# 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 $\alpha_i$ Check the documentation for better understanding of these attributes: [https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html (https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html)](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 $\alpha_i$ 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 (https://scikit-learn.org/stable/modules/svm.html#svm-mathematical-formulation)](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 [ ]:
```
import numpy as np
import pandas as pd
from sklearn.datasets import make_classification
import numpy as np
from sklearn.svm import SVC
from tqdm import tqdm
```

In [ ]:
```
X, y = make_classification(n_samples=5000, n_features=5, n_redundant=2,
                           n_classes=2, weights=[0.7], class_sep=0.7, random_stat
```

### 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 $y_i$, $\alpha_i$, 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 [ ]:
```
# split test train and cross validation data
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.4, random_
X_cv,X_test, y_cv, y_test = train_test_split(X_test, y_test, test_size = 0.5, ran
# fit RBF svc to Xtrain data
clf_svm = SVC(gamma=0.001, C= 100)
clf_svm.fit(X_train,y_train)
# support vector indices
sv_indices = clf_svm.support_
sv_dual_coeff = clf_svm.support_vectors_
```

In [ ]:
```
def rbf(xi,xq, gamma):
  p1 = np.sum((xi-xq) **2,axis= -1)
  return np.exp(-gamma * p1)
```

In [ ]:
```
def getKernel(supportVectors, X, gamma):
  Kernels = np.zeros((X.shape[0], supportVectors.shape[0]))
  for indices, point in enumerate(X):
    for idexes, vector in enumerate(supportVectors):
      rbf_ = rbf(point, vector, gamma)
      Kernels[indices][idexes] = rbf_
  return Kernels
```

In [ ]:
```
def decision_function_custom(X, intercept, dual_coeff, support_vector, gamma ) :
  Kernels = getKernel(support_vector, X,  gamma)
  decision_custom = np.sum(dual_coeff * Kernels, axis = -1) + intercept
  return decision_custom
```

In [ ]:
```
custom_decision = decision_function_custom(X_cv,clf_svm.intercept_, clf_svm.dual_
```

In [ ]:
```
custom_decision
```

Out[31]:
```
array([-4.54631892e+00, -3.18769119e+00,  1.62139697e+00,  8.61360038e-02,
        1.75246241e+00, -9.76830808e-01, -3.20246796e+00, -2.62082863e+00,
       -2.42937891e+00,  1.57993649e+00, -2.06329703e+00,  9.02139954e-01,
       -2.49633212e+00, -3.12153041e+00,  2.98793441e-01, -9.79851167e-02,
       -2.10058057e+00, -3.05987657e+00,  6.09426723e-01, -2.28247363e+00,
        1.83705137e+00, -1.49963660e+00,  1.64502533e+00,  1.74774640e+00,
        8.64699998e-01, -2.44402367e+00, -2.77765859e+00,  2.81547371e+00,
        5.69770433e-01, -2.75397962e+00, -3.38300646e+00, -2.98418538e+00,
       -3.96622930e+00, -3.17731359e+00, -2.46000821e+00, -2.51646933e+00,
       -3.78091581e+00, -2.92728890e+00,  1.55489810e+00,  1.38727315e+00,
        1.44977701e+00,  9.80246803e-01, -3.50913018e+00,  3.68527290e+00,
        9.93865029e-01,  1.64063405e+00,  1.44693130e+00, -3.46194602e+00,
       -2.92257250e+00, -3.04151815e-01, -2.88716371e+00, -2.01001616e+00,
       -2.96415173e+00, -3.93243751e+00, -3.16152581e+00, -3.16365721e+00,
        2.31341730e+00, -3.37939147e+00, -7.22598505e-01,  2.08339272e+00,
       -2.13517154e+00,  1.48945168e+00, -1.96099648e+00,  1.81353239e+00,
       -1.72974263e+00, -4.05107267e+00, -1.25869622e+00, -3.50988882e+00,
        2.22264304e+00, -1.55737048e+00,  1.82902013e+00, -3.13187903e+00,
       -2.06958284e+00,  3.06721490e-01, -3.38649015e+00, -2.16142201e+00,
```

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

Check this [PDF (https://drive.google.com/open?id=133odBinMOIVb_rh_GQxxsyMRyW-Zts7a)](https://drive.google.com/open?id=133odBinMOIVb_rh_GQxxsyMRyW-Zts7a)

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

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

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

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

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

# 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

In [ ]:
```python
def getCalibratedProb(Y):
  N_Positive = np.count_nonzero(Y)
  N_negative = len(Y) - N_Positive
  Calibrated_Y_positive = (N_Positive + 1) / (N_Positive + 2)
  Calibrated_Y_negative = 1 / (N_negative + 2)
  return Calibrated_Y_positive, Calibrated_Y_negative
```

In [ ]:
```python
Calibrated_Y_positive, Calibrated_Y_negative = getCalibratedProb(y_cv)
print("Calibrated_Y_positive ", Calibrated_Y_positive)
print("Calibrated_Y_negative ", Calibrated_Y_negative)
```

```
Calibrated_Y_positive  0.9967637540453075
Calibrated_Y_negative  0.0014388489208633094
```

In [ ]:
```python
def updateY(Y, y_train ):
  updated_y = []
  Calibrated_Y_positive, Calibrated_Y_negative = getCalibratedProb(y_train)
  for p in Y:
    if p == 1:
      updated_y.append(Calibrated_Y_positive)
    else:
      updated_y.append(Calibrated_Y_negative)
  return updated_y
```

In [ ]:
```
def sigmoid(w,x,b):
  z=np.dot(x,w.T)+b
  return 1/(1+np.exp(-z))
```

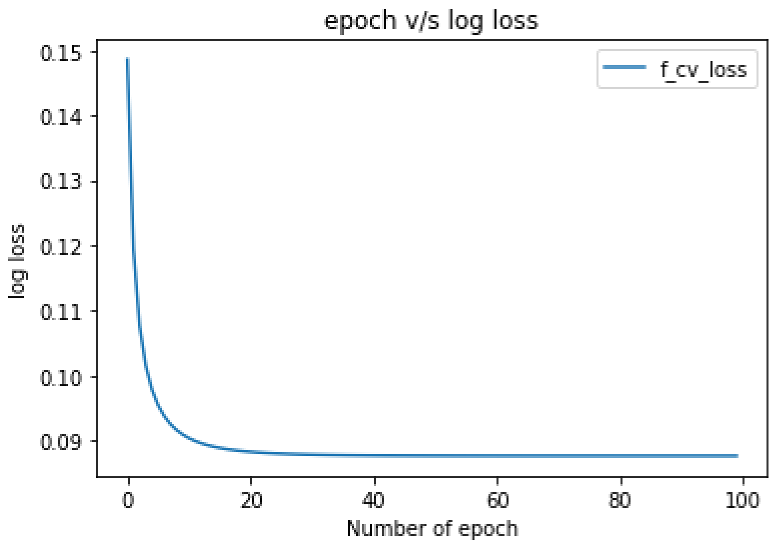
In [ ]:
```
def log_loss(W,b,X,Y): #log loss function N=len(X)
  sum_log=0
  for i in range(N):
    sum_log+=Y[i]*np.log10(sigmoid(W,X[i],b)) + (1-Y[i])*np. log10(1-sigmoid(W,X[
  return -1*sum_log/N
```

In [ ]:
```
def getoptimized_w_b():
  N = len(custom_decision)
  f_cv = custom_decision
  w = np.zeros_like(f_cv[0],)
  b = 0
  eta0 = 0.001
  alpha = 0.0001
  cvLoss = []
  EPOCH = 1
  y = updateY(y_cv,y_train)
  for epoch in tqdm(range(0,100)):
    for j in (range(N)):
      w =  ((1-((eta0*alpha)/N)) * w) + (eta0 * f_cv[j]) * (y[j] -sigmoid(w,f_cv[
      b = b + (eta0)*(y[j] - sigmoid(w,f_cv[j],b))
    losss = log_loss(w,b,f_cv,y)
    cvLoss.append(losss)
  return cvLoss,w,b
```

In [ ]:
```
cvLoss,w,b=getoptimized_w_b()
```

```
100%|██████████| 100/100 [00:03<00:00, 30.99it/s]
```

In [ ]:
```
import matplotlib.pyplot as plt
epoch=np.arange(0,100)
plt.plot(epoch,cvLoss,label='f_cv_loss')
plt.legend()
plt.xlabel("Number of epoch")
plt.ylabel("log loss")
plt.title("epoch v/s log loss")
plt.show()
```




In [ ]:
```
optimized_w = w
optimized_b = b
print(f"optimized_w {optimized_w} optimized_b {optimized_b}" )
```

```
optimized_w 1.8663229498110596 optimized_b 0.16670021101813404
```

In [ ]:
```
f_test = decision_function_custom(X_test,clf_svm.intercept_, clf_svm.dual_coef_,
```

In [ ]:
```
probs = sigmoid(optimized_w, f_test, optimized_b)
for prob in probs:
  print(prob)
```

```
0.004075255838046246
0.9686522507821004
0.19485616612913761
0.009435096499164143
0.381194357275506
0.006306669890403943
0.003827468684052202
0.051442179910544304
0.962633415491325
0.021959119241278725
0.8448423460243107
0.00290510743492829
0.0025784374968731744
0.06361316786753854
0.006365527694801174
0.9367295594322528
0.018197705576088057
0.9253246368560888
0.0009557860530703487
```

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