Stat 432 Homework 8

Assigned: Mar 15, 2019; Due: 11:59PM Mar 29, 2019

Before starting this homework, you should read the rlab file of Class on the course website carefully.

Question 1 (logistic regression) [5 points]

```
library(ElemStatLearn)
data(SAheart)

heart = SAheart
heart$famhist = as.numeric(heart$famhist)-1
n = nrow(heart)
p = ncol(heart)

heart.full = glm(chd~., data=heart, family=binomial)

# fitted value
# yhat = (heart.full$fitted.values>0.5)
# table(yhat, SAheart$chd)
```

```
# the coefficients and significance
round(summary(heart.full)$coef, dig=3)
```

We can write a function that computes the Hessian matrix based on page 12 of the lecture note. (1 point.)

```
# the negative log-likelihood function of logistic regression
my.loglik <- function(b, x, y)
{
    bm = as.matrix(b)
    xb =  x %*% bm
    # this returns the negative loglikelihood
    return(sum(y*xb) - sum(log(1 + exp(xb))))
}
# Gradient
my.gradient <- function(b, x, y)
{
    bm = as.matrix(b)
    expxb =  exp(x %*% bm)
    return(t(x) %*% (y - expxb/(1+expxb)))
}
# Hessian
my.hessian <- function(b, x, y)
{
    bm = as.matrix(b)
    expxb =  exp(x %*% bm)
    x1 = sweep(x, 1, expxb/(1+expxb)^2, "*")
    return(-t(x) %*% x1)
}
```

Now we are ready to use Newton–Raphson (1 point) to obtain the optimal β (1 point) in logistic regression. We first set the initial value as `rep(0, ncol(x))` and replicate (1 point) the summary of `glm` fit for logistic regression.

```r
    # Newton-Ralphson
    my.logistic <- function(b, x, y, tol = 1e-10, maxitr = 30, gr, hess, verbose = FALSE)
    {
        b_new = b

        for (j in 1:maxitr) # turns out you don't really need many iterations
        {
            b_old = b_new
            b_new = b_old - solve(hess(b_old, x, y)) %*% gr(b_old, x, y)

            if (verbose)
            {
                cat(paste("at iteration ", j,", current beta is \n", sep = ""))
                cat(round(b_new, 3))
                cat("\n")
            }
            if (sum(abs(b_old - b_new)) < 1e-10) break;
        }
        return(b_new)
    }

    # prepare the data matrix, I am adding a column of 1 for intercept
    x = as.matrix(cbind("intercept" = 1, heart[, 1:9]))
    y = as.matrix(heart[,10])

    # set initial value and optimize beta using Newton-Ralphson
    b = rep(0, ncol(x))

    mybeta = my.logistic(b, x, y, tol = 1e-10, maxitr = 20,
                         gr = my.gradient, hess = my.hessian, verbose = TRUE)
```

```
## at iteration 1, current beta is
## -4.032 0.005 0.066 0.133 0.009 0.694 0.024 -0.045 -0.001 0.027
## at iteration 2, current beta is
## -5.684 0.006 0.077 0.167 0.017 0.884 0.037 -0.061 0 0.041
## at iteration 3, current beta is
## -6.127 0.007 0.079 0.174 0.019 0.924 0.039 -0.063 0 0.045
## at iteration 4, current beta is
## -6.151 0.007 0.079 0.174 0.019 0.925 0.04 -0.063 0 0.045
## at iteration 5, current beta is
## -6.151 0.007 0.079 0.174 0.019 0.925 0.04 -0.063 0 0.045
## at iteration 6, current beta is
## -6.151 0.007 0.079 0.174 0.019 0.925 0.04 -0.063 0 0.045
## at iteration 7, current beta is
## -6.151 0.007 0.079 0.174 0.019 0.925 0.04 -0.063 0 0.045
```

```r
    # get the standard error estimation
    mysd = sqrt(diag(solve(-my.hessian(mybeta, x, y))))

    # my summary matrix
    round(data.frame("beta" = mybeta, "sd" = mysd, "z" = mybeta/mysd,
          "pvalue" = 2*(1-pnorm(abs(mybeta/mysd)))), dig=5)
```

```
##               beta      sd        z  pvalue
## intercept -6.15072 1.30826 -4.70145 0.00000
```

```
## sbp         0.00650 0.00573  1.13500 0.25637
## tobacco     0.07938 0.02660  2.98376 0.00285
## ldl         0.17392 0.05966  2.91517 0.00355
## adiposity   0.01859 0.02929  0.63458 0.52570
## famhist     0.92537 0.22789  4.06053 0.00005
## typea       0.03960 0.01232  3.21382 0.00131
## obesity    -0.06291 0.04425 -1.42176 0.15509
## alcohol     0.00012 0.00448  0.02714 0.97835
## age         0.04523 0.01213  3.72846 0.00019
```

```
    # check that with the glm fitting
    # round(summary(heart.full)$coef, dig=5)
```

Now we change the initial value to `rep(1, ncol(x))`. Then the iteration breaks down due to overflow. This issue could possibly be solved by introducing learning rate to adapt the step size of Newton Ralphson. (1 point for comments.)

```
    # change the initial value and optimize beta using Newton-Ralphson
    b = rep(1, ncol(x))

    tryCatch(
    {mybeta = my.logistic(b, x, y, tol = 1e-10, maxitr = 20,
                          gr = my.gradient, hess = my.hessian, verbose = TRUE)
    # get the standard error estimation
    mysd = sqrt(diag(solve(-my.hessian(mybeta, x, y))))

    # my summary matrix
    round(data.frame("beta" = mybeta, "sd" = mysd, "z" = mybeta/mysd,
          "pvalue" = 2*(1-pnorm(abs(mybeta/mysd)))), dig=5)
    # check that with the glm fitting
    # round(summary(heart.full)$coef, dig=5)
    }, error=function(e)e)
```

```
## at iteration 1, current beta is
## -1.903282e+99 1.589904e+97 -7.130201e+97 1.276984e+98 3.435697e+96 -2.646404e+99 -9.937334e+96 1.088
## at iteration 2, current beta is
## NaN NaN NaN NaN NaN NaN NaN NaN NaN NaN
```

```
## <simpleError in if (sum(abs(b_old - b_new)) < 1e-10) break: missing value where TRUE/FALSE needed>
```

Question 2 (LDA and QDA)

Load the handwritten digit data with the following code to generate training and testing data

```
    # a plot of some samples
    findRows <- function(zip, n) {
        # Find n (random) rows with zip representing 0,1,2,...,9
        res <- vector(length=10, mode="list")
        names(res) <- 0:9
        ind <- zip[,1]
        for (j in 0:9) {
        res[[j+1]] <- sample( which(ind==j), n ) }
        return(res)
    }

    set.seed(1)

    # find 100 samples for each digit for both the training and testing data
```

```
    train.id <- findRows(zip.train, 100)
    train.id = unlist(train.id)

    test.id <- findRows(zip.test, 100)
    test.id = unlist(test.id)

    X = zip.train[train.id, -1]
    Y = zip.train[train.id, 1]
    dim(X)
```

```
## [1] 1000  256
```

```
    Xtest = zip.test[test.id, -1]
    Ytest = zip.test[test.id, 1]
    dim(Xtest)
```

```
## [1] 1000  256
```

a) [5 points]

Now we write our own implemenation of LDA. (2 points.)

```
# my LDA
myLDA = function(X,Y){
  n=dim(X)[1]; p=dim(X)[2]
  cls=unique(Y)
  K=length(cls)
  pihat=rep(NA,K); muhat=matrix(NA,p,K); Sigmahat=array(NA,dim=c(p,p,K))
  for(k in 1:K){
    n_k=sum(Y==cls[k])
    pihat[k]=n_k/n
    muhat[,k]=colMeans(X[Y==cls[k],])
    Sigmahat[,,k]=(n_k-1)*cov(X[Y==cls[k],])
  }
  Sigmahat=rowSums(Sigmahat,dims=2)/(n-K) # pooled
  w=solve(Sigmahat,muhat) # (p,K)
  b=t(log(pihat)-colSums(muhat*w))/2 # (1,K)
  return(list(delta=function(x)sweep(x%*%w,2,b,'+'),class=cls)) # return the discriminant function
}
```

Now we use this manually defined LDA function to train the model and predict on the testing data (2 points). Then we calculate the confusion matrix and the prediction accuracy (1 point). We get the same results as output by the `R` function `lda`.

```
# train the model using myLDA
mylda=myLDA(X,Y)
# predict
pred_idx=apply(mylda$delta(Xtest),1,which.max)
Ypred=mylda$class[pred_idx]
# confusion matrix
table(Ypred,Ytest)
```

```
##       Ytest
## Ypred  0  1  2  3  4  5  6  7  8  9
##     0 93  0  1  2  4  4  2  0  3  0
##     1  1 93  2  0  2  0  1  1  0  0
##     2  0  0 77  3  4  2  1  0  0  0
```

```
##      3  0  0  3 77  0 13  1  1  8  0
##      4  0  2  3  2 78  4  2  6  6  4
##      5  0  0  1  8  0 73  3  1  3  0
##      6  6  0  3  0  3  2 89  0  0  0
##      7  0  0  2  1  1  1  0 84  1  5
##      8  0  1  8  5  0  0  1  0 75  2
##      9  0  4  0  2  8  1  0  7  4 89
```

```
# accuracy
(acc=mean(Ypred==Ytest))
```

```
## [1] 0.828
```

```
# check with built-in LDA
library(MASS)
lda.fit=lda(X,Y)
Ypred1=predict(lda.fit,Xtest)$class
table(Ypred1,Ytest)
```

```
##       Ytest
## Ypred1  0  1  2  3  4  5  6  7  8  9
##      0 93  0  1  2  4  4  2  0  3  0
##      1  1 93  2  0  2  0  1  1  0  0
##      2  0  0 77  3  4  2  1  0  0  0
##      3  0  0  3 77  0 13  1  1  8  0
##      4  0  2  3  2 78  4  2  6  6  4
##      5  0  0  1  8  0 73  3  1  3  0
##      6  6  0  3  0  3  2 89  0  0  0
##      7  0  0  2  1  1  1  0 84  1  5
##      8  0  1  8  5  0  0  1  0 75  2
##      9  0  4  0  2  8  1  0  7  4 89
```

```
mean(Ypred1==Ytest)
```

```
## [1] 0.828
```

b) [extra 3 points]

Now we apply regularized discrinimant analysis by calling `rda` function in `klaR` package. We can obtain better prediction accuracy commpared to LDA. (1 point: any 'rda' package works.)

```
# RDA fit: takes some time
require(klaR)
rda.fit=rda(X,Y)
# test
Ypred2=predict(rda.fit,Xtest)$class
(acc2=mean(Ypred2==Ytest))
```

```
## [1] 0.905
```

Alternatively, we can do dimension reduction using PCA and then run QDA on the resulting data. It improves LDA but not better than the previous RDA. (2 points: 1 for dimension reduction; 1 for QDA.)

```
# dimension reduction
digit_pc=prcomp(X)
# choose the dimension based on the stdandard deviations of pincipal components
r=sum(digit_pc$sdev>=digit_pc$sdev[1]*.1)
# run QDA on the reduced covariates
```

```
qda.fit=qda(digit_pc$x[,1:r],Y)
# predict on the transformed test data
Xtest_dr=sweep(Xtest,2,digit_pc$center)%*%digit_pc$rotation[,1:r]
Ypred3=predict(qda.fit,Xtest_dr)$class
(acc3=mean(Ypred3==Ytest))
```

```
## [1] 0.846
```

Question 3 [extra 3 points]

Again we can write our own naive Bayes implementation for categorical variables. (2 points.)

```
# my naive Bayes
myNB=function(X,Y){
  n=dim(X)[1]; p=dim(X)[2]
  Y=as.factor(Y)
  cls=levels(Y)
  K=length(cls)
  priprob=rep(NA,K)
  condprob=list(p); lvl=list(p)
  for(k in 1:K){
    n_k=sum(Y==cls[k])
    priprob[k]=n_k/n
    for(j in 1:p){
      x_j=as.factor(X[,j])
      lvl[[j]]=levels(x_j)
      L_j=length(lvl[[j]])
      if(k==1){condprob[[j]]=matrix(NA,L_j,K)}
      for(l in 1:L_j){
        condprob[[j]][l,k]=mean(X[Y==cls[k],j]==lvl[[j]][l])
      }
    }
  }
  return(list(priprob=priprob,condprob=condprob,class=cls,level=lvl))
}
# precition function for my naive Bayes
myNB_pred=function(nb.fit,Xtest){
  K=length(nb.fit$class); p=length(nb.fit$condprob)
  nte=dim(Xtest)[1]; Ypred=rep(NA,nte)
  postprob=matrix(NA,nte,K)
  for(i in 1:nte){
    for(k in 1:K){
      postprob[i,k]=nb.fit$priprob[k]
      for(j in 1:p){
        lvl_j=which(nb.fit$level[[j]]%in%Xtest[i,j])
        postprob[i,k]=postprob[i,k]*nb.fit$condprob[[j]][lvl_j,k]
      }
    }
    Ypred[i]=nb.fit$class[which.max(postprob[i,])]
  }
  return(list(class=Ypred,prob=postprob))
}
```

Now we use the data in the example to train naive Bayes and test the model on the instance `today`. (1 point.)

```r
# golf play data
golf=data.frame(OUTLOOK=c('Rainy','Rainy','Overcast','Sunny','Sunny','Sunny','Overcast','Rainy','Rainy'
                TEMPERATURE=c('Hot','Hot','Hot','Mild','Cool','Cool','Cool','Mild','Cool','Mild','Mild'
                HUMIDITY=c('High','High','High','High','Normal','Normal','Normal','High','Normal','Norma
                WINDY=c('False','True','False','False','False','True','True','False','False','False','T
                PLAYGOLF=c('No','No','Yes','Yes','Yes','No','Yes','No','Yes','Yes','Yes','Yes','Yes','N
# today
today=c('Sunny','Hot','Normal','False')
# train naive Bayes
X=golf[,-5]; Y=golf[,5]
mynb=myNB(X,Y)
# predict
(Ypred=myNB_pred(mynb,t(today))$class)
```

```
## [1] "Yes"
```

```r
# posterior probility (calculation in the lecture is wrong!)
myNB_pred(mynb,t(today))$prob
```

```
##              [,1]       [,2]
## [1,] 0.004571429 0.02116402
```