Micro Array data and penalized regression

Matt Richey

03/18/2019

```r
library(glmnet)
```

```
## Loading required package: Matrix
```

```
## Loaded glmnet 3.0-2
```

```r
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------------------------------- tidyverse 1.3.0
```

```
## v ggplot2 3.3.0     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.5
## v tidyr   1.0.2     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.5.0
```

```
## -- Conflicts ------------------------------------------------------------------ tidyverse_conflicts()
## x tidyr::expand() masks Matrix::expand()
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## x tidyr::pack()   masks Matrix::pack()
## x tidyr::unpack() masks Matrix::unpack()
```

Introduction

This data set uses microarray data for 72 individuals some of whom had leukimia. The response variable is discrete, hence this is a classification problem. In this case, the response variables indicate whether the individual had ALL (acute lymphocytic leukemia, 47 cases) or AML (acute myelogenous leukemia, 25 cases).

Fortunately, GLMNET does classification as well as regression. For classification, you need to include the argument **family="binomial"** as part of the glmmnet function call.

Load the data and inspect dimensions. Note that in this case the data are stored in a "*.RData" file. This is R's form of saving data. The data will be loaded into the variable "Leukemia".

```r
load("Leukemia.RData")
```

Look at the structure of Leukemia. We have n=72 samples and p=3571 predictors (!)

```
str(Leukemia)
```

```
## List of 2
##  $ x: num [1:72, 1:3571] 0.562 -0.623 -0.815 0.229 -0.706 ...
##   ..- attr(*, "scaled:center")= num [1:3571] -1.048 -0.867 -1.121 -0.875 0.208 ...
##   ..- attr(*, "scaled:scale")= num [1:3571] 0.462 0.514 0.461 0.451 0.487 ...
##  $ y: num [1:72] 0 0 0 0 0 0 0 0 0 0 ...
```

Pull off the x and y values.

```
xVals<-Leukemia$x
yVals<-Leukemia$y
```

xVals is a very wide matrix

```
dim(xVals)
```

```
## [1]   72 3571
```

72 rows and 3571 columns.

```
numObserve <- nrow(xVals)
```

The yVals are the binary responses. How are these distributed?

```
table(yVals)
```

```
## yVals
##  0  1
## 47 25
```

The goal is to see if we can predict yVals from the (very wide) xVals.

Model building

As usual, create train and test data sets

```
train.vals <- sample(1:numObserve,numObserve/2,rep=F)
train.x <- xVals[train.vals,]
train.y <- yVals[train.vals]
```

Table the training data

```
table(train.y)
```

```
## train.y
##  0  1
## 23 13
```

Make sure this looks ok.

```
dim(train.x)
```

```
## [1]   36 3571
```

Now build the test data the same way

```
test.x <- xVals[-train.vals,]
test.y <- yVals[-train.vals]
```

Step One: Try Logistic regression.

There is an obvious problem: Too many predictor variables approximately 3700!

Note: Traditional subset selection methods are out of the question.

What the heck. . . .Give logistic regression a shot

```
train.df <- data.frame(train.x,y=train.y)
test.df <- data.frame(test.x,y=test.y)
## names(train.df) ## take a peek if you want.
mod.log <- glm(y ~ .,
               data=train.df,
               family="binomial")
```

A summary of the model demonstrates that logistic regression simply is using the first 35 or so predictors.

```
## summary(mod.log) ## take a peek if you want.
```

How well does this work? You will encounter some problems but you can still attempt a classification.

```
pred.log <- predict(mod.log,
                    newdata=test.df,
                    type="response")
```

```
## Warning in predict.lm(object, newdata, se.fit, scale = 1, type = if (type == :
## prediction from a rank-deficient fit may be misleading
```

A "rank-deficient matrix" means the columns of the input matrix were not linearly independent. We knew this since the matrix was 36 x 3571. Hence it has 3571 columns in R^{36}!

```
dim(test.x)
```

```
## [1]   36 3571
```

What to do. . . ..

Step Two: KNN

Will KNN works any better? Give it a shot as a point of comparison.

In this case, it would be helpful to scale the inputs.

```
library(class) ##  for knn
```

```
train.x.sc <- scale(train.x)
test.x.sc <- scale(test.x)
```

Get started with particular value of k

```
kVal <- 10
knn.mod <- knn(train.x.sc,test.x.sc,train.y,k=kVal)
table(test.y,knn.mod)
```

```
##       knn.mod
## test.y  0  1
##      0 24  0
##      1  5  7
```

```
mean(test.y != knn.mod)
```

```
## [1] 0.1388889
```

So that seems to work.

Cross-validation of KNN

Let's use 5-fold cross-validation on the full data set to indentify a candidate for an optimal value of k.

We'll use 5-fold (vs 10-fold) since there are only 72 observations.

```
numFolds <- 5
folds <- sample(1:numFolds,numObserve,rep=T)
## Scale the original data
xVals.sc <- scale(xVals)
```

```
kVal <- 10
errs <- numeric(numFolds)
fold <- 1
for(fold in 1:numFolds){
  train.x <- xVals.sc[folds != fold,]
  test.x <- xVals.sc[folds = fold,]
  train.y <- yVals[folds != fold]
  test.y <- yVals[folds == fold]
  knn.mod <- knn(train.x,test.x,train.y,k=kVal)
  errs[fold] <- mean(test.y !=knn.mod)
}
errs
```

```
## [1] 0.2222222 0.3333333 0.1250000 0.5333333 0.5000000
```

```
mean(errs)
```

```
## [1] 0.3427778
```

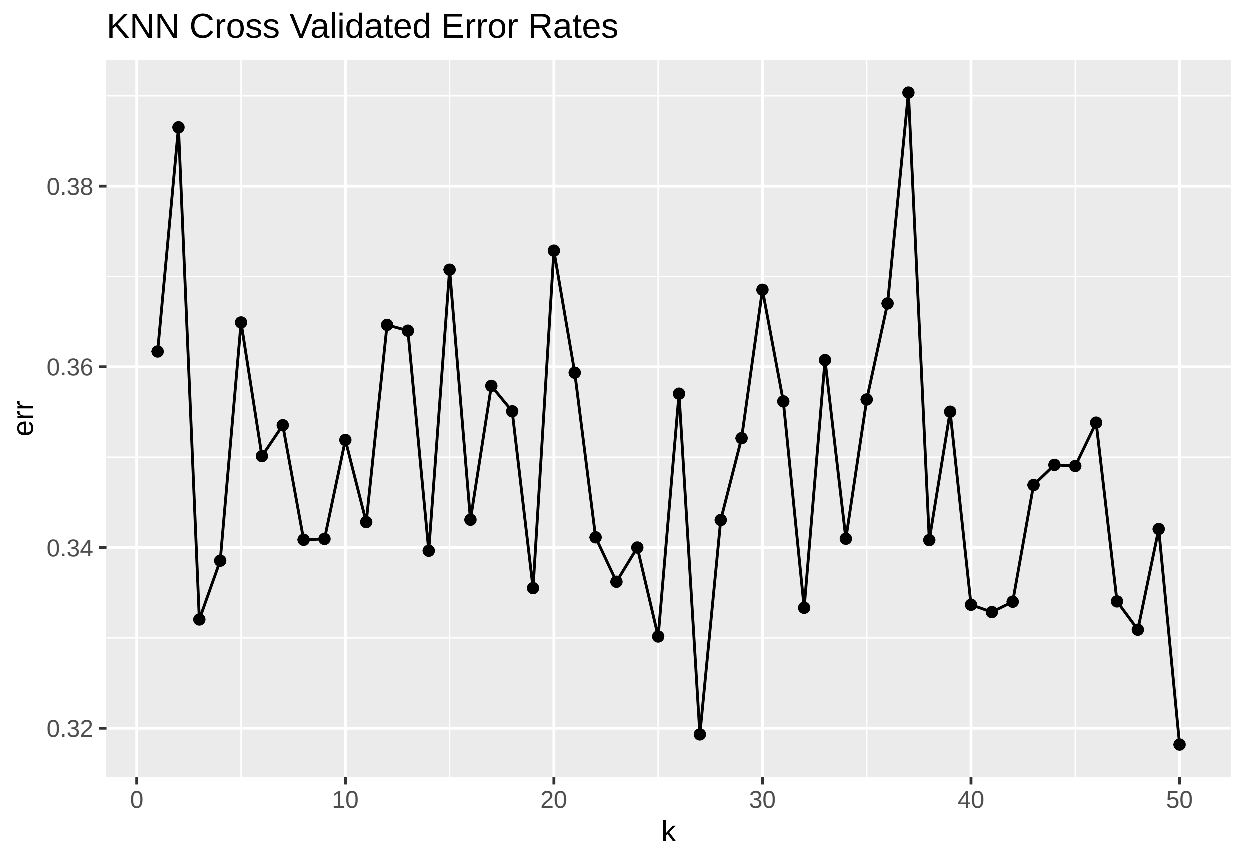
Now make this step a function and repeat for different kVals

```r
cv.knn <- function(kVal,numFolds=5){
  folds <- sample(1:numFolds,numObserve,rep=T)
  errs <- numeric(numFolds)
  for(fold in 1:numFolds){
    train.x <- xVals.sc[folds != fold,]
    test.x <- xVals.sc[folds = fold,]
    train.y <- yVals[folds != fold]
    test.y <- yVals[folds == fold]
    knn.mod <- knn(train.x,test.x,train.y,k=kVal)
    errs[fold] <- mean(test.y !=knn.mod)
  }
  mean(errs)
}
cv.knn(50)
```

```
## [1] 0.3459951
```

```r
maxK <- 50
errs.knn <- map_dbl(1:maxK,cv.knn)

data.frame(k=1:maxK,err=errs.knn) %>%
  ggplot()+
  geom_point(aes(k,err))+
    geom_line(aes(k,err))+
  labs(title="KNN Cross Validated Error Rates")
```



Not much to say here. . . .As it turns out, KNN is not very useful with "wide" data sets.

The problem lies with the so-called "curse of dimensionality." The key point is that in high-dimensional space, nothing is really close together. Hence the KNN method, which depends on "nearest", breaks down.

Oh well. . .

Step Three: Ridge Regression

Explore this over-determined model with ridge regression. Penalized methods were devised as a way of working with really wide data sets.

Plan: Use glmnet to perform classification (glmnet has the **family="binomial"** option). Then use cross-validation on the training to estimate error rates. Finally, get the optimal lambda value and make a prediction on testing data.

Of course, we'll let glmnet do its own cross-validation.

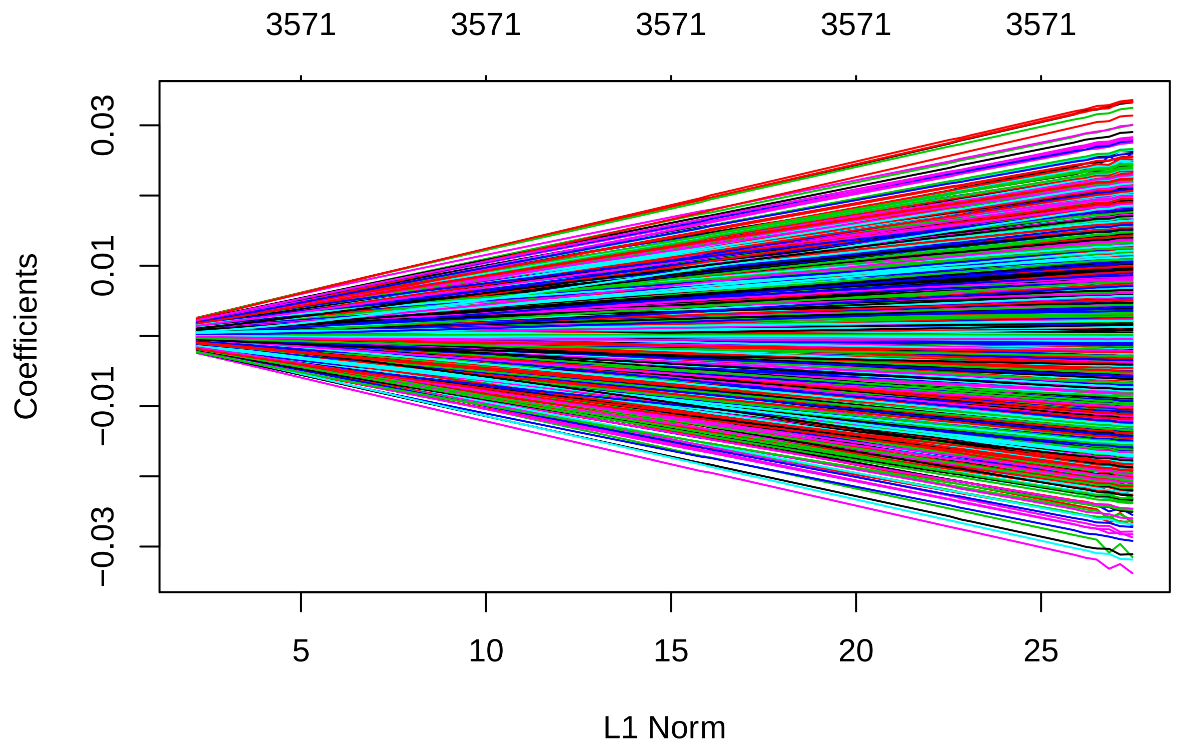
A couple points to keep in mind.
* Use alpha=0 for ridge regression. alpha=1 means lasso, which we'll talk about later * Like KNN, penalized methods are much happier with scaled input data. This is because the goal is to "penalize" large coefficients. If the data are unscaled, the magnitudes of the coefficients are on different scales. This means the penalization process isn't equitable. * There is no a priori notion of a good lambda value. Hence it is best to use a geometric sequence of value (versus an arithmetic) so that you can cover more ground!

Build a geometrically spaced lambda grid. This might take some experimentation to get a good range of values.

```
lambda.grid=10^seq(-2,2,length=100)
```

Look at the model over the lambda grid. This plot isn't particularly revealing.

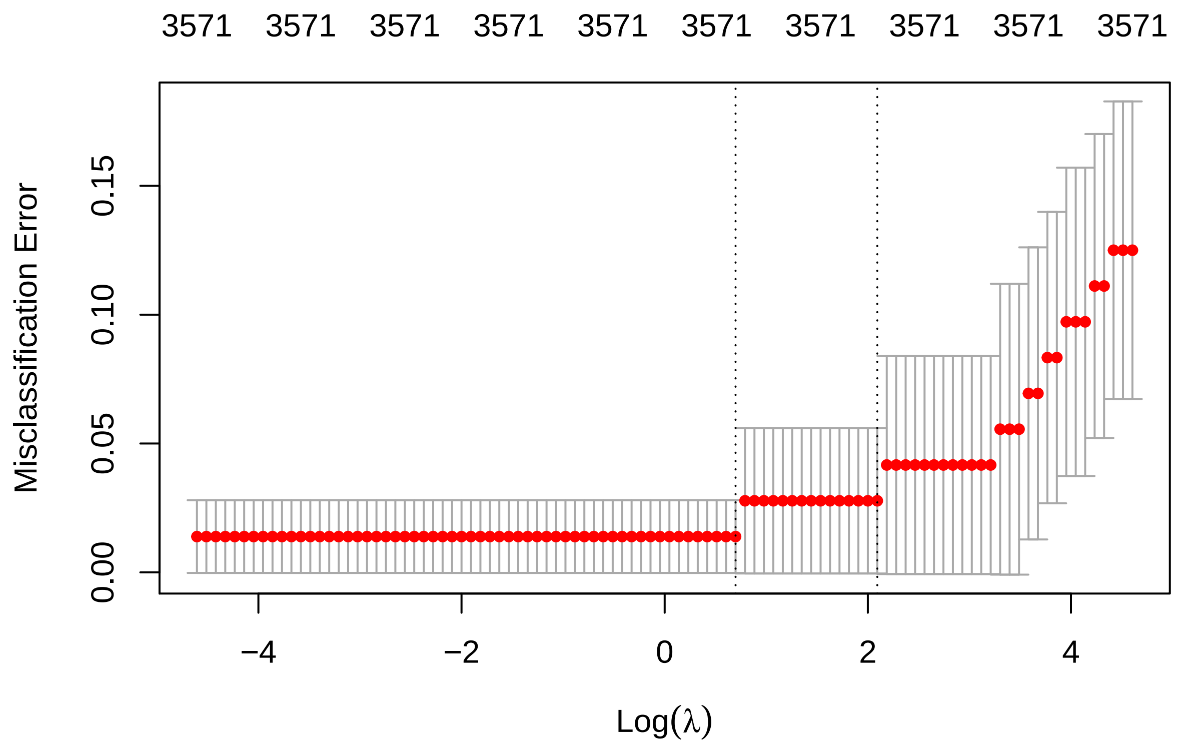
```
mod.ridge <- glmnet(xVals.sc, ## use  scaled
                    yVals,
                    lambda=lambda.grid,
                    family="binomial",
                    type.measure="class",
                    alpha=0)
plot(mod.ridge)
```



Now use cv.glmnet to cross-validate across the lambda.grid.

```
mod.cv.ridge <- cv.glmnet(xVals.sc, ## use  scaled
                          yVals,
                          lambda=lambda.grid,
                          family="binomial",  ## classification
                          type.measure="class", ## predict the class
                          alpha=0) ##for ridge
```

```
plot(mod.cv.ridge)
```



Extract the optimal lambda value and evaluate.

```
lambda.opt <- mod.cv.ridge$lambda.min
mod.ridge <- glmnet(xVals.sc, ## use  scaled
                    yVals,
                    lambda=lambda.opt,
                    family="binomial",
                    type.measure="class",
                    alpha=0)
```

```
pred <- predict(mod.ridge,newx=xVals.sc,type="response")
pred
```

```
##                s0
##  [1,] 0.009002018
##  [2,] 0.048395160
##  [3,] 0.014314241
##  [4,] 0.013054447
##  [5,] 0.011421708
##  [6,] 0.021213151
##  [7,] 0.044465263
##  [8,] 0.031223196
##  [9,] 0.016597657
## [10,] 0.040700478
## [11,] 0.016425887
## [12,] 0.085315496
## [13,] 0.001910438
## [14,] 0.027218540
## [15,] 0.001849425
## [16,] 0.006360119
## [17,] 0.046074169
## [18,] 0.045268344
## [19,] 0.014673992
## [20,] 0.001340273
## [21,] 0.005970847
## [22,] 0.032996564
## [23,] 0.018748614
## [24,] 0.007606940
## [25,] 0.049723066
## [26,] 0.014784069
## [27,] 0.030452673
## [28,] 0.923446423
## [29,] 0.946931652
## [30,] 0.982734601
## [31,] 0.943816322
## [32,] 0.898117109
## [33,] 0.978306800
## [34,] 0.965654900
## [35,] 0.920988136
## [36,] 0.985962834
## [37,] 0.984784127
## [38,] 0.943474009
## [39,] 0.023860700
## [40,] 0.061539360
## [41,] 0.006698080
## [42,] 0.043019102
## [43,] 0.036583210
## [44,] 0.004640374
## [45,] 0.013269322
## [46,] 0.011824339
## [47,] 0.041245200
## [48,] 0.004152148
## [49,] 0.033642864
## [50,] 0.976188331
## [51,] 0.977812003
## [52,] 0.981633691
## [53,] 0.955031662
## [54,] 0.926307391
## [55,] 0.077690552
## [56,] 0.030408694
## [57,] 0.956924405
## [58,] 0.964398713
## [59,] 0.029118594
## [60,] 0.939352567
## [61,] 0.911907389
## [62,] 0.960344243
## [63,] 0.967834673
## [64,] 0.955542555
## [65,] 0.939015361
## [66,] 0.842930834
## [67,] 0.095586316
## [68,] 0.006755801
## [69,] 0.009316482
## [70,] 0.029998775
## [71,] 0.031495316
## [72,] 0.022605776
```

```
table(yVals,pred > 0.5)
```

```
##
## yVals FALSE TRUE
##     0    47    0
##     1     0   25
```

Pretty impressive. Granted this was build and evaluated on the same data set, but to get all but one observation correctly classified is pretty impressive.

One downside of Ridge regression is this setting is that we still have a very large predictor set. Lasso is an effective way of performing subset selection on wide data sets such as this.

Step Four: Repeat with Lasso.

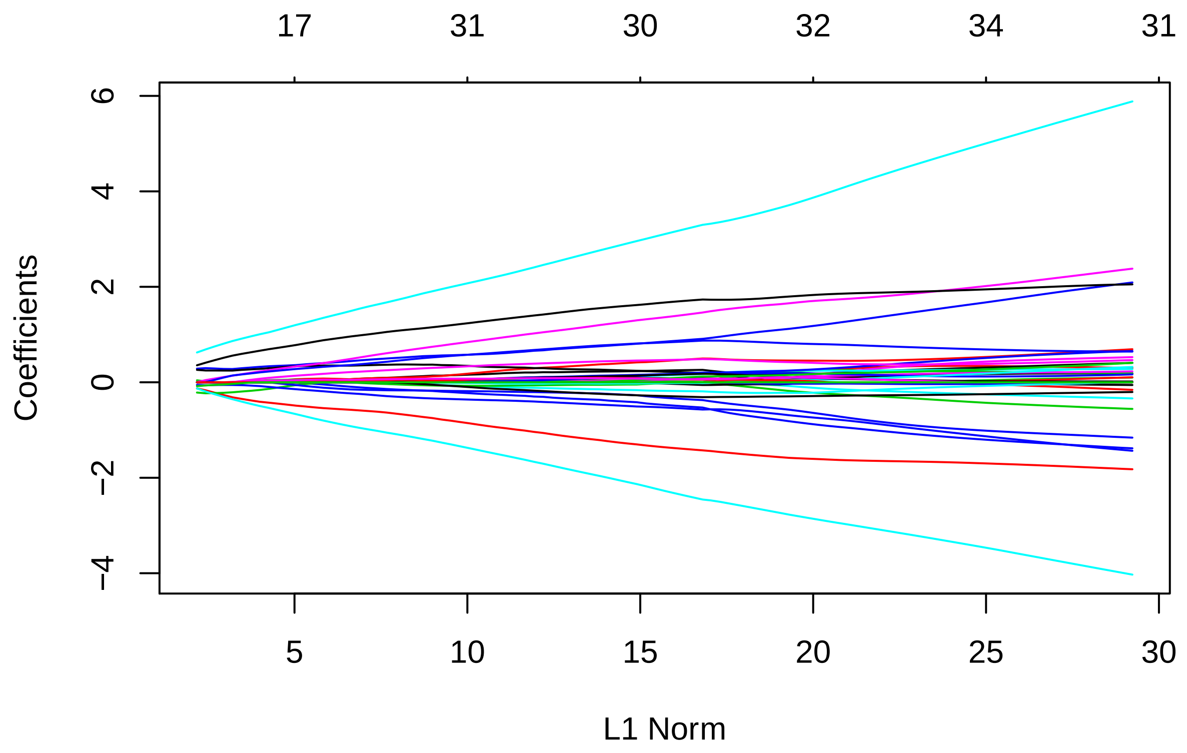
Here we use alpha=1. Everything is similar How many coefficents are selected (i.e., how many non-zero coefficients)? Build a geometrically spaced lambda grid.

Try the same one (as ridge) first.

```
lambda.grid=10^seq(-8,-1,length=100)
```

Look at the model over the lambda grid. This plot isn't particularly revealing.

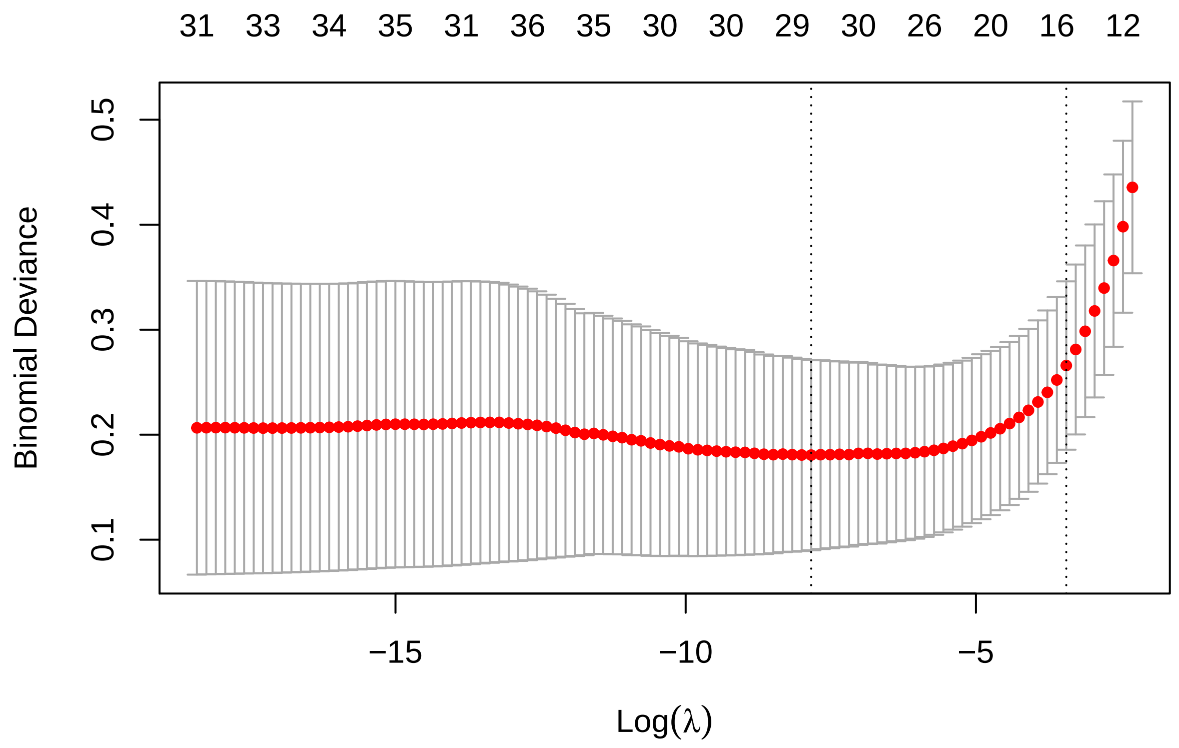
```
mod.lasso <- glmnet(xVals.sc, ## use  scaled
                    yVals,
                    lambda=lambda.grid,
                    family="binomial",
                    type.measure="class",
                    alpha=1)  ## For  Lasso
plot(mod.lasso)
```



Now use cv.glmnet to cross-validate across the lambda.grid.

```
mod.cv.lasso <- cv.glmnet(xVals.sc, ## use  scaled
                          yVals,
                          lambda=lambda.grid,
                          family="binomial",  ## classification
                          ##type.measure="class", ## predict the class
                          alpha=1) ##for lasso
```

```
plot(mod.cv.lasso)
```



Extract the optimal lambda value and evaluate.

```
(lambda.opt <- mod.cv.lasso$lambda.min)
```

```
## [1] 0.0003944206
```

```
log(lambda.opt)
```

```
## [1] -7.838093
```

```
mod.lasso <- glmnet(xVals.sc, ## use  scaled
                    yVals,
                    lambda=lambda.opt,
                    family="binomial",
                    #type.measure="class",
                    alpha=1)
```

```
prob.lasso <- predict(mod.lasso,
                      newx=xVals.sc,
                      family="binomial",
                      type="response")

pred.lasso <- prob.lasso > 0.5
table(yVals,pred.lasso > 0.5)
```

```
##
## yVals FALSE TRUE
##     0    47    0
##     1     0   25
```

The perfect fit is important. In this case, it is equally important to identify a smaller set of predictor variables.

Let's extract the identities of the non-zero coeffients.

It's no as easy as it could be, but this works

```
coefs.lasso <- coef(mod.lasso)
(nonZeroIndices <- coefs.lasso@i)  ##
```

```
##  [1]    0  219  456  657  672  888  918  926  956  979 1007 1219 1569 1652 1835
## [16] 1946 2230 2239 2481 2727 2831 2859 2888 2929 3038 3098 3125 3158 3181 3292
```

This tells us that the intercept along with indices 219, 456,. . . 3181 form a predictor set. Quite the reduction from the original set of predictors.

Back to Logistic Regression.

Now we can repeat logistic regression (if we want) on the reduced variable set.

```
xVals.red <- xVals[,nonZeroIndices]
data.red.df <- data.frame(y=yVals,xVals.red)
mod.log.red <- glm(y ~., data=data.red.df,family="binomial")
```

```
## Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

The warning indicates a perfect fit.

Finish it off.

```
prob <- predict(mod.log.red,type="response")
pred <- prob > 0.5
table(yVals,pred)
```

```
##      pred
## yVals FALSE TRUE
##     0    47    0
##     1     0   25
```

It works great!

Summary

This example indicates the power of using penalized regression with extremely wide data sets. In this case p>3000 while N<100. In particular, Lasso can work extemely well as a means not only of prediction, but as a way of reducing the size of the predictor set.