Homework 1

Solution

Installing and loading packages

```
packages = c('biglm','foreach','doParallel')
for(package in packages){
  if(!require(package,character.only=TRUE,quietly=TRUE)){
    install.packages(package,repos='http://cran.us.r-project.org')
    require(package,character.only=TRUE,quietly=TRUE)
  }
}
```

Out-of-memory Least Squares via biglm

An important part of being a data scientist is getting comfortable experimenting via simulation. Simulation is the process of generating data from a known distribution. Then, the performance of any procedure(s) can be evaluated/explored precisely due to knowing the truth (which would be unknowable for a data set). Let's first experiment with fitting the least squares procedure to simulated data.

```
set.seed(1)
n = 2000
p = 500
X = matrix(rnorm(n*p),nrow=n,ncol=p)
X[,1] = 1
format(object.size(X),units='auto')#memory used by X
```

```
## [1] "7.6 Mb"
```

```
b = rep(0,p)
b[1:5] = 25
b_0 = 0
Xdf = data.frame(X)
Y = b_0 + X %*% b + rnorm(n)
hatBeta = coef(lm(Y~X-1)) #Here, the [-1] ignores the intercept
```

Part a

Report the first 5 entries in $\hat{\beta}$ (that is, hatBeta in the above code) using lm on all the data simultaneously

```
#SOLUTION
print(hatBeta[1:5])
```

```
##       X1       X2       X3       X4       X5
## 24.99160 24.98724 24.96734 25.05268 25.04096
```

Part b

Though this is practically speaking a small problem (see the 'object.size' above), let's pretend it is too large to fit in memory and hence directly using lm is infeasible. The first step is to get the feature matrix into chunks on the hard drive:

```
#Using out-of-core technique
write.table(X[1:500,],file='Xchunk1.txt',sep=',',row.names=F,col.names=names(Xdf))
write.table(X[501:1000,],file='Xchunk2.txt',sep=',',row.names=F,col.names=names(Xdf))
write.table(X[1001:1500,],file='Xchunk3.txt',sep=',',row.names=F,col.names=names(Xdf))
write.table(X[1501:2000,],file='Xchunk4.txt',sep=',',row.names=F,col.names=names(Xdf))
write.table(Y[1:500],file='Ychunk1.txt',sep=',',row.names=F,col.names=F)
write.table(Y[501:1000],file='Ychunk2.txt',sep=',',row.names=F,col.names=F)
write.table(Y[1001:1500],file='Ychunk3.txt',sep=',',row.names=F,col.names=F)
write.table(Y[1501:2000],file='Ychunk4.txt',sep=',',row.names=F,col.names=F)
```

Now, we can read in each chunk individually and update the least squares solution using biglm. This will alleviate the need for reading all of X into memory at the same time. The below code is only partially completed. Complete the procedure in the natural way on the remaining chunks.

```
# Chunk 1
Xchunk = read.table(file='Xchunk1.txt',sep=',',header=T)
Ychunk = scan(file='Ychunk1.txt',sep=',')
form = as.formula(paste('Ychunk ~ -1 + ',paste(names(Xchunk),collapse=' + '),collapse=''))
out.biglm = biglm(formula = form,data=Xchunk)
hatBeta[1:5]
```

```
##       X1       X2       X3       X4       X5
## 24.99160 24.98724 24.96734 25.05268 25.04096
```

```
coef(out.biglm)[1:5]
```

```
##       X1       X2       X3       X4       X5
## 25.08815 24.48229 26.09057 26.84305 24.64633
```

```
# Chunk 2
Xchunk = read.table(file='Xchunk2.txt',sep=',',header=T)
Ychunk = scan(file='Ychunk2.txt',sep=',')
out.biglm = update(out.biglm,moredata=Xchunk)
hatBeta[1:5]
```

```
##       X1       X2       X3       X4       X5
## 24.99160 24.98724 24.96734 25.05268 25.04096
```

```
coef(out.biglm)[1:5]
```

```
##       X1       X2       X3       X4       X5
## 24.96665 25.00382 24.97887 25.01876 25.08741
```

```
# Chunk 3
Xchunk = read.table(file='Xchunk3.txt',sep=',',header=T)
Ychunk = scan(file='Ychunk3.txt',sep=',')
out.biglm = update(out.biglm,moredata=Xchunk)
hatBeta[1:5]
```

```
##       X1       X2       X3       X4       X5
## 24.99160 24.98724 24.96734 25.05268 25.04096
```

```
coef(out.biglm)[1:5]
```

```
##       X1       X2       X3       X4       X5
## 24.98077 25.00492 24.97527 25.02329 25.08050
```

```
## Can you figure out the final steps? Have we updated on all of the chunks?
#Solution
# Chunk 4
Xchunk = read.table(file='Xchunk4.txt',sep=',',header=T)
Ychunk = scan(file='Ychunk4.txt',sep=',')
out.biglm = update(out.biglm,moredata=Xchunk)
```

Solution to part b:

Compare the first 5 entries in $\hat{\beta}$ formed by this method with the entries in (a)

```
#SOLUTION
print(hatBeta[1:5])
```

```
##       X1       X2       X3       X4       X5
## 24.99160 24.98724 24.96734 25.05268 25.04096
```

```
print(coef(out.biglm)[1:5])
```

```
##       X1       X2       X3       X4       X5
## 24.99160 24.98724 24.96734 25.05268 25.04096
```

Yes all chunks updated because we see the two solutions match

Cross Validation and Parallelism

First, let's explore parallelism in R. Parallelism can be used to speed up computations by using multiple processors/CPUs at the same time. It is especially useful if we need to run a lot of computations at the same time that don't depend on eachother (this is often called "trivially" or "embarrassingly" parallel)

Note: The -1 in the "nCores" below is if using a GUI and hence needing system resources for other than processing. Usually, parallel processing is meant to be run in "batch" mode (R CMD BATCH myRscript.r &) though we won't be running in batch for this assignment.

```
nCores  = detectCores(all.tests = FALSE, logical = TRUE) - 1
cat('My work station has ',nCores,' cores \n')
```

```
## My work station has  3  cores
```

```
workers = makeCluster(nCores)
registerDoParallel(workers)

tmp  = rep(.1,nCores*10)
wait = function(tmp_i) Sys.sleep(tmp_i)
system.time(sapply(tmp,wait))#single processor
```

```
##    user  system elapsed
##   0.003   0.001   3.076
```

```
system.time(foreach(tmp_i = tmp) %dopar% {wait(tmp_i)})#multi-processor
```

```
##    user  system elapsed
##   0.138   0.005   1.192
```

Comment on the relative sizes of nCores vs. the system.time with and without parallelism (note that if you happen to have a 2 core or fewer work station, you won't see any difference. If you have 2 cores, try and eliminate the '-1' from the code above)

Solution:

My work station has 3 available cores (the 4th core is currently running the R session). The time for multi-processor computations takes about 1.215/3.025 = .40 times as long.

Cross-Validation

Now, we want to implement our own CV and apply it to the simulated data from the previous question.

```
K        = 10# Do K fold CV
folds    = sample(rep(1:K, length.out = n))
CVoutput = rep(0,K)

for(k in 1:K){
  validIndex  = which(folds == k)
  YhatValid   = X[validIndex,]%*%coef(lm(Y~X-1,subset=-validIndex))
  CVoutput[k] = mean( (YhatValid-Y[validIndex])**2  )
}
cat('CV estimate of the risk: ',mean(CVoutput),'\n')
```

```
## CV estimate of the risk:  1.369106
```

```
cat('Standard error of CV estimate of the risk: ',sd(CVoutput)/sqrt(K),'\n')
```

```
## Standard error of CV estimate of the risk:  0.04290718
```

What could the standard error of CV estimate be used for?

Solution:

The standard error of the CV estimate could be used to quantify uncertainty with respect to the risk estimate. In general, the variability of CV will increase with K. (see the plot in the next questions for some evidence of that in terms of the standard errors for various K).

Note that in the homework I didn't have the standard deviation divided by sqrt(K). I was looking to see if anyone would think critically about computing the standard deviation of a set of numbers vs. the standard deviation of their average. Sometimes the denominator is sqrt(K-1) instead.

Cross-validation in Parallel

CV runs well in parallel, so let's try that out. First, let's define a function:

```
cvF = function(k){
  validIndex  = which(folds == k)
  YhatValid   = X[validIndex,]%*%coef(lm(Y~X-1,subset=-validIndex))
  return(mean( (YhatValid-Y[validIndex])**2  ))
}
```

Now, we want to extend the above parallel code to CV

```
K        = 9# Do K fold CV
folds    = sample(rep(1:K, length.out = n))
startTime = proc.time()[3]
CVoutputParallel = foreach(k = 1:K) %dopar%{cvF(k)}
endTime = proc.time()[3]
cat('CV time: ',endTime-startTime,'\n')
```

```
## CV time:  3.035
```

```
cat('CV estimate of the risk: ',mean(CVoutput),'\n')
```

```
## CV estimate of the risk:  1.369106
```

```
K        = 9# Do K fold CV
folds    = sample(rep(1:K, length.out = n))
startTime = proc.time()[3]
CVoutputParallel = foreach(k = 1:K) %do%{cvF(k)}
endTime = proc.time()[3]
cat('CV time: ',endTime-startTime,'\n')
```

```
## CV time:  3.868
```

```
cat('CV estimate of the risk: ',mean(CVoutput),'\n')
```

```
## CV estimate of the risk:  1.369106
```

The above code can be made to not run in parallel by replacing "dopar" with "do". Compare the time for CV with and without parallelism.

Solution:

Here, the parallel CV took about 2/3s as long. Here, we didn't get as much improvement in time as the previous example. One reason is that with K = 10 and my 3 cores, we aren't being as efficient as we could be (K=9 would be much faster..)

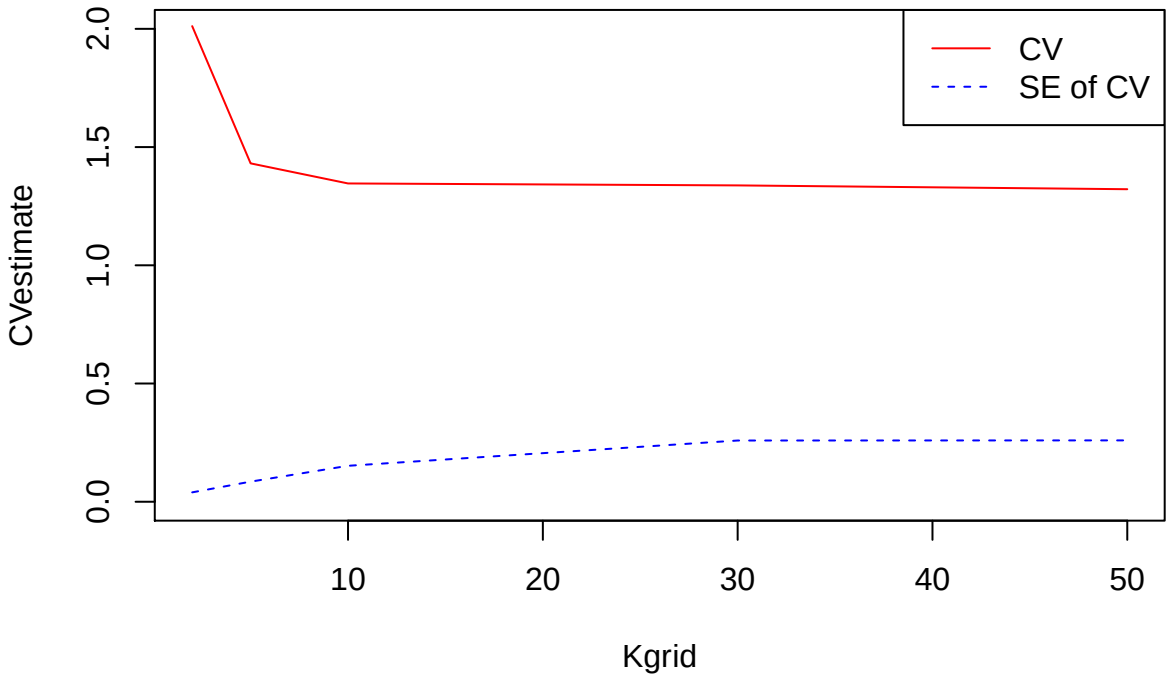
Solution

Also, make a plot of the CV estimate of the risk for a variety of K values (note that LOOCV would be of interest, but would take awhile on this problem):

```
Kgrid = c(2,5,10,30,50)
CVestimate = rep(0,length(Kgrid))
CVse       = rep(0,length(Kgrid))
Kiter = 0
for(K in Kgrid){
  Kiter = Kiter + 1
  folds    = sample(rep(1:K, length.out = n))
  CVestimateVec     = unlist(foreach(k = 1:K) %dopar%{cvF(k)})
  CVestimate[Kiter] = mean(CVestimateVec)
  CVse[Kiter]       = sd(CVestimateVec)
}
plot(Kgrid,CVestimate,col='red',type='lines',ylim=c(0,2))
```

```
## Warning in plot.xy(xy, type, ...): plot type 'lines' will be truncated to
## first character
```

```
lines(Kgrid,CVse,col='blue',lty=2)
legend(x='topright',legend=c('CV','SE of CV'),lty=1:2,col=c('red','blue'))
```



Possibly tricky question: For which K is the K-Fold CV the best estimate of the risk?

Solution

This is a multifaceted question. In general, small values of K will have a high bias/low variance and large values of K (in particular K = n) will have low/no bias and high variance. So, picking intermediate values (say K = 10) is a good general rule. Some exceptions: if you have very small (here, if you have n=20, choosing a smaller K makes sense) or very large (here there are computational considerations about the number of folds) data sets.

Cross-validation in Caret

The "caret" package can do some commonly done machine learning tasks for you. In particular, create the folds:

```
if(!require(caret)) install.packages("caret", dependencies = c("Depends", "Suggests"));require(caret)
```

```
## Loading required package: caret
## Loading required package: lattice
## Loading required package: ggplot2
```

```
#See https://cran.r-project.org/web/packages/caret/vignettes/caret.pdf
K = 10
folds = createFolds(Y, k = K, list = TRUE, returnTrain = FALSE)
```

Solution

In our CV implementation, we make a length n vector of permuted values of 1 up to K. Hence, we grab the indices for k = 1 to get the first validation set. The caret package makes a list with K entries. The kth entry

is a vector of the indices of the kth validation set (note this more directly mimics the V = {v_1,v_2,. . . ,v_K} notation from the lecture notes).

Forward Selection

Using the same simulated X and Y generated in the previous problem, use forward selection and AIC to estimate the a constrained version of the least squares problem

```
if(!require(leaps)){install.packages('leaps',repos='http://cran.us.r-project.org');require(leaps)}
```

```
## Loading required package: leaps
```

Solution

Compare the first 5 entries of the coefficient vector estimated via forward selection to the first 5 entries of the coefficient vector found using lm (or biglm). How many nonzero entries do each have?

```
if(!require(leaps)){install.packages('leaps',repos='http://cran.us.r-project.org');require(leaps)}
outFor     = regsubsets(x=X,y=Y,nvmax=p,method='forward',intercept=FALSE)
#Note: the intercept = FALSE is because we defined a column of 1's in X already)
sumFor     = summary(outFor)
modelFor   = sumFor$which[which.min(sumFor$cp),]
#note that I mentioned in riskEstimation lectures that cp and AIC will select
#the same model
(leapsModel = as.numeric(which(modelFor)))
```

```
##  [1]   1   2   3   4   5   7  12  31  35  42  46  56  69  72  74  77  88
## [18]  93  97 109 117 138 140 141 152 160 165 179 188 189 190 199 211 212
## [35] 215 226 231 256 257 261 267 270 276 278 284 286 288 301 308 314 336
## [52] 347 352 364 369 370 372 379 380 390 399 407 410 414 426 427 433 441
## [69] 451 457 458 482 485 493
```

```
#Some people chose to use stepAIC in MASS, so I'm including it here
#  It's quite a bit slower than regsubsets, I haven't looked into why
#  Also, it is much less computationally efficient due to needing to specify the "scope"
#      and hence have to fit the full least squares problem (even though we stop well short)
require(MASS)
```

```
## Loading required package: MASS
```

```
X.df   = data.frame(X)
upper  = lm(Y~.-1,data=X.df)
lower  = lm(Y~-1,data=X.df)
out.aic = stepAIC(lower,scope=list(upper=upper,lower=lower),direction='forward',trace=0)#trace just sto
attributes(out.aic$terms)$variables
```

```
## list(Y, X4, X1, X5, X2, X3, X231, X256, X257, X278, X152, X72,
##     X88, X390, X56, X380, X370, X46, X31, X314, X451, X276, X301,
##     X35, X138, X212, X270, X364, X117, X347, X433, X458, X399,
##     X261, X77, X160, X485, X288, X226, X286, X369, X12, X427,
##     X190, X189, X426, X199, X74, X140, X69, X410, X267, X215,
##     X109, X352, X141, X165, X336, X372, X308, X97, X93, X179,
##     X441, X42, X211, X493, X379, X414, X7, X284, X457, X188,
##     X407, X482, X112, X64, X318, X58, X491, X366, X111, X460,
##     X350, X135, X173, X206, X54, X118, X480, X376)
```

Extra part

Some people went through (perhaps by accident) and tried to implement their own forward stepwise following the pseudo-code in lectures. I've followed up on this idea here in case it is useful. You can do this in an out of core manner, which I've implemented here. Note that the out of core implementation will be very slow compared to the in core implementation. This is in general always true, as reading from the hard-drive is very slow. However, if the data set is large enough this becomes the only reasonable alternative.

Save the $\mathbb{X}$ generated in the previous problem to a .csv file. Using forward selection and AIC, estimate b without having $\mathbb{X}$ stored in memory. Verify that your answer matches (a)

```
write.csv(x=X,file='featureMat.csv')
p          = ncol(X)
n          = nrow(X)
sigmaSq    = NULL#Try sigmaSq = out.biglm$qr$ss/(n-p)#
gicType    = 'AIC'
outOfCore = FALSE### To do forward selection out of core, will be slow

GICf = function(ind,gicType = 'AIC', sigmaSq = NULL,outOfCore = FALSE){
  if(outOfCore){
    grabVec = rep('NULL',p)
    grabVec[ind] = NA
    featureMat = read.csv('featureMat.csv', colClasses=grabVec)
    lm.out = lm(Y~.-1,data=featureMat)
  }else{
    lm.out = lm(Y~X[,ind]-1)
  }
  if(gicType == 'AIC'){
    scaleTerm = 2
  }else if(gicType == 'BIC'){
    scaleTerm = log(n)
  }else{stop('Only supports AIC or BIC')}

  if(!is.null(sigmaSq)){
    if(class(sigmaSq) != class(1) | sigmaSq < 0){stop('Invalid variance estimate')}
    return(sum(lm.out$residuals**2)/n + scaleTerm/n * length(ind)*sigmaSq )
  }else{
    return(n*log(sum(lm.out$residuals**2)/n) + scaleTerm * length(ind) )
  }
}


GIC          = Inf#initialize
gicOutput    = list()
indSelect    = c(1)#initialize
indSet       = 2:p#initialize
importantVar = 0#initialize
addedNewVar  = FALSE#initialize
verbose      = 0#Controls the amount of output, larger number -> more output
sweepRepeat = 0

repeat{
  sweepRepeat = sweepRepeat + 1
  if(verbose > 0){
    cat('We have selected thus far: ',indSelect,'\n')
```

```
  }
  countFeatures = 0
  indSetSweep   = 0#this gets the index in indSet of importantVar
  for(j in indSet){
    indSetSweep = indSetSweep + 1
    countFeatures = countFeatures + 1
    if(verbose > 1){
      if(countFeatures %% round(length(indSet)/5) == 0){
        cat('We have looked at the first: ', ... =
            countFeatures/length(indSet),' fraction of features \n')
      }
    }
    indTmp = c(indSelect,j)
    GICnew = GICf(indTmp, gicType = gicType,
                  sigmaSq = sigmaSq,outOfCore = outOfCore)
    if(GICnew < GIC){
      GIC = GICnew
      importantVar      = j
      importantVarIndex = indSetSweep
      addedNewVar       = TRUE
    }

  }
  if(!addedNewVar){
    break
  }else{
    indSet    = indSet[-importantVarIndex]
    indSelect = c(indSelect,importantVar)
  }
  gicOutput[[sweepRepeat]] = GIC
  addedNewVar = FALSE
}
setdiff(indSelect,leapsModel)
```

```
##  [1] 112  64 318  58 491 366 111 460 350 135 173 206  54 118 480 376
```

```
setdiff(leapsModel,indSelect)
```

```
## numeric(0)
```