STAT230 HW 11
University of California, Berkeley

Thibault Doutre, Student ID 26980469

April 22, 2016

I would like to thank Shamindra for discussing the project with me.

1

1.1 Load data

```
rm(list = ls())
cat("\014")
```

```
# Data ----------------------------------------------------

makedata=function(p=20,wh=15,n=100){
# This function generates p independent X variables and then uses wh of them
# to generate Y based on coefficients determined by exp(exps) below.
# Outside of the function, the data frame named data is further changed to
# randomize the order of the variables so it's hard to tell for sure which
# ones were used to generate Y.  You can always go back and examine switch
# to figure out if your models include the right variables and which ones.
  X=matrix(rnorm(n*p),n,p)
  exps=seq(-1,-2.5,length=wh)
  beta=rep(0,p)
  beta[1:wh]=exp(exps)
  Y=.5+X%*%beta+rnorm(n)
  data = data.frame(Y,X)
  colnames(data)=c("Y",letters[1:20])
  switch=sample(20)+1
```

```r
  data=data[,c(1,switch)]
  return(data)
}


set.seed(1)
data=makedata()
lmout=lm(Y~.,data=data)
summary(lmout)
```

```
##
## Call:
## lm(formula = Y ~ ., data = data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -2.27948 -0.62608 -0.05717  0.66909  2.24809
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  0.385093   0.122642   3.140 0.002377 **
## b            0.561787   0.123555   4.547 1.94e-05 ***
## c            0.462793   0.117937   3.924 0.000185 ***
## i            0.140704   0.121406   1.159 0.249965
## l            0.164197   0.116121   1.414 0.161285
## p            0.045910   0.116150   0.395 0.693713
## t            0.059449   0.116261   0.511 0.610539
## m           -0.077337   0.117840  -0.656 0.513546
## h           -0.041968   0.114228  -0.367 0.714297
## g            0.185011   0.111205   1.664 0.100136
## j            0.270331   0.125661   2.151 0.034511 *
## s            0.077381   0.118734   0.652 0.516475
## r            0.004575   0.109184   0.042 0.966680
## n            0.047872   0.137054   0.349 0.727796
## e            0.251664   0.099387   2.532 0.013319 *
## k            0.192793   0.113346   1.701 0.092891 .
## d            0.277622   0.127615   2.175 0.032584 *
## a            0.479756   0.131251   3.655 0.000461 ***
## q            0.004262   0.125687   0.034 0.973037
```

```
## f            0.234947   0.124049   1.894 0.061887 .
## o            0.131873   0.121603   1.084 0.281460
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.091 on 79 degrees of freedom
## Multiple R-squared:  0.5177,Adjusted R-squared:  0.3955
## F-statistic: 4.239 on 20 and 79 DF,  p-value: 1.927e-06
```

```
coef(summary(lmout))[-1,]
```

```
##       Estimate Std. Error     t value     Pr(>|t|)
## b  0.561786874 0.12355483  4.54686293 1.936786e-05
## c  0.462792924 0.11793750  3.92405250 1.847248e-04
## i  0.140704418 0.12140598  1.15895787 2.499646e-01
## l  0.164196922 0.11612062  1.41402032 1.612848e-01
## p  0.045909884 0.11614996  0.39526386 6.937135e-01
## t  0.059448890 0.11626102  0.51133985 6.105393e-01
## m -0.077336903 0.11783998 -0.65628749 5.135462e-01
## h -0.041968451 0.11422842 -0.36740813 7.142970e-01
## g  0.185010780 0.11120500  1.66369119 1.001364e-01
## j  0.270330537 0.12566141  2.15126129 3.451109e-02
## s  0.077380836 0.11873362  0.65171800 5.164747e-01
## r  0.004575346 0.10918416  0.04190485 9.666803e-01
## n  0.047872444 0.13705381  0.34929670 7.277959e-01
## e  0.251664055 0.09938718  2.53215806 1.331884e-02
## k  0.192793127 0.11334639  1.70091994 9.289138e-02
## d  0.277622444 0.12761473  2.17547342 3.258437e-02
## a  0.479755896 0.13125131  3.65524661 4.609670e-04
## q  0.004261679 0.12568653  0.03390720 9.730367e-01
## f  0.234946715 0.12404874  1.89398708 6.188702e-02
## o  0.131873284 0.12160279  1.08445934 2.814597e-01
```

1.2 Compute CV MSE

```
## # Cross validated MSE

MSE_cv = function(data, nfold=10, formula="Y~."){
```

```
  Y_cv = c()
  nrows = nrow(data)
  MSE_test = c()
  for (i in seq(0, nrows-nfold, by=nrows/nfold)){
    test = (i+1):(nfold+i)
    train = -test
    lm.fit = lm(formula, data=data[train,])
    Ytest = predict(lm.fit, data[test,])
    MSE_test = c(MSE_test, mean((data$Y[test]-Ytest)^2))
  }
  return(mean(MSE_test))
}

MSE_cv(data)

## [1] 1.526669
```

1.3 Perform backward selection

```
# Backward selection ------------------------------------

backward_lm = function(data,nfold=10){
  # Initialize with OLS
  formula = "Y~."
  lm.fit = lm(formula, data=data)
  # Initialize outputs
  MSE_test = c()
  AIC = c()
  BIC = c()
  Cp = c()
  next_to_remove = ""
  variables = c()
  n = nrow(data)
  while(length(names(lm.fit$model))>1){
    MSE = MSE_cv(data, nfold=nfold, formula)
    MSE_test = c(MSE,MSE_test)
    AIC = c(AIC(lm.fit),AIC)
    BIC = c(BIC(lm.fit),BIC)
```

```
    d = length(names(lm.fit$coefficients))-1
    Cp = c(MSE + 2*d*mean((lm.fit$residuals)^2)/n,Cp)
    t_values = coef(summary(lm.fit))[,"t value"]
    # Variable with smallest t-value
    next_to_remove = names(which.min(t_values[-1]))
    # Store removed variables in the order
    variables = c(next_to_remove,variables)
    # Update formula
    formula = paste(formula,"-",next_to_remove,sep="")
    # Update model using new formula
    lm.fit = update(lm.fit, formula)
  }
  # Intercept only
  variables = variables
  MSE = MSE_cv(data, nfold, formula)
  MSE_test = c(MSE,MSE_test)
  AIC = c(AIC(lm.fit),AIC)
  BIC = c(BIC(lm.fit),BIC)
  Cp = c(MSE,Cp)
  return(list(variables = variables, MSE_test = MSE_test,
              AIC=AIC, BIC=BIC, Cp=Cp))
}

backward = backward_lm(data)
backward
```

```
## $variables
##  [1] "a" "b" "c" "d" "e" "j" "f" "k" "g" "i" "l" "o" "s" "t" "p" "r" "n"
## [18] "q" "h" "m"
##
## $MSE_test
##  [1] 1.980806 1.831603 1.647572 1.511213 1.399686 1.376298 1.383213
##  [8] 1.346406 1.315683 1.319069 1.284828 1.301687 1.336176 1.356568
## [15] 1.373401 1.394812 1.415984 1.450824 1.487838 1.521835 1.526669
##
## $AIC
##  [1] 354.5459 345.6160 335.7371 326.7194 319.8356 315.5372 313.3477
##  [8] 311.4899 309.4718 308.0315 307.5168 307.0589 307.3790 308.9469
## [15] 310.6606 312.5164 314.3918 316.3248 318.2997 320.1792 321.6355
```

```
##
## $BIC
##  [1] 359.7563 353.4315 346.1578 339.7452 335.4666 333.7734 334.1891
##  [8] 334.9365 335.5235 336.6883 338.7788 340.9261 343.8514 348.0245
## [15] 352.3433 356.8043 361.2849 365.8231 370.4031 374.8878 378.9493
##
## $Cp
##  [1] 1.980806 1.866557 1.709649 1.594614 1.501435 1.495721 1.520640
##  [8] 1.500670 1.485041 1.503154 1.484287 1.515765 1.565825 1.604282
## [15] 1.639406 1.679407 1.719175 1.772749 1.828613 1.881109 1.902802

# Generate data and performs backward selection
backward_data = function(n=100,nfold=10){
  data = makedata(n=n)
  back=backward_lm(data,nfold=nfold)
  back$names=names(data)[-1]
  return(back)
}
```

1.4 Replicate

Here I parallelize the code in order to run it faster. I make use of Amazon AWS in order to run the simulations more quickly.

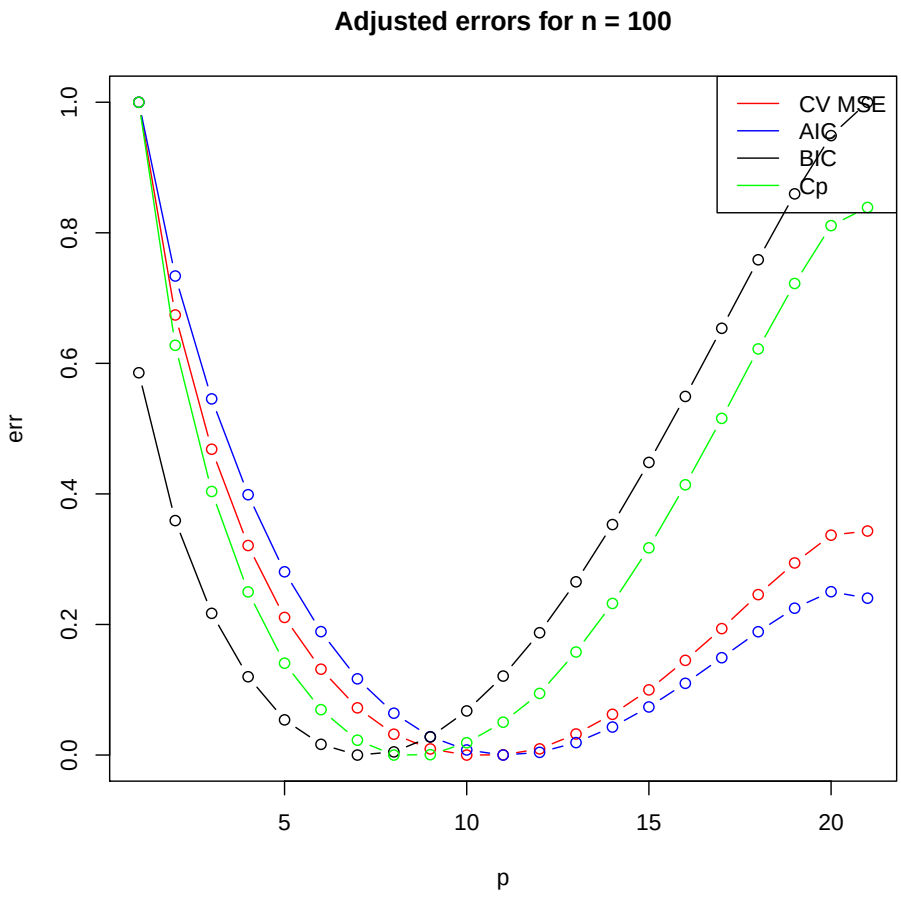
```
## # REPLICATE
library(parallel)
metric_values = function(B,n=100){
  ncores = detectCores()-1
  print(paste('Starting ', ncores, ' cores...'))
  cl = makeCluster(ncores)
  clusterExport(cl,list("makedata", "backward_lm", "MSE_cv",
                        "backward_data"))
  R = parSapply(cl, 1:B, function(i,...)
    { backward_data(n=n) } )
  MSE_values = matrix(unlist(R["MSE_test",]), nrow = B, byrow = T)
  AIC = matrix(unlist(R["AIC",]), nrow = B, byrow = T)
  BIC = matrix(unlist(R["BIC",]), nrow = B, byrow = T)
  Cp_values  = matrix(unlist(R["Cp",]), nrow = B, byrow = T)
  variables = matrix(unlist(R["variables",]), nrow = B, byrow = T)
```

```
  names = matrix(unlist(R["names",]), nrow = B, byrow = T)
  stopCluster(cl)
  print('Done.')
  return(list(MSE=MSE_values, AIC=AIC, BIC=BIC, Cp=Cp_values,
              variables=variables, names=names))
}

B = 1000
Rep_backward100 = metric_values(B,n=100)

## [1] "Starting  39  cores..."
## [1] "Done."

Rep_backward1000 = metric_values(B,n=1000)

## [1] "Starting  39  cores..."
## [1] "Done."
```

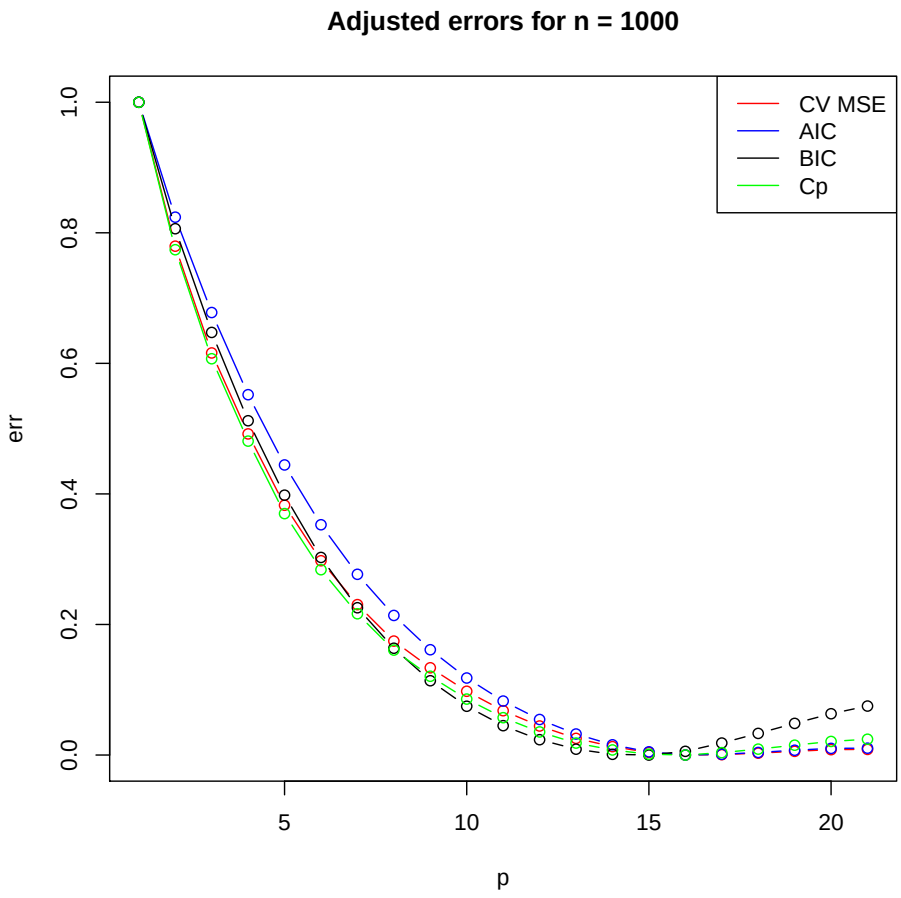
1.5 Plots

The plots are scaled between 0 and 1 for each of the four criteria.

```
# Plots ------------------------------------------------------

plot_err = function(metrics,n){
  m = sapply(metrics, colMeans)
  scale_m = apply(m, MARGIN = 2,
                  FUN = function(X) (X - min(X))/diff(range(X)))
  plot(scale_m[,1],type="b",xlab="p",ylab="err",
       main=paste("Adjusted errors for n =",n), col="red")
  lines(type="b",scale_m[,2],col="blue")
  lines(type="b",scale_m[,3],col="black")
  lines(type="b",scale_m[,4],col="green")
  legend("topright",c("CV MSE","AIC","BIC","Cp"),
         col=c("red","blue","black","green"),lty=1)
}

plot_err(Rep_backward100[-c(5,6)],100)
```

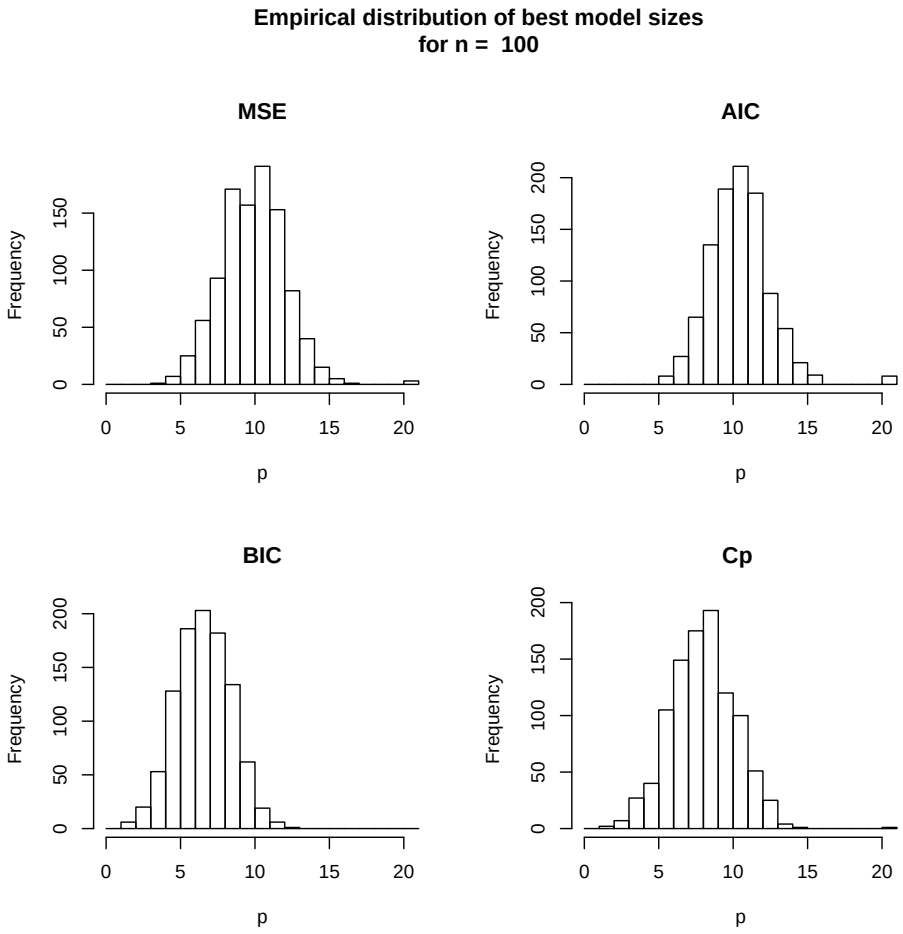



```
plot_err(Rep_backward1000[-c(5,6)],1000)
```

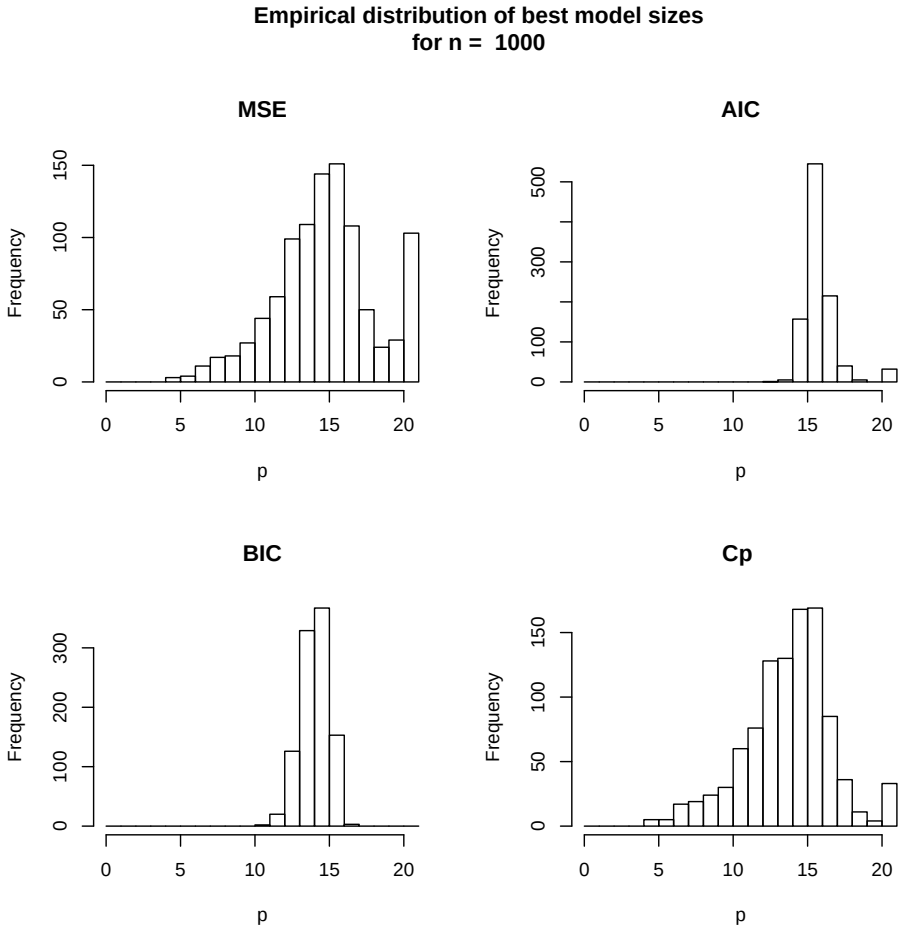



```
plot_model_sizes = function(metrics,n){
  m = lapply(metrics, function(X) {apply(X,1,which.min)})
  par(mfrow=c(2,2),oma=c(0,0,3,0))
  hist(m$MSE,main="MSE",xlim=c(0,21),breaks=0:21,xlab="p")
  hist(m$AIC,main="AIC",xlim=c(0,21),breaks=0:21,xlab="p")
  hist(m$BIC,main="BIC",xlim=c(0,21),breaks=0:21,xlab="p")
  hist(m$Cp ,main="Cp ",xlim=c(0,21),breaks=0:21,xlab="p")
  title(paste("Empirical distribution of best model sizes\nfor n = ",n),
        outer=TRUE)
  par(mfrow=c(1,1),oma=c(0,0,0,0))
}
```

```
plot_model_sizes(Rep_backward100[-c(5,6)],n=100)
```




```
plot_model_sizes(Rep_backward1000[-c(5,6)],n=1000)
```



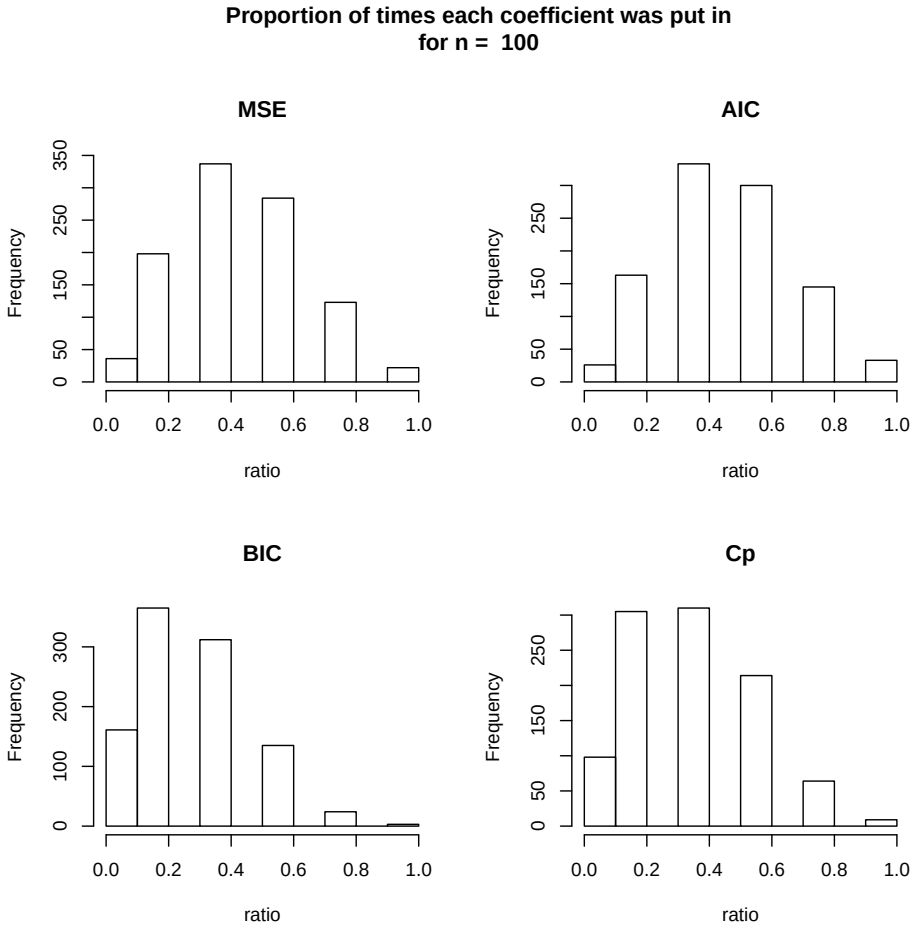
1.6 Proportions

Here I calculate the proportion of times each coefficient was left out (for the first 15) and put in (for the last 5).

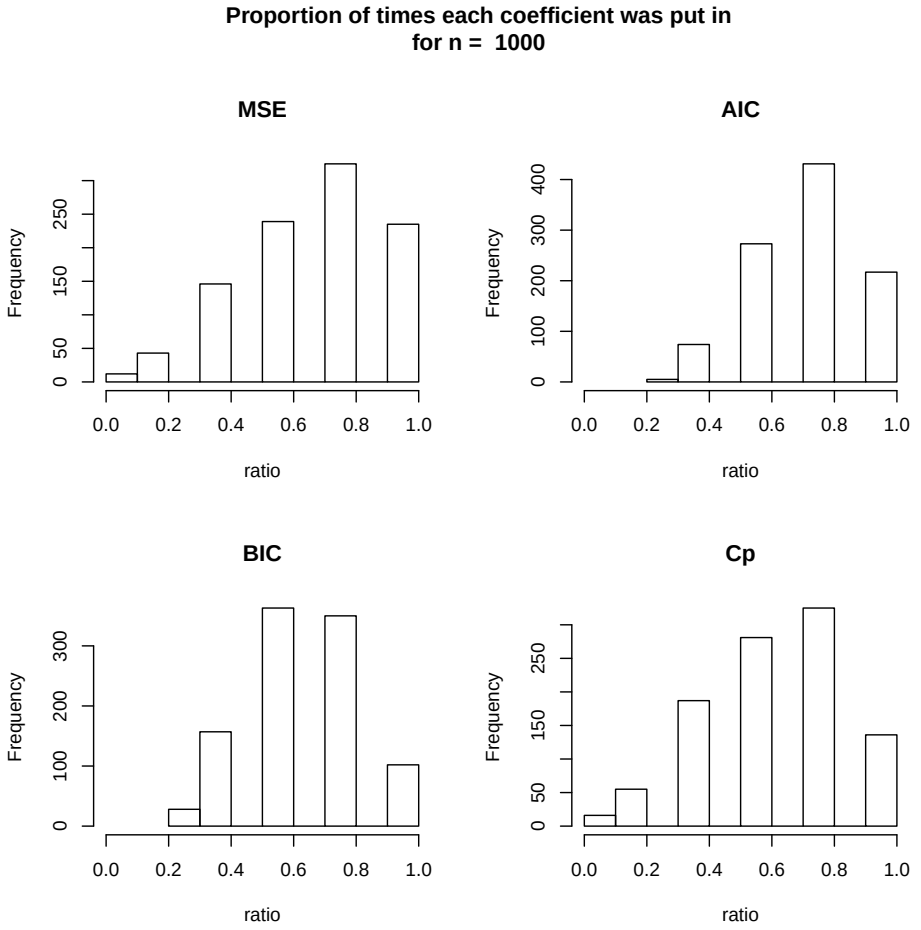
```
# Proportions ------------------------------------------------

## # left out first 15
prop_left = function(metrics){
  optimum = lapply(metrics[-c(5,6)], function(X)
    {apply(X,1,which.min)})
  proportions_left = optimum
```

```
  for (i in 1:B){
    proportions_left$MSE[i] = 1-length(intersect(
      metrics$variables[i,1:(optimum$MSE[i]-1)],
      metrics$names[i,1:15]))/15
    proportions_left$AIC[i] = 1-length(intersect(
      metrics$variables[i,1:(optimum$AIC[i]-1)],
      metrics$names[i,1:15]))/15
    proportions_left$BIC[i] = 1-length(intersect(
      metrics$variables[i,1:(optimum$BIC[i]-1)],
      metrics$names[i,1:15]))/15
    proportions_left$Cp[i] =  1-length(intersect(
      metrics$variables[i,1:(optimum$Cp[i] -1)],
      metrics$names[i,1:15]))/15
  }
  proportions_left
}

proportions_left100 = prop_left(Rep_backward100)
proportions_left1000 = prop_left(Rep_backward1000)


## # kept last 5
prop_kept = function(metrics){
  optimum = lapply(metrics[-c(5,6)], function(X)
    {apply(X,1,which.min)})
  proportions_kept = optimum
  for (i in 1:B){
    proportions_kept$MSE[i] = length(intersect(
      metrics$variables[i,1:(optimum$MSE[i]-1)],
      metrics$names[i,16:20]))/5
    proportions_kept$AIC[i] = length(intersect(
      metrics$variables[i,1:(optimum$AIC[i]-1)],
      metrics$names[i,16:20]))/5
    proportions_kept$BIC[i] = length(intersect(
      metrics$variables[i,1:(optimum$BIC[i]-1)],
      metrics$names[i,16:20]))/5
    proportions_kept$Cp[i] =  length(intersect(
      metrics$variables[i,1:(optimum$Cp[i])-1 ],
      metrics$names[i,16:20]))/5
```

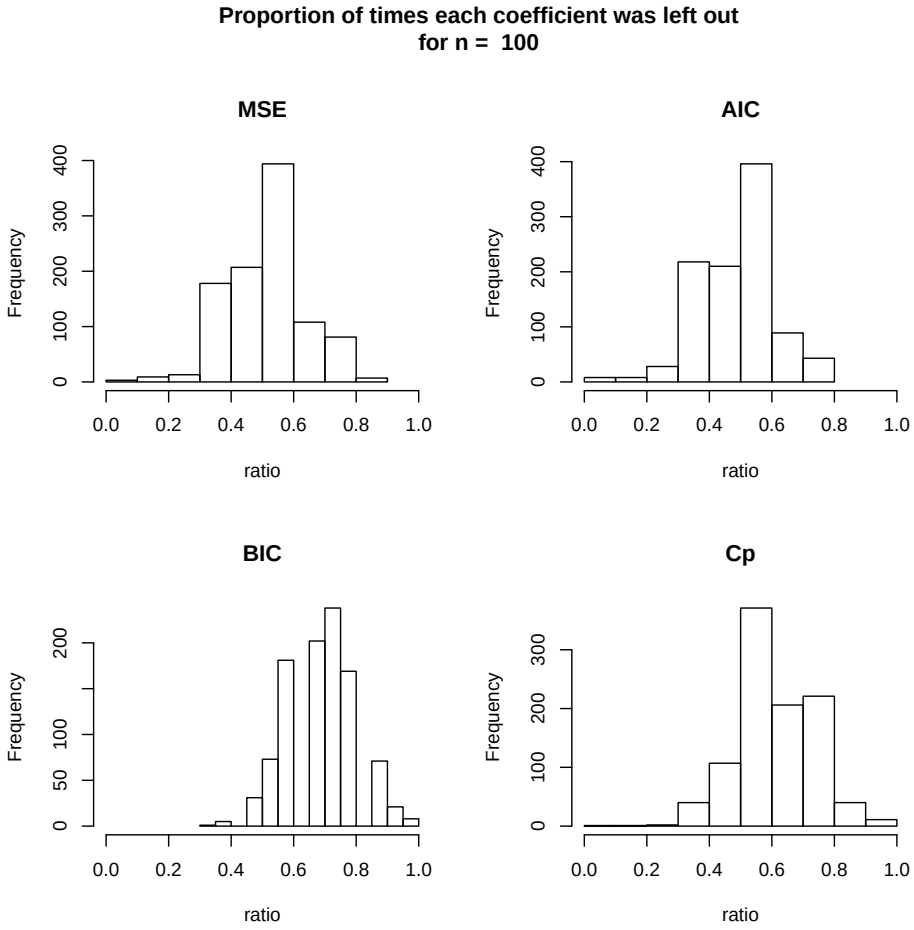
```
  }
  proportions_kept
}

proportions_kept100 = prop_kept(Rep_backward100)
proportions_kept1000 = prop_kept(Rep_backward1000)



# Plot ----------------------------------------------------

plot_kept = function(proportions_kept,n){
  par(mfrow=c(2,2),oma=c(0,0,3,0))
  hist(proportions_kept$MSE,main="MSE",
       xlim=c(0,1),xlab="ratio")
  hist(proportions_kept$AIC,main="AIC",
       xlim=c(0,1),xlab="ratio")
  hist(proportions_kept$BIC,main="BIC",
       xlim=c(0,1),xlab="ratio")
  hist(proportions_kept$Cp, main="Cp",
       xlim=c(0,1),xlab= "ratio")
  title(paste("Proportion of times each coefficient was put in\nfor n = ",n),
        outer=TRUE)
  par(mfrow=c(1,1),oma=c(0,0,0,0))
}

plot_kept(proportions_kept100,100)
```



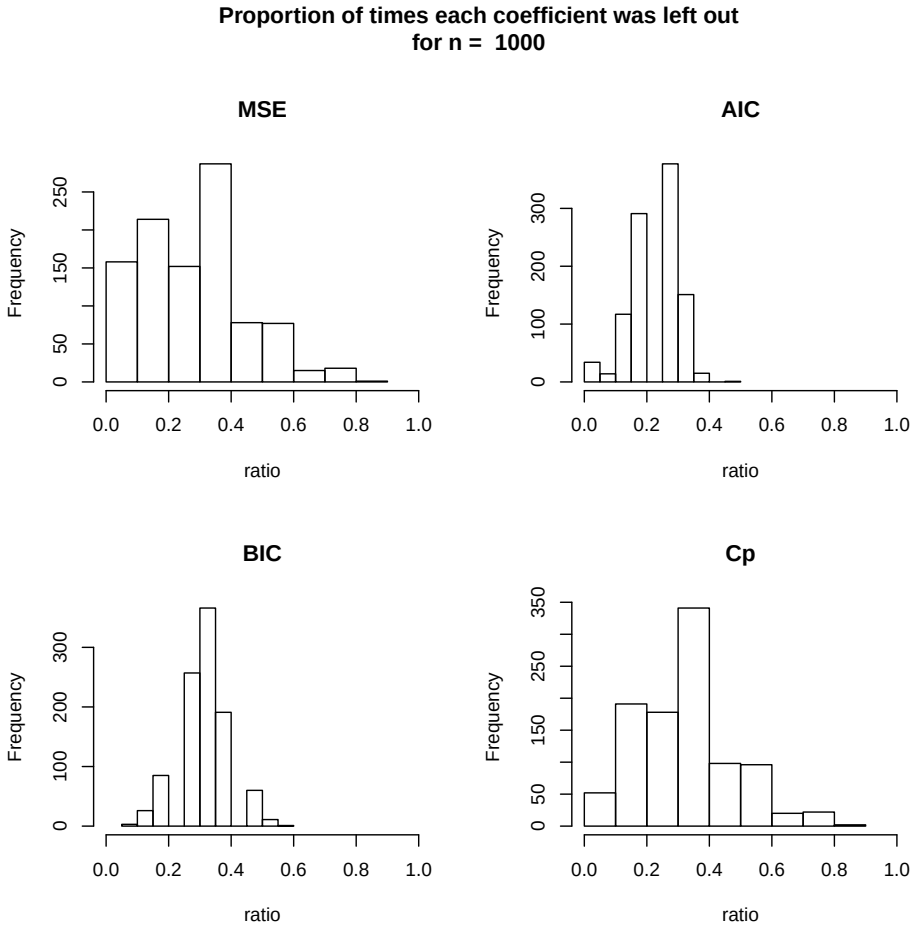

```
plot_kept(proportions_kept1000,1000)
```




```
plot_left = function(proportions_left,n){
  par(mfrow=c(2,2),oma=c(0,0,3,0))
  hist(proportions_left$MSE,main="MSE",xlim=c(0,1),xlab="ratio")
  hist(proportions_left$AIC,main="AIC",xlim=c(0,1),xlab="ratio")
  hist(proportions_left$BIC,main="BIC",xlim=c(0,1),xlab="ratio")
  hist(proportions_left$Cp, main="Cp",xlim=c(0,1),xlab= "ratio")
  title(paste("Proportion of times each coefficient was left out\nfor n = ",n),
        outer=TRUE)
  par(mfrow=c(1,1),oma=c(0,0,0,0))
}

plot_left(proportions_left100,100)
```




```
plot_left(proportions_left1000,1000)
```



BIC has a heavier penalty term than AIC or Cp so it tends to favor simpler models, we can see it here: the proportion of left out parameters for the first 15 variables is greater for this statistic. This is also similar to Mallow's Cp. AIC has the lowest error, as expected because it favors more complex models than BIC.We can also notice that as n increases, the empirical variance reduces.

2

2.1 Load data

Here I load the data with outputing the true beta.

```r
library(pls)

##
## Attaching package:  'pls'
##
## The following object is masked from 'package:stats':
##
##     loadings

rm(list = ls())
cat("\014")
```

```r
# Data ------------------------------------------------

makedata=function(p=20,wh=15,n=100){
  X=matrix(rnorm(n*p),n,p)
  exps=seq(-1,-2.5,length=wh)
  beta=rep(0,p)
  beta[1:wh]=exp(exps)
  Y=.5+X%*%beta+rnorm(n)
  data = data.frame(Y,X)
  colnames(data)=c("Y",letters[1:20])
  switch=sample(20)+1
  data=data[,c(1,switch)]
  names(beta)=names(data)[switch]
  return(list(df=data,beta=beta))
}

set.seed(1)
data=makedata()
data$beta

##          c          i          g          r          d          o
## 0.36787944 0.33050190 0.29692203 0.26675395 0.23965104 0.21530185
##          n          h          m          j          f          q
## 0.19342660 0.17377394 0.15611805 0.14025603 0.12600565 0.11320313
##          e          p          s          l          b          a
## 0.10170139 0.09136826 0.08208500 0.00000000 0.00000000 0.00000000
##          t          k
## 0.00000000 0.00000000
```

2.2 Replicate

Here again I make use of parrallelization.

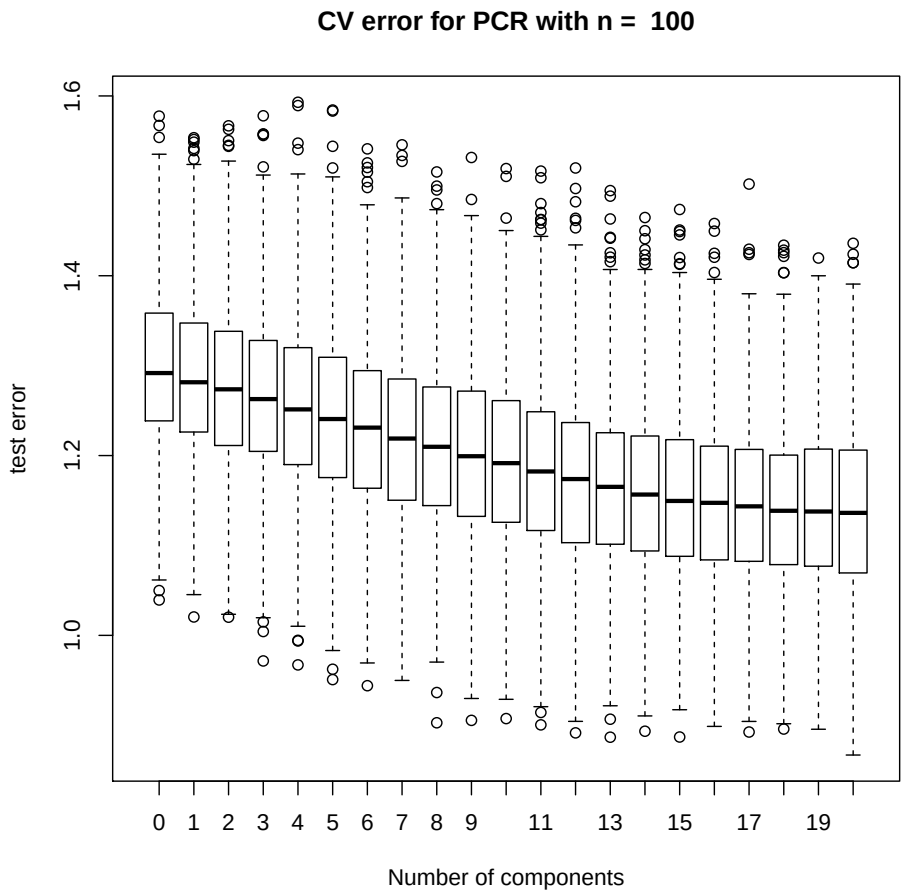
```
# Replicate ------------------------------------------------
cv_pcr = function(n){
  data = makedata(n=n)
  df = data$df
  beta0 = data$beta
  pcr.fit = pcr(Y~0+., ncomp=20, data=df, validation ="CV")
  cv_error = as.data.frame(RMSEP(pcr.fit)$val)[1,]
  names(cv_error) = 0:20
  # beta pcr
  beta0 = data$beta
  beta_pcr = rev(sort(pcr.fit$coefficients[,,20]))
  return(list(CV=cv_error, beta0=beta0, beta=beta_pcr))
}


library(parallel)

pcr_rep = function(B,n=100){
  ncores = detectCores()-1
  print(paste('Starting ', ncores, ' cores...'))
  cl = makeCluster(ncores)
  clusterEvalQ(cl,library(pls))
  clusterExport(cl,list("cv_pcr","makedata"))
  R = parSapply(cl, 1:B, function(i,...)
  { cv_pcr(n=n) } )
  CV = matrix(unlist(R["CV",]), nrow = B, byrow = T)
  beta0 = matrix(unlist(R["beta0",]), nrow = B, byrow = T)[1,]
  beta = matrix(unlist(R["beta",]), nrow = B, byrow = T)
  stopCluster(cl)
  print('Done.')
  return(list(CV=CV,beta=beta,beta0=beta0))
}

B = 1000
Rep_pcr100 = pcr_rep(B,n=100)
```

```
## [1] "Starting  39  cores..."
```

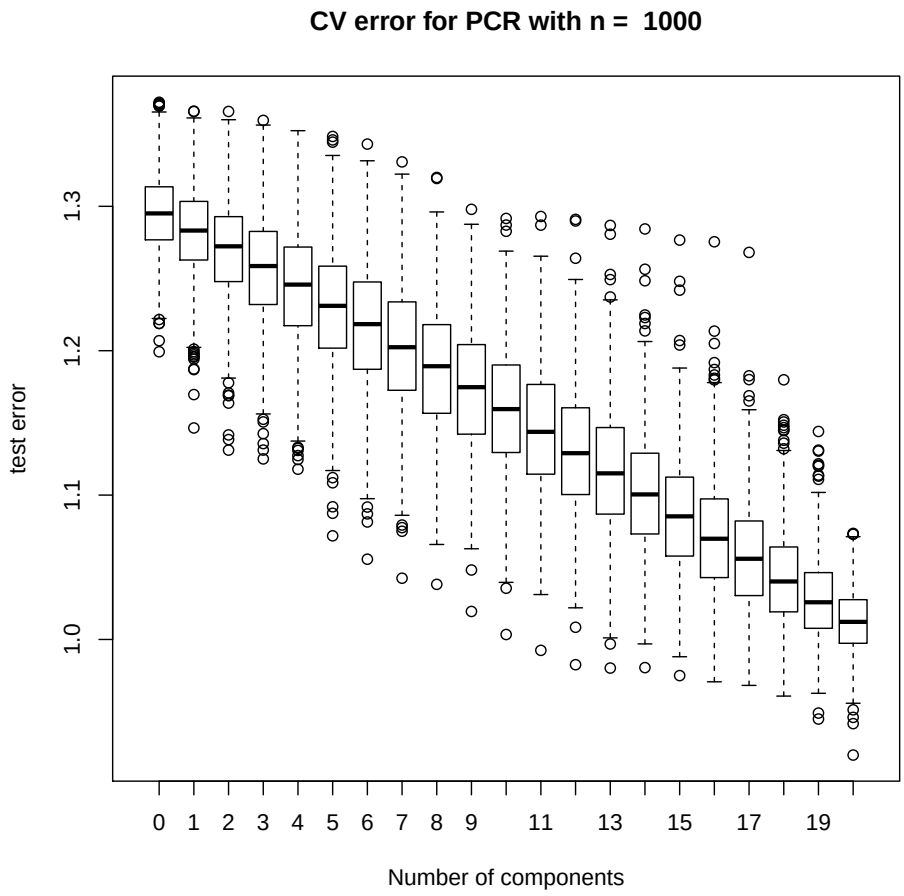
```
## [1] "Done."

Rep_pcr1000 = pcr_rep(B,n=1000)

## [1] "Starting  39  cores..."
## [1] "Done."
```

2.3 Plots

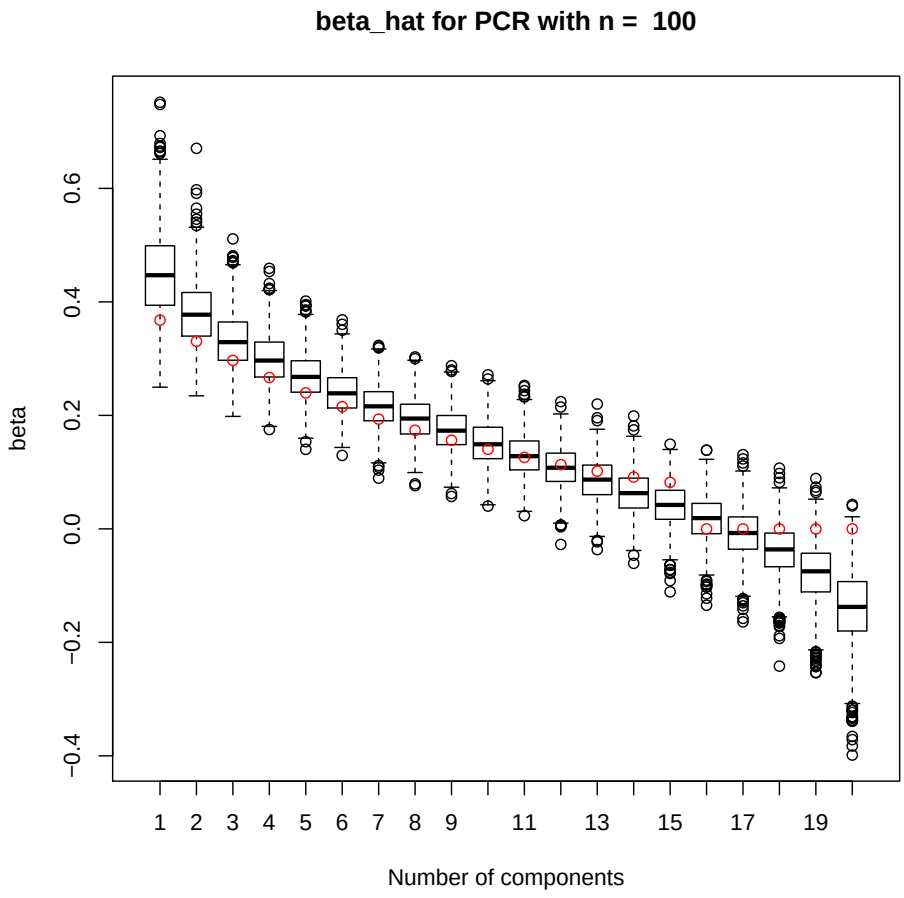
```
# Plot ----------------------------------------------------

# n=100
boxplot(Rep_pcr100$CV,
        main=paste("CV error for PCR with n = ",100),
        xlab = "Number of components",
        ylab="test error",names=0:20)
```



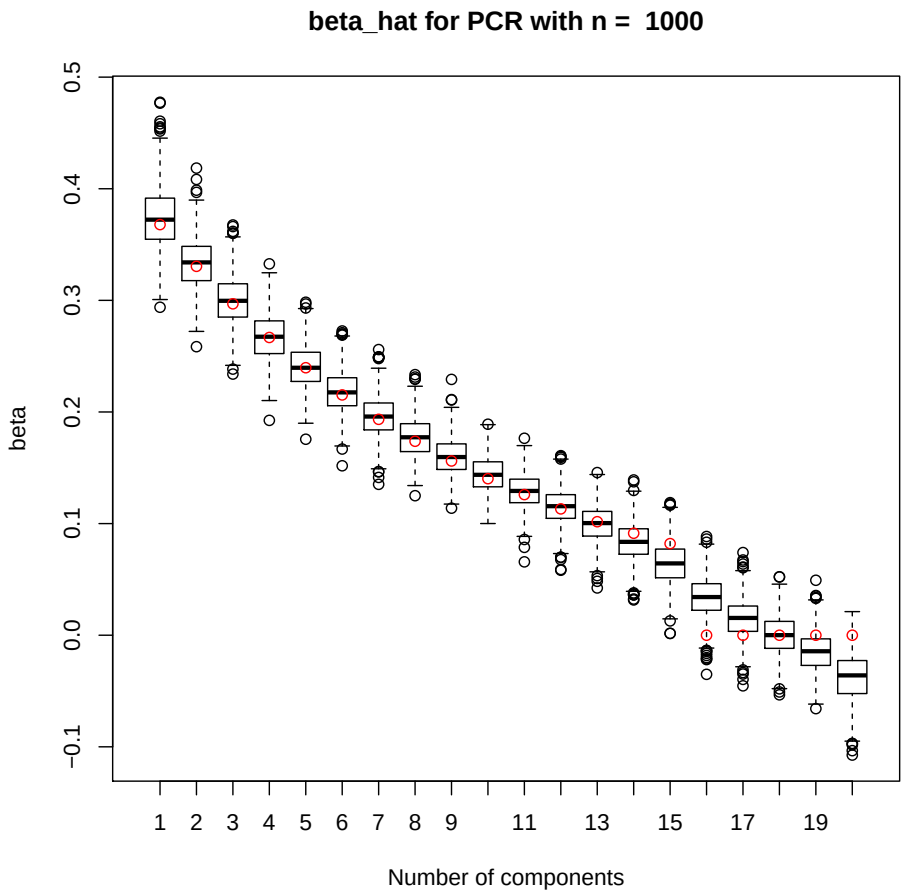

```
#n=1000
boxplot(Rep_pcr1000$CV,
        main=paste("CV error for PCR with n = ",1000),
        xlab = "Number of components",
        ylab="test error",names=0:20)
```




```
#n=100
boxplot(Rep_pcr100$beta,
        main=paste("beta_hat for PCR with n = ",100),
        xlab = "Number of components",
        ylab="beta")
points(1:20,Rep_pcr1000$beta0,col="red")
```




```
#n=1000
boxplot(Rep_pcr1000$beta,
        main=paste("beta_hat for PCR with n = ",1000),
        xlab = "Number of components",
        ylab="beta")
points(1:20,Rep_pcr1000$beta0,col="red")
```



The CV error goes down as we increase the number of components. The true beta remains in the confidence interval even if there seems to be a little biased.