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 $sign(\sum_{i=1}^{n}(y_i\alpha_i K(x_i, x_q)) + intercept)$, 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()

```
import numpy as np
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.svm import SVC
import math
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
```

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)

```
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)
```

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)
X_train, X_cv, y_train, y_cv = train_test_split(X_train, y_train, test_size=0.25, random_state=1)
clf = SVC(gamma=0.001, C=100)
clf.fit(X_train, y_train)
```

```
SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
    decision_function_shape='ovr', degree=3, gamma=0.001, kernel='rbf',
    max_iter=-1, probability=False, random_state=None, shrinking=True,
    tol=0.001, verbose=False)
```

```
def K(x, x_q, gamma):
  distance = np.linalg.norm(x - x_q) ** 2
  return np.exp(- gamma * distance)

def decision_function(x_cv, x_train, sv_indices, coef, intercept, gamma):
```

```
def decision_function(x_cv, x_train, sv_indices, coef, intercept, gamma):
  decision_weights = []
  for x_q in x_cv:
    z = 0
    for i, j in zip(sv_indices, range(len(coef))):
      z += coef[j] * K(x_train[i], x_q, gamma)
    value = z + intercept
    decision_weights.append(float(value))
  return np.array(decision_weights)

sv_indices = clf.support_
coef = clf.dual_coef_.reshape(-1, 1)
intercept = clf.intercept_
gamma = clf.gamma
```

```
# Returns True if two arrays are element-wise equal within a tolerance.
np.allclose(decision_function(X_cv, X_train, sv_indices, coef, intercept, gamma), clf.decision_function(X_cv))
```

```
True
```

F: 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:

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

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:

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

where

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

Two questions arise: where does the sigmoid train set come 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

$$y_+ = \frac{N_+ + 1}{N_+ + 2}; \; y_- = \frac{1}{N_- + 2} \tag{4}$$

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

Check this PDF

```
Np, Nn = 0, 0
for target in y_cv:
  if target == 0:
    Np += 1
  else:
    Nn += 1

ypos = round((Np + 1)/(Np + 2), 8)
```

```
yneg = round(1 / (Nn + 2), 8)

print("positive examples : {0} \nnegative examples : {1}".format(Np, Nn))
print("\nPlatt Callibrated positive target : {0}   \nPlatt Callibrated negative target : {1}".format(ypos, yneg))
```

```
positive examples : 714
negative examples : 286

Platt Callibrated positive target : 0.99860335
Platt Callibrated negative target : 0.00347222
```

```
platt_scaled_y_cv = np.array([yneg if target==0 else ypos for target in y_cv])
platt_scaled_y_cv[:5]
```

```
array([0.00347222, 0.00347222, 0.00347222, 0.99860335, 0.99860335])
```

TASK F

4. 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

5. 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 including 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

```
def sigmoid(w,x,b):
    return 1/(1+np.exp(-(np.dot(x,w)+b)))
```

```
def reg_cost(y,pred,alpha,N):
  return(-y * np.log(pred) -(1-y) * np.log(1-pred)).mean() + (alpha/(2*N)) * np.sum(N**2)
```

```
def compute_log_loss(true,pred):
```

```
    loss = 0
    for true, pred in zip(true,pred):
        loss += (true * np.log(pred)) + ((1-true) * np.log(1-pred))
    return -1*(loss)/len(true)
```

```
svmrbf = clf.decision_function(X_cv)
X_train = svmrbf
y_train = platt_scaled_y_cv
```

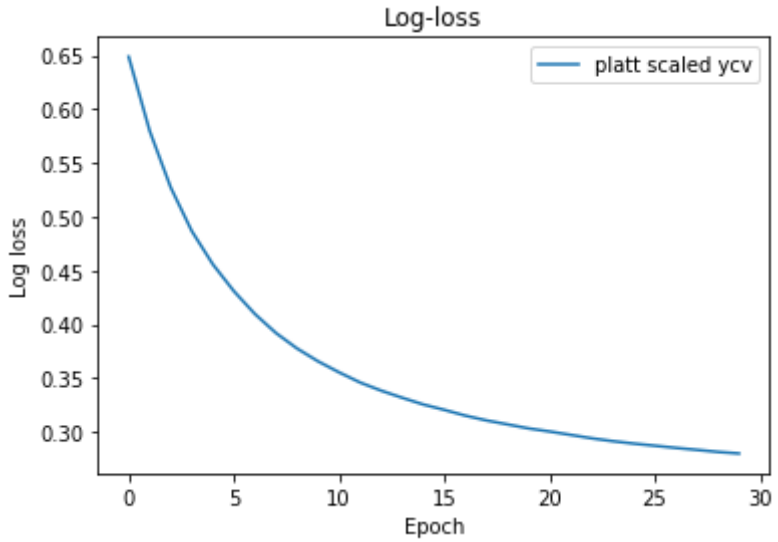
```
from tqdm import tqdm
train_loss = []
# test_loss = []
print("epoch\t log loss")

w = np.zeros_like(X_train[0])
b = 0
eta0  = 0.0001
alpha = 0.0001
N = len(X_train)

for epoch in range(30):
    for j in range(N):
        r = np.random.randint(N)
        Xn = X_train[r]
        yn = y_train[r]

        weight_update = ( 1- (alpha*eta0)/N) *w + (alpha*(Xn*(yn-sigmoid(w,Xn,b))))
        intercept_update = (1 - (alpha*eta0)/N) *b + (alpha*(yn-sigmoid(w,Xn,b)))

        w = weight_update
        b = intercept_update

    # y_train_pred = map(lambda i: sigmoid(w,i,b), X_train)
    y_train_pred = sigmoid(w,X_train,b)
    # y_train_pred = [i for i in map(lambda i: sigmoid(w,i,b), X_train)]
    # y_test_pred = map(lambda i: sigmoid(w,i,b), X_test)
    # print(y_train_pred, y_test_pred)
```

```
    # loss = compute_log_loss(y_train,y_train_pred)
    loss = reg_cost(y_train, y_train_pred, alpha, N)

    train_loss.append(loss)

    print(epoch,'\t',loss)
```

```
epoch    log loss
0        0.648641058734492
1        0.5796761151982028
2        0.527141351882481
3        0.48670791601570557
4        0.455873671995446
5        0.430939336471167
6        0.4097124751721089
7        0.392063309197011
8        0.37773697139374757
9        0.36577624493226857
10       0.3555205457254072
11       0.34609859502364737
12       0.3385354004717094
13       0.3319281027034656
14       0.32568497952638614
15       0.3206763420035529
16       0.31532928566413015
17       0.31086352367536974
18       0.3072022642420477
19       0.30345680127099706
20       0.30059266207467894
21       0.29746507064475836
22       0.29427999704919083
23       0.29161706541432064
24       0.2893653514641676
25       0.28738234021450926
26       0.2853683135933077
27       0.2835310269828128
28       0.28160921334535677
29       0.2801772393612103
```

```
%matplotlib inline
plt.plot(train_loss, label = 'platt scaled ycv')
```

```
plt.title("Log-loss")
plt.xlabel("Epoch")
plt.ylabel("Log loss")
plt.legend()
plt.show();
```



Platt probability scores for Optimal w and b

```
sigmoid(w, svmrbf, b)[:5]
```

```
array([0.0439985 , 0.18208117, 0.09107683, 0.85066219, 0.80443747])
```