8E and 8F: Finding the Probability P(Y==1|X)

8E: Implementing Decision Function of SVM RBF Kernel

After we train a kernel SVM model, we will be getting support vectors and their corresponsing coefficients α_i

Check the documentation for better understanding of these attributes:

https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html

As a part of this assignment you will be implementing the `decision_function()` of kernel SVM, here decision_function() means based on the value return by `decision_function()` model will classify the data point either as positive or negative

Ex 1: In logistic regression After traning the models with the optimal weights w we get, we will find the value $\frac{1}{1+\exp(-(wx+b))}$, if this value comes out to be < 0.5 we will mark it as negative class, else its positive class

Ex 2: In Linear SVM After traning the models with the optimal weights w we get, we will find the value of $sign(wx+b)$, if this value comes out to be -ve we will mark it as negative class, else its positive class.

Similarly in Kernel SVM After traning the models with the coefficients α_i we get, we will find the value of $sign(\sum_{i=1}^{n}(y_i\alpha_i K(x_i,x_q)) + intercept)$, here $K(x_i,x_q)$ is the RBF kernel. If this value comes out to be -ve we will mark x_q as negative class, else its positive class.

RBF kernel is defined as: $K(x_i,x_q) = exp(-\gamma ||x_i - x_q||^2)$

For better understanding check this link: https://scikit-learn.org/stable/modules/svm.html#svm-mathematical-formulation

Task E

> 1. Split the data into X_{train}(60), X_{cv}(20), X_{test}(20)
> 2. Train $SVC(gamma = 0.001, C = 100.)$ on the (X_{train}, y_{train})
> 3. Get the decision boundry values f_{cv} on the X_{cv} data i.e. f_{cv} `= decision_function(` X_{cv} `)` you need to implement this decision_function()

In [2]:
```python
import numpy as np
import pandas as pd
from sklearn.datasets import make_classification
import numpy as np
import math
from sklearn import metrics
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
```

In [8]:
```python
X, y = make_classification(n_samples=5000, n_features=5, n_redundant=2,
                           n_classes=2, weights=[0.7], class_sep=0.7, random_state=15)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2)
X_train, X_cv, y_train, y_cv = train_test_split(X_train, y_train, test_size = 0.25)
```

Pseudo code

clf = SVC(gamma=0.001, C=100.)
clf.fit(Xtrain, ytrain)

def decision_function(Xcv, ...): #use appropriate parameters
  for a data point x_q in Xcv:
    #write code to implement $(\sum_{i=1}^{\text{all the support vectors}}(y_i\alpha_i K(x_i,x_q)) + intercept)$, here the values y_i, α_i, and $intercept$ can be obtained from the trained model
return *# the decision_function output for all the data points in the Xcv*

fcv = decision_function(Xcv, ...) *# based on your requirement you can pass any other parameters*

Note: Make sure the values you get as fcv, should be equal to outputs of clf.decision_function(Xcv)

Let's try for Linear Kernel

In [9]:
```python
from sklearn.svm import LinearSVC

linear_clf = LinearSVC()
linear_clf.fit(X_train, y_train)
```

Out[9]:
```
LinearSVC()
```

In [10]:
```python
intercept = linear_clf.intercept_[0]
decision_func_op = []
for i in range(len(X_cv)):
    prod = np.dot(X_cv[i], linear_clf.coef_[0]) + intercept
    decision_func_op.append(prod)
```

In [11]:
```python
np.array(decision_func_op)[0:5]
```

Out[11]:
```
array([-1.12725643, -1.08019728, -1.07912335,  0.00963751, -0.4754098 ])
```

In [12]:
```python
linear_clf.decision_function(X_cv)[0:5]
```

Out[12]:
```
array([-1.12725643, -1.08019728, -1.07912335,  0.00963751, -0.4754098 ])
```

So the values of the implemented function matches with the model output.

FOR RBF KERNEL

In [20]:
```python
#For RBF Kernel
from numpy import linalg

clf = SVC(gamma=0.001, C=100, kernel='rbf')
clf.fit(X_train, y_train)

#Support Vectors
Xi = clf.support_vectors_
#Xi = clf.support_vectors_.tolist()

#Query Point
Xq = X_cv
#Xq = X_cv.tolist()
```

In [21]:
```python
def decision_func(data, support_vectors, clf, gamma = 0.001 ):

    aiyi = clf.dual_coef_.tolist()
    intercept = clf.intercept_[0]
    decision_function_outputs = []

    #Selecting Query points
    for query_point in data:

        interim_sum = 0

        #Selecting Each Support Vector
        for i in range(len(support_vectors)):
            dist = query_point - support_vectors[i]
            l2_norm = linalg.norm(dist,2)
            interim_val = math.exp(-gamma * math.pow(l2_norm,2))
            interim_sum += aiyi[0][i] * interim_val
        final_val = np.round((interim_sum + intercept),8)
        decision_function_outputs.append(final_val)

    return np.array(decision_function_outputs)
```

In [22]:
```python
decision_function_output = decision_func(X_cv, Xi, clf)
decision_function_output[0:5]
```

Out[22]:
```
array([-2.06763167, -2.22715144, -2.03757667, -0.03701432, -0.97272742])
```

In [23]:
```python
clf.decision_function(X_cv)[0:5]
```

Out[23]:
```
array([-2.06763167, -2.22715144, -2.03757667, -0.03701432, -0.97272742])
```

8F: Implementing Platt Scaling to find P(Y==1|X)

Check this PDF

TASK F

> 1. Apply SGD algorithm with (f_{cv}, y_{cv}) and find the weight W intercept b `Note: here our data is of one dimensional so we will have a one dimensional weight vector i.e W.shape (1,)`
>
> Note1: Don't forget to change the values of y_{cv} as mentioned in the above image. you will calculate y+, y- based on data points in train data
>
> Note2: the Sklearn's SGD algorithm doesn't support the real valued outputs, you need to use the code that was done in the `'Logistic Regression with SGD and L2'` Assignment after modifying loss function, and use same parameters that used in that assignment. if Y[i] is 1, it will be replaced with y+ value else it will replaced with y- value
>
> 1. For a given data point from X_{test}, $P(Y=1|X) = \frac{1}{1+exp(-(W*f_{test}+b))}$ where f_{test} `= decision_function(` X_{test} `)`, W and b will be learned as metioned in the above step

Note: in the above algorithm, the steps 2, 4 might need hyper parameter tuning, To reduce the complexity of the assignment we are excluding the hyerparameter tuning part, but intrested students can try that

If any one wants to try other calibration algorithm istonic regression also please check these tutorials

1. http://fa.bianp.net/blog/tag/scikit-learn.html#fn:1
2. https://drive.google.com/open?id=1MzmA7QaP58RDzocB0RBmRiWfl7Co_VJ7
3. https://drive.google.com/open?id=133odBinMOIVb_rh_GQxxsyMRyW-Zts7a
4. https://stat.fandom.com/wiki/Isotonic_regression#Pool_Adjacent_Violators_Algorithm

Here first task is to find y+ and y-

In [25]:
```python
targets = y_cv.tolist()
targets.count(0), targets.count(1)
```

Out[25]:
```
(693, 307)
```

In [26]:
```python
 #calculate y+, y-
N_plus = targets.count(1)
y_pos = (N_plus + 1) / (N_plus + 2)

N_minus = targets.count(0)
y_neg = 1 / (N_minus + 2)
```

In [27]:
```python
y_pos, y_neg
```

Out[27]:
```
(0.9967637540453075, 0.0014388489208633094)
```

In [28]:
```python
y_modified = [y_pos if i == 1 else y_neg for i in y_cv]
y_modified = np.array(y_modified)
y_modified[0:5]
```

Out[28]:
```
array([0.00143885, 0.00143885, 0.00143885, 0.99676375, 0.00143885])
```

First need to modify **Sigmoid and Loss Function**

Here use manual implementation of SGD code from previous assignmemnt

In [29]:
```python
def logloss(y_modified,y_pred):
    '''In this function, we will compute log loss '''
    sub_sum = 0

    for i in range(len(y_modified)):
        sub_x = y_modified[i]*(np.log10(y_pred[i])) + (1 - y_modified[i])*(np.log10(1-y_pred[i]))
        sub_sum += sub_x
    loss = -(sub_sum)/len(y_true)

    return loss
```

In [30]:
```python
epochs=50
alpha=0.0001
eta0=0.0001
tolerace = 0.00001

#from tqdm import tqdm

def initialize_weights(dim):
    ''' In this function, we will initialize our weights and bias'''
    #initialize the weights to zeros array of (1,dim) dimensions
    #you use zeros_like function to initialize zero, check this link https://docs.scipy.org/doc/numpy/reference/generated/numpy.zeros_like.html
    #initialize bias to zero

    w = np.zeros_like(dim)
    b = 0

    return w,b

def sigmoid(z):
    ''' In this function, we will return sigmoid of z'''
    # compute sigmoid(z) and return
    sigmoid = 1 / (1 + np.exp(-z))

    return sigmoid

def logloss(y_true,y_pred):
    '''In this function, we will compute log loss '''
    sub_sum = 0

    for i in range(len(y_true)):
        sub_x = y_true[i]*(np.log10(y_pred[i])) + (1 - y_true[i])*(np.log10(1-y_pred[i]))
        sub_sum += sub_x
    loss = -(sub_sum)/len(y_true)

    return loss

def gradient_dw(x,y,w,b,alpha,N):
    '''In this function, we will compute the gardient w.r.to w '''

    dw = x*(y - sigmoid(np.dot(w, x+b))) - (alpha*w)/N


    return dw

def gradient_db(x,y,w,b):

    '''In this function, we will compute gradient w.r.to b '''

    db = y - sigmoid(np.dot(w, x+b))

    return db
```

In [31]:
```python
def train(X_train, y_train ,X_test, y_test, epochs ,alpha, eta0, tolerace):

    ''' In this function, we will implement logistic regression'''

    N=len(X_train)

    w, b = initialize_weights(X_train[0])
    training_loss = []
    test_loss = []

    #converged = False

    '''while not converged :
        for every_epoch in tqdm(range(epochs)):'''

    y_predicted_train = []
    y_predicted_test = []

    for i in range(len(X_train)):
        grad_w = gradient_dw(X_train[i], y_train[i], w, b, alpha, N)
        grad_B = gradient_db(X_train[i], y_train[i], w, b)

        #Updating weights and intercept
        w = w + eta0 * grad_w
        b = b + eta0 * grad_B


    for i in range(len(X_train)):
        y_pred_train = sigmoid(np.dot(w,X_train[i])+b) #changed here, using sigmoid
        y_predicted_train.append(y_pred_train)

    loss_train = logloss(list(y_train), y_predicted_train)
    training_loss.append(loss_train)

    #predicting y_test with updated values of w and b
    for i in range(len(X_test)):
        y_pred_test = sigmoid(np.dot(w,X_test[i])+b)
        y_predicted_test.append(y_pred_test)

    loss_test = logloss(y_test, y_predicted_test)
    test_loss.append(loss_test)

    ''''if every_epoch != 0 and (training_loss[every_epoch-1] - training_loss[every_epoch]) <= tolerace:
                max_epoch = every_epoch
                converged = True
                print("Converged at {0} th epoch with tolerance = {1} ".format(every_epoch, tolerace))
                break'''

    return w,b #,training_loss,test_loss, y_predicted_train
```

In [32]:
```python
w, b = train(y_cv, y_modified ,X_test, y_test, epochs ,alpha, eta0, tolerace)
w,b
```

Out[32]:
```
(0.015193200827233874, -0.019355390099394014)
```

> 1. For a given data point from X_{test}, $P(Y=1|X) = \frac{1}{1+exp(-(W*f_{test}+b))}$ where f_{test} `= decision_function(` X_{test} `)`, W and b will be learned as metioned in the above step

Now that we have value of w and b, write a function to compute probabilities.

In [33]:
```python
'''def decision_function_linearSVM(data, classifier):

    decision_func_op = []
    for i in range(len(data)):
        prod = np.dot(data[i], classifier.coef_[0]) + classifier.intercept_[0]
        decision_func_op.append(prod)

    return decision_func_op

ftest = decision_function_linearSVM(X_test, classifier=linear_clf)'''
```

In [37]:
```python
ftest = decision_func(X_cv, Xi, clf)
```

In [38]:
```python
platts_prob = []
for i in range(len(X_test)):
    prob = np.round(1/(1+math.exp(w*ftest[i]+b)),5)
    platts_prob.append(prob)
```

In [40]:
```python
platts_prob[0:10]
```

Out[40]:
```
[0.51269,
 0.5133,
 0.51258,
 0.50498,
 0.50853,
 0.50762,
 0.5149,
 0.51017,
 0.49946,
 0.49805]
```
