Advances Computational Econometrics. Chapter 7: Neural networks

Thierry Denoeux

2023-05-28

Exercise 1

We start by loading package `nnet` and preprocessing the data (note that we scale the predictors):

```
library(nnet)
library(MASS)
n<-nrow(Boston)
Boston[,-14]<-scale(Boston[,-14])
set.seed(2022)
ntrain<-round(2*n/3)
ntest<-n-ntrain
train<-sample(n,ntrain)
Boston.train<-Boston[train,]
Boston.test<-Boston[-train,]
y.test<-Boston.test$medv
```

We write a function that runs function `nnet` several times with different random initial conditions:

```
nnet_rep<-function(formula,data,linout,size,decay=0,trace=FALSE,maxit=100,ntrials){
  best.error<-Inf
  for(i in 1:ntrials){
    fit<-nnet(formula=formula,data=data,linout=linout,size=size,decay=0.1,maxit=maxit,
              trace=trace)
    if(fit$value<best.error){
      best.error<-fit$value
      best.fit<-fit
    }
  }
  return(best.fit)
}
```

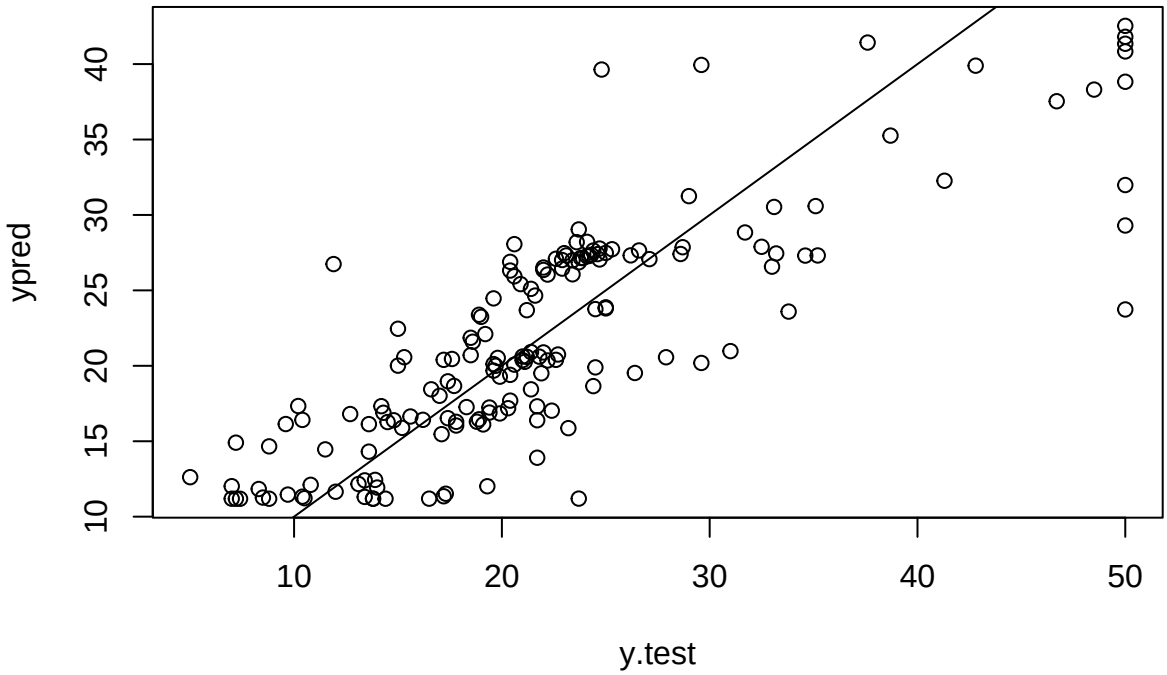
Part I: Using predictor `lstat` only

Let us fit a neural network with 5 hidden units and no weight decay, et compute the test MSE

```
fit<-nnet_rep(medv ~ lstat,data=Boston.train,linout=TRUE,size=5,decay=0,trace=FALSE,
              maxit=1000,ntrials=5)
ypred<-predict(fit,newdata=Boston.test)
print(sqrt(mean((y.test-ypred)^2)))
```

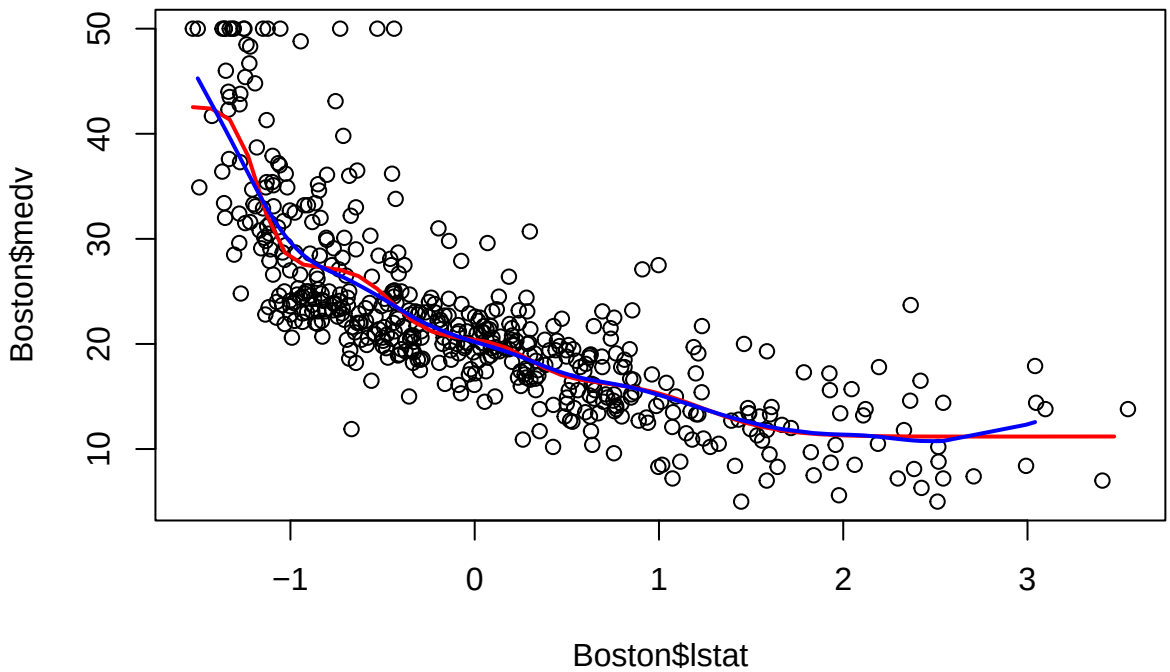
```
## [1] 5.579597
```

```
plot(y.test,ypred)
abline(0,1)
```



We can compare the result with that of smoothing splines and plot the two prediction functions:

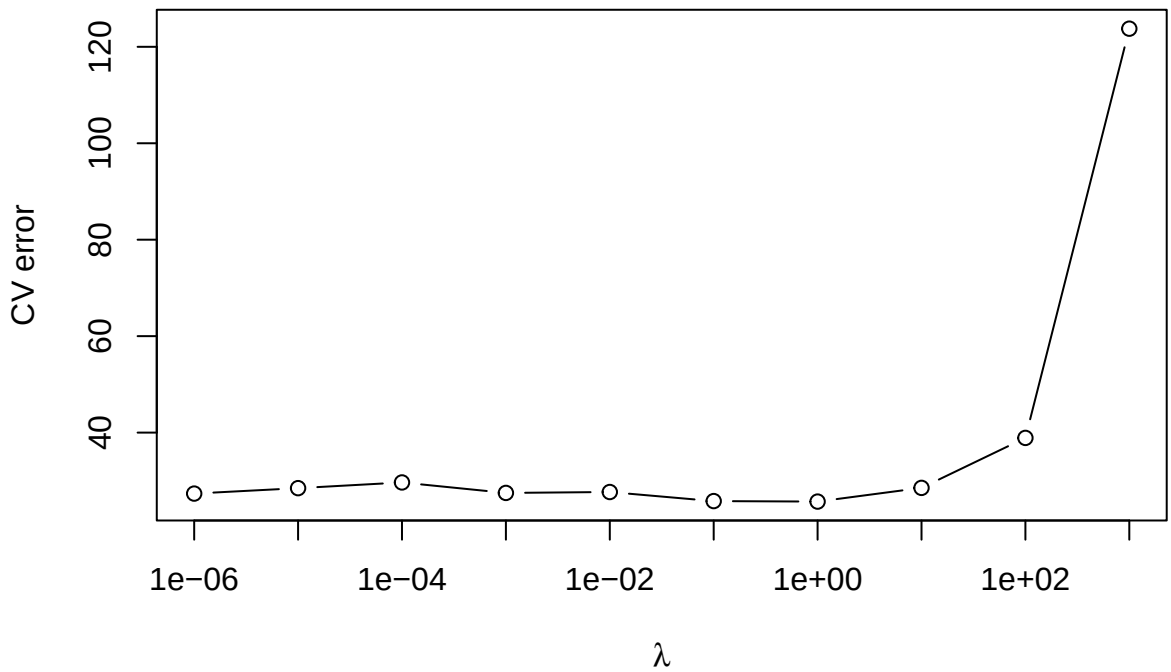
```
library(splines)
fit.ss<-smooth.spline(Boston.train$lstat,Boston.train$medv,cv=TRUE)
xx<-seq(min(Boston$lstat),max(Boston$lstat),0.1)
yy<-predict(fit,newdata=data.frame(lstat=xx))
pred.ss<-predict(fit.ss,newdata=data.frame(lstat=xx))
plot(Boston$lstat,Boston$medv)
lines(xx,yy,col='red',lwd=2)
lines(pred.ss$x,pred.ss$y,col='blue',lwd=2)
```



They are very close.

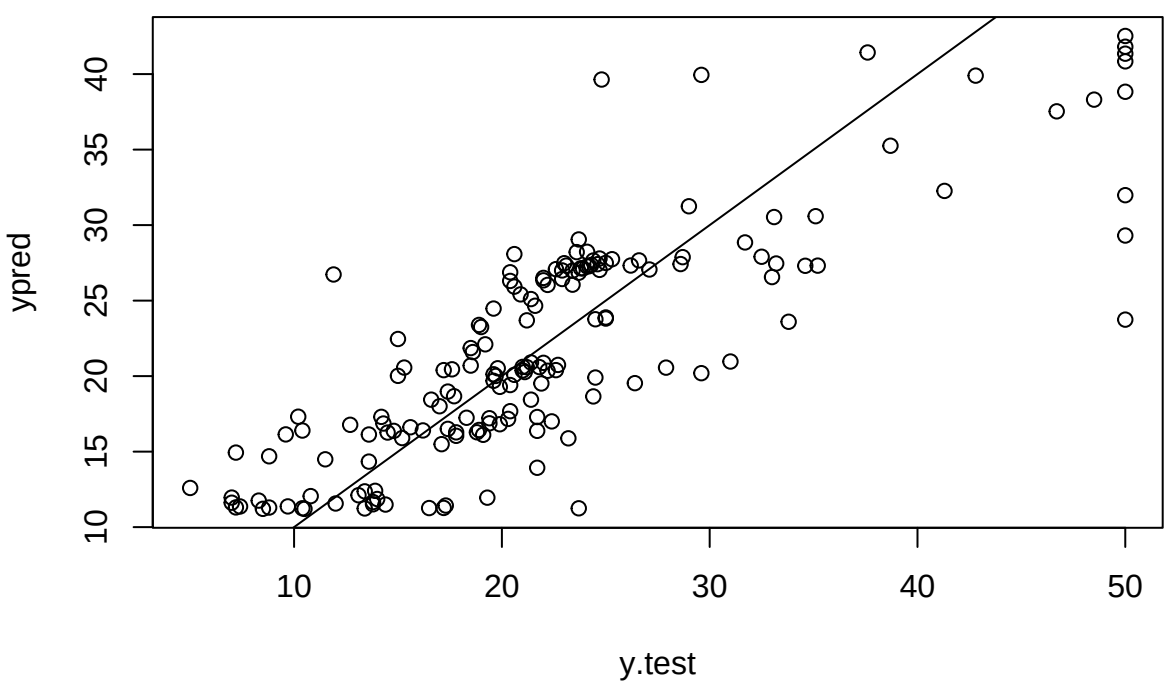
Let us now tune the weight decay parameter using 10-fold cross-validation. We will use an architecture with 10 hidden units:

```
K<-10
folds=sample(1:K,ntrain,replace=TRUE)
Lambda<-c(1e-6,1e-5,1e-4,0.001,0.01,0.1,1,10,100,1000)
N<-length(Lambda)
CV1<-rep(0,N)
for(i in (1:N)){
  for(k in (1:K)){
    fit<-nnet(medv~lstat,data=Boston.train[folds!=k,],size=10,decay=Lambda[i],lin=TRUE,
              trace=FALSE)
    pred<-predict(fit,newdata=Boston.train[folds==k,])
    CV1[i]<-CV1[i]+ sum((Boston.train$medv[folds==k]-pred)^2)
  }
  CV1[i]<-CV1[i]/ntrain
}
plot(Lambda,CV1,type='b',xlab=expression(lambda),ylab='CV error',log="x")
```



We now retrain the network with $\lambda = 1$:

```
best.fit<-nnet_rep(medv ~ lstat,data=Boston.train,linout=TRUE,size=10,decay=1,trace=FALSE,
                   maxit=1000,ntrials=5)
ypred<-predict(best.fit,newdata=Boston.test)
plot(y.test,ypred)
abline(0,1)
```

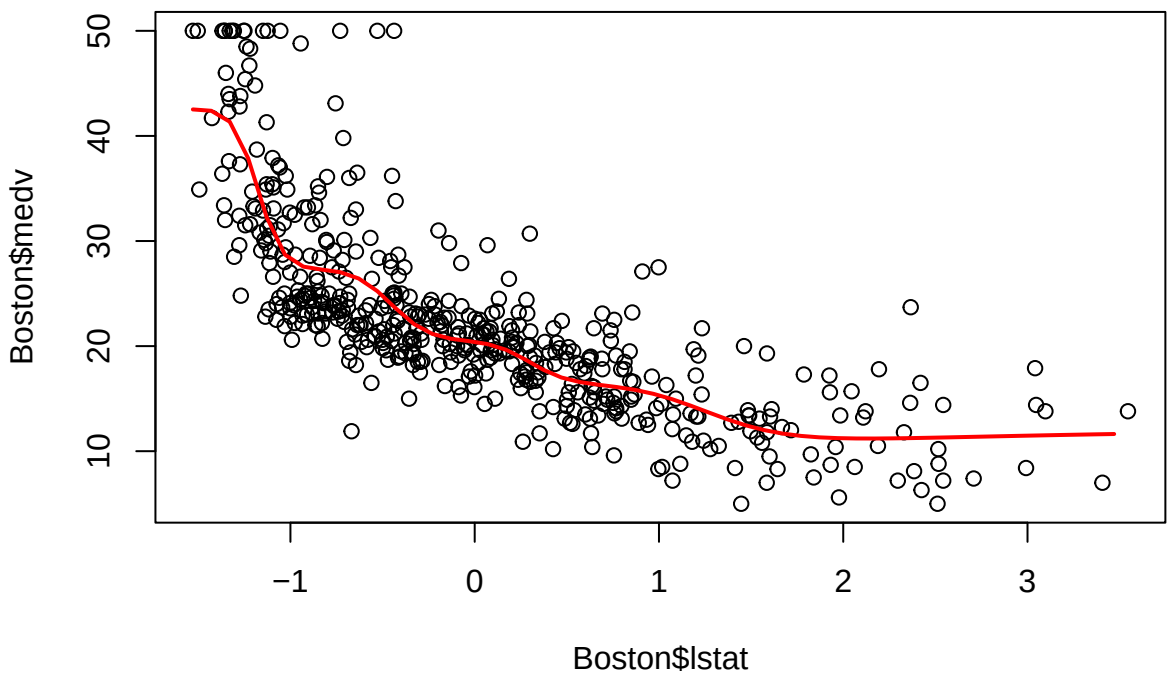



```
print(sqrt(mean((y.test-ypred)^2)))
```

```
## [1] 5.579438
```

The result is similar to that of the network with 5 hidden units and no decay.

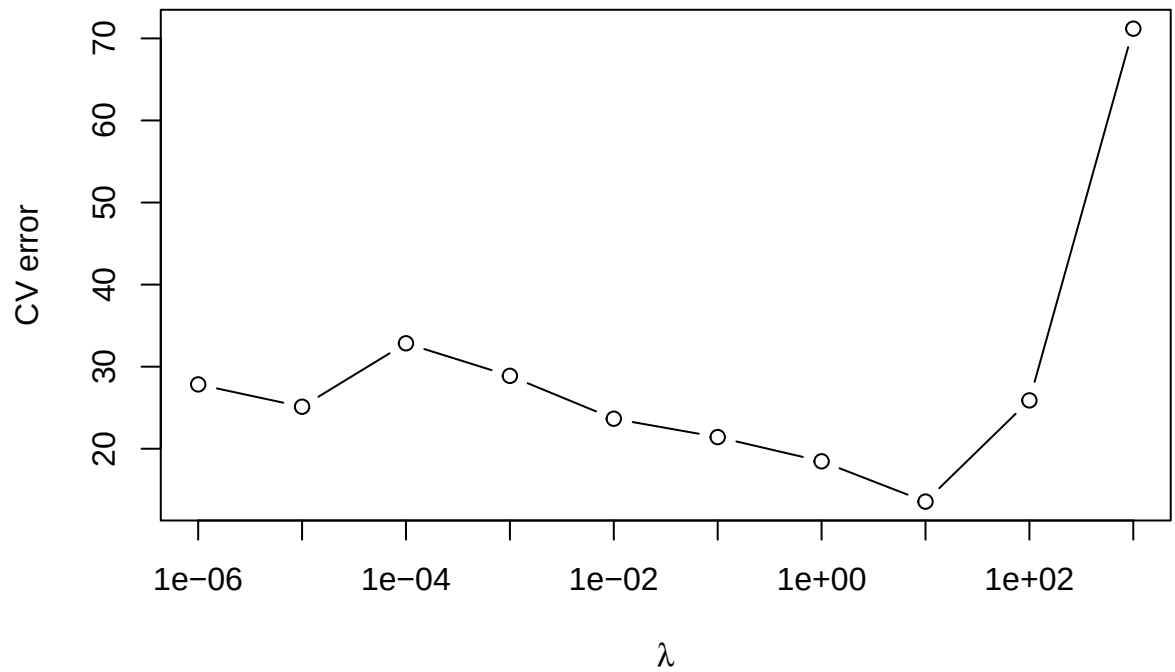
```
plot(Boston$lstat,Boston$medv)
xx<-seq(min(Boston$lstat),max(Boston$lstat),0.1)
yy<-predict(best.fit,newdata=data.frame(lstat=xx))
lines(xx,yy,col='red',lwd=2)
```



Part II: Using all predictors

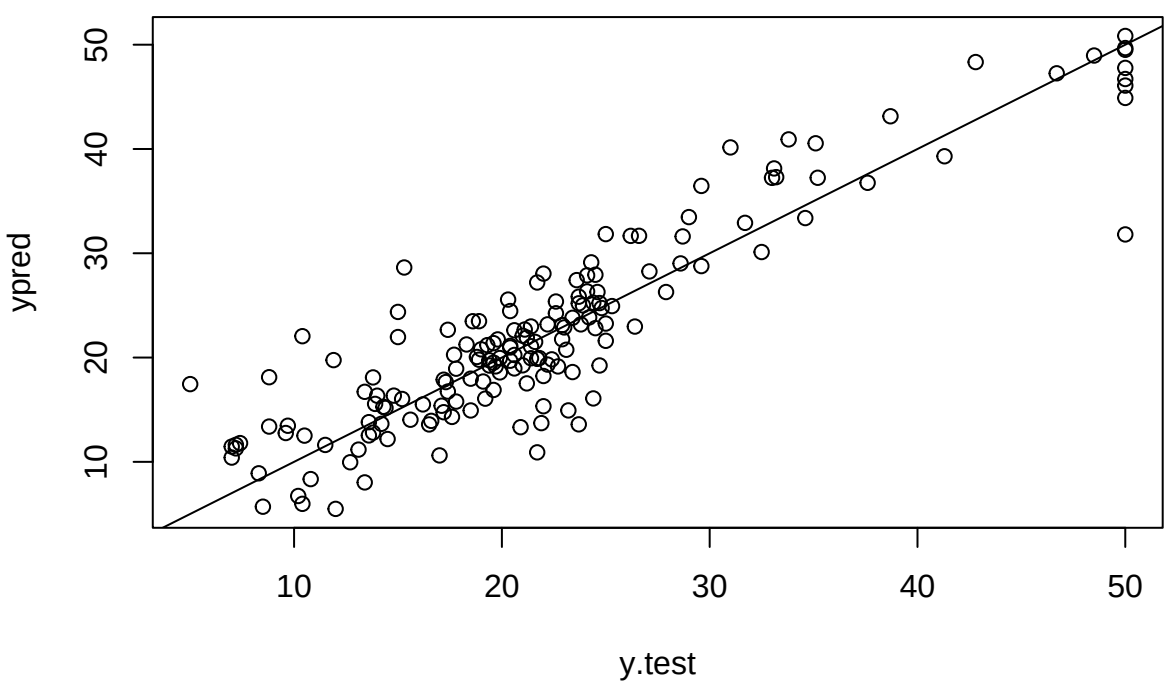
Let us now use the 13 predictors. As before, we consider a network with 30 hidden units and tune the weight decay hyperparameter by 5-fold cross-validation:

```
K<-5
folds=sample(1:K,ntrain,replace=TRUE)
Lambda<-c(1e-6,1e-5,1e-4,0.001,0.01,0.1,1,10,100,1000)
N<-length(Lambda)
CV1<-rep(0,N)
ntrials<-5
nunits<-30
for(i in (1:N)){
  for(k in (1:K)){
    fit<-nnet(medv ~., data=Boston.train,subset=which(folds!=k),
              size=nunits,decay=Lambda[i],lin=TRUE,trace=FALSE)
    pred<-predict(fit,newdata=Boston.train[folds==k,])
    CV1[i]<-CV1[i]+ sum((Boston.train[folds==k,14]-pred)^2)
  }
  CV1[i]<-CV1[i]/ntrain
}
plot(Lambda,CV1,type='b',xlab=expression(lambda),ylab='CV error',log="x")
```



Let us fit the model on the training data with $\lambda = 10$:

```
best.fit<-nnet_rep(medv ~.,data=Boston.train,linout=TRUE,size=nunits,
                   decay=10,trace=FALSE,maxit=1000,ntrials=ntrials)
ypred<-predict(best.fit,newdata=Boston.test)
plot(y.test,ypred)
abline(0,1)
```




```
print(sqrt(mean((y.test-ypred)^2)))
```

```
## [1] 4.25315
```

Comparison with random forests:

```r
library(randomForest)
fit.rf<-randomForest(medv~.,data=Boston.train)
pred.rf<-predict(fit.rf,newdata=Boston.test)
print(sqrt(mean((y.test-pred.rf)^2)))
```

```
## [1] 3.54585
```

Comparison with SVR (RBF kernel):

```r
library(kernlab)
fit<-ksvm(medv~.,data=Boston.train,kernel="rbfdot",C=10)
pred.svr<-predict(fit,newdata=Boston.test)
print(sqrt(mean((y.test-pred.svr)^2)))
```

```
## [1] 4.086596
```

The random forest algorithm seems to work better. However, the result may depend on the training/test split. We will repeat the whole procedure with 10 different random splits. We will only compare neural networks and random forests:

```r
M<-10
K<-5
Lambda<-c(0.01,0.1,1,10,100)
N<-length(Lambda)
RMS.NN<-rep(0,M)
RMS.RF<-rep(0,M)
for(j in 1:M){
  train<-sample(n,ntrain)
  Boston.train<-Boston[train,]
  Boston.test<-Boston[-train,]
  y.test<-Boston.test$medv
  folds=sample(1:K,ntrain,replace=TRUE)
  CV1<-rep(0,N)
  ntrials<-10
  for(i in (1:N)){
    for(k in (1:K)){
      fit<-nnet(medv ~., data=Boston.train,subset=which(folds!=k),
                size=nunits,decay=Lambda[i],lin=TRUE,trace=FALSE)
      pred<-predict(fit,newdata=Boston.train[folds==k,])
      CV1[i]<-CV1[i]+ sum((Boston.train[folds==k,14]-pred)^2)
    }
    CV1[i]<-CV1[i]/ntrain
  }
  lambda<-Lambda[which.min(CV1)]
  best.fit<-nnet_rep(medv ~.,data=Boston.train,linout=TRUE,
                     size=nunits,decay=lambda,trace=FALSE,maxit=1000,ntrials=5)
  ypred<-predict(best.fit,newdata=Boston.test)
  RMS.NN[j]<-sqrt(mean((y.test-ypred)^2))
  fit.rf<-randomForest(medv~.,data=Boston.train)
  pred.rf<-predict(fit.rf,newdata=Boston.test)
  RMS.RF[j]<-sqrt(mean((y.test-pred.rf)^2))
}
print(cbind(RMS.NN,RMS.RF))
```

```
##         RMS.NN   RMS.RF
##  [1,] 3.934257 3.643151
```

```
##  [2,] 3.470692 3.192156
##  [3,] 4.008096 3.701860
##  [4,] 3.540960 2.897352
##  [5,] 3.686888 3.332863
##  [6,] 3.725156 3.245906
##  [7,] 4.032422 3.872565
##  [8,] 4.134240 3.675811
##  [9,] 4.690825 3.663249
## [10,] 3.659787 3.141067
```

Means and standard errors:

```
print(c(mean(RMS.NN),mean(RMS.RF)))
```

```
## [1] 3.888332 3.436598
```

```
print(c(sd(RMS.NN),sd(RMS.RF))/sqrt(10))
```

```
## [1] 0.11334358 0.09981595
```

We can compare the two sets of results using a paired Wilcoxon test:

```
wilcox.test(RMS.NN,RMS.RF,paired = TRUE, alternative="two.sided")
```

```
##
##  Wilcoxon signed rank exact test
##
## data:  RMS.NN and RMS.RF
## V = 55, p-value = 0.001953
## alternative hypothesis: true location shift is not equal to 0
```

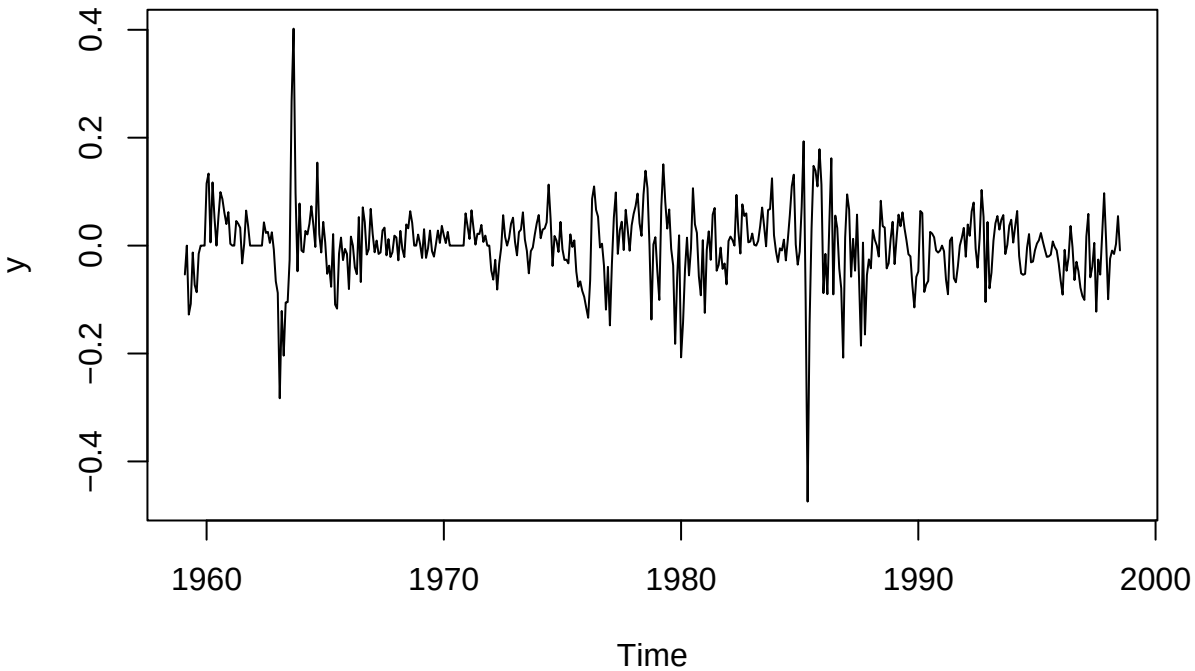
They are significantly different at the 5% level.

Exercise 2

Question 1

We read the data and apply the log-difference transformation:

```
data<-read.csv("/Users/Thierry/Documents/R/Data/Economics/CP.csv",header=TRUE)
x<-ts(data$Cp,start = c(1959,1),frequency=12)
y<-diff(log(x))
plot(y)
```



This function reformats the data in the form of a matrix of predictors and a matrix of traget values:

```
preproc<-function(y,W,H){
  n<-length(y)
  Lag<-W+H
  Z<-y[1:(n-Lag+1)]
  for(i in 2:Lag) Z<-cbind(Z,y[i:(n-Lag+i)])
  X<-Z[,1:W]
  Y<-Z[,(W+1):(W+H)]
  return(list(X=X,Y=Y))
}
```

We use it with a window size $W = 3$ and $H = 1$ (one-step-ahead prediction):

```
W<-3
data1<-preproc(y,W,H=1)
X<-data1$X
Xs<-scale(X)
Y<-data1$Y
```

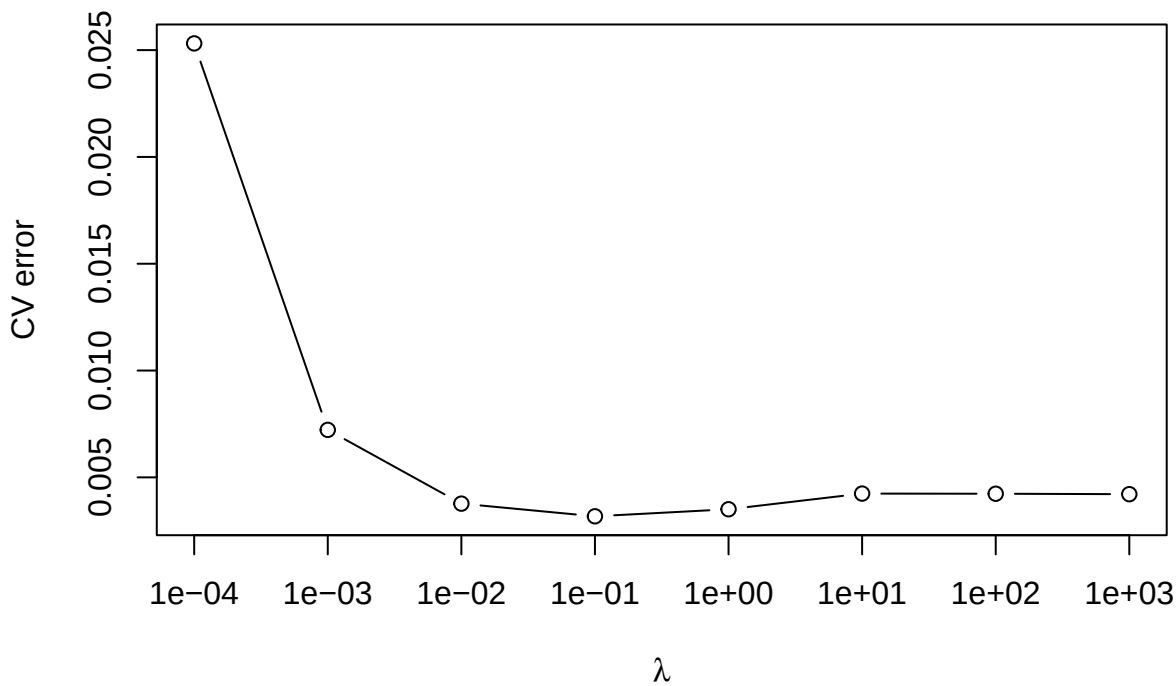
Finally, we split the data into training and test sets:

```
n<-nrow(X)
ntrain<-n-174
X.train<-X[1:ntrain,]
Xs.train<-Xs[1:ntrain,]
Y.train<-Y[1:ntrain]
X.test<-X[(ntrain+1):n,]
Xs.test<-Xs[(ntrain+1):n,]
Y.test<-Y[(ntrain+1):n]
```

Question 2

Let us consider a neural network architecture with 10 hidden units. We will tune the weight decay parameter using 5-fold cross-validation:

```
K<-5
folds<-sample(1:K,ntrain,replace=TRUE)
Lambda<-c(1e-4,0.001,0.01,0.1,1,10,100,1000)
N<-length(Lambda)
CV1<-rep(0,N)
for(i in (1:N)){
  for(k in (1:K)){
    fit<-nnet(Xs.train[folds!=k,],Y.train[folds!=k], linout = TRUE,size=10,
              decay=Lambda[i],maxit = 500,trace=FALSE)
    pred<-predict(fit,newdata=Xs.train[folds==k,])
    CV1[i]<-CV1[i]+ sum((Y.train[folds==k]-pred)^2)
  }
  CV1[i]<-CV1[i]/ntrain
}
plot(Lambda,CV1,type='b',xlab=expression(lambda),ylab='CV error',log="x")
```




```
lambda<-Lambda[which.min(CV1)]
```

We fit the network with teh best value of λ and compute the test MSE:

```
fit<-nnet(Xs.train,Y.train, linout = TRUE,size=10,decay=lambda,maxit = 1000)
```

```
## # weights:  51
## initial  value 301.743590
## iter  10 value 1.978827
## iter  20 value 1.071213
## iter  30 value 0.964489
## iter  40 value 0.936090
## iter  50 value 0.929308
```

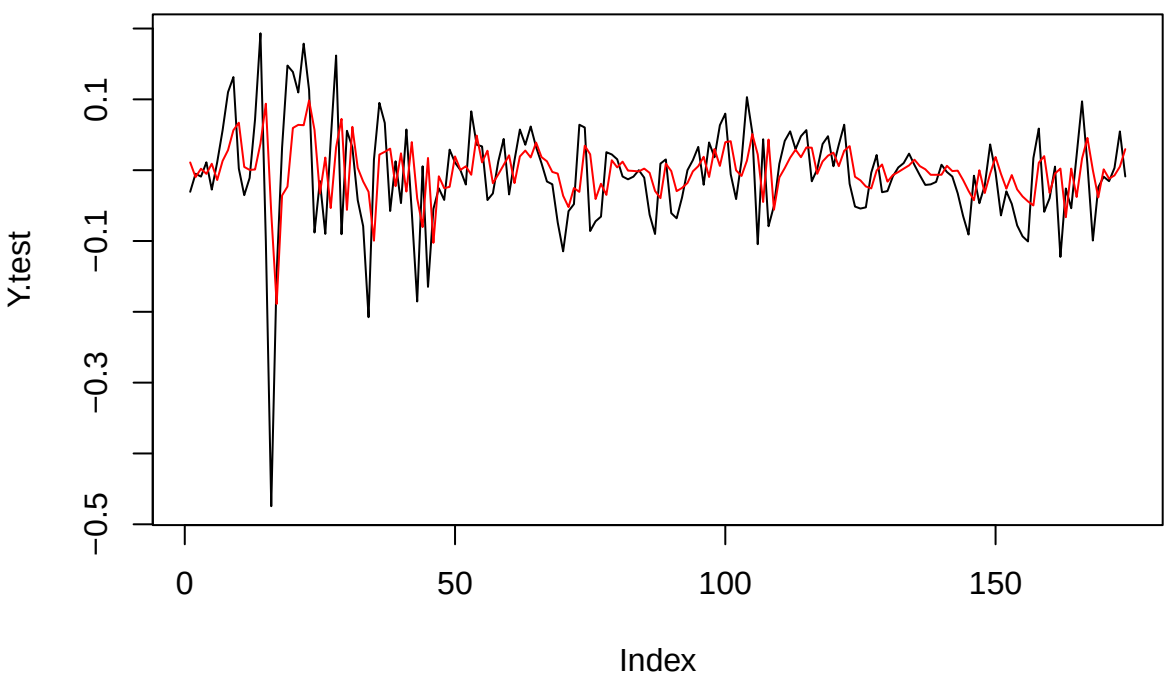
```
## iter  60 value 0.927961
## iter  70 value 0.927238
## iter  80 value 0.927030
## iter  90 value 0.926963
## iter 100 value 0.926908
## iter 110 value 0.926875
## iter 120 value 0.926547
## iter 130 value 0.926411
## iter 140 value 0.926389
## iter 150 value 0.926384
## final  value 0.926383
## converged
```

```
y.pred<-predict(fit,newdata=Xs.test)
err.NN<-mean((Y.test-y.pred)^2)
print(err.NN)
```

```
## [1] 0.004743213
```

We plot the test data with the predictions:

```
plot(Y.test,type="l")
lines(y.pred,col="red")
```



Comparison with the naive predictions:

```
pred.naive<-X.test[,W]
err.naive<-mean((Y.test-pred.naive)^2)
print(err.naive)
```

```
## [1] 0.006875965
```