Albert Osom
STA 6107
Report

Problem 1

1.1

Write a Python code to solve the LS-SVM problem using matrix in- version. Evaluate the performance of your LS-SVM by evaluating the percentage or rate of misclassification on the test set.

Solution

LS-SVM problem is postulated by solving the following constraint minimization problem

$$\begin{aligned} \min_{w,b,e} \quad & \frac{1}{2}||w||^2 + \frac{C}{2}\sum_{i=1}^{N} e_i \\ \text{such that} \quad & Y_i(w'\varphi(x_i) + b) = 1 - e_i, \quad i = 1, 2, 3, .., N \end{aligned}$$

The lagrangian associated with this problem is

$$L(w, b, e, \alpha) = \frac{1}{2}||w||^2 + \frac{C}{2}\sum_{i=1}^{N} e_i - \sum_{i=1}^{N} \alpha_i(Y_i(w'\varphi(x_i) + b) - 1 + e_i)$$

Taking the partial derivative of the lagrangian and setting them to zero, we will obtain the following equations in matrix form as

$$\begin{bmatrix} 0 & \mathbf{Y}' \\ \mathbf{Y} & \Omega + \frac{1}{C}\mathbf{I} \end{bmatrix} \begin{bmatrix} b \\ \alpha \end{bmatrix} = \begin{bmatrix} 0 \\ \mathbf{1} \end{bmatrix}$$

where $\mathbf{1} = (1, 1, 1, ..., 1)'$, $\mathbf{Y} = (Y_1, Y_2, .., Y_N)'$ and $\alpha = (\alpha_1, \alpha_2, ...\alpha_N)'$ are n $\times$ 1

$\mathbf{I}$ is the identity matrix of rank N and Ω is the matrix is

$$\Omega = [y_i y_j k(x_i, x_j)]_{i,j=1}^{N} \text{ with elements } \Omega_{i,j} = y_i y_j k(x_i, x_j)$$

if we let

$$A = \begin{bmatrix} 0 & \mathbf{Y}' \\ \mathbf{Y} & \Omega + \frac{1}{C}\mathbf{I} \end{bmatrix}_{(N+1,N+1)} \quad x = \begin{bmatrix} b \\ \alpha \end{bmatrix}_{(N+1,1)}, B = \begin{bmatrix} 0 \\ \mathbf{1} \end{bmatrix}_{(N+1,1)}$$

then we can obtain x by solving $x = A^{-1}B$

We will implement this method in Python to obtain the solution for b and α which would be used to classify the observations.

We first define the gaussian kernel

$$K(x, y) = exp\left(\frac{-||x - y||^2}{\sigma^2}\right)$$

```
def rbf_kernel(x1,x2,gamma):
    u=np.exp(-np.matmul((x1-x2).T,(x1-x2))/gamma**2)
    return(u)
```

The following function "LS_SVMtrain" computes b and α

```
def LS_SVMtrain(X,Y,C,gamma=1.5):

    N=len(Y)
    Dm=np.zeros((N,N))
    #X=matrix(X);
    #Y=matrix(Y,nrow=N)
    Y_prime=np.transpose(Y)
    zeros=np.zeros((1,1))

    top=np.hstack([zeros,Y_prime])

    for i in np.arange(N):
        for j in np.arange(N):
            Dm[i,j]=Y[i,:]*Y[j,:]*rbf_kernel(X[i,:],X[j,:],gamma)
    Dm=Dm+np.eye(N)*(1/C)

    down=np.hstack([Y,Dm])

    A=np.vstack([top,down])

    ones=np.ones((N,1))

    B= np.vstack([0,ones])

    alpha=np.matmul(np.linalg.inv(A),B)

    b = alpha[0]
    alpha_star=alpha[1:,]

    return({'alpha':alpha_star,'b':b, 'Xv':X, 'Yv':Y, 'gamma':gamma})
```

We now classify observations using the following function

$$g(u) = sign(f(u)) = sign\left(\sum_j \alpha_j y_j K(u, x_j) + b\right)$$

The function "LS_svmpredict " is defined and used to predicts the observation into one of the two labels $(i.e. \{-1, 1\})$

```
def LS_svmpredict(X,model):
    alpha=model['alpha']
    b=model['b']
    Yv=model['Yv']
    Xv=model['Xv']
    #C=model['C']
    gamma=model['gamma']
    result=np.empty(len(X))

    for j in np.arange(len(X)):

        ayK=np.empty(len(Xv))

        for i in np.arange(len(Xv)):

            if(abs(alpha[i]*Yv[i]*rbf_kernel(Xv[i,:],X[j,:],gamma))==0):
                ayK[i]=0

            else:
                ayK[i] =alpha[i,:]*Yv[i,:]*rbf_kernel(Xv[i,:],X[j,:],gamma)

                #ayK[i] =rbf_kernel(Xv[i,:],X[4,:],gamma)

        result[j] = np.sign(np.sum(ayK)+b)
    return(result)
```

- We can now train our model using the training set and predict using testing set. Choosing $C = 3$ and $\sigma = 1/\sqrt{0.01}$

```
model=LS_SVMtrain(X_train,Y_train,C=3,gamma=1/np.sqrt(0.01))

Y_pred=LS_svmpredict(X_test,model)

Y_pred=Y_pred.reshape((400,1))

np.unique(Y_pred,return_counts=True)

from sklearn.metrics import confusion_matrix

confusion_matrix(Y_test,Y_pred,labels=(-1,1))
```

```
array([[132,  68],
       [ 92, 108]], dtype=int64
```

Therefore the percentage of misclassification is obtained by

$$\frac{92+68}{132+68+92+108} \times 100\% = \frac{160}{400} \times 100\% = 40\%$$

1.2

Use the large scale algorithm (Hestenes-Stiefel algorithm) discussed in Suykens et al. (1999), available on the class webcourse, to solve the LS-SVM problem. Evaluate the performance of your LS-SVM by evaluating the percentage or rate of misclassification on the test set.

Solution

LS-SVM- Large Scale Algorithm is given below

1. Solve η, v from $\Omega\eta = Y$ and $\Omega v = \mathbf{1}$ using Conjugate Gradient Method
2. Compute $s = Y^T\eta$
3. Find solution
   $b = \eta^T d_2/s$
   $\alpha = v - \eta b$

where $\mathbf{1} = (1,1,1,...,1)'$, $\mathbf{Y} = (Y_1, Y_2, ..., Y_N)'$ and $\alpha = (\alpha_1, \alpha_2, ..., \alpha_N)'$ are $N \times 1$

We want to to use the LS-SVM- Large Scale Algorithm to solve the following problem below

$$\begin{bmatrix} 0 & \mathbf{Y}' \\ \mathbf{Y} & \Omega + \frac{1}{C}\mathbf{I} \end{bmatrix} \begin{bmatrix} b \\ \alpha \end{bmatrix} = \begin{bmatrix} 0 \\ \mathbf{1} \end{bmatrix}$$

where $\mathbf{1} = (1,1,1,...,1)'$, $\mathbf{Y} = (Y_1, Y_2, .., Y_N)'$ and $\alpha = (\alpha_1, \alpha_2, ...\alpha_N)'$ are n $\times$ 1

I is the identity matrix of rank N and Ω is the matrix is

$$\Omega = [y_i y_j k(x_i, x_j)]_{i,j=1}^N \text{ with elements } \Omega_{i,j} = y_i y_j k(x_i, x_j)$$

We first define the Conjugate Gradient Method in Python so we can use when implementing the Large Scale LSSVM.

The conjugate gradient algorithm is use for solving $Ax = \beta$

The algorithm to obtain solution x is below:

1. initialize $i = 0; x_0 = 0; r_o = \beta$

```
2. while $r_i \neq 0$
   i=i+1
   if i=1
   $p_1 = r_0$
   else
   $\beta_i = r_{i-1}^T r_{i-1} / r_{i-2}^T r_{i-2}$
   $p_i = r_{i-1} + \beta_i p_{i-1}$
   end
   $\lambda_i = r_{i-1}^T r_{i-1} / p_i^T A p_i$
   $x_i = x_{i-1} + \lambda_i p_i$
   $r_i = r_{i-1} - \lambda_i A p_i$
   end
   $x = x_i$
```

we wrote a function that implements the algorithm and it is given below

```
def conjugate_gradient(x0,r0,A,eps=1e-16):
    i=0
    r_prev2=r0
    x_prev=x0
    r_new=r0
    for i in np.arange(1,10000000):

        if abs(np.max(r_new)) <= eps:
            break
        else:
        #i=i+1

            if i==1:
                p_new=r0
                r_prev=r_prev2
            else:
                B_new= np.matmul(r_prev.T,r_prev)/np.matmul(r_prev2.T,r_prev2)
                p_new= r_prev + B_new*p_prev
            lam=np.matmul(r_prev.T,r_prev)/np.matmul(np.matmul(p_new.T,A),p_new)
            x_new=x_prev + lam*p_new

            r_new=r_prev-lam*np.matmul(A,p_new)

            r_prev2=r_prev

            r_prev=r_new

            p_prev=p_new

            x_prev=x_new

            x=x_new
```

```
          #if(i==1000):

              #print(x_new)

    return({'x':x_new,'r':r_new,'iteration':i})
```

Now, we are ready to solve for b and α in the SVM problem using the Large Scale SVM algorithm.

The following Python codes define the function "Large_Scale_LSSVM" that solves for b and α using the algorithm discussed.

```
def Large_Scale_LSSVM(X,Y,C,gamma):

    N=len(Y)
    Dm=np.zeros((N,N))

    for i in np.arange(N):
        for j in np.arange(N):
            Dm[i,j]=Y[i,:]*Y[j,:]*rbf_kernel(X[i,:],X[j,:],gamma)

    H=Dm+np.eye(N)*(1/C)

    d2=np.ones((N,1))

    x0=np.zeros(N)

    x0=x0.reshape([N,1])

  ### computing eta and v, step 1 of the algorithm

    eta=conjugate_gradient(x0,Y,H)

    eta=eta['x']

    v=conjugate_gradient(x0,d2,H)

    v=v['x']

    ### step 2 of the algorithm

    s=np.matmul(Y.T,eta)

    ### obtaining b and v using formula in step 3 of algorithm

    b=np.matmul(eta.T,d2/s)
    alpha=v-eta*b

    return({'alpha':alpha,'b':b, 'Xv':X, 'Yv':Y, 'gamma':gamma})
```

We can now train our model using the training set and predict using testing set. Choosing $C = 3$ and $\sigma = 1/\sqrt{0.01}$

```
model_LS=Large_Scale_LSSVM(X_train,Y_train,C=3,gamma=1/np.sqrt(0.01))

Y_pred=LS_svmpredict(X_test,model_LS)

Y_pred=Y_pred.reshape((len(Y_pred),1))

from sklearn.metrics import confusion_matrix

confusion_matrix(Y_test,Y_pred,labels=(-1,1))

array([[132,  68],
       [ 92, 108]], dtype=int64)
```

Therefore the percentage of misclassification is obtained by

$$\frac{92+68}{132+68+92+108} \times 100\% = \frac{160}{400} \times 100\% = 40\%$$

Conclusion
The two methods discussed produce consistent results

Problem 2

We will use Python for this problem. Also, we will use ridge regression to model our response.

1. Use the ridge regression formula discussed in lectures to obtain the ridge estimators $\hat{\beta_R}$. Then, evaluate the training and testing errors.

Solution

To obtain the ridge regression estimators $\hat{\beta_R}$, we will solve

$$\hat{\beta_R} = (X'X + \lambda I)^{-1} X'Y$$

Data Processing
We will delete some variables that are not needed and obtain our training and testing set. For the purpose of comparison, we decided to scale both the response variable and the predictors so our results can be consistent with the results from the "glmnet" package. The following codes, does that

```
library(readxl)
library(glmnet)

data<-read_excel('Residential-Building-Data-Set.xlsx',
sheet='Data',skip=1,col_names=TRUE)

df<-data.frame(data)

### Deleting categorical variables

df<-df[,-c(1,2,3,4,5,17,26,36,56,74,93)]

###Splitting to training set
X_trainset<-df[1:50,]

Y_trainset<- X_trainset[,97]

X_trainset<-X_trainset[,-(97:98)]

X_trainM<-as.matrix(X_trainset)

Y_trainM<-as.matrix(Y_trainset)

str(X_trainset)

### Splitting to testing set

X_testset<-df[51:60,]

Y_testset<- X_testset[,97]

X_testset<-X_testset[,-(97:98)]

X_testM<-as.matrix(X_testset)

Y_testM<-as.matrix(Y_testset)

###### scaling Data

X_trainM_S<-scale(X_trainM)

X_testM_S<-scale(X_testM)

Y_trainM_S<-scale(Y_trainM)

Y_testM_S<- scale(Y_testM)
```

Obtaining λ value from cross-validation using function from the "glmnet" package.

```
set.seed(2)
cv.out=cv.glmnet(X_trainM,Y_trainM_S,alpha=0,standardize = TRUE)

plot(cv.out)

bestlam =cv.out\$lambda.min

bestlam
9.497212
```

The lambda obtained from cross validation is = 9.497212

Now that we have our data sets, we are now ready to compute the beta parameters of the ridge regression. The following Python codes computes the parameters.

```
lamb=9.497212
N=X_trainM.shape[1]

A=np.linalg.inv(np.matmul(X_trainM.T,X_trainM)+ np.eye(N)*lamb)

B=np.matmul(X_trainM.T,Y_trainM_S)

beta_coef=np.matmul(A,B)
```

We now compute our $MSE = \frac{1}{N}\sum_{i=1}^{N}(Y-\hat{Y})^2$ where $\hat{Y} = X\hat{\beta_R}$

```
######## MSE for training set

Ridge_pred_train=np.matmul(X_trainM,beta_coef)

MSE=np.mean((Ridge_pred_train-Y_trainM_S)**2)

MSE

0.046970006503752416

####### MSE for testing set

Ridge_pred_test=np.matmul(X_testM_S,beta_coef)

MSE=np.mean((Ridge_pred_test-Y_testM_S)**2)

MSE

0.035727
```

2.2

Use the Least Squares formula discussed in lectures with the augmented data approach to obtain the ridge estimators $\hat{\beta}_R$. Again, evaluate the training and testing errors.

Solution

We want to obtain the parameters of our ridge regression model using augmented data approach. We will obtain the coefficients as

$$\hat{\beta} = (X'_\lambda X_\lambda)^{-1} X'_\lambda Y_\lambda$$

where

$$X_\lambda = \begin{bmatrix} X \\ \sqrt{\lambda} I_p \end{bmatrix}_{(N+p,p)} \quad Y_\lambda = \begin{bmatrix} Y \\ \mathbf{0} \end{bmatrix}_{(N+p,1)}$$

We now implement this in Python to compute the coefficients.

```
p=X_trainM.shape[1]

lamb_matrix=np.eye(p)*np.sqrt(lamb)

X_lamb=np.vstack([X_trainM,lamb_matrix])

zeros=np.zeros(p)

zeros=zeros.reshape([p,1])

Y_lamb=np.vstack([Y_trainM_S,zeros])


A_lamb= np.linalg.inv(np.matmul(X_lamb.T,X_lamb))

B_lamb=np.matmul(X_lamb.T,Y_lamb)

beta_coef=np.matmul(A_lamb,B_lamb)
```

We now compute our $MSE = \frac{1}{N}\sum_{i=1}^{N}(Y - \hat{Y})^2$ where $\hat{Y} = X\hat{\beta}_R$

```
######## MSE for training set

Ridge_pred_train=np.matmul(X_trainM,beta_coef)

MSE=np.mean((Ridge_pred_train-Y_trainM_S)**2)

MSE
```

```
0.046970006503752416

####### MSE for testing set

Ridge_pred_test=np.matmul(X_testM_S,beta_coef)

MSE=np.mean((Ridge_pred_test-Y_testM_S)**2)

MSE

0.035727
```

Results from "glmnet" R package

```
########### R package ############

### Training set
> ridge.pred_test=predict(ridge.mod,s=bestlam,newx=X_testM_S)
> mean((ridge.pred_test - Y_testM_S)^2)
[1]0.2345413

#### Testing set
ridge.pred_test=predict(ridge.mod,s=bestlam,newx=X_testM_S)
> mean((ridge.pred_test - Y_testM_S)^2)
[1] 0.09133485
```

MSE	Ridge(3.d.p)	Ridge Augmented(3.d.p)	glmnet package (3.d.p)
Training	0.047	0.047	0.23
Testing	0.035	0.035	0.091

As expected, results from the Ridge and the augmented methods coincide but the results from the R package are little higher

Problem 3

3.1

Use the lasso coordinate descent solution with p = 1 to model the response 'Sale Prices' using the first continuous variable, i.e. by finding the lasso estimator $\hat{\beta}_L$.

Solution

We will compute the $\hat{\beta}_L$ as follows

$$\hat{\beta_L} = \begin{cases} \sum X_i Y_i - \lambda & if \quad \sum X_i Y_i > \lambda \\ \sum X_i Y_i + \lambda & if \quad \sum X_i Y_i < -\lambda \\ 0 & if \quad \sum X_i Y_i \in [-\lambda, \lambda] \end{cases}$$

Choosing λ to be 0.1 we will implement the algorithm to compute the regression coefficient as follows

We first define the soft-thresholding operator

```
soft_threshold <- function(a, lambda) {
    if (a  < (-lambda)){
        return(a+lambda)
    }
    if (a > lambda){
        return(a-lambda)
    }

    else {
        return(0)
    }
}
```

Lets now compute the lasso estimator of $\hat{\beta_L}$. Since we standardized both the response and the predictor, the intercept parameter is zero.

```
X<-X_trainM_S[,1]

y<-Y_trainM_S

a<-crossprod(X,y)

lambda<-bestlam

soft_threshold(a,lambda=0.1)
 3.565946
```

Therefore $\hat{\beta_L} = 3.565946$

3.2

Use the lasso coordinate descent solution with all continuous variables to model the response 'Sale Prices', i.e. by finding the lasso estimator $\hat{\beta_L}$.

Solution

The coordinate descent algorithm updates each coordinate of the lasso estimator using the following update

$$\hat{\beta}_j^{\,new} = \begin{cases} \sum X_{ij}r_i - \lambda & if \quad \sum X_{ij}r_i > \lambda \\ \sum X_{ij}r_i + \lambda & if \quad \sum X_{ij}r_i < -\lambda \\ 0 & if \quad \sum X_{ij}r_i \in [-\lambda, \lambda] \end{cases}$$

where $r_i = Y_i - \sum_{k \neq j} X_{ik}\hat{\beta}_k = Y_i - \hat{Y}_i + X_j\hat{\beta}_j$

We will now implement the algorithm in R. We choose our initial start point to be the solution from the ridge regression. The function "lasso" below implements the algorithm and outputs the lasso coefficients.

We adopt our stopping rule to be similar to the one used in "lasso2.r" in webcourses. The rule compares the change in the average loss a window of length 'window/2' to determine if the average loss has changed more than ('tol') some tolerance .

```
should_stop <- function(loss, window, k, tol = 1e-10) {
  window_length = floor(window / 2) - 1
  prev_loss <- mean(loss[(k-(window-1)):(k-(window-1)+window_length)])
  curr_loss <- mean(loss[(k-(window-1)+window_length):k])

  abs(curr_loss - prev_loss) < tol
}
```

We use this in our lasso function below

```
lasso <- function(
    X,                    # model matrix
    y,                    # target
    lambda  = .1,         # penalty parameter
    soft    = TRUE,       # soft vs. hard thresholding
    tol     = 1e-6,       # tolerance
    iter    = 100        # number of max iterations
    ){

    #### ridge regression  solution

    beta= solve(crossprod(X) + diag(lambda, ncol(X))) %*% crossprod(X,y)

    i = 1

    loss<-numeric(ncol(X))

    while ( i < iter){

        beta_old=beta

        # the full residual used to check the value of the loss function
        residual <- y - X %*% beta_old
        loss[i] <- mean(residual * residual)
        for (j in 1:ncol(X)) {
            # partial residual (effect of all the other co-variates)
            r_j <- residual + X[, j] * beta[j]

            # single variable OLS estimate
            beta_ols_j <- mean(r_j * X[, j])

            # soft-threshold the result
            beta[j] <- soft_thresh(beta_ols_j, lambda)
            residual <- r_j - X[, j] * beta[j]

        }

    #}
        if (i > 10) {

        if (should_stop(loss, k = i, window = 10, tol = tol)) {
            break
        }
        }

    i = i + 1

    if (i==2){
        print(beta[1:4])
    }

}
    print(i)

    beta
}
```

We obtained our lambda value through cross validation, we used the "glmnet" package in R.

```
set.seed(2)
cv.out=cv.glmnet(X_trainM_S,Y_trainM_S,alpha=1,standardize = FALSE)

bestlam_lasso =cv.out$lambda.min

bestlam_lasso
 0.009497212
```

Finding parameters to fit model to obtain coefficients

```
lambda=bestlam_lasso
```

```
beta_coef<-lasso(X_trainM_S,Y_trainM_S,iter=1000,lambda)

zero_lasso<-beta_coef==0

### Lasso Estimators of beta

beta_coef1<-beta_coef[!zero_lasso]

as.matrix(beta_coef1,ncol=3)

              [,1]
 [1,]  3.235834e-03
 [2,]  9.288885e-02
 [3,]  7.966353e-01
 [4,]  7.223034e-02
 [5,] -1.428939e-01
 [6,]  6.980123e-03
 [7,]  5.351404e-03
 [8,]  2.858713e-03
 [9,]  8.803192e-03
[10,]  6.706099e-03
[11,]  6.917371e-02
[12,]  3.123220e-02
[13,]  3.439154e-03
[14,]  1.330731e-02
[15,] -6.390231e-03
[16,]  5.381846e-03
[17,]  1.621516e-01
[18,]  7.850656e-05
[19,] -2.023587e-02
[20,] -6.591092e-02
[21,]  5.028220e-03
[22,]  7.887148e-02
[23,]  3.482375e-05
```

3

From (2), how many coefficients $\hat{\beta}_j$. are equal to 0? What can you conclude in terms of model selection?

```
length(beta_coef[beta_coef==0])

[1] 74
```

We can conclude that the lasso regularization select 23 features to be included in the model out of 97 features.

3.4

Compare your results from (1) and (2) with the ones using the ‘glmnet’ package from R.

Solution

From (2), it is best to compare the MSEs since the ”glmnet” package from R included some features in the model but our algorithm excluded them and vice versa

”glmnet” Results

```
lasso.mod=glmnet(X_trainM_S,Y_trainM_S,alpha=1,thresh=1e-12,standardize = FALSE)

lasso_coef<-predict(lasso.mod,s=bestlam_lasso,type="coefficients")

lasso_coef<-as.vector(lasso_coef)

non_zero_coef<-lasso_coef[lasso_coef!=0]

as.matrix(non_zero_coef[-1])
              [,1]
 [1,]  0.093314430
 [2,]  0.886242069
 [3,]  0.055996586
 [4,] -0.175340676
 [5,]  0.002501534
 [6,]  0.005291476
 [7,]  0.006183039
 [8,]  0.092767631
 [9,]  0.021524334
[10,] -0.008865563
[11,]  0.191649213
[12,]  0.010534389
[13,] -0.099533337
[14,]  0.042099233
[15,]  0.001835725
```

The ”glmnet” R package included 15 features in the model

We want to compare the training and testing error between our lasso algorithm and the results from the ”glmnet” package

```
#### our lasso function

######## training set

Y_pred<-X_trainM_S%*%beta_coef
> mean((Y_pred - Y_trainM_S)^2)
```

```
[1] 0.01604175

########## testing set

> Y_pred<-X_testM_S%*%beta_coef

> mean((Y_pred - Y_testM_S)^2)
[1] 0.01209568

###### R package ############

##### training set
lasso.pred=predict(lasso.mod,s=bestlam_lasso,newx=X_trainM_S)
mean((lasso.pred - Y_trainM_S)^2)

[1] 0.01256805

##### Testing set
lasso.pred=predict(lasso.mod,s=bestlam_lasso,newx=X_testM_S)
mean((lasso.pred - Y_testM_S)^2)

0.01298502
```

MSE	Lasso(3.d.p)	glmnet package (3.d.p)
Training	0.016	0.013
Testing	0.012	0.013

The MSEs from both training set and testing set of both methods seems to be consistent.

From (1), we cannot compute the estimate using "glmnet" package because we have only one feature.