MASSEY UNIVERSITY
Institute of Fundamental Sciences / School of Engineering and Advanced Technology

161.326 Statistical Machine Learning

Solutions to Assignment 3, 2010

Question 1

CODE

```
from pylab import *
from numpy import *

def fit_beta(X1,y):
    # fits a logistic regression model to the data in X1,y
    N = len(y)        # number of data
    P1 = X1.shape[1]  # number of predictors
    # initialize
    betaold = zeros((P1,1))
    tol = 0.000001
    stepdiff = 1
    while stepdiff > tol:
        p = exp(X1*betaold)
        p = p/(1+p)
        temp = multiply(p,(ones(shape(p)) - p))
        W = zeros((N,N))
        for j in range(N):
            W[j,j] = temp[j,0]
        betanew = betaold +
(linalg.inv(transpose(X1)*W*X1))*transpose(X1)*(matrix(y) - p)
        stepdiff = abs(betaold - betanew).max()
        betaold = betanew
    return betaold

def error_rate(X1,betahat,y):
    # calculates the error rate for a predictor betahat on data in X1,y, and
returns z-scores
    N = len(y) # number of predictors
    p = exp(X1*betahat)
    p = p/(1+p)
    yhat = (p >= 0.5)
    errorrate = (0. + N - sum(y == yhat))/N
    # check z-scores
    temp = multiply(p,(ones(shape(p)) - p))
    W = zeros((N,N))
    for j in range(N):
        W[j,j] = temp[j,0]
    varbeta = linalg.inv(transpose(X1)*W*X1)
    z = multiply(betahat,1/transpose(matrix(sqrt(diag(varbeta)))))
    return errorrate,concatenate((y,p),axis = 1),concatenate((betahat,z),axis=1)

data = loadtxt('saheartdis.txt')
P = 9
data = data.reshape(-1,P+1)
# divide into 2/3 training, 1/3 test
traindata = data[0:308,:]
testdata = data[308:462,:]
```

```
## reformat data as X and Y
ytrain = traindata[:,9:10]
Ntrain = len(ytrain)
ytest = testdata[:,9:10]
Ntest = len(ytest)
K = 2  # number of classes
X = data[:,0:9]
X = matrix(X)
# add in quadratic terms, except for x4 (famhist)
for i in range(4):
    X = concatenate((X,multiply(X[:,i],X[:,i])), axis = 1)
for i in range(5,P,1):
    X = concatenate((X,multiply(X[:,i],X[:,i])), axis = 1)
X1 = concatenate((ones((shape(X)[0],1)),X),axis=1)

# divide into 2/3 training, 1/3 test
X1train = X1[0:308,:]
X1test = X1[308:462,:]

# fit full model
betanew = fit_beta(X1train,ytrain)
#print 'beta= ',betanew
print 'beta, z = ',error_rate(X1train,betanew,ytrain)[2]
print 'error rate (train) = ',error_rate(X1train,betanew,ytrain)[0]
print ''

# fit parsimonious models based on previous z-values
print 'error rate (test)'

Xselect0 = array([5, 7, 9])
betanew = fit_beta(X1train[:,Xselect0],ytrain)
print 'Model x5,x7,x9 =',error_rate(X1test[:,Xselect0],betanew,ytest)[0]
print 'betanew =',betanew
print ''

Xselect1 = array([0, 5, 7, 9])
betanew = fit_beta(X1train[:,Xselect1],ytrain)
print 'Model x0,x5,x7,x9 =',error_rate(X1test[:,Xselect1],betanew,ytest)[0]
print 'betanew =',betanew
print ''

Xselect2 = array([0, 5, 7, 9, 15])
betanew = fit_beta(X1train[:,Xselect2],ytrain)
print 'Model x0,x5,x7,x9,x7*x7
=',error_rate(X1test[:,Xselect2],betanew,ytest)[0]
print 'betanew =',betanew
print ''

Xselect3 = array([0, 3, 5, 7, 9, 15])
betanew = fit_beta(X1train[:,Xselect3],ytrain)
print 'Model x0,x3,x5,x7,x9,x7*x7
=',error_rate(X1test[:,Xselect3],betanew,ytest)[0]
print 'betanew =',betanew
print ''

Xselect_1 = array([5, 7])
betanew = fit_beta(X1train[:,Xselect_1],ytrain)
print 'Model x5,x7 =',error_rate(X1test[:,Xselect_1],betanew,ytest)[0]
print 'betanew =',betanew
print ''
```

OUTPUT

```
beta, z =  [[  1.44188467e+01   1.93864561e+00]
 [ -1.32219303e-01  -1.79871600e+00]
 [  4.80069260e-02   6.33585009e-01]
 [  4.81474340e-01   1.81961421e+00]
 [ -8.58362672e-02  -5.46764589e-01]
 [  7.19626836e-01   2.51385972e+00]
 [ -4.94802063e-02  -4.67278217e-01]
 [ -7.51629907e-01  -2.19791478e+00]
 [  1.14893427e-02   7.46893102e-01]
 [  1.80774292e-01   2.12714513e+00]
 [  4.16651093e-04   1.69039017e+00]
 [  6.27676327e-04   1.51869965e-01]
 [ -2.56223200e-02  -1.33041977e+00]
 [  3.02948801e-03   1.02871739e+00]
 [  9.12921080e-04   9.19561308e-01]
 [  1.09748352e-02   1.92246695e+00]
 [ -3.44675313e-05  -2.26708507e-01]
 [ -1.55062876e-03  -1.65345515e+00]]
error rate (train) =  0.279220779221

error rate (test)
Model x5,x7,x9 = 0.292207792208
betanew = [[ 0.82147749]
 [-0.1118822 ]
 [ 0.04785873]]

Model x0,x5,x7,x9 = 0.24025974026
betanew = [[-3.30797815]
 [ 0.78074739]
 [-0.0036852 ]
 [ 0.05699442]]

Model x0,x5,x7,x9,x7*x7 = 0.25974025974
betanew = [[ 1.69430717]
 [ 0.81554705]
 [-0.37325739]
 [ 0.0612015 ]
 [ 0.00635251]]

Model x0,x3,x5,x7,x9,x7*x7 = 0.227272727273
betanew = [[ 2.42491839]
 [ 0.15796926]
 [ 0.74392281]
 [-0.4534542 ]
 [ 0.05776885]
 [ 0.0074428 ]]

Model x5,x7 = 0.318181818182
betanew = [[ 0.98779285]
 [-0.03484767]]
```

DISCUSSION

The purpose of the question is NOT to use the test RSS to select a model, but to evaluate how well a policy of selecting a model containing all predictors with |z| > 2 on the training data performs. The additional predictors are chosen on the basis of the training data, and as the beta is likewise fitted to

the training data, adding them in does not necessarily improve the error rate on the test data. However, in this case, the selected model appears to be smaller than desirable. A stepwise approach might seem attractive, but we lack the statistical theory to decide whether a predictor to be added/dropped is significant/insignificant.

COMMENTS

Note that the beta is refitted on the training data restricted to the included predictors.

Question 2

CODE

```
from numpy import *
from pylab import *

def lda(X1,y1,X2,y2,K):
    # performs LDA using training data (X1,y1) and outputs error rate on test
data (X2,y2)
    N1 = len(y1)
    N2 = len(y2)
    nk = zeros((K,1))
    for k in range(K):
        nk[k] = sum(y1 == k)
    pi = (0. + nk)/N1
    #print "pi= ",transpose(pi)
    mu = zeros((K,X1.shape[1]))
    for k in range(K):
        for j in range(N1):
            mu[k:k+1,:] = mu[k:k+1,:] + (y1[j] == k)*X1[j:j+1,:]/nk[k]
    #print "mu= ",mu
    Sigma = zeros((X1.shape[1],X1.shape[1]))
    for k in range(K):
        for j in range(N1):
            temp = X1[j:j+1,:] - mu[k:k+1,:]
            Sigma = Sigma + (int(y1[j]) == k)*transpose(temp)*temp/(N1-K)
    #print "Sigma= ",Sigma
    # discriminant function
    invSigma = linalg.inv(Sigma)
    deltak = zeros((N2,K))
    for j in range(N2):
        for k in range(K):
            deltak[j,k] = (X2[j:j+1,:] -
0.5*mu[k:k+1,:])*invSigma*transpose(mu[k:k+1,:]) + log(pi[k])
    #print "deltak= ",deltak
    #classify
    deltakT = transpose(deltak)
    y2hatT = matrix(deltakT.argmax(0))
    y2hat = transpose(y2hatT)
    errorrate = (0. + N2 - sum(y2 == y2hat))/N2
    #print "errorrate= ",errorrate
    return errorrate,pi,mu,Sigma

def nbayesclass(X1,y1,X2,y2):
    # naive Bayes classifier for y = {0,1,2}, X a matrix of p predictors
    N1 = len(y1)
```

```
    N2 = len(y2)
    p = X1.shape[1]
    f0k = zeros(shape(X1),dtype=float) # density estimate for y = 0, kth
predictor
    f1k = zeros(shape(X1),dtype=float) # density estimate for y = 1, kth
predictor
    f2k = zeros(shape(X1),dtype=float) # density estimate for y = 2, kth
predictor
    for k in range(p):
        x = X1[:,k:k+1] # get kth predictor
        x0 = x[y1 == 0]  # and divide according
        x1 = x[y1 == 1]  # value of y
        x2 = x[y1 == 2]
        xk = X2[:,k:k+1] # test values, kth predictor
        lam = 0.05*(max(x) - min(x)) # bandwidth
        lsq = float(lam**2)
        # calculate density estimates
        for i in range(N2):
            xki = float(xk[i])
            #print pi,lsq,x0.shape,xk.shape
            K0 = (1/sqrt(2*pi))*exp(-(multiply(x0-xki,x0-xki))/(2*lsq))
            K1 = (1/sqrt(2*pi))*exp(-(multiply(x1-xki,x1-xki))/(2*lsq))
            K2 = (1/sqrt(2*pi))*exp(-(multiply(x2-xki,x2-xki))/(2*lsq))
            f0k[i,k] = sum(K0)/(lam*len(x0))
            f1k[i,k] = sum(K1)/(lam*len(x1))
            f2k[i,k] = sum(K2)/(lam*len(x2))
    # multiply over predictors
    f0 = transpose(matrix(f0k[:,0]))
    f1 = transpose(matrix(f1k[:,0]))
    f2 = transpose(matrix(f2k[:,0]))
    for k in range(p-1):
        f0 = multiply(f0,f0k[:,k+1:k+2])
        f1 = multiply(f1,f1k[:,k+1:k+2])
        f2 = multiply(f2,f2k[:,k+1:k+2])
    nbc = zeros(shape(y2))
    for i in range(N2):
        if f0[i] > f1[i]:
            if f0[i] > f2[i]:
                nbc[i] = 0
            else:
                nbc[i] = 2
        else:
            if f1[i] > f2[i]:
                nbc[i] = 1
            else:
                nbc[i] = 2
    errorrate = (0. + N2 - sum(y2 == nbc))/N2
    return errorrate

# get iris data
data = loadtxt('iris.txt')
p = 4
data = data.reshape(-1,p+1)
## reformat data as X and Y
y = data[:,p]
N = len(y)
y = transpose(matrix(y))
X = data[:,0:p]
X = matrix(X)
K = 3 # number of classes
```

```
M = 100 # number of random training/test assignments
ER = zeros((M,2)) # Storage for error rates
for m in range(M):
    # randomly divide data into training and test data
    setsize = int(N*2/3)
    setno0 = random_sample((N,))
    setno1 = sort(setno0)
    boundary = setno1[setsize]
    X1 = X[setno0 < boundary,:]
    y1 = y[setno0 < boundary]
    X2 = X[setno0 >= boundary,:]
    y2 = y[setno0 >= boundary]
    ldaer = lda(X1,y1,X2,y2,K)[0]
    nbcer = nbayesclass(X1,y1,X2,y2)
    ER[m,0] = ldaer
    ER[m,1] = nbcer
    print 'LDA | Naive Bayes error rate =',ldaer,nbcer

print ' '
print 'Mean Error Rate [LDA | NBC]=',mean(ER, axis = 0)
```

OUTPUT

```
LDA | Naive Bayes error rate = 0.02 0.08
LDA | Naive Bayes error rate = 0.0 0.04
LDA | Naive Bayes error rate = 0.02 0.06
...
LDA | Naive Bayes error rate = 0.02 0.08

Mean Error Rate [LDA | NBC]= [ 0.0218  0.0496]
```

DISCUSSION

- Linear discriminant analysis is obviously more powerful in this case, because the normality assumptions appear reasonable.
- The error rate in the Naive Bayes Classifier can be optimized by adjusting the bandwidth, using a leave-one-out approach. Note that optimizing on the training data without leaving one out results in a bandwidth near zero (reasonable, as then the only point contributing is the one being examined) and a zero error rate on the training data, but a very large error rate on the test data.