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

Attributes:

support_ : *array-like, shape = [n_SV]*
Indices of support vectors.

support_vectors_ : *array-like, shape = [n_SV, n_features]*
Support vectors.

n_support_ : *array-like, dtype=int32, shape = [n_class]*
Number of support vectors for each class.

dual_coef_ : *array, shape = [n_class-1, n_SV]*
Coefficients of the support vector in the decision function. For multiclass, coefficient for all 1-vs-1 classifiers. The layout of the coefficients in the multiclass case is somewhat non-trivial. See the section about multi-class classification in the SVM section of the User Guide for details.

coef_ : *array, shape = [n_class * (n_class-1) / 2, n_features]*
Weights assigned to the features (coefficients in the primal problem). This is only available in the case of a linear kernel.

coef_ is a readonly property derived from `dual_coef_` and `support_vectors_`.

intercept_ : *array, shape = [n_class * (n_class-1) / 2]*
Constants in decision function.

fit_status_ : *int*
0 if correctly fitted, 1 otherwise (will raise warning)

probA_ : *array, shape = [n_class * (n_class-1) / 2]*

probB_ : *array, shape = [n_class * (n_class-1) / 2]*
If probability=True, the parameters learned in Platt scaling to produce probability estimates from decision values. If probability=False, an empty array. Platt scaling uses the logistic function `1 / (1 + exp(decision_value * probA_ + probB_))` where `probA_` and `probB_` are learned from the dataset [R20c70293ef72-2]. For more information on the multiclass case and training procedure see section 8 of [R20c70293ef72-1].

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 $sign(\sum_{i=1}^{n}(y_i\alpha_iK(x_i,x_q)) + intercept)$, 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 [88]:

```
# necessary libraries
import numpy as np
```

```
import pandas as pd
from sklearn.datasets import make_classification
import numpy as np
from sklearn.svm import SVC
```

In [89]:

```
# obtaining the data
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)
#y = np.where(y == 0,-1,1)
```

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)

In [90]:

```
#Necessary libraries:
from sklearn.model_selection import train_test_split
import math

# Split the dataset into train,Cv and test set:
Xtrain,Xtest,Ytrain,Ytest = train_test_split(X,y,test_size=0.20,random_state=15)

X_train,X_cv,Y_train,Y_cv = train_test_split(Xtrain,Ytrain,
                                             test_size=0.20,random_state=15)
```

```
# Create a SVM classifier with the parameters:
clf = SVC(gamma=0.001,C=100,kernel='rbf',random_state = 15)
clf.fit(X_train,Y_train)

# Assign coefficient values:
alpha_i = clf.dual_coef_
sv_indices = clf.support_
intercept = clf.intercept_

# Output values of decision_function(X_cv)
f = clf.decision_function(X_cv)
print("(A) Sklearn's implementation: \n",f[1:10])


# define required functions:
def kernel(x,y): # rbf kernel function
    return math.exp(-1*0.001*(np.sum([(m-n)*(m-n) for m,n in zip(x,y)])))


# lets define our custom decision function:-
def decision_function(X_cv,X_train,Y_train,intercept,alpha_i):

    fcv = [] # to append decision func output

    for pt in range(len(X_cv)): # for each point in cv data

        sign_value = 0   # lets initialize the sign value

        for i,j in enumerate(sv_indices):  # iterating through supportvector indices

            sign_value +=(alpha_i[0,i]*kernel(X_train[j],X_cv[pt]))

        fcv.append(sign_value+intercept)  # append output value

    return fcv  # return the fcv list

print("\n")

# Call the function and verify the output:

fcv = decision_function(X_cv,X_train,Y_train,intercept,alpha_i)
```

```
print("(B) Custom implementation : \n",fcv[1:10])# lets display first 10 values alone.
```

```
(A) Sklearn's implementation:
 [-1.10424672 -2.02119883 -3.08410483 -2.7867225  -3.17760331 -3.06190422
 -2.6512218   3.32813573 -1.27477059]


(B) Custom implementation :
 [array([-1.10424672]), array([-2.02119883]), array([-3.08410483]), array([-2.7867225]), array([-3.17760331]), array
([-3.06190422]), array([-2.6512218]), array([3.32813573]), array([-1.27477059])]
```

OBSERVATION :

Hence, we can observe that the sklearn's implementation of decision function & my custom implementation of decision function is perfectly matching.

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

Let the output of a learning method be $f(x)$. To get calibrated probabilities, pass the output through a sigmoid:

$$P(y=1|f) = \frac{1}{1+exp(Af+B)} \qquad (1)$$

where the parameters A and B are fitted using maximum likelihood estimation from a fitting training set (f_i, y_i). Gradient descent is used to find A and B such that they are the solution to:

$$\underset{A,B}{argmin}\{-\sum_i y_i log(p_i) + (1-y_i)log(1-p_i)\}, \qquad (2)$$

where

$$p_i = \frac{1}{1+exp(Af_i+B)} \qquad (3)$$

Two questions arise: where does the sigmoid train set come from? and how to avoid overfitting to this training set?

from? and how to avoid overfitting to this training set?

If we use the same data set that was used to train the model we want to calibrate, we introduce unwanted bias. For example, if the model learns to discriminate the train set perfectly and orders all the negative examples before the positive examples, then the sigmoid transformation will output just a 0,1 function. So we need to use an independent calibration set in order to get good posterior probabilities. This, however, is not a draw back, since the same set can be used for model and parameter selection.

To avoid overfitting to the sigmoid train set, an out-of-sample model is used. If there are N_+ positive examples and N_- negative examples in the train set, for each training example Platt Calibration uses target values y_+ and y_- (instead of 1 and 0, respectively), where

$$y_+ = \frac{N_+ + 1}{N_+ + 2};\ y_- = \frac{1}{N_- + 2} \qquad (4)$$

For a more detailed treatment, and a justification of these particular target values see (Platt, 1999).

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.

```
def log_loss(w, b, X, Y):
    N = len(X)
    sum_log = 0
    for i in range(N):
        sum_log += Y[i]*np.log10(sig(w, X[i], b)) + (1-Y[i])*np.log10(1-sig(w, X[i], b))
    return -1*sum_log/N
```

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

In [91]:

```
# lets define initial variables (input & output)
data_fcv = fcv
data_ycv = Y_cv.tolist()

#lets obtain Y_plus & Y_minus from the training data
N_plus = np.unique(Y_train, return_counts=True)[1][1] # get the positive count values
Y_plus = (N_plus + 1) / (N_plus + 2)
print("The Y_plus value :- ",Y_plus)

N_minus = np.unique(Y_train, return_counts=True)[1][0] # get the negative count values
Y_minus = (1 / (N_minus + 2))
```

```
print("The Y_minus value :- ",Y_minus)


# Now lets replace 1 with yplus & 0 with yminus
for j,val in enumerate(data_ycv):
    if val == 0:
        data_ycv[j] = Y_minus
    else:
        data_ycv[j] = Y_plus

print("Replaced values in the Ycv list :-",data_ycv[0:10]) # display first 10 values
```

```
The Y_plus value :-  0.9989743589743589
The Y_minus value :-  0.00044863167339614175
Replaced values in the Ycv list :- [0.9989743589743589, 0.00044863167339614175, 0.00044863167339614175, 0.00044863167
339614175, 0.00044863167339614175, 0.00044863167339614175, 0.00044863167339614175, 0.00044863167339614175, 0.99897435
89743589, 0.00044863167339614175]
```

In [92]:

```
# lets initialize the weights
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/gene
    #initialize bias to zero
    w= np.zeros_like((dim))
    b= 0
    return w,b
```

In [93]:

```
# obtain the initialized weights
dim = data_fcv[0]
w,b = initialize_weights(dim)
print('w =',(w))
print('b =',str(b))
```

```
w = [0.]
b = 0
```

In [94]:

```
# sigmoid function:
def sigmoid(z):
```

```
    ''' In this function, we will return sigmoid of z'''
    # compute sigmoid(z) and return

    return (1/(1 + np.exp(-1*z)))
```

In [95]:

```
# Loss function:
def logloss(y_true,y_pred):
    '''In this function, we will compute log loss '''
    n = len(data_ycv)
    loss = 0.0

    for (y_true,y_pred) in zip(y_true,y_pred):

        loss += ((y_true*np.log10(y_pred)) + ((1-y_true) * np.log10(1-y_pred)))

    loss = -1 *(loss/n)
    return loss # return the loss value
```

In [96]:

```
# calculating gradient w.r.t weight w for single data point.
def gradient_dw(x,y,w,b,alpha,N):
    '''In this function, we will compute the gardient w.r.to w '''

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

    return dw  # returns derivative value
```

In [97]:

```
# calculating gradient w.r.t weight b for single data point.
def gradient_db(x,y,w,b):
    '''In this function, we will compute gradient w.r.to b '''
    z = np.dot(x,w.T)+b
    db = y - sigmoid(z)
    return db
```

In [98]:

```
# predict function
def pred(w,b,X):
    N = len(X)  # No of points in data_fcv
```

```
    predicted_prob = []
    for i in range(N):
        z = np.dot(w,X[i]) + b
        predicted_prob.append(sigmoid(z))
    return np.array(predicted_prob)
```

In [99]:

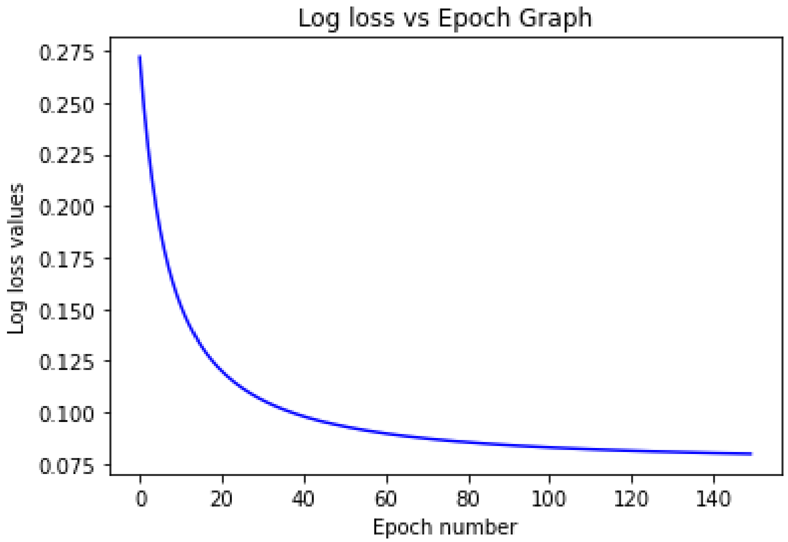
```
# implement SGD algo with log loss:
def train(X_train,y_train,epochs,alpha,eta0):
    ''' In this function, we will implement logistic regression'''
    #initialize the weights:
    w,b = initialize_weights(X_train[0])
    N = len(X_train) # No of points in data_fcv

    # define lists to store train & test loss:
    train_loss_list = []

    for e in range(epochs):
        grad_w = 0
        grad_b = 0

        for row in range(N):# pass each instance of training data to update weights

            #compute gradient:
            grad_w = gradient_dw(X_train[row],y_train[row],w,b,alpha,N)
            grad_b = gradient_db(X_train[row],y_train[row],w,b)

            #update w & b:
            w = w + (eta0*grad_w)
            b = b + (eta0*grad_b)


        # using updated weights(each epoch) predict for X_train:
        pred_train = pred(w,b,X_train)

        # compute loss between predicted values & actual values:
        train_loss = logloss(y_train,pred_train)

        # append the loss obtained in each epoch:
        train_loss_list.append(train_loss)
```

```
    return w,b,train_loss_list
```

In [113…

```
# call the train function:(defined above)

alpha= 0.01
eta0= 0.0001
epochs = 150  # for 150 epochs

w,b,trainlosses = train(data_fcv,data_ycv,epochs,alpha,eta0)

#Slope coefficients:(weights)
print("The weight value after 150 epochs:",w)

# intercept value:
print("The intercept value after 150 epochs :",b)

# lets obtain best weights & intercept by plotting epoch vs loss
import matplotlib.pyplot as plt

epoch_range = range(150)
plt.plot(epoch_range,trainlosses,c = 'blue')
plt.title("Log loss vs Epoch Graph")
plt.xlabel('Epoch number')
plt.ylabel('Log loss values')
plt.show()
```

```
The weight value after 150 epochs: [1.59755791]
The intercept value after 150 epochs : -0.17691930871968428
```



OBSERVATION :

I choose 80 as the final epoch number because after that loss values does not reduce significantly.

In [118…

```
# call the train function:(defined above)
alpha= 0.01
eta0= 0.0001
epochs = 80  # we choose 80 as the optimal epoch number

w,b,trainlosses = train(data_fcv,data_ycv,epochs,alpha,eta0)

#Slope coefficients:(weights)
print("The optimal weight value after 80 epochs:",w)

# intercept value:
print("The optimal intercept value after 80 epochs :",b)
```

```
The optimal weight value after 80 epochs: [1.3305034]
The optimal intercept value after 80 epochs : -0.17815361956395273
```

In [119…

```
# decision values function for X_test :
```

```python
x_test_dfvalues = clf.decision_function(Xtest)

# performing calibration on test data outputs
prob_values = (1 / (1 + np.exp((-1*w*x_test_dfvalues + b))))

# calibrated probabilities: #lets display first 15 values
print("Calibrated probabilities for test data :-\n",prob_values[0:15])
```

```
Calibrated probabilities for test data :-
 [2.04127092e-01 9.42828020e-01 2.40489988e-01 7.05625281e-01
 8.44278330e-04 9.50581189e-01 1.50421929e-01 9.33001739e-01
 1.57113217e-01 9.12896512e-01 8.34994719e-02 9.18430566e-01
 4.87894043e-02 1.33727775e-02 1.41743431e-02]
```