,1:1})
    clf.fit(X_train, Y_train)
    for metric in metric_list:
        print("For " + metric + ", performance is " + str(cv_performance(clf, X_train, Y_t
rain, metric=metric)))
```

```
    print("For test 4.1.c")
    for metric in metric_list:
        y_pred = []
        if metric == "auroc":
            y_pred = clf.decision_function(X_test)
        if metric != "auroc":
            y_pred = clf.predict(X_test)
        print("For " + metric + ", performance is " + str(performance(Y_test, y_pred, metr
ic=metric)))

    print()
    # 4.2.a
    print("============================================")
    clf = select_classifier(penalty='l2', c=0.01, degree=1, class_weight={-1:1,1:1})
    clf.fit(IMB_features, IMB_labels)
    for metric in metric_list:
        if metric == "auroc":
            y_pred = clf.decision_function(IMB_test_features)
            print("For " + metric + ", performance is " + str(performance(IMB_test_labels,
 y_pred, metric=metric)))
        if metric != "auroc":
            y_pred = clf.predict(IMB_test_features)
            print("For " + metric + ", performance is " + str(performance(IMB_test_labels,
 y_pred, metric=metric)))

    # 4.3
    best_performance = 0
    best_C_val = 0
    for c in C_range:
        clf = select_classifier(penalty='l2', c=c, degree=1, class_weight={-1:4,1:1})
        clf.fit(IMB_features, IMB_labels)
        # y_pred = clf.predict(IMB_test_features)
        current_performance = cv_performance(clf, IMB_features, IMB_labels, metric="specif
icity")
        if current_performance > best_performance:
            best_C_val = c
            best_performance = current_performance
    print(best_C_val)

    clf = select_classifier(penalty='l2', c=best_C_val, class_weight={-1:4,1:1})
    clf.fit(IMB_features, IMB_labels)
    y_pred = []
    for metric in metric_list:
        if metric == "auroc":
            y_pred = clf.decision_function(IMB_test_features)
        if metric != "auroc":
            y_pred = clf.predict(IMB_test_features)
        print("For " + metric + ", performance is " + str(performance(IMB_test_labels, y_p
red, metric=metric)))
```

```
    clf1 = select_classifier(penalty='l2', c=0.01, degree=1, class_weight={-1:1,1:1})
    clf2 = select_classifier(penalty='l2', c=0.01, degree=1, class_weight={-1:4,1:1})
    clf1.fit(IMB_features, IMB_labels)
    clf2.fit(IMB_features, IMB_labels)
    fpr_1, tpr_1, thresholds = metrics.roc_curve(y_true=IMB_test_labels, y_score=clf1.deci
sion_function(IMB_test_features))
    fpr_2, tpr_2, thresholds = metrics.roc_curve(y_true=IMB_test_labels, y_score=clf2.deci
sion_function(IMB_test_features))
    plt.plot(fpr_1, tpr_1, label="Wn=1, Wp=1")
    plt.plot(fpr_2, tpr_2, label="Wn=4, Wp=1")
    plt.xlabel("fpr")
    plt.ylabel("tpr")
    plt.title("ROC_curve")
    plt.legend()
    plt.savefig("roc_curve.png")




    # 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_train
ing_data()
    # heldout_features = get_heldout_reviews(multiclass_dictionary)




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

In helper.py

```
def get_multiclass_training_data_challenge(class_size=750):
    """
    Reads in the data from data/dataset.csv and returns it using
    extract_dictionary and generate_feature_matrix as a tuple
    (X_train, Y_train) where the labels are multiclass as follows
        -1: poor
         0: average
         1: good
    Also returns the dictionary used to create X_train.
    Input:
        class_size: Size of each class (pos/neg/neu) of training dataset.
    """
    fname = "data/dataset.csv"
    dataframe = load_data(fname)
    neutralDF = dataframe[dataframe['label'] == 0].copy()
```

```
    positiveDF = dataframe[dataframe['label'] == 1].copy()
    negativeDF = dataframe[dataframe['label'] == -1].copy()
    X_train = pd.concat([positiveDF[:class_size], negativeDF[:class_size], neutralDF[:clas
s_size]]).reset_index(drop=True).copy()
    dictionary = project1.extract_dictionary(X_train)
    Y_train = X_train['label'].values.copy()
    # X_train = project1.challenge_feature_matrix(X_train, dictionary)

    return (X_train, Y_train, dictionary)


def get_heldout_reviews(dictionary):
    """
    Reads in the data from data/heldout.csv and returns it as a feature
    matrix based on the functions extract_dictionary and generate_feature_matrix
    Input:
        dictionary: the dictionary created by get_multiclass_training_data
    """
    fname = "data/heldout.csv"
    dataframe = load_data(fname)
    # print(dataframe.values)
    # neutralDF = dataframe[dataframe['label'] == 0].copy()
    # positiveDF = dataframe[dataframe['label'] == 1].copy()
    # negativeDF = dataframe[dataframe['label'] == -1].copy()
    # X_train = pd.concat([positiveDF[:class_size], negativeDF[:class_size], neutralDF[:cl
ass_size]]).reset_index(drop=True).copy()
    # X = project1.challenge_feature_matrix(dataframe, dictionary)
    return dataframe
```