Machine Learning for Finance Applications - Elastic Net (Ridge and LASSO) Regression

Brian Clark

2020

Contents

1 Introduction

This vignette introduces Ridge and LASSO regression methods for finance applications. The topics closely follow Chapter 6 of James et al. (2013).[1]

LASSO and Ridge regressions are implementations of linear regression with penalty functions that penalize the number of parameters. They are part of a class of shrinkage methods used to reduce the dimensions of the data. They can be adapted to regression or classificaiton problems (e.g., for logistic regressions).

2 Ridge Regression

Both ridge and lasso regression are a variant of linear regression with a penalty function. Recall that linear regression minimizes the residual sum of squares (RSS):

$$RSS = \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 . \tag{1}$$

Ridge regression adds a penalty function so we have a loss plus penalty equation. The goal is to minimize the cost function:

$$\min_{\beta_0, \beta_1, ..., \beta_p} \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 + \lambda \sum_{j=1}^{p} \beta_j^2. \tag{2}$$

The last term is simply the l_2 norm (or the square the way it is written) which acts as the shrinkage penalty. The parameter λ is the tuning parameter which determines the influence of the penalty. Large values of λ reduce variance but increase bias by reducing the size of the $\beta's$. Small values do the opposite. λ can be chosen via cross-validation. Note that the penalty term is sometimes written as the square of the l_2 norm: $||\beta||_2^2$, where β is the vector of $[\beta_1, \beta_2, ..., \beta_p]$ and $||\beta||_2 = \sqrt{\beta_1^2 + \beta_2^2 + ... + \beta_p^2}$. Moreover, a scaling factor of $\frac{1}{2}$ is sometimes written but it effectively inluded in λ in the above notation.

One practical note is that like methods such as k-NN, the scale of the features matters. Therefore, you should standardize the variables before running the model. One such approach is to use the following equation (see page 217 of James et al. (2013)):

$$\hat{x_{ij}} = \frac{x_{ij}}{\sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_{ij} - \bar{x_j})^2}} \tag{3}$$

3 Lasso Regression

One drawback to ridge regression is that the coefficients will never equal zero. As such, they have less influence and overfitting can be mitigated. However, it may still be difficult to interpret the results if there are a lot of predictors.

The Lasso regression is essentially the same as the ridge except it has a different penalty function. In particular it replaces the l_2 norm with the l_1 norm:

$$\min_{\beta_0, \beta_1, ..., \beta_p} \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 + \lambda \sum_{j=1}^{p} |\beta_j|. \tag{4}$$

[1] The full text is available at http://www-bcf.usc.edu/ gareth/ISL/.

Both models can be written as constrained optimization problems as shown in Section 6.2.2 of James et al. (2013).

Lasso:

$$\min_{\beta} \left[\sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right) \right] s.t. \sum_{j=1}^{p} |\beta_j| \leq s \tag{5}$$

Ridge:

$$\min_{\beta} \left[\sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right) \right] s.t. \sum_{j=1}^{p} \beta_j^2 \leq s \tag{6}$$

4 Elastic Net Regression

The elastic net regression is a generalization of the above Lasso and Ridge regressions. Elastic net regression introduces an additional parameter, α, that defines the weights on the Lasso and Ridge penalties. More formally, the elastic net regression minimizes the following cost function:[2]

$$\min_{\beta_0, \beta_1, \ldots, \beta_p} \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 + \lambda \left[(1-\alpha) \sum_{j=1}^{p} \beta_j^2 + \alpha \sum_{j=1}^{p} |\beta_j| \right]. \tag{7}$$

5 Implementaiton in R

The following examples are from the Chapter 6 lab in James et al. (2013). We will use the `glmnet` package which is described **here**. `alpha` is the tuning parameter that ranges from 0 to 1. `alpha = 0` is the ridge and `alpha=1` is the lasso. Values inbetween are essentially a blend of the two and are known as the elastic net model.

Note that the objective (cost) function in the `glmnet` package documentation is written slightly different than in the above sections. In particular, it is written as follows:

$$\min_{(\beta_0, \beta) \in \mathbb{R}^{p+1}} \frac{1}{N} \sum_{i=1}^{N} w_i l \left(y_i, \beta_0 - x_i^T \beta \right) + \lambda \left[(1-\alpha) \frac{||\beta||_2^2}{2} + \alpha ||\beta||_1 \right], \tag{8}$$

where $l(y_i, \beta_0 - x_i^T \beta)$ is the loss function and β refers to the vector of coefficients, excluding the constant term β_0. β_0 is separated because we do not want to shrink the constant paramter in the model.

Note that the above difference in the way the regressions are defined in the `glmnet` package versus in James et al. (2013) highlights an important practical point implementing pre-packaged software packages. The definitions of variables often varies across packages. For example, λ in one package may be different than λ in another package. This is especially important when using parameters that have no "physical" meaning such as the tuning or hyper-parameters such as λ and α in the `glmnet` package.

For a linear regression (i.e., the Gaussian case), the loss function is the residual sum of squares:

$$l(y_i, \beta_0 - x_i^T \beta) = \frac{1}{2} \left(y_i - \beta_0 - x_i^T \beta \right)^2, \tag{9}$$

[2]Note that the examples herein are for the case of Gaussian regression. The cost functions for classification problems such as a logistic regression framework are slightly different but very similar in structure.

which makes the objective function:

$$\min_{(\beta_0,\beta)\in\mathbb{R}^{p+1}} \frac{1}{2N}\sum_{i=1}^{N} w_i \left(y_i - \beta_0 - x_i^T\beta\right)^2 + \lambda\left[(1-\alpha)\frac{||\beta||_2^2}{2} + \alpha||\beta||_1\right]. \tag{10}$$

Note that w_i is an optional weighting parameter that can be used to control the weight on certain observations akin to a weighted least squares model.

For a logistic regression, `glmnet` implements a negative binomial log-likelihood function that is written as follows:

$$\min_{(\beta_0,\beta)\in\mathbb{R}^{p+1}} -\left[\frac{1}{N}\sum_{i=1}^{N} y_i\left(\beta_0 + x_i^T\beta\right) - log\left(1 + e^{(\beta_0 + x_i^T\beta)}\right)\right] + \lambda\left[(1-\alpha)\frac{||\beta||_2^2}{2} + \alpha||\beta||_1\right]. \tag{11}$$

Additional functional forms included within the `glmnet` package include multinomoal logistic models, Poisson models, and Cox proportional hazard models.

5.1 Example using Baseball Data in R

For the first example, we will consider a sample of baseball data described in James et al. (2013). The dependent variable is a continuous variable contining the baseball players' salaries. Note there are 20 potential features in the data. Start by loading in and describing the data.

5.1.1 Step 1: Prepare and Describe the Data

```
rm(list=ls())
library("glmnet")
```

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

```
## Loading required package: foreach
```

```
## Loaded glmnet 2.0-18
```

```
library("ISLR")
```

```
## Warning: package 'ISLR' was built under R version 3.6.2
```

```
# Use some baseball data:
# fix(Hitters)
Hitters <- na.omit(Hitters)
dim(Hitters)
```

```
## [1] 263  20
```

```
names(Hitters)
```

```
##  [1] "AtBat"     "Hits"      "HmRun"     "Runs"      "RBI"
##  [6] "Walks"     "Years"     "CAtBat"    "CHits"     "CHmRun"
## [11] "CRuns"     "CRBI"      "CWalks"    "League"    "Division"
## [16] "PutOuts"   "Assists"   "Errors"    "Salary"    "NewLeague"
```

```
# Produce a table of basic summary stats
Hitters.numeric <- Filter(is.numeric,Hitters)
my.stats <- as.data.frame(matrix(NA,ncol=8,nrow=length(names(Hitters.numeric))))
names(my.stats) <- c("N","Mean","Min","p.25","p.50","p.75","Max","SD")
my.stats[,1] <- apply(Hitters.numeric,2,function(x) length(is.na(x)))
```

```
my.stats[,2] <- apply(Hitters.numeric,2,mean,na.rm=T)
my.stats[,3] <- apply(Hitters.numeric,2,min,na.rm=T)
v1 <- c(0.25,0.50,0.75)
my.stats[,c(4:6)] <- t(sapply(Hitters.numeric, function(x) quantile(x,probs = v1)))
my.stats[,7] <- apply(Hitters.numeric,2,max,na.rm=T)
my.stats[,8] <- apply(Hitters.numeric,2,sd,na.rm=T)
rownames(my.stats) <- names(Hitters.numeric)
```

Print out a nice version of the table based on the **knitr** package.

```
library("knitr")
kable(my.stats,digits=c(0,1,1,1,1,1,1,1,1),format.args = list(big.mark=","))
```

	N	Mean	Min	p.25	p.50	p.75	Max	SD
AtBat	263	403.6	19.0	282.5	413	526.0	687	147.3
Hits	263	107.8	1.0	71.5	103	141.5	238	45.1
HmRun	263	11.6	0.0	5.0	9	18.0	40	8.8
Runs	263	54.7	0.0	33.5	52	73.0	130	25.5
RBI	263	51.5	0.0	30.0	47	71.0	121	25.9
Walks	263	41.1	0.0	23.0	37	57.0	105	21.7
Years	263	7.3	1.0	4.0	6	10.0	24	4.8
CAtBat	263	2,657.5	19.0	842.5	1,931	3,890.5	14,053	2,286.6
CHits	263	722.2	4.0	212.0	516	1,054.0	4,256	648.2
CHmRun	263	69.2	0.0	15.0	40	92.5	548	82.2
CRuns	263	361.2	2.0	105.5	250	497.5	2,165	331.2
CRBI	263	330.4	3.0	95.0	230	424.5	1,659	323.4
CWalks	263	260.3	1.0	71.0	174	328.5	1,566	264.1
PutOuts	263	290.7	0.0	113.5	224	322.5	1,377	279.9
Assists	263	118.8	0.0	8.0	45	192.0	492	145.1
Errors	263	8.6	0.0	3.0	7	13.0	32	6.6
Salary	263	535.9	67.5	190.0	425	750.0	2,460	451.1

The above table shows the descriptive statistics for the full data. As discussed above, we want to standardize the data prior to running our model. Note that we will ignore categorical variables for the following analysis - although they could easily be included as a series of dummy or categorical variables.

Let's use Eq. (3) to standardize the data. That is, we will create a new dataset where all the variables are standardized to have unit variance.

```
# Generate a matrix of standard deviations for each variable:
std.vars <- matrix(rep(my.stats[,8],nrow(Hitters.numeric)),ncol=ncol(Hitters.numeric),byrow = T)

# Standardize the data:
Hitters.numeric.std <- Hitters.numeric / std.vars

# Make a new summary stats table:
my.stats.std <- as.data.frame(matrix(NA,ncol=8,nrow=length(names(Hitters.numeric.std))))
names(my.stats.std) <- c("N","Mean","Min","p.25","p.50","p.75","Max","SD")
my.stats.std[,1] <- apply(Hitters.numeric.std,2,function(x) length(is.na(x)))
my.stats.std[,2] <- apply(Hitters.numeric.std,2,mean,na.rm=T)
my.stats.std[,3] <- apply(Hitters.numeric.std,2,min,na.rm=T)
v1 <- c(0.25,0.50,0.75)
my.stats.std[,c(4:6)] <- t(sapply(Hitters.numeric.std, function(x) quantile(x,probs = v1)))
my.stats.std[,7] <- apply(Hitters.numeric.std,2,max,na.rm=T)
```

```
my.stats.std[,8] <- apply(Hitters.numeric.std,2,sd,na.rm=T)
rownames(my.stats.std) <- names(Hitters.numeric.std)

# Print the table:
kable(my.stats.std,digits=c(0,3,3,3,3,3,3,3,3),format.args = list(big.mark=","))
```

	N	Mean	Min	p.25	p.50	p.75	Max	SD
AtBat	263	2.740	0.129	1.918	2.804	3.571	4.664	1
Hits	263	2.390	0.022	1.584	2.283	3.136	5.274	1
HmRun	263	1.327	0.000	0.571	1.028	2.055	4.568	1
Runs	263	2.144	0.000	1.312	2.036	2.858	5.090	1
RBI	263	1.989	0.000	1.159	1.816	2.743	4.675	1
Walks	263	1.893	0.000	1.059	1.704	2.625	4.835	1
Years	263	1.525	0.209	0.834	1.252	2.086	5.007	1
CAtBat	263	1.162	0.008	0.368	0.844	1.701	6.146	1
CHits	263	1.114	0.006	0.327	0.796	1.626	6.566	1
CHmRun	263	0.842	0.000	0.182	0.487	1.125	6.667	1
CRuns	263	1.091	0.006	0.319	0.755	1.502	6.537	1
CRBI	263	1.022	0.009	0.294	0.711	1.313	5.130	1
CWalks	263	0.986	0.004	0.269	0.659	1.244	5.931	1
PutOuts	263	1.038	0.000	0.405	0.800	1.152	4.919	1
Assists	263	0.819	0.000	0.055	0.310	1.323	3.391	1
Errors	263	1.301	0.000	0.454	1.060	1.968	4.844	1
Salary	263	1.188	0.150	0.421	0.942	1.663	5.453	1

The `glmnet` package will standardize the data as we just did by default. However you may turn off the option to do so.

The next step is to prepare the data to run the models. Let's test the model to see what the results look like. For now, we will simply run a Ridge regression on the full dataset (thus ignoring in-sampe versis out-of-sample considerations).

Define the independent variables as everything by Salary. Note that the [,-1] means to use all rows (the blank space before the comma) and all but the last column (-1). Define the dependent variable as Salary. Note that we will run the models both with and without standardized data to compare the differences.

```
x     <- model.matrix(Salary~.,Hitters.numeric)[,-1]
x.std <- model.matrix(Salary~.,Hitters.numeric.std)[,-1]
head(x)
```

```
##                   AtBat Hits HmRun Runs RBI Walks Years CAtBat CHits
## -Alan Ashby         315   81     7   24  38    39    14   3449   835
## -Alvin Davis        479  130    18   66  72    76     3   1624   457
## -Andre Dawson       496  141    20   65  78    37    11   5628  1575
## -Andres Galarraga   321   87    10   39  42    30     2    396   101
## -Alfredo Griffin    594  169     4   74  51    35    11   4408  1133
## -Al Newman          185   37     1   23   8    21     2    214    42
##                   CHmRun CRuns CRBI CWalks PutOuts Assists Errors
## -Alan Ashby           69   321  414    375     632      43     10
## -Alvin Davis          63   224  266    263     880      82     14
## -Andre Dawson        225   828  838    354     200      11      3
## -Andres Galarraga     12    48   46     33     805      40      4
## -Alfredo Griffin      19   501  336    194     282     421     25
## -Al Newman             1    30    9     24      76     127      7
```

```
y     <- Hitters.numeric$Salary
y.std <- Hitters.numeric$Salary #Note, we'll keep the outcome raw
head(y)
```

```
## [1] 475.0 480.0 500.0  91.5 750.0  70.0
```

5.1.2 Run the Models - Ridge Regression

Now, we have te data setup so that we can test the models. We'll consider three models:

1. Use the raw (unstandardized) data *Hitters.numeric* and set the 'glmnet' option 'standardize = TRUE'. This means that the program will automatically standardize the data.
2. Use the raw (unstandardized) data *Hitters.numeric* and set the 'glmnet' option 'standardize = FALSE'. This means that the program will not standadize the data and instead use the raw data.
3. Use the standardized data *Hitters.numeric.std*. By default, 'glmnet' sets 'standardize=T' so there is no need to specify the option.

Let's estimate each model for various values of $\lambda = [10^{10}, ..., 10^{-2}]$. Set this option using the `lambda=` option. Note that we set $alpha = 0$ for the Ridge regression as per Eq. (9).

```
grid <- c(10^seq(10,-2,length=99),0)

# First the unstandardized data with glmnet's standardize = T:
ridge.mod.T <- glmnet(x,y,alpha=0,lambda=grid,standardize = T)

# First the unstandardized data with glmnet's standardize = F:
ridge.mod.F <- glmnet(x,y,alpha=0,lambda=grid,standardize = F)

# Second, the standardized model:
ridge.mod.std <- glmnet(x.std,y.std,alpha=0,lambda=grid)
```

Let's view what is saved in the model files. A useful command is `str()` which displays the contents of an object in R.

```
str(ridge.mod.T)
```

```
## List of 12
##  $ a0       : Named num [1:100] 536 536 536 536 536 ...
##   ..- attr(*, "names")= chr [1:100] "s0" "s1" "s2" "s3" ...
##  $ beta     :Formal class 'dgCMatrix' [package "Matrix"] with 6 slots
##   .. ..@ i       : int [1:1600] 0 1 2 3 4 5 6 7 8 9 ...
##   .. ..@ p       : int [1:101] 0 16 32 48 64 80 96 112 128 144 ...
##   .. ..@ Dim     : int [1:2] 16 100
##   .. ..@ Dimnames:List of 2
##   .. .. ..$ : chr [1:16] "AtBat" "Hits" "HmRun" "Runs" ...
##   .. .. ..$ : chr [1:100] "s0" "s1" "s2" "s3" ...
##   .. ..@ x       : num [1:1600] 5.44e-08 1.97e-07 7.96e-07 3.34e-07 3.53e-07 ...
##   .. ..@ factors : list()
##  $ df       : int [1:100] 16 16 16 16 16 16 16 16 16 16 ...
##  $ dim      : int [1:2] 16 100
##  $ lambda   : num [1:100] 1.00e+10 7.54e+09 5.69e+09 4.29e+09 3.24e+09 ...
##  $ dev.ratio: num [1:100] 2.72e-07 3.61e-07 4.79e-07 6.35e-07 8.41e-07 ...
##  $ nulldev  : num 53319113
##  $ npasses  : int 2194
##  $ jerr     : int 0
```

```
##  $ offset   : logi FALSE
##  $ call     : language glmnet(x = x, y = y, alpha = 0, lambda = grid, standardize = T)
##  $ nobs     : int 263
##  - attr(*, "class")= chr [1:2] "elnet" "glmnet"
```

We can also check the dimensions of the stored coefficients.

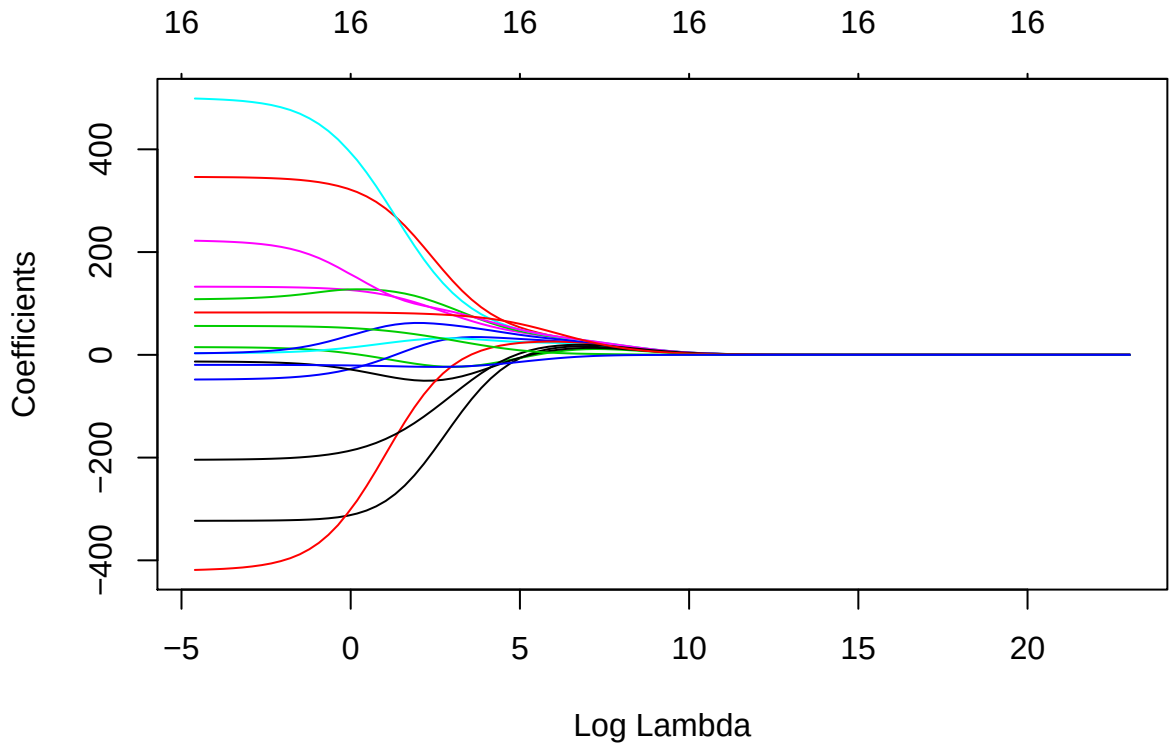
```
dim(ridge.mod.T$beta)
```

```
## [1]  16 100
```

The coefficients are stored in a matrix with $p = 16$ rows and and 100 columns (where each column represents a set of coefficients for each λ).

First, let's plot the coefficeints to see how they vary with λ.

```
plot(ridge.mod.std, xvar = "lambda", label = TRUE)
```



Let's next view the first and last few columns:

```
# Print the first and last two columns:
print(ridge.mod.T$beta[,c(1:2,99:100)],digits=2)
```

```
## 16 x 4 sparse Matrix of class "dgCMatrix"
##                s0       s1    s98    s99
## AtBat    5.4e-08  7.2e-08 -2.193 -2.193
## Hits     2.0e-07  2.6e-07  7.671  7.677
## HmRun    8.0e-07  1.1e-06  1.708  1.728
## Runs     3.3e-07  4.4e-07 -1.881 -1.892
## RBI      3.5e-07  4.7e-07  0.126  0.120
```

```
## Walks    4.2e-07  5.5e-07  6.105  6.108
## Years    1.7e-06  2.3e-06 -2.726 -2.685
## CAtBat   4.7e-09  6.2e-09 -0.183 -0.184
## CHits    1.7e-08  2.3e-08  0.167  0.166
## CHmRun   1.3e-07  1.7e-07  0.037  0.031
## CRuns    3.5e-08  4.6e-08  1.506  1.511
## CRBI     3.6e-08  4.7e-08  0.687  0.690
## CWalks   3.8e-08  5.0e-08 -0.773 -0.774
## PutOuts  2.2e-08  2.9e-08  0.295  0.295
## Assists  3.6e-09  4.7e-09  0.387  0.388
## Errors  -1.7e-08 -2.2e-08 -2.955 -2.953
```

Note that the first few columns are the models where λ was large and hence the penalty function was huge. As such, the coefficients are small. Conversely, the last few columns have large coefficients and converge toward a regular OLS regression.

To compare to OLS regression, we can fit a linear model and compare the results.

```
ols.model <- lm(y.std ~ x.std)
str(ols.model)
```

```
## List of 12
##  $ coefficients : Named num [1:17] 126.1 -324.5 353.2 18.9 -53.6 ...
##   ..- attr(*, "names")= chr [1:17] "(Intercept)" "x.stdAtBat" "x.stdHits" "x.stdHmRun" ...
##  $ residuals    : Named num [1:263] 82.3 -313.3 -584.7 -390 178.5 ...
##   ..- attr(*, "names")= chr [1:263] "1" "2" "3" "4" ...
##  $ effects      : Named num [1:263] -8691.3 -2882.6 -1595.6 -1120.1 85.6 ...
##   ..- attr(*, "names")= chr [1:263] "(Intercept)" "x.stdAtBat" "x.stdHits" "x.stdHmRun" ...
##  $ rank         : int 17
##  $ fitted.values: Named num [1:263] 393 793 1085 481 572 ...
##   ..- attr(*, "names")= chr [1:263] "1" "2" "3" "4" ...
##  $ assign       : int [1:17] 0 1 1 1 1 1 1 1 1 1 ...
##  $ qr           :List of 5
##   ..$ qr   : num [1:263, 1:17] -16.2173 0.0617 0.0617 0.0617 0.0617 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:263] "1" "2" "3" "4" ...
##   .. .. ..$ : chr [1:17] "(Intercept)" "x.stdAtBat" "x.stdHits" "x.stdHmRun" ...
##   .. ..- attr(*, "assign")= int [1:17] 0 1 1 1 1 1 1 1 1 1 ...
##   ..$ qraux: num [1:17] 1.06 1.03 1.03 1.01 1.02 ...
##   ..$ pivot: int [1:17] 1 2 3 4 5 6 7 8 9 10 ...
##   ..$ tol  : num 1e-07
##   ..$ rank : int 17
##   ..- attr(*, "class")= chr "qr"
##  $ df.residual  : int 246
##  $ xlevels      : Named list()
##  $ call         : language lm(formula = y.std ~ x.std)
##  $ terms        :Classes 'terms', 'formula'  language y.std ~ x.std
##   .. ..- attr(*, "variables")= language list(y.std, x.std)
##   .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. ..$ : chr [1:2] "y.std" "x.std"
##   .. .. .. ..$ : chr "x.std"
##   .. ..- attr(*, "term.labels")= chr "x.std"
##   .. ..- attr(*, "order")= int 1
##   .. ..- attr(*, "intercept")= int 1
##   .. ..- attr(*, "response")= int 1
```

```
##   .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. ..- attr(*, "predvars")= language list(y.std, x.std)
##   .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "nmatrix.16"
##   .. .. ..- attr(*, "names")= chr [1:2] "y.std" "x.std"
##  $ model        :'data.frame':   263 obs. of  2 variables:
##   ..$ y.std: num [1:263] 475 480 500 91.5 750 ...
##   ..$ x.std: num [1:263, 1:16] 2.14 3.25 3.37 2.18 4.03 ...
##   .. ..- attr(*, "dimnames")=List of 2
##   .. .. ..$ : chr [1:263] "-Alan Ashby" "-Alvin Davis" "-Andre Dawson" "-Andres Galarraga" ...
##   .. .. ..$ : chr [1:16] "AtBat" "Hits" "HmRun" "Runs" ...
##   ..- attr(*, "terms")=Classes 'terms', 'formula'  language y.std ~ x.std
##   .. .. ..- attr(*, "variables")= language list(y.std, x.std)
##   .. .. ..- attr(*, "factors")= int [1:2, 1] 0 1
##   .. .. .. ..- attr(*, "dimnames")=List of 2
##   .. .. .. .. ..$ : chr [1:2] "y.std" "x.std"
##   .. .. .. .. ..$ : chr "x.std"
##   .. .. ..- attr(*, "term.labels")= chr "x.std"
##   .. .. ..- attr(*, "order")= int 1
##   .. .. ..- attr(*, "intercept")= int 1
##   .. .. ..- attr(*, "response")= int 1
##   .. .. ..- attr(*, ".Environment")=<environment: R_GlobalEnv>
##   .. .. ..- attr(*, "predvars")= language list(y.std, x.std)
##   .. .. ..- attr(*, "dataClasses")= Named chr [1:2] "numeric" "nmatrix.16"
##   .. .. .. ..- attr(*, "names")= chr [1:2] "y.std" "x.std"
##  - attr(*, "class")= chr "lm"
```

```
tmp <- as.data.frame(matrix(cbind(ridge.mod.T$beta[,c(1:2,99:100)],ols.model$coefficients[c(2:17)]/my.s

names(tmp) <- c("s0","s1","s98","s99","OLS")
kable(tmp, digits=7)
```

s0	s1	s98	s99	OLS
1.0e-07	1.0e-07	-2.1927513	-2.1933643	-2.2030219
2.0e-07	3.0e-07	7.6707642	7.6771459	7.8277616
8.0e-07	1.1e-06	1.7082161	1.7277653	2.1635460
3.0e-07	4.0e-07	-1.8808923	-1.8917189	-2.0995672
4.0e-07	5.0e-07	0.1259213	0.1199157	-0.0229159
4.0e-07	6.0e-07	6.1048074	6.1084546	6.1510607
1.7e-06	2.3e-06	-2.7261922	-2.6845141	-2.5923684
0.0e+00	0.0e+00	-0.1830911	-0.1837237	-0.1762761
0.0e+00	0.0e+00	0.1670993	0.1663457	0.0697616
1.0e-07	2.0e-07	0.0372519	0.0305062	-0.2330881
0.0e+00	0.0e+00	1.5061547	1.5105471	1.6100500
0.0e+00	0.0e+00	0.6868874	0.6900174	0.8014277
0.0e+00	0.0e+00	-0.7732270	-0.7741168	-0.7939414
0.0e+00	0.0e+00	0.2946138	0.2946136	0.2945734
0.0e+00	0.0e+00	0.3874545	0.3877721	0.3839953
0.0e+00	0.0e+00	-2.9548449	-2.9525474	-2.8787143

Note that there are some slight differences between the `glmnet` results with $\lambda = 0$ and the OLS results. The reason is that the `glmnet` uses a numerical optimization technique to fit the parameters. To return results closer to the OLS coefficients, we can simply tighten the tolerance on the Ridge regression using the `thresh` option.

```
library("kableExtra")
```

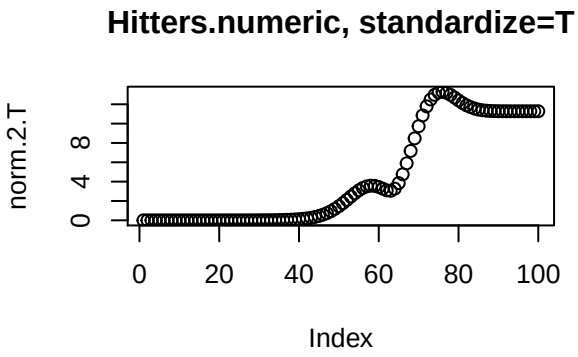
```
## Warning: package 'kableExtra' was built under R version 3.6.2
```

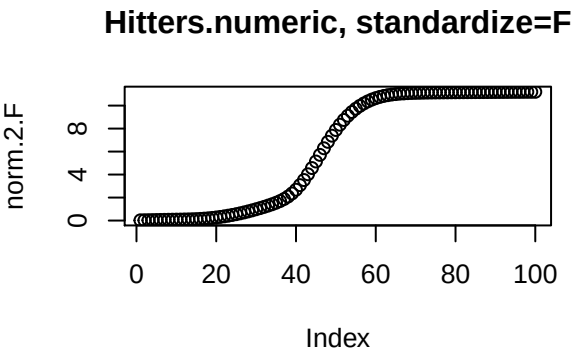
```
# Fit the tight model:
ridge.mod.T.tight <- glmnet(x,y,alpha=0,lambda=grid,standardize = T,thresh=1e-16)


tmp <- as.data.frame(matrix(cbind(ridge.mod.T$beta[,c(1:2,99:100)],ridge.mod.T.tight$beta[,100],ols.mod

names(tmp) <- c("s0","s1","s98","s99","s99 - tight","OLS")

kable(tmp, digits=7) %>%
  row_spec(c(3,4,5,9,10,12), bold = T, color = "white", background = "#D7261E")
```

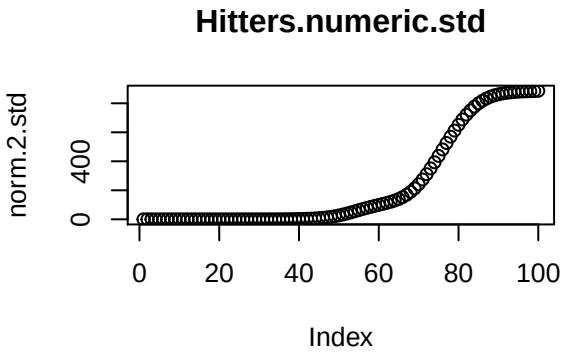
s0	s1	s98	s99	s99 - tight	OLS
1.0e-07	1.0e-07	-2.1927513	-2.1933643	-2.2030209	-2.2030219
2.0e-07	3.0e-07	7.6707642	7.6771459	7.8277568	7.8277616
8.0e-07	**1.1e-06**	**1.7082161**	**1.7277653**	**2.1635412**	**2.1635460**
3.0e-07	**4.0e-07**	**-1.8808923**	**-1.8917189**	**-2.0995633**	**-2.0995672**
4.0e-07	**5.0e-07**	**0.1259213**	**0.1199157**	**-0.0229147**	**-0.0229159**
4.0e-07	6.0e-07	6.1048074	6.1084546	6.1510589	6.1510607
1.7e-06	2.3e-06	-2.7261922	-2.6845141	-2.5923498	-2.5923684
0.0e+00	0.0e+00	-0.1830911	-0.1837237	-0.1762766	-0.1762761
0.0e+00	**0.0e+00**	**0.1670993**	**0.1663457**	**0.0697642**	**0.0697616**
1.0e-07	**2.0e-07**	**0.0372519**	**0.0305062**	**-0.2330844**	**-0.2330881**
0.0e+00	0.0e+00	1.5061547	1.5105471	1.6100485	1.6100500
0.0e+00	**0.0e+00**	**0.6868874**	**0.6900174**	**0.8014262**	**0.8014277**
0.0e+00	0.0e+00	-0.7732270	-0.7741168	-0.7939407	-0.7939414
0.0e+00	0.0e+00	0.2946138	0.2946136	0.2945734	0.2945734
0.0e+00	0.0e+00	0.3874545	0.3877721	0.3839955	0.3839953
0.0e+00	0.0e+00	-2.9548449	-2.9525474	-2.8787160	-2.8787143

To summarize the results, we can compute the penalty function for each column, or more specifically the l_2 norm compnonent of each column. Check the l_2 norm of the coefficients as λ decreases.

```
par(mfrow=c(2,2))

# Unstandardized data, standardize=T
norm.2.T <- apply(ridge.mod.T$beta[,],2,function(x){norm(x,type="2")})
plot(norm.2.T,main="Hitters.numeric, standardize=T")

# Unstandardized data, standardize=F
norm.2.F <- apply(ridge.mod.F$beta[,],2,function(x){norm(x,type="2")})
plot(norm.2.F,main="Hitters.numeric, standardize=F")

# Standardized data
norm.2.std <- apply(ridge.mod.std$beta[,],2,function(x){norm(x,type="2")})
plot(norm.2.std,main="Hitters.numeric.std")
```







Notice the differences in the behavior of the l_2 norms in the above plots. The first plot uses the results form the non-standardized data while the second uses the standardized data.

The above results bring about a few quesitons:

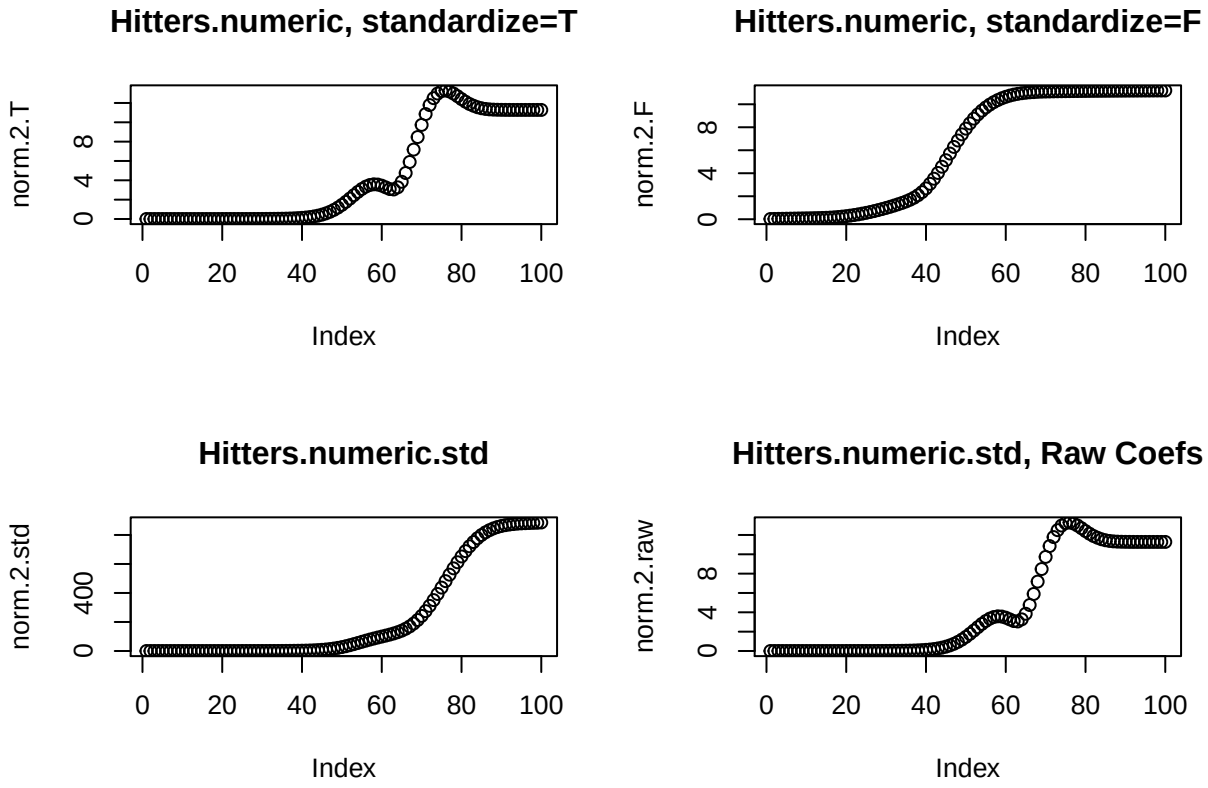
1. Why do the plots look different?
2. Which is "correct"?

The most obvious answer is they look different because of the differences in how we standardized the data. The first plot shows some odd behavior. In particular, why is the l_2 norm not monotonically increasing with decreasing values of λ? The answer is due to the way in which the `glmnet` package handles the coefficients. In particular, when the standardize option is set to true, the algorithm fits the regression models to standardized data. However, it returns the coefficients in raw form. That is, it transforms the coefficients back to their raw values.

To see how this works, let's try and convert our standardized coefficients to the raw coefficients.

```
par(mfrow=c(2,2))

# Unstandardized data, standardize=T
norm.2.T <- apply(ridge.mod.T$beta[,],2,function(x){norm(x,type="2")})
plot(norm.2.T,main="Hitters.numeric, standardize=T")

# Unstandardized data, standardize=F
norm.2.F <- apply(ridge.mod.F$beta[,],2,function(x){norm(x,type="2")})
plot(norm.2.F,main="Hitters.numeric, standardize=F")

# Standardized data
```

```r
norm.2.std <- apply(ridge.mod.std$beta[,],2,function(x){norm(x,type="2")})
plot(norm.2.std,main="Hitters.numeric.std")

# Standardized data
my.stds <- matrix(rep(my.stats[c(1:16),8],100),ncol=100,nrow=16,byrow=F)
ridge.betas.raw <- ridge.mod.std$beta / my.stds
norm.2.raw <- apply(ridge.betas.raw,2,function(x){norm(x,type="2")})
plot(norm.2.raw,main="Hitters.numeric.std, Raw Coefs")
```



Note that now the bottom right and top left plots are exactly the same. The reasaon is that we have now transformed our results back to the scale of the original raw data. This is equivalent to what the `glmnet` package does automatically and again highlights the importance of understanding how specific packages work. For example, in this case in order to make the predicitons we need to make sure the scale of the coefficients and data are the same. If we were to use the coefficients underlying the bottom-left plot to make predicitons, we would have to input standardized data. If we wanted to make predicitons using the top-left (or bottom-right) plots, we would input the raw (unadjusted) data.

In sum, the top left and bottom two graphs are equivalent - conditional on getting the scaling correct. However, shoule we use thos results or the ones from the top-right plot where we fit the data using unstandardized data? A general rule of thumb is that we want to use the standardized data when fitting the model. Otherwise, the scale of the data will impact the results. For example, if we had input the variable $Years$ as $Months$ instead, it would be penalized much more severely and hence be less likely to have an impact on the result. However, since it is standardized, there would be no difference.

5.1.3 Choosing Lambda (i.e., Calibrating the Model)

The above exmaples show how to fit some basic models but are of little practical use. Let's now do a more realistic example and choose the optimal λ. We will still use the Ridge regression as an example (again, to change from a RIdge to Lasso, simply change α from $1 \rightarrow 0$.

In order to choose an "optimal" value of λ, we need to first define a target value that we want to optimize. Here, we want to find a balance between bias and variance. In particular, we will use the Mean-Square Error or MSE. Consider the following expression,

$$E\left[y_0 - \hat{f}(x_0)\right] = Var\left[\hat{f}(x_0)\right] + \left[Bias(\hat{f}(x_0))\right]^2 + Var[\epsilon], \tag{12}$$

where, $E\left[y_0 - \hat{f}(x_0)\right]$ refers to the expected test MSE. In practice, we can compute the MSE for various values of λ based on out-of-sample (i.e., test) data. The test statistic will be the following:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y_i}), \tag{13}$$

where, $\hat{y_i}$ is the model prediction and y_i is the actual outcome for each test observation $i...N$.

5.1.3.1 Step 1.: Split the Data into a Training and Test Sample

Start by splitting the data into a test and training set. Note that we will now use the raw data $Hitters.numeric$ in this seciton and rely on `glmnet`'s standardizing option to handle the standardization of the data automatically.

First, select a random sample of data for the test and training samples. We will use the `sample.int()` function to randomly split the data. Note the random number gerating seed is fixed so you can replicate the results.

First generate a vector of $1's$ and $2's$ to define the training and test samples. Recall that x contians the raw independent vairables and y contains the raw dependent variables.

```
set.seed(1111) #set the RNG seed to any value
idx <- sample.int(2,size=nrow(x),replace=T,prob=c(0.5,0.5))
```

Next, define the training set as observations where $idx = 1$ and the test sample as $idx = 2$.

```
x.train <- x[which(idx==1),]
y.train <- y[which(idx==1)]
x.test <- x[which(idx==2),]
y.test <- y[which(idx==2)]
```

Alternatively, if we wanted to set a fixed percentage, do the following.

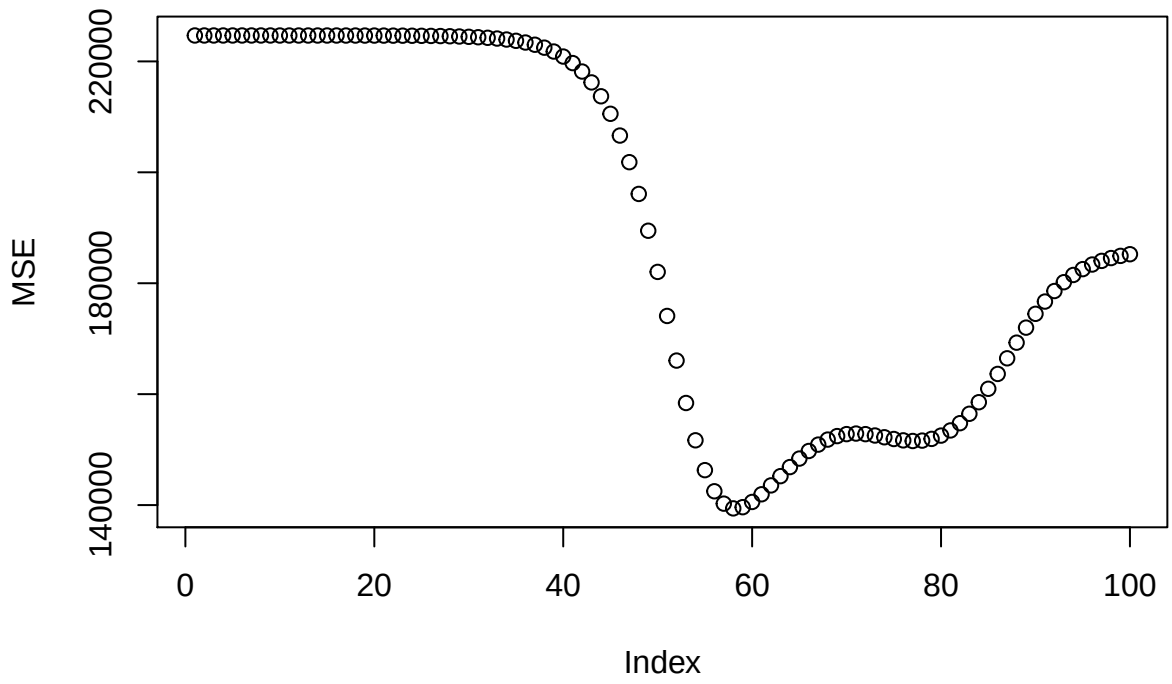
```
# Another way to sample:
set.seed(1)
train <- sample(1:nrow(x),nrow(x)/2)
test <- (-train)
x.test <- x[test,]
x.train <- x[train,]
y.test <- y[test]
y.train <- y[train]
```

Next, write a function to compute MSE based on Eq. (11).

```r
# Use MSE as a performance measure:
my.mse <- function(pred,act){
  mse <- mean((pred-act)^2)
  return(mse)
}
```

Next, we can fit a series of models as above and save them as a `glmnet` object *ridge.mse*.

```r
# Fit some models and save MSE:
MSE <- c(NA)
grid <- 10^seq(10,-2,length=100)
ridge.mse <- glmnet(x.train,y.train,alpha=0,lambda=grid,thresh=1e-12)

# Compute the MSE of each model
for(i in 1:length(grid)){
  ridge.pred.tmp <- predict(ridge.mse,s=grid[i],newx <- x.test)
  MSE[i] <- my.mse(ridge.pred.tmp,y.test)
}
plot(MSE)
```



The final step is to choose the value of λ that minimizes the MSE.

```r
lambda.star <- grid[which.min(MSE)]
sprintf("Optimal value of lambda is %.1f",lambda.star)
```

```
## [1] "Optimal value of lambda is 1232.8"
```

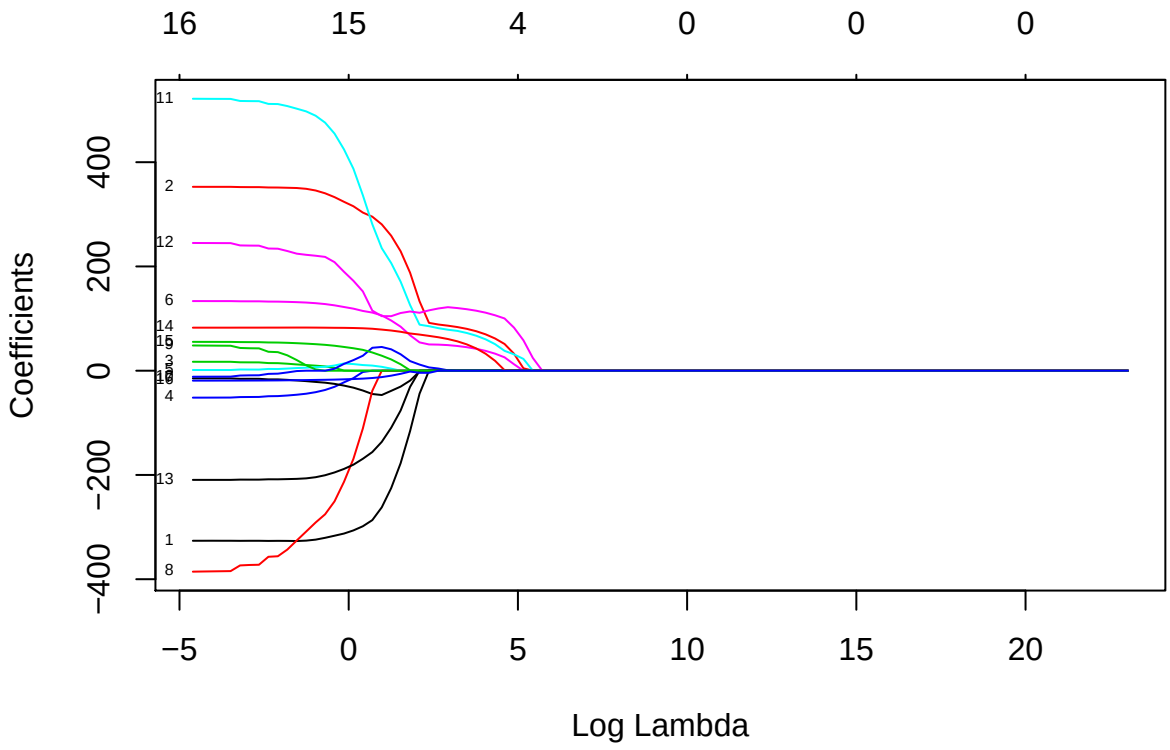
5.1.4 Run the Models - Lasso Regressions

To run the lasso regression, we simply need to change α to 1. We will repeat the above analysis.

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

# First the unstandardized data with glmnet's standardize = T:
lasso.mod.T <- glmnet(x,y,alpha=1,lambda=grid,standardize = T)

# First the unstandardized data with glmnet's standardize = F:
lasso.mod.F <- glmnet(x,y,alpha=1,lambda=grid,standardize = F)

# Second, the standardized model:
lasso.mod.std <- glmnet(x.std,y.std,alpha=1,lambda=grid)
```

Now, plot the coefficients as a function of *lambda*.

```
plot(lasso.mod.std, xvar = "lambda", label = TRUE)
```



Let's view the first and last few columns:

```
# Print the first and last two columns:
print(lasso.mod.T$beta[,c(1:2,99:100)],digits=2)
```

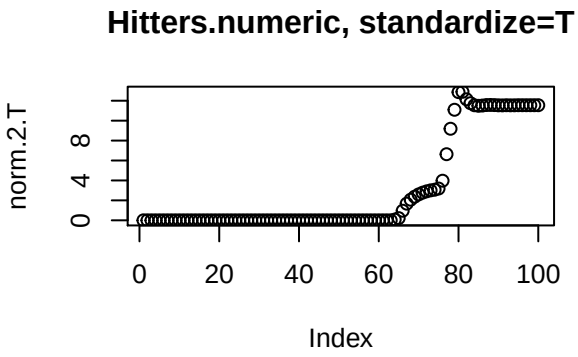
```
## 16 x 4 sparse Matrix of class "dgCMatrix"
##         s0 s1    s98    s99
## AtBat    .  . -2.215 -2.215
## Hits     .  .  7.817  7.817
## HmRun    .  .  1.952  1.954
```

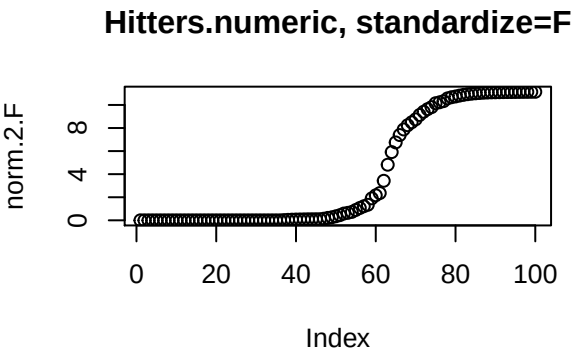
```
## Runs      .   . -2.023 -2.024
## RBI       .   .  0.052  0.052
## Walks     .   .  6.149  6.150
## Years     .   . -3.058 -3.055
## CAtBat    .   . -0.168 -0.169
## CHits     .   .  0.074  0.075
## CHmRun    .   . -0.140 -0.141
## CRuns     .   .  1.575  1.575
## CRBI      .   .  0.758  0.758
## CWalks    .   . -0.793 -0.793
## PutOuts   .   .  0.295  0.295
## Assists   .   .  0.381  0.381
## Errors    .   . -2.884 -2.886
```

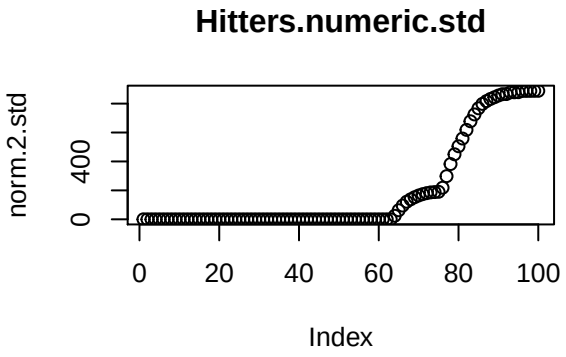
Note that the first few columns are the models where λ was large and hence the penalty function was huge. For the Lasso regressios, the variables drop out as denoted by the "."'s. Conversely, the last few columns have large coefficients and converge toward a regular OLS regression. Note that the are very similar to the last couple columns of the Ridge regressions.

To summarize the results, we can compute the penalty function for each column, or more specifically the l_1 norm compnonent of each column. Check the l_1 norm of the coefficients as λ decreases.

```r
par(mfrow=c(2,2))

# Unstandardized data, standardize=T
norm.2.T <- apply(lasso.mod.T$beta[,],2,function(x){norm(x,type="2")})
plot(norm.2.T,main="Hitters.numeric, standardize=T")

# Unstandardized data, standardize=F
norm.2.F <- apply(lasso.mod.F$beta[,],2,function(x){norm(x,type="2")})
plot(norm.2.F,main="Hitters.numeric, standardize=F")

# Standardized data
norm.2.std <- apply(lasso.mod.std$beta[,],2,function(x){norm(x,type="2")})
plot(norm.2.std,main="Hitters.numeric.std")
```







Notice the differences in the behavior of the l_1 plots from the Lasso regressions versus the l_2 plots from the Ridge regressions. The main difference is that the l_1 norm is exactly equal to zero for high values of λ whereas the l_2 norm is close but not exactly equal to zero.

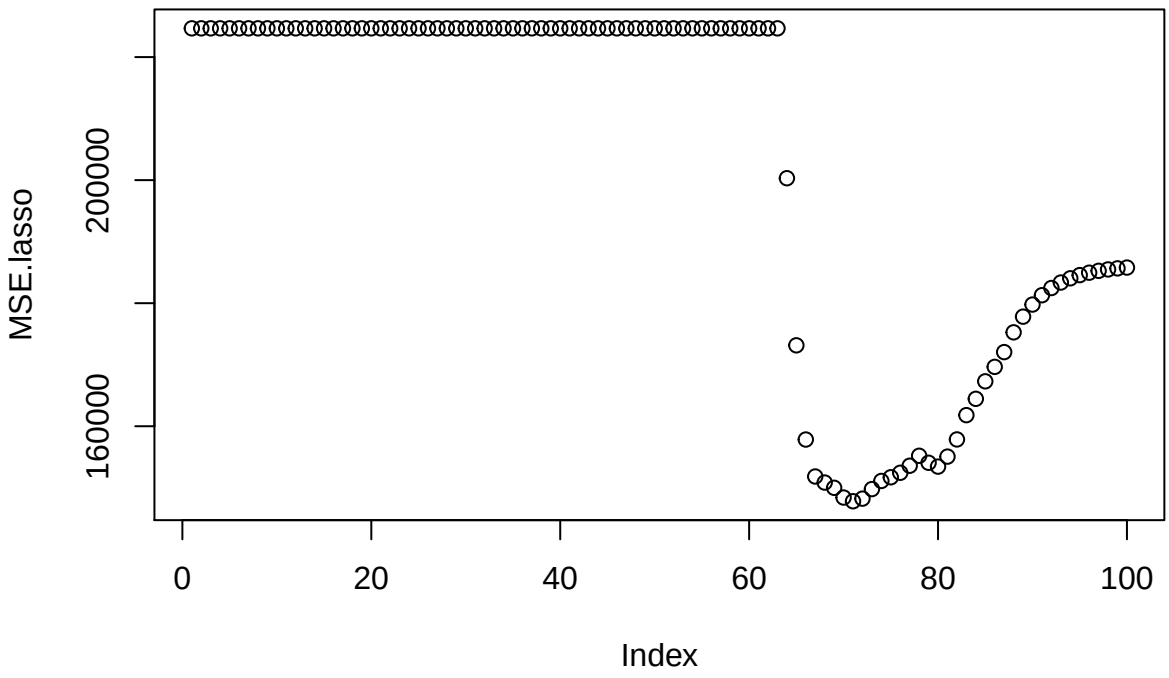
The same logic regarding the standardized and raw data apply to the Lasso models as the Ridge reressions.

5.1.4.1 Choosing Lambda (i.e., Calibrating the Model)

Just as for the ridge regressions, in order to choose an "optimal" value of λ, we need to first define a target value that we want to optimize. We will again use the out-of-sample MSE. We'll use the same training and test samples as the above examples.

Next, fit a series of Lasso models as above and save them as a `glmnet` object *lasso.mse*.

```
# Fit some models and save MSE:
MSE.lasso <- c(NA)
grid <- 10^seq(10,-2,length=100)
lasso.mse <- glmnet(x.train,y.train,alpha=1,lambda=grid,thresh=1e-12)

# Compute the MSE of each model
for(i in 1:length(grid)){
  lasso.pred.tmp <- predict(lasso.mse,s=grid[i],newx <- x.test)
  MSE.lasso[i] <- my.mse(lasso.pred.tmp,y.test)
}
plot(MSE.lasso)
```



Note again that the diffrence fro teh ridge regressions is that until λ drops below a certain value, no variables are selected as indicated by the constant MSE in the left part of the plot. The final step is to choose the value of λ that minimizes the MSE.

```
lambda.star.lasso <- grid[which.min(MSE.lasso)]
sprintf("Optimal value of lambda is %.1f",lambda.star.lasso)
```

```
## [1] "Optimal value of lambda is 32.7"
```

References

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2013. *An Introduction to Statistical Learning*. Vol. 112. Springer.