Homework 3

Thibault Doutre, ID : 26980469

STAT 241 : Statistical Learning Theory

1 Preliminaries

First, store the data into two data.frames.

```
# Load data
setwd('/Users/doutre/Desktop/Homework3')
data.test=read.table('./hw3-2-test.data')
names(data.test)=c('v1','v2','v3','y')
data.train=read.table('./hw3-2-train.data')
names(data.train)=c('v1','v2','v3','y')
sum(data.train$y==1)

## [1] 27

# Set global variables
k=max(data.train$y)
d=3
n=dim(data.train)[1]

# Preview
head(data.test)

##            v1         v2 v3 y
## 1 -1.1474741 1.1053061  1 1
## 2 -0.8809272 1.1149605  1 1
## 3 -0.9865532 0.8745638  1 1
## 4 -0.9316571 1.2960280  1 1
## 5 -0.5685454 1.1019466  1 1
## 6 -1.0507876 1.2569895  1 1
```

2 Logistic Regression

As described in the algorithm, I create a function to initialize N the matrix of nijs.

```r
# Init N matrix
init_N=function(){
  N=matrix(0,ncol=k,nrow=n)
  for (i in 1:n){
    N[i,data.train$y[i]]=1
  }
  N
}
# Example
head(init_N())
```

```
##       [,1] [,2] [,3] [,4]
## [1,]    1    0    0    0
## [2,]    1    0    0    0
## [3,]    1    0    0    0
## [4,]    1    0    0    0
## [5,]    1    0    0    0
## [6,]    1    0    0    0
```

Then I create the Pi functions which takes for argument the data, i, j and beta and returns the value of the softmaw function.

```r
# Pi
Pi = function(i,j,beta,x){
  xi=x[i,]
  num=exp(-t(beta[(1+(j-1)*d):(j*d),])%*%xi)
  den=1+sum(sapply(1:(k-1),function(i) exp(-t(beta[(1+(i-1)*d):(i*d),])%*%xi)))
  return(num/den)
}
# Example
beta0=matrix(1,nrow=d*(k-1),ncol=1)
Pi(1,1,beta0,as.matrix(data.train[,1:3]))
```

```
##             [,1]
## [1,] 0.1220311
```

In order to compute the Newton-Raphson algorithm, we need the gradient vector and the hessian matrix. So I create two functions which compute both and then I compute the Newton-Raphson algorithm. The algorithm takes as inputs:

- beta0: initial vector of beta
- eps: parameter of the stopping criteria
- rho: step
- data: dataset

And as output, a list of three elements:

- beta: final vector
- diff: the difference between beta before the last iteration and beta, times rho
- iteration: the number of iteration of the algorithm

```
gradient = function(beta,data){
  res=matrix(0,ncol=1,nrow=d*(k-1))
  x=as.matrix(data[,1:3])
  for (j in 1:(k-1)){
    res_i=matrix(0,ncol=1,nrow=d)
    for (i in 1:n){
      ni=sum(N[i,])
      nij=N[i,j]
      pi_ij=Pi(i,j,beta,x)
      res_i=res_i+x[i,]*(ni*pi_ij-nij)
    }
    res[(1+(j-1)*d):(j*d),]=res_i
  }
  return(res)
}

hessien = function(beta,data){
  x=as.matrix(data[,1:3])
  res=matrix(0,ncol=d*(k-1),nrow=d*(k-1))
  for (i in 1:n){
    ni=sum(N[i,])
    Pi=sapply(1:(k-1),function(j) {Pi(i,j,beta,x)})
    Pi_matrix=as.matrix(Pi,ncol=1)
    Di=ni*(Pi_matrix%*%t(Pi_matrix)-diag(Pi))
    Xi=matrix(0,ncol=k-1,nrow=d*(k-1))
    xi=x[i,]
    for (j in 1:(k-1) ){
      Xi[(1+(j-1)*d):(j*d),j]=xi
    }
    res=res+Xi%*%Di%*%t(Xi)
  }
  return(res)
}

newton_raphson = function(beta0,eps,rho,data){
  beta=beta0
  s=matrix(eps+1,1)
  i=1
  print('Reaching optimum...')
```

```
  while(norm(s,"1")>eps){
    H=hessien(beta,data.train)
    G=gradient(beta,data.train)
    s=solve(H,G)
    beta=beta-rho*s
    i=i+1
  }
  list(beta=beta,diff=s,iteration=i)
}
```

Now, I compute the algorithm with an initial beta of zeros, a stopping criteria of 0.1 and a step of 0.1.

```
N=init_N()
beta0=matrix(1,nrow=d*(k-1),ncol=1)
nr=newton_raphson(beta0,0.1,0.1)

## [1] "Reaching optimum..."

beta=nr$beta
# Number of iterations
nr$iteration

## [1] 71

# Beta
beta

##              [,1]
##  [1,]  4.2403541
##  [2,] -3.0034901
##  [3,]  3.5941898
##  [4,] -2.6413339
##  [5,]  3.1868102
##  [6,]  2.8161760
##  [7,]  0.6116957
##  [8,]  0.3377679
##  [9,]  0.3046426
```

Once beta computed, we need to assess the accuracy of our results. I create a first function "probs", which displays the probabilities given the data and beta. And then, I create a second function "predict" which assigns the more likely class for each observation.

```r
probs = function(beta,data){
  x=as.matrix(data[,1:3])
  p=as.matrix(
    sapply(1:n,function(i)
      sapply(1:(k-1),function(j) Pi(i,j,beta,x))
      )
    )
  s=colSums(p)
  return(rbind(p,1-s))
}

predict=function(beta,data){
  x0=apply(probs(beta,data),2,which.max)
  x=x0
  y=unique(x)
  for (i in 1:4){
    x[x0==y[i]]=i
  }
  return(x)
}
# Example
predict(beta,data.train)
```

```
##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
    1 2 2
##  [28] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3
    3 3 3
##  [55] 3 3 3 3 3 3 3 3 1 3 3 3 3 3 3 3 3 3 3 3 4 4 4
    4 4 4
##  [82] 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 2 3 3 1 3
    3 3 3
## [109] 3 2
```

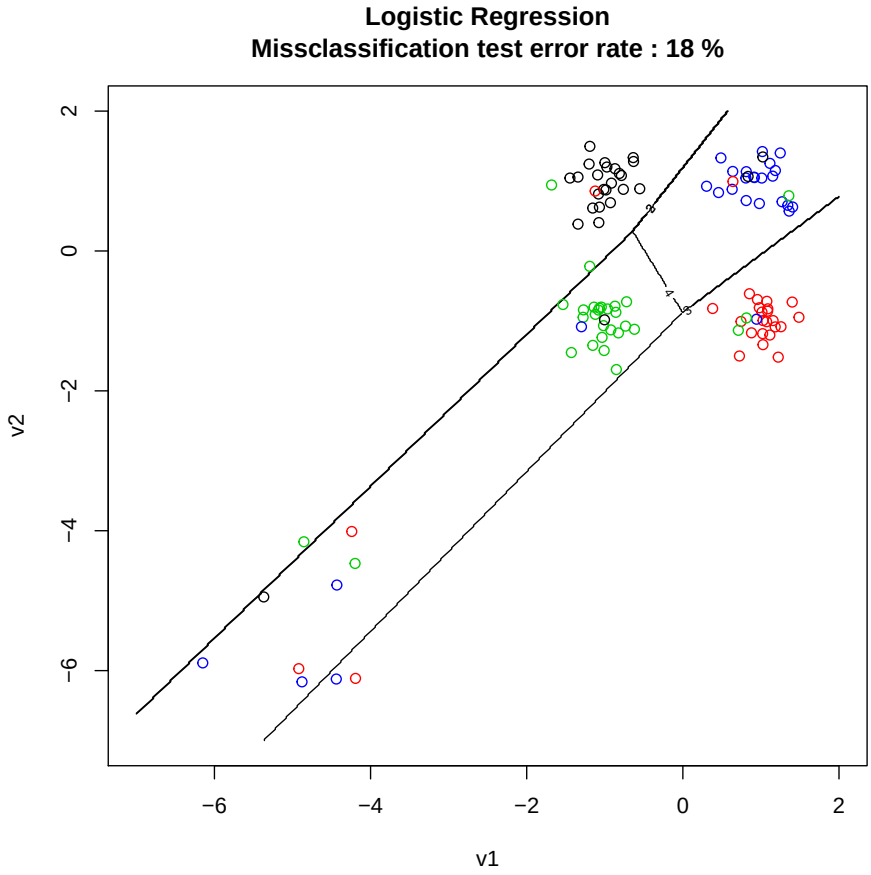
The key thing here is to order the result of predict in a good way. Indeed, labels must correspond to each other in order to compute the missclassification error rate. With this setting the error rate is easy to compute for both the training set and the test set.

```r
error_rate=function(beta,data){
  m.predict=predict(beta,data)
  sum(m.predict!=data$y)/n
}
# Training Error
error_rate(beta,data.train)
```

```
## [1] 0.08181818

# Test Error
error_rate(beta,data.test)

## [1] 0.1818182
```

Finally, I can plot the result with boundaries associated with the beta obtained by using the Newto-Raphson algorithm. The labels correspond to the true labels for the test set and the boundaries are the ones I found using the training set.

```
# Plotting function
f1=function(x,y) {
  #x=as.matrix(data[,1:3])
  m=matrix(cbind(x,y,rep(1,length(x))),ncol=d)
  p=as.matrix(
    sapply(1:length(x),function(i)
      sapply(1:(k-1),function(j) Pi(i,j,beta,m))
    )
  )
  s=colSums(p)
  m=rbind(p,1-s)
  apply(m,2,which.max)
}
# Plot
x=seq(-7,2,0.02)
contour(x,x,outer(x,x,f1),levels=c(1,2,3,4))
points(data.train[,1:2],col=data.test$y)
title(paste('Logistic Regression\nMissclassification test error rate :',
            round(error_rate(beta,data.test)*100),"%"),xlab ='v1',
      ylab = 'v2')
```



3 Linear Regression

Now, I do the same thing with linear regression, i.e. using the training set, I compute beta and then I use a predict function to predict the results of the test set. Then I plot the results.

```
linear_estimate=function(data){
  x=as.matrix(data[,1:3])
  names(x)=NULL
  y0=as.matrix(data$y)
  y=matrix(0,nrow=n,ncol=k)
  for (i in 1:n){
    y[i,y0[i,]]=1
  }
  X=matrix(0,d,d)
```

```
  for (i in 1:n){
    X=X+x[i,]%*%t(x[i,])
  }
  XY=matrix(0,d,k)
  for (i in 1:n){
    xi=as.matrix(x[i,])
    yi=as.matrix(y[i,])
    XY=XY+xi%*%t(yi)
  }
  Beta=2*solve(X,XY)
  row.names(Beta)=NULL
  return(Beta)
}


f2=function(x,y) {
  m=matrix(cbind(x,y,rep(1,length(x))),ncol=d)%*%beta
  res=matrix(0,ncol=1,nrow=length(x))
  for (i in 1:length(x)){
    res[i,]=which.max(m[i,])
  }
  res
}
error_rate2=function(beta,data){
  1-mean(f2(data.train[,1],data.train[,2])==data$y)
}
```

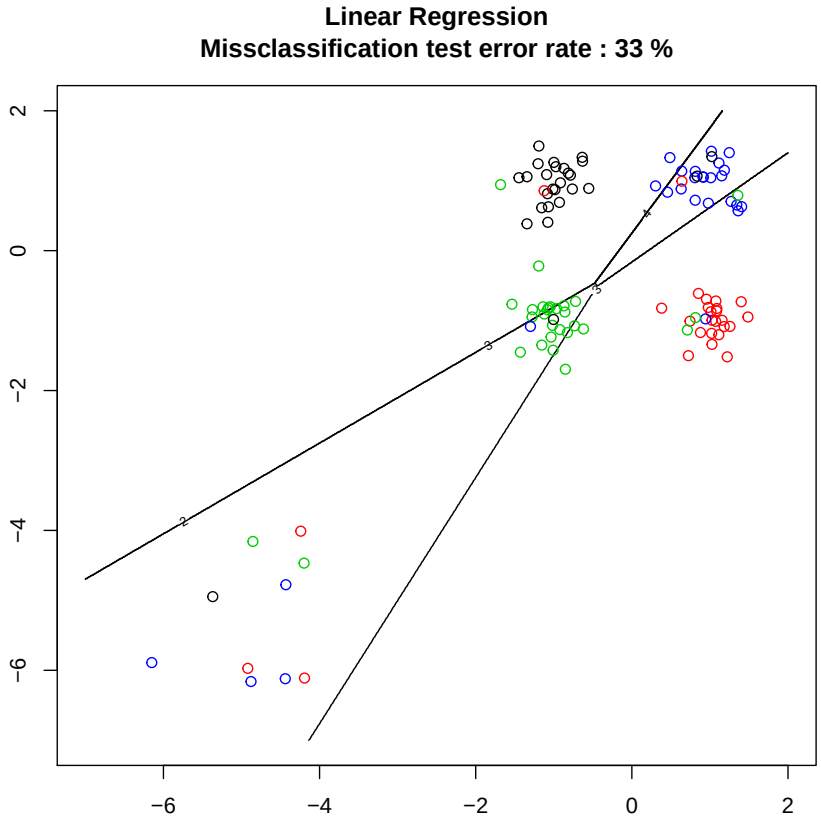
Here are the results.

```
# Estimate
beta=linear_estimate(data.train)
# Training Error
error_rate2(beta,data.train)

## [1] 0.2272727

# Test Error
error_rate2(beta,data.test)

## [1] 0.3272727

# Confusion matrix
table(f2(data.train[,1],data.train[,2]),data.test$y)
```

```
##
##        1   2   3   4
##    1  23   1   9   2
##    2   0  22   6   5
##    3   2   3  14   5
##    4   2   1   0  15
```

And the plot:

```
x=seq(-7,2,0.01)
contour(x,x,outer(x,x,f2),levels=c(1,2,3,4))
points(data.train[,1:2],col=data.test$y)
title(paste('Linear Regression\nMissclassification test error rate :',
            round(error_rate2(beta,data.test)*100),"%"))
```



4 Comparison of the two algorithms

We can see here that the test error rate for the Logistic regression is significantly better than for the Linear Regression. When seeing the boundaries of the predicted classes we can acknoledge the fact that the boundaries of the Logistic Regression are more flexible and fit the data well.
Actually the linear regression method would have well fitted the data without the outliers (in the bottom left corner). As we have seen in class, logistic regression performs particulary well with outliers as compared to linear regression, which is not very suited for classification.