Cross-validation and Model Selection

Rob Leonard (robleonard@tamu.edu (mailto:robleonard@tamu.edu))

Out-of-memory Least Squares via biglm

An important part of being a statistician 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.

The full least squares solution

```
set.seed(1)
n = 4000
p = 500
X = matrix(rnorm(n*p),nrow=n,ncol=p)
format(object.size(X),units='auto')  #memory used by X
```

```
## [1] "15.3 Mb"
```

```
b = rep(0,p)
b[1:5] = 25
Xdf = data.frame(X)
Y = 5 + X %*% b + rnorm(n)
betaHat = coef(lm(Y~X))
betaHat[1:6]
```

```
## (Intercept)          X1          X2          X3          X4          X5
##    4.991328   25.009889   24.992888   24.988658   25.011266   24.995551
```

Report the first 5 coefficient estimates and the intercept of the least squares solution: The intercept is 4.99. X1 through X5 are, respectively: 25.01, 24.99, 24.99, 25.01, 25.

Iteratively updated least squares via biglm

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:

```
write.table(X[1:1000,],file='Xchunk1.txt',sep=',',row.names=F,col.names=names(Xdf))
write.table(X[1001:2000,],file='Xchunk2.txt',sep=',',row.names=F,col.names=names(Xdf))
write.table(X[2001:3000,],file='Xchunk3.txt',sep=',',row.names=F,col.names=names(Xdf))
write.table(X[3001:4000,],file='Xchunk4.txt',sep=',',row.names=F,col.names=names(Xdf))
write.table(Y[1:1000],file='Ychunk1.txt',sep=',',row.names=F,col.names=F)
write.table(Y[1001:2000],file='Ychunk2.txt',sep=',',row.names=F,col.names=F)
write.table(Y[2001:3000],file='Ychunk3.txt',sep=',',row.names=F,col.names=F)
write.table(Y[3001:4000],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.

```
# Chunk 1
Xchunk = read.table(file='Xchunk1.txt',sep=',',header=T)
Ychunk = scan(file='Ychunk1.txt',sep=',')
form = as.formula(paste('Ychunk ~ ',paste(names(Xchunk),collapse=' + '),collapse=''))
out.biglm = biglm(formula = form,data=Xchunk)
coef(out.biglm)[1:6]
```

```
## (Intercept)          X1          X2          X3          X4          X5
##    4.970058   25.087823   24.984586   24.948675   25.003253   24.931791
```

```
rm(Xchunk)

# Chunk 2
Xchunk = read.table(file='Xchunk2.txt',sep=',',header=T)
Ychunk = scan(file='Ychunk2.txt',sep=',')
out.biglm = update(out.biglm,moredata=Xchunk)
coef(out.biglm)[1:6]
```

```
## (Intercept)          X1          X2          X3          X4          X5
##    4.968055   25.014846   24.980627   24.985553   24.983485   24.963039
```

```
rm(Xchunk)

# Chunk 3
Xchunk = read.table(file='Xchunk3.txt',sep=',',header=T)
Ychunk = scan(file='Ychunk3.txt',sep=',')
out.biglm = update(out.biglm,moredata=Xchunk)
coef(out.biglm)[1:6]
```

```
## (Intercept)          X1          X2          X3          X4          X5
##    4.974551   24.999340   24.985498   24.979917   25.009175   24.985895
```

```
rm(Xchunk)

# Chunk 4
Xchunk = read.table(file='Xchunk4.txt',sep=',',header=T)
Ychunk = scan(file='Ychunk4.txt',sep=',')
out.biglm = update(out.biglm,moredata=Xchunk)
coef(out.biglm)[1:6] # changed from 5 to 6 to get 5 coefficients + intercept
```

```
## (Intercept)          X1          X2          X3          X4          X5
##    4.991328   25.009889   24.992888   24.988658   25.011266   24.995551
```

```
rm(Xchunk)
```

Compare the first 5 non-intercept entries in betaHat to the coefficient estimate . What is the advantage of computing the coefficient estimates via biglm versus lm?
The first 5 non-intercept entries are exactly the same. One advantage of using biglm is that lm may not run if the data set is simply too large to read in. Another advantage of biglm is that you can see the impact that each chunk of data has on the coefficient estimates. If there are any big changes in the coefficients, it's a signal to look closer into that data chunk.

Cross Validation and Parallelism

Problem 2.1. Simple example of 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 remove since more than 2 cores
cat('My work station has ',nCores,' cores \n')
```

```
## My work station has  24  cores
```

```
cl = makeCluster(nCores)
registerDoParallel(cl)

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

```
##    user  system elapsed
##    0.00    0.00   26.19
```

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

```
##    user  system elapsed
##    0.06    0.00    1.14
```

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) It appears that the system.time without parallelism is greater than with parallelism by a factor ~ # cores.

Cross-Validation

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

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

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

```
## CV estimate of the risk:  1.171231
```

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

```
## Standard error of CV estimate of the risk:  0.02930568
```

What could the standard error of the CV estimate be used for? It could be used as a measurement of the accuracy of the risk estimate and helpful for determining if the value of K chosen is appropriate in the bias-variance tradeoff framework.

Cross-validation in Parallel

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

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

Now, we want to extend the above parallel code to CV. If we use ‘dopar’, then we will be using the parallel backend we defined earlier. If we use ‘do’, this is just another way of writing a for loop in R and it won’t use the parallel backend (i.e. it will be doing sequential processing)

```
startTime        = proc.time()[3]
CVoutputParallel = foreach(k = 1:K) %dopar%{cvF(k, folds)}
endTime          = proc.time()[3]
parallelTime     = endTime-startTime
parallelCV       = mean(unlist(CVoutputParallel))


startTime          = proc.time()[3]
CVoutputSequential = foreach(k = 1:K) %do%{cvF(k, folds)}
endTime            = proc.time()[3]
seqTime            = endTime-startTime
seqCV              = mean(unlist(CVoutputSequential))
cat('Parallel CV estimate of the risk: ',parallelCV,'\n')
```

```
## Parallel CV estimate of the risk:  1.171231
```

```
cat('Parallel Run Time: ',parallelTime,'\n')
```

```
## Parallel Run Time:  1.67
```

```
cat('Sequential CV estimate of the risk: ',seqCV,'\n')
```

```
## Sequential CV estimate of the risk:  1.171231
```

```
cat('Sequential Run Time: ',seqTime,'\n')
```

```
## Sequential Run Time:  5.66
```

Compare the time for CV and the values of the K-fold CV estimator of the risk with and without parallelism. Similar to before, the parallel run time significantly reduces total run time, but the reduction factor is now only 3:1 and not 22:1, so running in parallel here is less efficient than in the simpler task in problem 2.1. The CV values are the same for both processes since they call the same function.

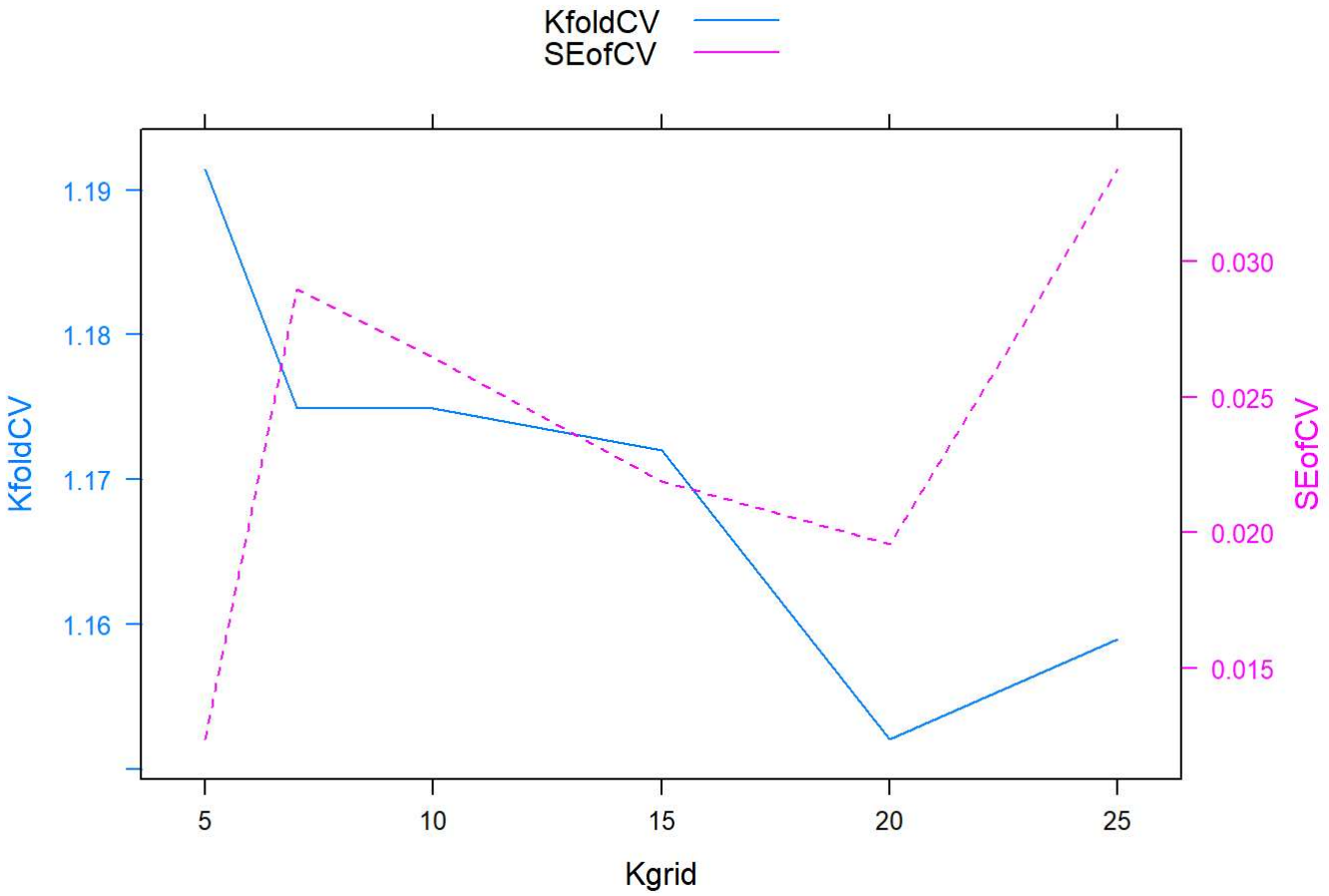
Let’s look at the CV estimate of the risk as a function of K.

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(5,7,10,15,20,25)
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, folds)})
  CVestimate[Kiter] = mean(CVestimateVec)
  CVse[Kiter]       = sd(CVestimateVec)/sqrt(K)
}


# Let's make a plot
df = data.frame('K' = Kgrid,
                'KfoldCV' = CVestimate,
                'SEofCV'  = CVse)
# This uses the latticeExtra package
plt1 = xyplot(KfoldCV ~ Kgrid, df, type = "l" , lty=1)
plt2 = xyplot(SEofCV ~ Kgrid, df, type = "l", lty=2)

doubleYScale(plt1, plt2, add.ylab2 = TRUE,
             text = names(df)[-1])
```



In general, what is the relationship between K & Risk? What aspects of this relationship are represented in this plot?
K and risk are (generally) inversely proportional. As K increases, the risk estimate decreases (for the most part, excluding k > 20). But as K increases, we generally are getting a larger standard error for the risk estimate, especially for values of K above 20. This is related to our lecture slides that use a heuristic of 5 to 10 for K, as the bias-variance tradeoff becomes too unbalanced with larger values of K but K=20 looks like a good K value for this specific problem.

Cross-validation in Caret

The caret (https://cran.r-project.org/web/packages/caret/vignettes/caret.pdf) package can do some commonly done statistical learning tasks for you. In particular, create the folds:

```
folds = createFolds(Y, k = K, list = TRUE)
```

What 'folds' object (this one or the previous one we defined earlier) more directly matches the definition of K-fold CV from the lectures? This object more directly matches the lectures. In prob 2.2, "folds" is a vector of 4000 entries (with numbers from 1 to 10 to represent which fold it belongs in). The object here is a list of the actual fold subsets of D.

It's best practice to stop the cluster when you're done with parallel computing and return to sequential computing

```
stopCluster(cl)
registerDoSEQ()
```

Forward Selection

Using the same simulated X and Y generated in the previous problem, use forward selection and K-fold CV to form an estimate betaHat. Let's get a training/test split so we can get different estimators of the risk.

Problem 3.1.

We can use caret for doing the data splitting:

```
trainingIndex = createDataPartition(Y, p = 0.5, list = FALSE)

Xdf = data.frame(X)

Xtrain = Xdf[trainingIndex,]
Xtest  = Xdf[-trainingIndex,]
Ytrain = Y[trainingIndex]
Ytest  = Y[-trainingIndex]
```

and the training:

```
K          = 10
trControl  = trainControl(method = 'cv', number = K)
tuneLength = 50
lmOut      = train(x = Xtrain, y = Ytrain,
                   method = "leapForward", tuneLength = tuneLength,
                   trControl = trControl)
```

What does 'tuneLength' do in this function in terms of forward selection (be specific)? Tunelength limits the maximum number of features to be included in model selection. Here we limit models to a maximum of 50 features.

Getting the risk estimate minimizing solution

Using BIC (which makes sense as we are interested in selecting the correct model), select the BIC minimizing solution

```
numSelectedCoefs = lmOut$bestTune$nvmax

betaHatForward   = coef(lmOut$finalModel, numSelectedCoefs)
betaHatForward[1:6]
```

```
## (Intercept)          X1          X2          X3          X4          X5
##    4.990367   25.036448   24.991004   24.975516   24.994184   25.025135
```

```
lmOut$bestTune # number of features in model, of p=500 possible, not including an intercept
```

```
##   nvmax
## 5     6
```

```
sum(betaHat==0)
```

```
## [1] 0
```

```
sum(betaHat<=abs(.0001))
```

```
## [1] 278
```

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? The coefficients from the lm model are slightly closer to the true value of 25 (except for X4). But the coefficients from the forward selection model really aren't that far off and it is a much simpler model.
The forward selection model only uses 5 features and an intercept term, so 495 coefficients have a zero value, and it has 5 nonzero entries. For the lm model, it technically doesn't have any 0 values, and uses all 500 features and the intercept. However, using a tolerance of .0001 to account for rounding in R, we could say that the lm model has 278 coefficients that are "close enough" to zero.

Comparing risk estimates

Compare the training error, K-fold CV, and test error for both the least squares solution and the forward selection solution. Look into train (https://topepo.github.io/caret/model-training-and-tuning.html) for more information about using ‘train’ and its output. In particular, the output of train has an object ‘resample’ which contains the square root of the loss evaluated on each fold, which it calls RMSE for ‘ root mean square error’.

```
lmOutLeastSq = train(x = Xtrain, y = Ytrain,
                     method = "lm", trControl = trControl)
YhatLS       = predict(lmOutLeastSq, Xtrain)
YhatTestLS   = predict(lmOutLeastSq, Xtest)

leastSquaresResults = data.frame('trainingError' = mean( (YhatLS - Ytrain)**2 ),
                                 'KfoldCV'       = mean( lmOutLeastSq$resample$RMSE**2 ),
                                 'testError'     = mean((YhatTestLS-Ytest)^2))

YhatForward      = predict(lmOut, Xtrain)
YhatTestForward  = predict(lmOut, Xtest)

forwardResults      = data.frame('trainingError' = mean( (YhatForward - Ytrain)**2 ),
                                 'KfoldCV'       = mean( lmOut$resample$RMSE**2 ),
                                 'testError'     = mean((YhatTestForward-Ytest)^2))

results = rbind(leastSquaresResults, forwardResults)
rownames(results) = c("lm","forward selection")
kable(results, caption = 'Comparison of least squares and forward selection')
```

Comparison of least squares and forward selection

	trainingError	KfoldCV	testError
lm	0.7789117	1.430215	1.270611
forward selection	0.9977138	1.008556	1.030358

Discuss this table. The training error is lower for the lm model, but the KfoldCV and test error is higher than it is in the forward selection model. This highlights how training error is optimistic and how we can correct this optimism by using data splitting, cross-validation and model selection procedures.