UNIVERSITY OF MICHIGAN

EECS 445 – Introduction to Machine Learning

Project 1 - Naitian's Super South Pole SVMs

Written by: Yigao Fang

Unique name: fgsepter

Feb.10^{th}, 2021

1 Introduction

2 Feature Extraction

2.1 Implementing the `extract_dictionary(df)` function

```
def extract_dictionary(df):
    word_dict = {}
    words = []
    num = 0
    for review in df['reviewText']:
        review = review.lower()
        for word in review:
            if word in string.punctuation:
                review = review.replace(word, " ")
        words = review.split()
        for word in words:
            if (word in word_dict) == False:
                word_dict[word] = num
                num = num+1
    return word_dict
```

2.2 Implementing the `generate_feature_matrix(df, word_dict)` function

```
def generate_feature_matrix(df, word_dict):
    number_of_reviews = df.shape[0]  # n
    number_of_words = len(word_dict) # d
    feature_matrix = np.zeros((number_of_reviews, number_of_words))
    for i, review in enumerate(df['reviewText']):
        review = review.lower()
        for word in review:
            if word in string.punctuation:
                review = review.replace(word, " ")
        words = review.split()
        for word in words:
            if word in word_dict:
                feature_matrix[i][word_dict[word]] = 1
    return feature_matrix
```

2.3 Apply the `get_split_binary_data(class_size=500)` function

```
def get_split_binary_data(class_size=500):
    fname = "data/dataset.csv"
    dataframe = load_data(fname)
    dataframe = dataframe[dataframe['label'] != 0]
    positiveDF = dataframe[dataframe['label'] == 1].copy()
    negativeDF = dataframe[dataframe['label'] == -1].copy()
    X_train = pd.concat([positiveDF[:class_size], negativeDF[:class_size]]).
                                                reset_index(drop=True).copy()
    dictionary = project1.extract_dictionary(X_train)
    X_test = pd.concat([positiveDF[class_size:(int(1.5 * class_size))],
                        negativeDF[class_size:(int(1.5 * class_size))]]).reset_index(
                                                                drop=True).copy()
    Y_train = X_train['label'].values.copy()
    Y_test = X_test['label'].values.copy()
    X_train = project1.generate_feature_matrix(X_train, dictionary)
    X_test = project1.generate_feature_matrix(X_test, dictionary)
    # The code for part 2c #
    print("The value of d is: ", len(dictionary))
    sum = 0
    for row in X_train:
        for col in row:
            sum = sum + col
    print("The number of non-zero features per rating in the training data is: ", sum
                                                /(2*class_size))
    word_dict = {}
    word_freq = {}
    for i in dictionary:
        word_dict[dictionary[i]] = i
        word_freq[dictionary[i]] = 0
    for i,row in enumerate(X_train):
        for j,col in enumerate(row):
            if col == 1:
                word_freq[j] = word_freq[j]+1
    maxnum = 0
    maxword = 0
    for num in word_freq:
        if word_freq[num]>maxnum:
            maxnum = word_freq[num]
            maxword = num
    print("The word appearing in the most number of reviews is: ", word_dict[maxword]
                                                )
    # The code for part 2c #
    return (X_train, Y_train, X_test, Y_test, dictionary)
```

The above algorithm prints:

1. The value of d is: 7508
2. The number of non-zero features per rating in the training data is: 53.794
3. The word appearing in the most number of reviews is: the

3 Hyperparameter and Model Selection

3.1 Hyperparameter Selection for a Linear-Kernel SVM

(a):

```
def performance(y_true, y_pred, metric="accuracy"):
    value = 0
    if metric == "auroc":
        value = metrics.roc_auc_score(y_true, y_pred)
    else:
        CM = metrics.confusion_matrix(y_true, y_pred, labels=[1,-1])
        if metric == "accuracy":
            value = (CM[0][0]+CM[1][1])/(CM[0][0]+CM[1][1]+CM[0][1]+CM[1][0])
        if metric == "f1-score":
            value = 2*CM[0][0]/(2*CM[0][0]+CM[0][1]+CM[1][0])
        if metric == "precision":
            value = CM[0][0]/(CM[0][0]+CM[1][0])
        if metric == "sensitivity":
            value = CM[0][0]/(CM[0][0]+CM[0][1])
        if metric == "specificity":
            value = CM[1][1]/(CM[1][1]+CM[1][0])
    return value.astype(np.float64)

def cv_performance(clf, X, y, k=5, metric="accuracy"):
    scores = []
    skf = StratifiedKFold(n_splits=k, shuffle=False)
    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)
        if metric=="auroc":
            y_pred = clf.decision_function(X_test)
        else:
            y_pred = clf.predict(X_test)
        scores = np.append(scores, performance(y_test, y_pred, metric) )
    return np.array(scores).mean()
```

(b):

Each fold will have similar proportion as the entire dataset, and hence they are similar to each other and can represent the entire dataset well, which lead to small bias.

On the contrary, if the class proportions are not maintained, the performance C may vary a lot among different folds. This is not conducive to our analysis of the average performance between several fold.

(c):

```
def select_classifier(penalty='l2', c=1.0, degree=1, r=0.0, class_weight='balanced'):
    if penalty == 'l2':
        if degree == 1:
            return SVC(C=c, kernel = 'linear', class_weight=class_weight)
        else:
            return SVC(C=c, coef0 = r, kernel = 'poly', degree=degree, class_weight=
                                                    class_weight, gamma='auto')
    else:
        return LinearSVC(penalty='l1', dual=False, C=c, class_weight=class_weight)

def select_param_linear(X, y, k=5, metric="accuracy", C_range = [], penalty='l2'):
    best_C_val=0.0
    max_score = 0
    for c in C_range:
        #clf = SVC(C=c, kernel = 'linear')
        clf = select_classifier(penalty=penalty, c=1.0, degree=1, r=0.0, class_weight
                                                    ='balanced')
        performance = cv_performance(clf,X,y,k,metric)
        print("Score for ", c , " under ", metric, " is ", performance)
        if performance > max_score:
            max_score = performance
            best_C_val = c
    return best_C_val
```

(d):

```
def Question3_1_d():
    X_train, Y_train, X_test, Y_test, dictionary = get_split_binary_data(500)
    metrics = ["accuracy", "f1-score","precision","sensitivity","specificity","auroc"
                                                    ]
    for metric in metrics:
        best_c = select_param_linear(X_train, Y_train, k=5, metric=metric, C_range =
                                                    [0.001,0.01,0.1,1,10,100,1000],
                                                    penalty='l2')
        print("best_c for ", metric, " is ", best_c )
```

Table 1: The Performance for Different Values of C

Performance Measures	C	Performance
Accuracy	0.01	0.817
F1-Score	0.01	0.816
AUROC	0.1	0.90474
Precision	0.1	0.82526
Sensitivity	0.001	0.89
Specificity	0.1	0.828

When the value of C increases, the performance first increase and then decrease when the value of C is getting too large. After some values of C, the performance relatively maintain unchanged.

I would choose *Precision* to optimize for when choosing C. *Precision* is calculated as:

$$\frac{TruePositive}{TruePositive + FalsePositive}$$

This can give us a proportion of correctly predicted positive points with all the points that predicted to be positive, which tells us the accuracy of predicting positive. Also, the performance of *Precision* with respect to C has a easy-recognizable trend. The performance for different C when using *Precision* is recorded in the following table:

Table 2: The Performance for Different Values of C when Choosing *Precision*

C	**Performance**
0.001	0.64689
0.01	0.82099
0.1	0.82526
1	0.80730
10	0.80730
100	0.80730
1000	0.80730

According to the above table, I notice that when the value of C increase, the performance first experience a huge increasing from *0.065* at `C = 0.001` to *0.1* at `C = 0.1`. Then the performance decreases with the increasing of C and maintains stable. This trend is very obvious from the table, and hence we can choose `Precision` when finding C. From here I know that *0.1* is much better than other values of C listed in the table, and I will use this value in the next question.

(e):

Table 3: The Performance for $C = 0.1$

Performance Measures	Performance
Accuracy	0.836
F1-Score	0.834677
AUROC	0.910912
Precision	0.8414634
Sensitivity	0.828
Specificity	0.844

```
def Question3_1_e(c): # Choose different C values here
    clf = select_classifier(c = c)
    X_train, y_train, X_test, y_test, dictionary = get_split_binary_data(500)
    clf.fit(X_train,y_train)
    metrics = ["accuracy", "f1-score","precision","sensitivity","specificity","auroc"
                                                                    ]
    for metric in metrics:
        if metric=="auroc":
            y_pred = clf.decision_function(X_test)
        else:
            y_pred = clf.predict(X_test)
        scores = performance(y_test, y_pred, metric)
        print("Performance for c = ", c, " under ", metric, " is ", scores)
```

(f):

```
def plot_weight(X,y,penalty,C_range):
    norm0 = []
    for c in C_range:
        clf = select_classifier(penalty=penalty, c=c, degree=1)
        clf.fit(X,y)
        theta = clf.coef_[0]
        L = 0
        for t in theta:
            if t != 0:
                L = L + 1
        norm0 = np.append(norm0, L)
    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()
```

(g):

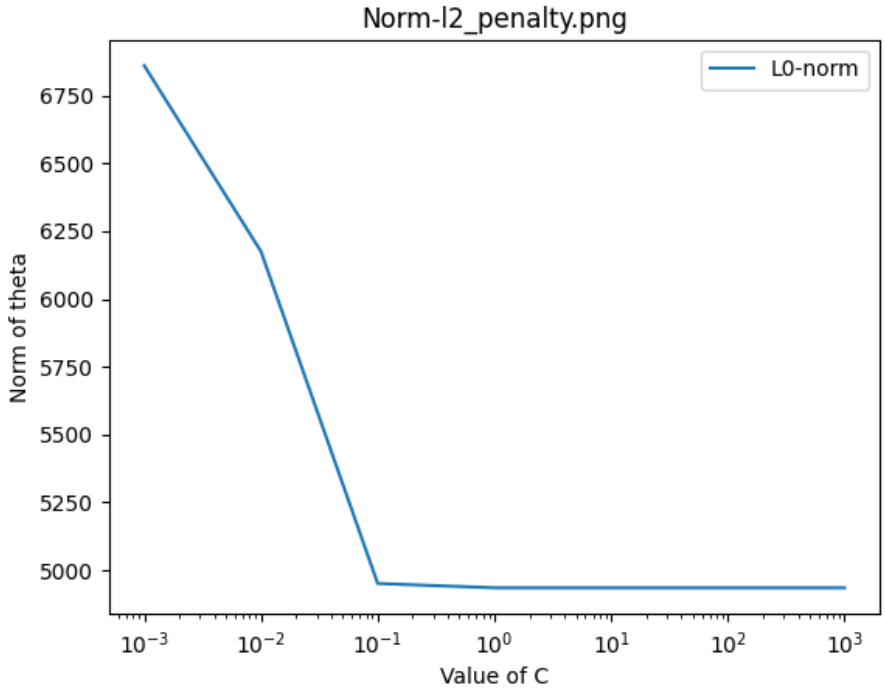

Figure 3.1: `L0-norm` of θ with respect to C under penalty `l2`

(h):

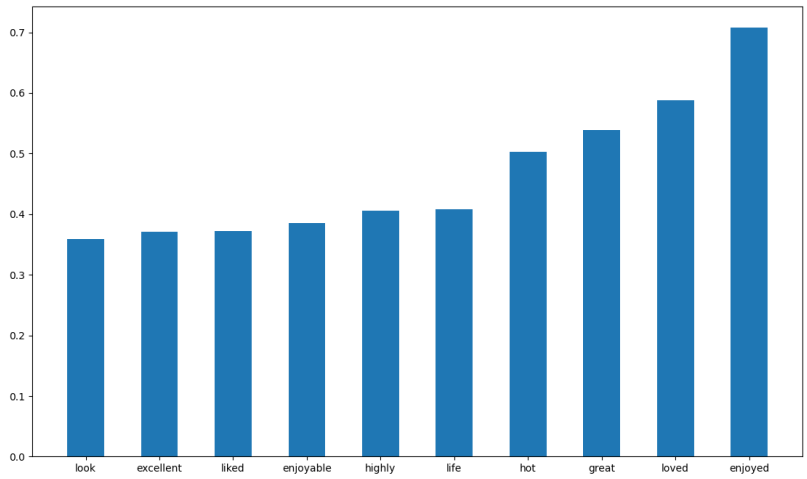

Figure 3.2: Most positive 10 words

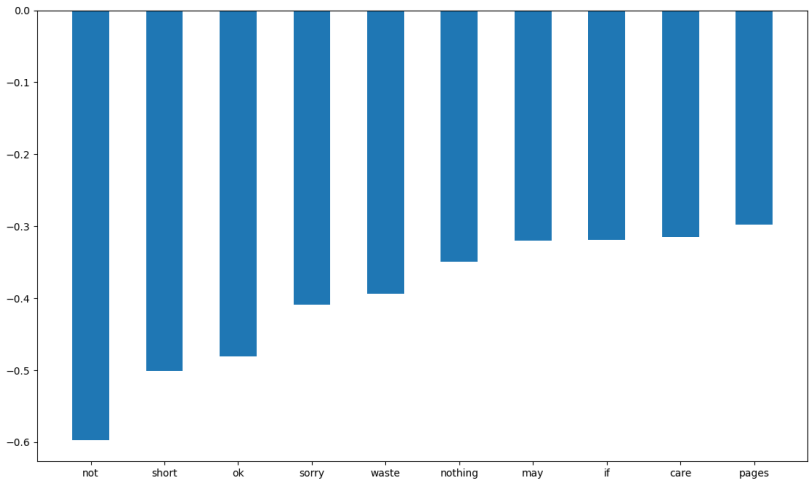

Figure 3.3: Most negative 10 words

The Python Code for this question is in next page.

```
def Question3_1_h(char):
    X_train, y_train, X_test, y_test, dictionary = get_split_binary_data(500)
    clf = SVC(C=0.1, kernel = 'linear')
    clf.fit(X_train,y_train)
    theta = clf.coef_[0]
    x,y = []
    new_dic = {v:k for k,v in dictionary.items()}
    if char == "positive":
        for i in range(10):
            max_t,max_index = 0
            for index, t in enumerate(theta):
                if t > max_t:
                    if (new_dic[index] in x) == False:
                        max_t = t
                        max_index = index
            x.append(new_dic[max_index])
            y.append(max_t)
        x.reverse()
        y.reverse()
    else:
        for i in range(10):
            min_t,max_index = 0
            for index, t in enumerate(theta):
                if t < min_t:
                    if (new_dic[index] in x) == False:
                        min_t = t
                        max_index = index
            x.append(new_dic[max_index])
            y.append(min_t)
    plt.bar(x, height = y, width = 0.5)
    plt.show()
```

(i):

I never **liked** this kind of book with bland story. You can only **look** the previous plot and then guess the entire whole story. No one would agree with that the design of a detective book is **great**. There are many **excellent** detective stories, why should I waste time reading this boring book?

3.2 Hyperparameter Selection for a Quadratic-Kernel SVM

(a):

```
def select_param_quadratic(X, y, k=5, metric="accuracy", param_range=[]):
    best_C_val,best_r_val = 0.0, 0.0
    max_score = 0
    for i in range(len(param_range)):
        c,r = param_range[i]
        clf = select_classifier(penalty='l2', c=c, degree=2, r=r)
        performance = cv_performance(clf,X,y,k,metric)
        print("Score for c = ", c,  " , r = ", r , " under ", metric, " is ",
                                                    performance)
        if performance > max_score:
            max_score = performance
            best_C_val = c
            best_r_val = r
    return best_C_val,best_r_val
```

(b):

```
def Question3_2_b(type, metric):
    X_train, y_train, X_test, y_test, dictionary = get_split_binary_data(500)
    if type == "grid":
        C, R = [0.001, 0.01, 0.1, 1, 10, 100, 1000]
        param_range = []
        for c in C:
            for r in R:
                param_range.append([c,r])
        best_C_val,best_r_val = select_param_quadratic(X_train, y_train, k=5, metric=
                                                    metric, param_range=param_range)
        print("Best C is: ", best_C_val)
        print("Best r is: ", best_r_val)
    elif type == 'random':
        C = np.random.uniform(-3,3,25)
        R = np.random.uniform(-3,3,25)
        param_range = []
        for i in range(25):
            param_range.append([10**C[i],10**R[i]])
        best_C_val,best_r_val = select_param_quadratic(X_train, y_train, k=5, metric=
                                                    metric, param_range=param_range)
        print("Best C is: ", best_C_val)
        print("Best r is: ", best_r_val)
    else:
        print("Error: Wrong Type!")
```

Table 4: Best C and r under Grid Search and Random Search Using CV

Turning Scheme	C	r	AUROC
Grid Search	10	10	0.90608
Random Search	825.1245290065642	0.13005772242050392	0.90612

Using the above best C and r obtained by Grid Search and Random Search, I can calculate the AUROC value for the test dataset, listing below in Table 5:

Table 5: Best C and r under Grid Search and Random Search for Test Dataset

Turning Scheme	C	r	AUROC
Grid Search	10	10	0.91314
Random Search	825.1245290065642	0.13005772242050392	0.913536

```
Score for c =  0.001  , r =  0.001  under  auroc  is  0.8382999999999999
Score for c =  0.001  , r =  0.01  under  auroc  is  0.8479000000000001
Score for c =  0.001  , r =  0.1  under  auroc  is  0.85002
Score for c =  0.001  , r =  1  under  auroc  is  0.8501999999999998
Score for c =  0.001  , r =  10  under  auroc  is  0.8501999999999998
Score for c =  0.001  , r =  100  under  auroc  is  0.8501999999999998
Score for c =  0.001  , r =  1000  under  auroc  is  0.8501999999999998
Score for c =  0.01  , r =  0.001  under  auroc  is  0.8382999999999999
Score for c =  0.01  , r =  0.01  under  auroc  is  0.8479000000000001
Score for c =  0.01  , r =  0.1  under  auroc  is  0.85002
Score for c =  0.01  , r =  1  under  auroc  is  0.8501999999999998
Score for c =  0.01  , r =  10  under  auroc  is  0.8501999999999998
Score for c =  0.01  , r =  100  under  auroc  is  0.8501999999999998
Score for c =  0.01  , r =  1000  under  auroc  is  0.86516
Score for c =  0.1  , r =  0.001  under  auroc  is  0.8382999999999999
Score for c =  0.1  , r =  0.01  under  auroc  is  0.8479000000000001
Score for c =  0.1  , r =  0.1  under  auroc  is  0.85002
Score for c =  0.1  , r =  1  under  auroc  is  0.8501999999999998
Score for c =  0.1  , r =  10  under  auroc  is  0.8501999999999998
Score for c =  0.1  , r =  100  under  auroc  is  0.8652599999999999
Score for c =  0.1  , r =  1000  under  auroc  is  0.9056
Score for c =  1  , r =  0.001  under  auroc  is  0.8382999999999999
Score for c =  1  , r =  0.01  under  auroc  is  0.8479000000000001
Score for c =  1  , r =  0.1  under  auroc  is  0.85002
Score for c =  1  , r =  1  under  auroc  is  0.8501999999999998
Score for c =  1  , r =  10  under  auroc  is  0.86528
Score for c =  1  , r =  100  under  auroc  is  0.90588
Score for c =  1  , r =  1000  under  auroc  is  0.8918800000000001
Score for c =  10  , r =  0.001  under  auroc  is  0.8382999999999999
Score for c =  10  , r =  0.01  under  auroc  is  0.8479000000000001
Score for c =  10  , r =  0.1  under  auroc  is  0.85002
Score for c =  10  , r =  1  under  auroc  is  0.86528
Score for c =  10  , r =  10  under  auroc  is  0.9060799999999999
Score for c =  10  , r =  100  under  auroc  is  0.8925599999999999
Score for c =  10  , r =  1000  under  auroc  is  0.8863
Score for c =  100  , r =  0.001  under  auroc  is  0.8382999999999999
Score for c =  100  , r =  0.01  under  auroc  is  0.8479000000000001
Score for c =  100  , r =  0.1  under  auroc  is  0.86602
Score for c =  100  , r =  1  under  auroc  is  0.90604
Score for c =  100  , r =  10  under  auroc  is  0.89208
Score for c =  100  , r =  100  under  auroc  is  0.8878599999999999
Score for c =  100  , r =  1000  under  auroc  is  0.8863
Score for c =  1000  , r =  0.001  under  auroc  is  0.8431
Score for c =  1000  , r =  0.01  under  auroc  is  0.86922
Score for c =  1000  , r =  0.1  under  auroc  is  0.9053800000000001
Score for c =  1000  , r =  1  under  auroc  is  0.89192
Score for c =  1000  , r =  10  under  auroc  is  0.8878199999999999
Score for c =  1000  , r =  100  under  auroc  is  0.8878599999999999
Score for c =  1000  , r =  1000  under  auroc  is  0.8863
Best C is:  10
Best r is:  10
```

Figure 3.4: Scores for different values of C and r

According to Figure **??**, we find that when the value of C increases, the performance scores first increase and then decrease. When the value of r increases, the performance scores also first increase and then decrease. Moreover, the higher the value of r, the more obvious of the phenomenon that performance score will first increase and then decrease with the increasing of C.

1. **The pros of Grid Search:** Grid Search is applicable to parameter search when the amount of parameters is relatively small. It can adjust parameters in order of step size within the specified parameter range. The parameter with the highest performance can be found within the range with high accuracy.

2. **The cons of Grid Search:** In the face of huge data set, large amount of parameters, or wide parameter range, this method is very time-consuming since every possibility needs to be enumerated.

3. **The pros of Random Search:**

   (a) The search is relatively fast for large dataset and more number of parameters.

   (b) We can conveniently set the number of searches to control the calculation and speed of parameter search.

   (c) It can search for larger amount of different values of each parameter, comparing with the Grid Search which only searches for a few value for each parameter.

4. **The cons of Random Search:** The accuracy is relatively low, and highly depends on randomness. The method generates numbers randomly, so the results given by this method can vary a lot when run multiple times.

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

(a):

Assume $\vec{x} = (x_1, x_2, ..., x_n)^T$:

$$\Phi(x) = (x_n^2, ..., x_1^2, \sqrt{2}x_n x_{n-1}, ..., \sqrt{2}x_n x_1, \sqrt{2}x_{n-1}x_{n-2}, ..., \sqrt{2}x_{n-1}x_1, ..., \sqrt{2}x_2 x_1, \sqrt{2r}x_n, ..., \sqrt{2r}x_1, r)^T$$

(b):

1. **The pros of explicit feature mapping:** This method can give a more precise and accurate boundary, and can easily classify some not linearly classifiable datasets by mapping them into higher dimensions. We could also not bothering for choosing parameters.

2. **The cons of explicit feature mapping:** It is much more complex, and the higher the dimension, the higher the complexity, and therefore it may be very time and memory consuming. It may also need approximations in order to calculate in higher dimension spaces.

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

(a):

Table 6: The AUROC performance for different values of C with `l1` penalty

C	Performance
0.001	0.8899
0.01	0.88978
0.1	0.89004
1	0.8898

According to Table **??**, we find the best C under AUROC is $C = 0.1$.

```
def Question3_4_a():
    X_train, Y_train, X_test, Y_test, dictionary = get_split_binary_data(500)
    best_c = select_param_linear(X_train, Y_train, k=5, metric="auroc", C_range = [0.
                                                    001,0.01,0.1,1], penalty='l1')
    print("best_c for auroc is ", best_c )
```

(b):

```
def Question3_4_b():
    X_train, Y_train, X_test, Y_test, dictionary = get_split_binary_data(500)
    plot_weight(X = X_train,y = Y_train,penalty='l1', C_range = [0.001,0.01,0.1,1])
```

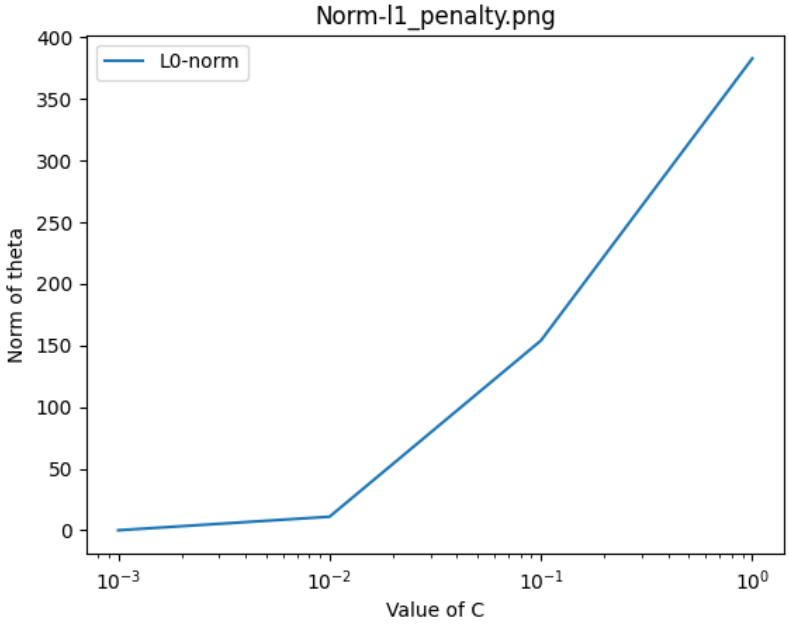

Figure 3.5: `L0-norm` of θ with respect to C under penalty `l1`

(c):

The norm with $l2$ penalty decreases when the value of C increases, while The norm with $l1$ penalty increases when the value of C. Also, the norm with $l2$ penalty is far larger than the norm with $l1$ penalty.

This is because that the gradient of $l1$ penalty with θ is constant, while the gradient of $l2$ penalty with θ is θ.

(d):

Consider the fact that when hinge loss less than 1, square will make the loss smaller; while when hinge loss greater than 1, square will make the loss larger. Hence the optimal solution will be in the center and more points within the boundary, and the hyperplane will be closer to the outliers.

3.5 Perceptron Classifier

(a):

```
def Question3_5_a():
    X_train, Y_train, X_test, Y_test, dictionary = get_split_binary_data(500)
    theta, b = train_perceptron(X_train,Y_train)
    Y_pred = []
    for i in range(X_test.shape[0]):
        if X_test[i].dot(theta) >= 0:
            Y_pred.append(1)
        else:
            Y_pred.append(-1)
    metrics = ["auroc", "accuracy", "f1-score", "precision", "sensitivity","
                                                specificity"]
    for metric in metrics:
        print("The performance of Perceptron algorithm under ", metric, " is ",
                                                    performance(Y_test,Y_pred,metric))
```

The output of this function is:

Table 7: The Performance for Perceptron algorithm

Performance Measures	Performance
Accuracy	0.818
F1-Score	0.815416
AUROC	0.818
Precision	0.82716
Sensitivity	0.804
Specificity	0.832

According to Table 7, the accuracy of perceptron algorithm is 0.818.

(b):

Comparing with Table 3 and Table 7, I notice that the performance of Perceptron algorithm under all of the six methods of measurement is smaller than the performance of SVM under the best choice of C, indicating that Perceptron performs worse than the SVM. For the accuracy, SVM has an accuracy of 0.836, while Perceptron algorithm is only 0.818.

Perceptron algorithm is only able to find one boundary that can classify most of the data correctly. However, it is unable to tell whether there is a better boundary or model that could classify the data points more accurately. SVM, on the other hand, compare different models under different parameters and choose the one that has the best performance, as what we did previously. Hence SVM will perform better and give more precise results.

4 Asymmetric Cost Functions and Class Imbalance

4.1 Arbitrary class weights

(a):

This modification will affect the weight of the positive and negative points, meaning that the misclassification of positive or negative points does not contribute to our result with the same proportion. We can change the class weights to indicate whether we care more about the positive points or negative points.

If W_n is much greater than W_p, we care more about the negative point. We would prefer to misclassify much more positive points rather than misclassify only one negative point. This is because that W_n is the coefficient of the negative terms, and larger W_n indicating that a small increase of $C\sum_{i|y_i=-1}\epsilon_i$ will increase the whole function a lot.

(b):

```
def Question4_1_b(X_train, Y_train, X_test, Y_test, dictionar, metric):
    clf = select_classifier(penalty='l2', c=0.1, degree=1, class_weight={-1:1,1:10})
    clf.fit(X_train,Y_train)
    if metric=="auroc":
        Y_pred = clf.decision_function(X_test)
    else:
        Y_pred = clf.predict(X_test)
    return Y_pred

def Question4_1_c():
    metrics = ["auroc", "accuracy", "f1-score", "precision", "sensitivity","
                                                specificity"]
    X_train, Y_train, X_test, Y_test, dictionary = get_split_binary_data(500)
    for metric in metrics:
        Y_pred = Question4_1_b(X_train, Y_train, X_test, Y_test, dictionary, metric)
        print("The performance of modified SVM under ", metric, " is ", performance(
                                                    Y_test,Y_pred,metric))
    pass
```

(c):

Table 8: The Performance for modified SVM

Performance Measures	Performance
Accuracy	0.824
F1-Score	0.828125
AUROC	0.904176
Precision	0.80916
Sensitivity	0.848
Specificity	0.8

(d):

Comparing with Table 3 and Table 8, I notice that *Specificity* was affected the most. This is due to the measuring equation of *Specificity*. It is calculated using the confusion matrix as:

$$Specificity = \frac{TN}{TN + FP}$$

In this case, the class weight of $-1 : 1$ is $1 : 10$, which means that we allow more negative points to be misclassified while decrease the number of misclassified positive points. Those misclassified negative points become FP, or $FalsePositive$, and significantly lower the value of the above expression, making the $Specificity$ affected the most.

4.2 Imbalanced data

(a):

```
IMB_features, IMB_labels = get_imbalanced_data(dictionary_binary)
IMB_test_features, IMB_test_labels = get_imbalanced_test(dictionary_binary)
def Question4_2_a(IMB_features, IMB_labels):
    clf = select_classifier(penalty='l2', c=0.1, degree=1, r=0.0, class_weight={-1:1,
                                                                                1:1})
    clf.fit(IMB_features, IMB_labels)
    return clf
```

(b):

```
def Question4_2_b(clf, IMB_test_features, IMB_test_labels):
    metrics = ["auroc", "accuracy", "f1-score", "precision", "sensitivity","
                                                specificity"]
    for metric in metrics:
        if metric=="auroc":
            Y_pred = clf.decision_function(IMB_test_features)
        else:
            Y_pred = clf.predict(IMB_test_features)
        print("The performance of SVM with imbalanced data under ", metric, " is ",
                                                    performance(IMB_test_labels,Y_pred
                                                    ,metric))
    pass
```

Table 9: The Performance for SVM with imbalanced data

Class Weight	Performance Measures	Performance
$W_n = 1, W_p = 1$	Accuracy	0.84
$W_n = 1, W_p = 1$	F1-Score	0.9029126
$W_n = 1, W_p = 1$	AUROC	0.8802
$W_n = 1, W_p = 1$	Precision	0.87735849
$W_n = 1, W_p = 1$	Sensitivity	0.93
$W_n = 1, W_p = 1$	Specificity	0.48

(c):

Comparing with Table 3 and Table 9, I find that the performance of the first five measuring metrics except for *Specificity* did not have a huge difference. Some of them become higher and some become lower. However, as for *Specificity*, the performance had a sharp decrease, from *0.844* to *0.48*

4.3 Choosing appropriate class weights

(a):

According to previous part, I notice that the performance given by metric *Specificity* experienced a huge decrease when training imbalanced dataset. Hence it is safe to assume that *Specificity* is informative in this situation and I will choose this metric to measure performance.

```
def Question4_3_a(IMB_features, IMB_labels):
    Best_Wn, Best_Wp, max_perf = 0, 0, 0
    Ns = [1,2,3,4,5]
    for Wn in Ns:
        for Wp in Ns:
            clf = select_classifier(penalty='l2', c=0.1, class_weight={-1:Wn,1:Wp})
            scores = []
            skf = StratifiedKFold(n_splits=5, shuffle=False)
            for train_index, test_index in skf.split(IMB_features,IMB_labels):
                X_train, X_test = IMB_features[train_index], IMB_features[test_index]
                y_train, y_test = IMB_labels[train_index], IMB_labels[test_index]
                clf.fit(X_train,y_train)
                Y_pred = clf.predict(X_test)
                scores.append(performance(y_test,Y_pred,"specificity"))
            perf = np.array(scores).mean()
            print("Wn: ",Wn," Wp: ",Wp, " Performance: " , perf)
            if perf > max_perf:
                max_perf = perf
                Best_Wn = Wn
                Best_Wp = Wp
    print("Best_Wn = ", Best_Wn)
    print("Best_Wp = ", Best_Wp)
    pass
```

We can use the above algorithm of Grid Search to determine the best setting for the class weights. 5-fold cross validation is applied to calculate the average performance. According to the two tables below, when $W_n = 3$, $W_p = 1$, the weights give the best performance with *Specificity* equals 0.57. Hence the weight parameters I choose are:

$$W_n = 3$$
$$W_p = 1$$

Class Weight	Specificity
$W_n = 1, W_p = 1$	0.51
$W_n = 2, W_p = 1$	0.55
$W_n = 3, W_p = 1$	0.57
$W_n = 4, W_p = 1$	0.57
$W_n = 5, W_p = 1$	0.565

Class Weight	Specificity
$W_n = 1, W_p = 1$	0.51
$W_n = 1, W_p = 2$	0.505
$W_n = 1, W_p = 3$	0.505
$W_n = 1, W_p = 4$	0.505
$W_n = 1, W_p = 5$	0.505

(b):

Table 10: The Performance for SVM with imbalanced data with $W_n = 3$, $W_p = 1$

Class Weight	Performance Measures	Performance
$W_n = 3, W_p = 1$	Accuracy	0.852
$W_n = 3, W_p = 1$	F1-Score	0.90773
$W_n = 3, W_p = 1$	AUROC	0.8647
$W_n = 3, W_p = 1$	Precision	0.9054726
$W_n = 3, W_p = 1$	Sensitivity	0.91
$W_n = 3, W_p = 1$	Specificity	0.62

4.4 The ROC curve

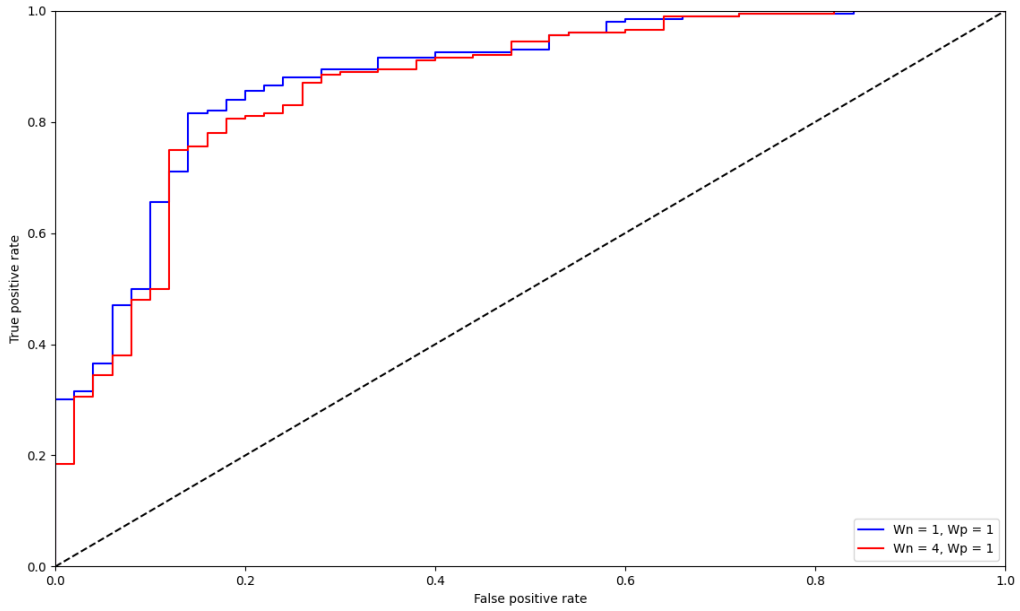

Figure 4.1: The ROC curve for $W_n = 1$, $W_p = 1$, and $W_n = 3$, $W_p = 1$

```
def Question4_4(IMB_features, IMB_labels, IMB_test_features, IMB_test_labels):
    clf = select_classifier(penalty='l2', c=0.1, class_weight={-1:1,1:1})
    clf.fit(IMB_features, IMB_labels)
    Y_pred = clf.decision_function(IMB_test_features)
    ftr1,tpr1,thresholds = metrics.roc_curve(IMB_test_labels, Y_pred)
    plt.plot(ftr1,tpr1,'b', label = "Wn = 1, Wp = 1")
    clf = select_classifier(penalty='l2', c=0.1, class_weight={-1:3,1:1})
    clf.fit(IMB_features, IMB_labels)
```

```
    Y_pred = clf.decision_function(IMB_test_features)
    ftr1,tpr1,thresholds = metrics.roc_curve(IMB_test_labels, Y_pred)
    plt.plot(ftr1,tpr1,'r', label = "Wn = 4, Wp = 1")
    plt.plot([0,1],[0,1],'k--')
    plt.xlabel("False positive rate")
    plt.ylabel("True positive rate")
    plt.ylim(0,1)
    plt.xlim(0,1)
    plt.legend(loc=4)
    plt.show()
    pass
```

5 Challenge

5.1 Write-Up and Code

Note: Due to the long running time, my tests for this section mainly based on 5-fold Cross Validation for $300 * 3$ review text.

5.1.1 Preparing Codes

In order to deal with three classes problems, I need to rewrite my previous functions so that perform well on this question and give the correct accuracy performance of $accuracy = (x_1 + x_2 + x_3)/n$. Also the `select_param` functions should be rewrite to give the correct best parameter values. Finally, `Challenge()` function is applied to give the predicting output `fgsepter.csv`.

```
def Challenge_performance(y_true, y_pred):
    CM = metrics.confusion_matrix(y_true, y_pred, labels=[1,0,-1])
    value = (CM[0][0] + CM[1][1] + CM[2][2])/len(y_true)
    return value.astype(np.float64)
```

```
def Challenge_cv_performance(clf, X, y, k):
    scores = []
    skf = StratifiedKFold(n_splits=k, shuffle=False)
    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 = clf.predict(X_test)
        scores.append(Challenge_performance(y_test, y_pred) )
    return np.array(scores).mean()
```

```
def Challenge_select_param_linear(X, y, k, C_range = [], function_shape = 'ovo'):
    best_C_val=0.0
    max_score = 0
    for c in C_range:
        clf = SVC(C=c, kernel = 'linear', decision_function_shape = function_shape)
        performance = Challenge_cv_performance(clf,X,y,k)
        print("Accuracy for ", c , " is ", performance)
        if performance > max_score:
            max_score = performance
            best_C_val = c
    return best_C_val, max_score
```

```
def Challenge_select_param_quadratic(X, y, k, param_range = [], function_shape = 'ovo
                                                                    '):
    best_C_val,best_r_val = 0.0, 0.0
    max_score = 0
    for i in range(len(param_range)):
        c,r = param_range[i]
        clf = SVC(C=c, coef0 = r, kernel = 'poly', degree=2, gamma='auto')
        performance = Challenge_cv_performance(clf,X,y,k)
        print("Accuracy for c = ", c,  " , r = ", r , " is ", performance)
        if performance > max_score:
            max_score = performance
```

```
            best_C_val = c
            best_r_val = r
    return best_C_val,best_r_val, max_score
```

```
def Challenge():
    print("begin:")
    multiclass_features, multiclass_labels, multiclass_dictionary =
                                                    get_multiclass_training_data()
    heldout_features = get_heldout_reviews(multiclass_dictionary)
    print("Step1:")
    clf = SVC(C=0.01, coef0 = 2600, kernel = 'poly', degree=2, gamma='auto',
                                                    decision_function_shape = 'ovo')
    # clf = SVC(C=0.595, kernel = 'linear')
    print("Step2:")
    clf.fit(multiclass_features,multiclass_labels)
    print("Step3:")
    y_pred = clf.predict(heldout_features)
    print("Step4:")
    generate_challenge_labels(y_pred, 'fgsepter')
```

5.1.2 Feature engineering

Instead of just binary sequence 0 and 1 that indicate the presence of word, I tried to use the number of times a word occurs in a comment as a feature and create the corresponding feature table. I rewrite the `generate_feature_matrix(df, word_dict)` function:

```
def generate_feature_matrix(df, word_dict):
    number_of_reviews = df.shape[0]  # n
    number_of_words = len(word_dict) # d
    feature_matrix = np.zeros((number_of_reviews, number_of_words))
    for i, review in enumerate(df['reviewText']):
        review = review.lower()
        for word in review:
            if word in string.punctuation:
                review = review.replace(word, " ")
        words = review.split()
        for word in words:
            if word in word_dict:
                feature_matrix[i][word_dict[word]] += 1
    return feature_matrix
```

Except the column of `reviewTest` we already looked at, the dataset also has a column named `summary`, which gives the key words of the comment. This column also gives us many useful information and can be applied into our feature table. To utilize this column, I rewrite the `extract_dictionary(df)` function as:

```
def extract_dictionary(df):
    word_dict = {}
    words = []
    num = 0
    for review in df['reviewText']:
        review = review.lower()
        for word in review:
            if word in string.punctuation:
```

```
                review = review.replace(word, " ")
        words = review.split()
        for word in words:
            if (word in word_dict) == False:
                word_dict[word] = num
                num = num + 1
    for summary in df['summary']:
        review = review.lower()
        for word in review:
            if word in string.punctuation:
                review = review.replace(word, " ")
        words = review.split()
        for word in words:
            if (word in word_dict) == False:
                word_dict[word] = num
                num = num + 1
    return word_dict
```

In this way, I extend my dictionary so that it contains words from the column of `summary`. Consider the fact that words in `summary` have a higher possibility of representing the core meaning of the comment, they should be given a higher weight when calculating the feature matrix, we note this weight as a parameter k. This means that a word appears once in `summary` equals the same word appears k times in the `viewText`. The value of k will be determined later. Since we already change the feature matrix to represent the number of times a word appears in the `reviewText`, we now need to further modify the `generate_feature_matrix(df,word_dict)` so that it can take the weight k of the summary words into consideration:

```
def generate_feature_matrix(df, word_dict):
    number_of_reviews = df.shape[0]  # n
    number_of_words = len(word_dict) # d
    k = 2 # We can change the value of k here.
    feature_matrix = np.zeros((number_of_reviews, number_of_words))
    for i, review in enumerate(df['reviewText']):
        review = review.lower()
        for word in review:
            if word in string.punctuation:
                review = review.replace(word, " ")
        words = review.split()
        for word in words:
            if word in word_dict:
                feature_matrix[i][word_dict[word]] += 1
    for i, review in enumerate(df['summary']):
        review = review.lower()
        for word in review:
            if word in string.punctuation:
                review = review.replace(word, " ")
        words = review.split()
        for word in words:
            if word in word_dict:
                feature_matrix[i][word_dict[word]] += k
    return feature_matrix
```

There are also some other columns in the `dataset.csv`, `unixReviewTime`, `helpful`, and `unhelpful`. However, after some testing, I did not notice any direct connecting between them and the rating. On the

contrary, adding these contributors increase the time complexity a lot. Hence I decide to not include them in my feature table.

Moreover, another method is applied to improve the behavior of feature matrix, which is known as `TF-IDF`. This method will be introduced in detail in Section 5.1.6.

5.1.3 Hyperparameter selection

I apply 5-fold Cross Validation and Grid Search to select the best parameter C. Comparing with Grid Search and Random Search introduced before, Grid Search has a better accuracy when the amount of parameters is small. Since we only have one parameter C for linear kernel here, I would prefer Grid Search.

Our previous Grid Search can only deal with C values that equals 10 to the power of n where n is an integer. However, in actual situations, it is almost impossible for the best C to be exactly 10^n. Hence, we need some technics to improve the search method. I tried the method of *Dichotomize Approximating Method*. This method will be introduced in details in Section 5.1.6. To select the parameter C, I write the following function:

```
def Challenge_findBestC():
    multiclass_features, multiclass_labels, multiclass_dictionary =
                                                get_multiclass_training_data()
    heldout_features = get_heldout_reviews(multiclass_dictionary)
    # Change the C_range here
    C_range = [0.00644,0.00648,0.00652,0.00656,0.00660,0.00664,0.00668,0.00672]
    best_C, max_score = Challenge_select_param_linear(X = multiclass_features, y =
                                                multiclass_labels, k = 5, C_range =
                                                C_range,  function_shape = 'ovo')
    print("Best C value is: ", best_C, " with accuracy ", max_score)
```

Table 11: The Performance for different values of C

C	Performance
0.001	0.5433
0.01	0.6122
0.1	0.5777
1	0.5844
10	0.5844
100	0.5844
1000	0.5844

According to Table 11, I know that the best C value is around 0.01. Using *Dichotomize Approximating Method*, more detailed value of C will be evaluated and give the result that when $C = 0.008$, the accuracy is 0.6211. The detailed information will be given in Section 5.1.6.

Using the similar method of *Dichotomize Approximating Method*, we can also find the parameters C and r for quadratic fitting. To select the parameters C and r, I write the following function:

```
def Challenge_findBestCandR():
    multiclass_features, multiclass_labels, multiclass_dictionary =
                                                get_multiclass_training_data()
    heldout_features = get_heldout_reviews(multiclass_dictionary)
```

```
# Change the range of C and R here:
C = [0.031, 0.032, 0.033, 0.034]
R = [1050,1070, 1100, 1125, 1150, 1175]
param_range = []
for c in C:
    for r in R:
        param_range.append([c,r])
best_C_val,best_r_val, max_score = Challenge_select_param_quadratic(
                                        multiclass_features, multiclass_labels
                                        , 5, param_range = param_range,
                                        function_shape = 'ovr')
print(best_C_val," ", best_r_val," ", max_score)
```

After running several times of this method, I finally choose $C = 0.034$, $r = 1100$, which gives the accuracy of 0.600. The output set is large and due to the page limit, I only list the last 15 choice of C and r in the following table:

Table 12: Dichotomize Approximating Method for finding C and r

C	r	Performance
c = 0.008	r = 2500	0.6144
c = 0.008	r = 2600	0.6144
c = 0.008	r = 2700	0.6115
c = 0.010	**r = 2500**	**0.6244**
c = 0.010	r = 2600	0.6222
c = 0.010	r = 2700	0.6244
c = 0.012	r = 2500	0.6167
c = 0.012	r = 2600	0.6144
c = 0.012	r = 2700	0.6100
c = 0.015	r = 2500	0.6133
c = 0.015	r = 2600	0.6100
c = 0.015	r = 2700	0.6167
c = 0.020	r = 2500	0.6144
c = 0.020	r = 2600	0.6067
c = 0.020	r = 2700	0.6089

Apart from C and r, another parameter we need to find is k, which introduced as the weight of words in `summary`. To select the parameter k, I choose different values of k in function `generate_feature_matrix(df word_dict)`. Using $c = 0.9$ and apply *Dichotomize Approximating Method*, I obtain the following table. According to the table, I choose $k = 2.6$, which gives the best performance.

Table 13: Dichotomize Approximating Method for finding k

k	Performance	k	Performance
0	0.5233	2.25	0.5811
1	0.5656	2.5	0.5833
1.5	0.5711	2.6	0.5844
2	0.5722	2.7	0.5822
3	0.5756	2.8	0.5822

5.1.4 Algorithm selection

Comparing with what we obtained in previous section of linear fitting, $C = 0.0080$ gives an accuracy of 0.6211, and the quadratic fitting, $C = 0.010$, $r = 2500$ gives an accuracy of 0.6244, I find that *Quadratic algorithm* gives a higher accuracy, and I will apply quadratic fitting when predict my results.

5.1.5 Multiclass method

The `SVC` function has one parameter named `decision_function_shape`, which has two possible inputs: `'ovo'` and `'ovr'`.

1. `'ovo'` means *One VS One*. This is a strategy to solve multiple classification problems by using binary classification algorithm. The algorithm obtains different combinations of two categories from multiple categories, and train multiple classifiers bases on them. For a given *n*-classification problem, it should train $\binom{n}{2} = n(n-1)/2$ binary classifiers. In this problem, $n = 3$, and 3 classifiers should be trained. Finally, the results of all classifiers are counted and the most common predictions are used as the final result.

2. `'ovr'` means *One VS Rest*. This is also a strategy to solve multiple classification problems by using binary classification algorithm. The algorithm picks one class and treat all of the rest classes as one class and run binary classification. After running, the algorithm picks another class and repeat. Consequently, this can translate the *n*-classification problem to *n* binary classification problem. Similar as `'ovo'`, the results of all classifiers are counted and the most common predictions are used as the final result.

For our current problem, we have $n = 3$. Both of the two methods will run 3 binary classifiers to obtain the results. I expect them to have the same complexity. The question is which one will give a better performance. Hence I rewrite the `select_classifier()` function to have one more input parameter `decision_function_shape` so that it can deal with multiclass methods. Then I wrong the following codes to test their performance:

```
def Challenge_ovoORovr():
    multiclass_features, multiclass_labels, multiclass_dictionary =
                                                get_multiclass_training_data(300)
    heldout_features = get_heldout_reviews(multiclass_dictionary)
    for c in [0.001, 0.01, 0.1, 1, 10, 100, 1000]:
        clf = SVC(C=c, kernel = 'linear', decision_function_shape = 'ovr')
        performance = Challenge_cv_performance(clf,multiclass_features,
                                                    multiclass_labels,5)
```

```
        print("Accuracy for ovr is: ", performance)
        clf = SVC(C=c, kernel = 'linear', decision_function_shape = 'ovo')
        performance = Challenge_cv_performance(clf,multiclass_features,
                                               multiclass_labels,5)
        print("Accuracy for ovo is: ", performance)
```

It is not surprise to me that both of these two methods gives exactly same results, and I can choose either one to predict the results. In the following training process, I will apply `'ovo'`.

5.1.6 Other techniques

The first technique I want to introduce is `TF-IDF`, which is used to improve the performance of feature matrix. The basic concept of this technique is to give each word a weight so that they can be judged more efficiently. In our dictionary, there are many useless words, such as the most commonly used word *the* as discovered before. This word does not contribute to the rating of the review, both positive and negative ones can have a lot of this word in their text. Including this in the dictionary will lead to more complexity and even over fitting, since there is no need for us to consider this word. There are many other words like this, such as *a*, *and*, or *it*. We need to lower their weight.

On the contrary, there are many other words that are important to our results. My previous research discover ten most positive words and ten most negative words, as shown in Figure 3.2 and Figure 3.3. Although containing these words is not guaranteed to be positive or negative, but it cannot be denied that these words may make a difference on the rating of each review. We need to give these words a higher weight.

After some testing, the words that are considered to be useless are given a weight equals zero, while the useful twenty words are given a weight equals 2.6. The method to determine the coefficient is also *Dichotomize Approximating Method*, which will be introduced in details later. To reduce complexity, I give other words a `TF-IDF` weight equals one. To further improve accuracy, I can reconsider their weight and give them more precise `TF-IDF` rate. In general, this approach really increased my accuracy a lot. Python code is recorded below:

```
def TFIDF(word):
    if word == "the" or word == "a" or word == "and" or word == "book" or word == "
                                                 that" or word == "for":
        return 0
    elif word == "enjoyed" or word == "loved" or word == "great" or word == "hot" or
                                                 word == "life" or word == "highly" or
                                                 word == "enjoyable" or word == "liked"
                                                  or word == "excellent" or word == "
                                                 look":
        return 2.7
    elif word == "not" or word == "short" or word == "ok" or word == "sorry" or word
                                                 == "waste" or word == "nothing" or
                                                 word == "may" or word == "if" or word
                                                 == "care" or word == "pages":
        return 2.7
    else:
        return 1

def generate_feature_matrix(df, word_dict):
    number_of_reviews = df.shape[0]  # n
    number_of_words = len(word_dict) # d
```

```
    k = 2.6 # We can change the value of k here.
    feature_matrix = np.zeros((number_of_reviews, number_of_words))
    for i, review in enumerate(df['reviewText']):
        review = review.lower()
        for word in review:
            if word in string.punctuation:
                review = review.replace(word, " ")
        words = review.split()
        for word in words:
            if word in word_dict:
                feature_matrix[i][word_dict[word]] += TFIDF(word)
    for i, review in enumerate(df['summary']):
        review = review.lower()
        for word in review:
            if word in string.punctuation:
                review = review.replace(word, " ")
        words = review.split()
        for word in words:
            if word in word_dict:
                feature_matrix[i][word_dict[word]] += k*TFIDF(word)
    return feature_matrix
```

From previous sections, we know that the performance of C first increase and then decrease due to over fitting. There must exist one maximal performance in between. Hence *Dichotomize Approximating Method* can be applied to approximate the maximum of C. The core of this method is that, when the best C among an array of values is discovered, we find the one before and one after, then the actual best C must lie in between. For example, Table **??** lists the performance of C with the seven values utilized before. We find $C = 0.01$ gives the best performance. Then we can choose the before value $C = 0.001$ and the after value $C = 0.1$, the actual best value of C must lie in between $[0.001, 0.1]$. Hence we can divide the section into several parts and apply the Grid Method again, as shown in the left-most table in Table 14. I apply this method four times and the corresponding outputs are listed in the four tables of Table 14. According to the right-most table, I finally choose $C = 0.0080$ as my final best value of C, since applying more times of Dichotomize Approximating does not improve accuracy any more. This gives an accuracy of 0.6211.

Table 14: Dichotomize Approximating Method for finding C

C1	Performance	C2	Performance	C3	Performance
0.004	0.6089	0.005	0.6144	0.0074	0.6189
0.008	0.6211	0.006	0.6133	0.0076	0.62
0.012	0.6177	0.007	0.6178	0.0078	0.6211
0.020	0.6022	0.009	0.6133	0.0082	0.6178
0.040	0.5889	0.010	0.6122	0.0084	0.62
0.080	0.5822	0.011	0.6122	0.0086	0.6167

Other techniques I applied including trying to convert words into vectors, so that I can easily analyze the phrases in the text. Also, I tried to emphasis some common used and important words, such as *not*, *but*. This words may easily lead to misclassified. *Not* plus a good word has a meaning of not good, but is easily predicted as good in our model. So does *but*. Consequently, it is useful for us to track the situations when

these words appears. Unfortunately, due to the high time-complexity and time limit, I did not finish these thoughts, and write it here as for future improvement.

5.1.7 Summary

1. The feature matrix is reconstructed so that it does not represent only the presence but the number of appearance of words.
2. The column of `summary` is applied and I assign a parameter k for words in `summary` to maximal the performance.
3. Grid Search is applied to obtain the range of parameters so that the algorithm gives best performance.
4. *Dichotomize Approximating Method* is applied to obtain more accurate value of parameters C, r, and k.
5. Comparing with the performance of linear and quadratic fitting, finally quadratic is chosen.
6. Comparing with `'ovo'` and `'ovr'`, finally `'ovo'` is chosen.
7. Using the method of TF-IDF, I give each word a corresponding weight so that they can be judged based on their importance.

Finally, the `clf` is written as:

```
clf = SVC(C=0.01, coef0 = 2500, kernel = 'poly', degree=2, gamma='auto',
                                          decision_function_shape = 'ovo')
```

When using 5-fold Cross Validation on 1500 review texts, this final result gave me an accuracy rate of 0.66733.

5.2 Test Scores

My predicted label file *fgsepter.csv* is uploaded on *Canvas*.

5.3 Social Bias

News articles are very easily influenced by political ideas, social organization, and other potential factors. Individuals' sentiment of comments on news are also high depend on their political stance, gender and race, or interest group they belong to. For example, a news article describing the government's increased investment in the social welfare system is likely to be applauded by most middle classes, although it may actually be fake news. We definitely cannot say that those news articles fulling of false information as good one, even though it is commented as good by most viewers. This is one of the big difference between books and news article.

The potential societal bias that might be present in the learned model is that people may misjudge the good or bad of a piece of news, because the masses often lack the ability to judge between facts and falsehood. Moreover, people's personal opinions also influence their comments on the news. However, our model trusts that people's judgement is trustworthy and take the average sentiment. The fact is not always the case. Stories written in books can of course be illusory: we want to read science fiction or fairy tales.

On the contrary, people read news articles because they want to read something that truly happened in the world. Therefore, the evaluation of a book can be subjective, but the evaluation of news needs enough objectivity. **There is strong subjectivity of people's comments towards news articles, and it is exactly a kind of bias.**

5.4 Possible Fix

As described before, people's comments are highly subjective. Therefore, we can't judge the good or bad of a piece of news only by taking the average sentiment of all the comments. We can define a good news as: accord with objective facts, and in line with the preferences of most readers. One possible fix is that, we can add more items into the feature table, which are able to evaluate the authenticity of news articles. Those items are extracted from the content of the news articles. For example, the number of detailed data presented in the articles, the name of an official institution, or comments from eminent scholars. Using this, we can evaluate whether a news article accord with truth and facts. Also, the comments made by viewers can evaluate whether the article conforms to the preferences of the public. Combining the above two factors, we can then evaluate whether the news article is good or not.

To be more precise, we can add more aspects except authenticity and preference of people, such as whether the news is time efficient. Those improvement can fix our model well.