Lab 21 - Regularized Regression

Chad Murchison, Guy Twa, Eddie-Williams Owiredu

2023-11-03

Today's Lab

Today's lab will introduce you to two supervised machine learning algorithm for regression, Ridge regression (L2) and Lasso (L1) regression. We will learn how to fit regularized regression models and compare them to simple ordinary least squares (OLS) regression. The goal of these regression methods is to perform prediction of outcome variables.

Loading Packages and Data

For today's lab, we will be using a new packages called `glmnet`, which you should go ahead and install prior to loading it in.

```
install.packages("glmnet")
```

I addition to loading our library, data, and setting our working directory, we'll use `set.seed` for replicability.

```
library(glmnet)
set.seed(112233)

setwd("~/Documents/GRD770/Lab 21 - Regularized Regression")

#Example data from glmnet
data(QuickStartExample)
```

Getting to Know Your Data

The dataset we'll be using for this lab is "QuickStartExample" from the glmnet package. This is a randomly generated data set with 100 observations of 20 numerical variables with one numerical outcome variable.

We can get a quick summary of the variables using the `sumarry()` function:

```
summary(QuickStartExample$x)
```

```
##        V1                 V2                  V3                  V4
##  Min.   :-1.8961   Min.   :-2.878895   Min.   :-2.63061   Min.   :-2.05928
##  1st Qu.:-0.4737   1st Qu.:-0.656813   1st Qu.:-0.63942   1st Qu.:-0.61508
##  Median : 0.2010   Median :-0.099319   Median : 0.07550   Median : 0.06699
##  Mean   : 0.2063   Mean   :-0.005649   Mean   : 0.07555   Mean   : 0.11810
##  3rd Qu.: 0.9952   3rd Qu.: 0.731664   3rd Qu.: 0.85611   3rd Qu.: 0.86831
##  Max.   : 3.1619   Max.   : 2.228786   Max.   : 2.55155   Max.   : 3.11291
##        V5                  V6                  V7                  V8
##  Min.   :-2.68543   Min.   :-2.78803   Min.   :-2.50549   Min.   :-3.6367
##  1st Qu.:-0.80457   1st Qu.:-0.78204   1st Qu.:-0.66256   1st Qu.:-0.8617
##  Median :-0.09333   Median : 0.06646   Median : 0.09519   Median :-0.1421
##  Mean   :-0.06115   Mean   : 0.01534   Mean   : 0.03528   Mean   :-0.1218
##  3rd Qu.: 0.58408   3rd Qu.: 0.70799   3rd Qu.: 0.63051   3rd Qu.: 0.5938
##  Max.   : 1.81981   Max.   : 2.27527   Max.   : 2.18573   Max.   : 2.6536
##        V9                 V10                 V11                V12
##  Min.   :-2.88985   Min.   :-2.27895   Min.   :-2.2393   Min.   :-2.44167
##  1st Qu.:-0.73077   1st Qu.:-0.54614   1st Qu.:-0.9854   1st Qu.:-0.88268
##  Median :-0.09803   Median : 0.02715   Median :-0.3196   Median : 0.03550
##  Mean   :-0.08380   Mean   : 0.10124   Mean   :-0.2116   Mean   :-0.02109
##  3rd Qu.: 0.47161   3rd Qu.: 0.81758   3rd Qu.: 0.4524   3rd Qu.: 0.68328
##  Max.   : 2.41487   Max.   : 2.68135   Max.   : 2.2831   Max.   : 2.34162
##       V13                 V14                V15                 V16
##  Min.   :-2.71954   Min.   :-2.4622   Min.   :-1.96111   Min.   :-2.22929
##  1st Qu.:-0.65109   1st Qu.:-0.9246   1st Qu.:-0.70695   1st Qu.:-0.60383
##  Median :-0.02552   Median :-0.1244   Median :-0.25940   Median :-0.00324
##  Mean   :-0.07845   Mean   :-0.1358   Mean   :-0.09221   Mean   : 0.06317
##  3rd Qu.: 0.40112   3rd Qu.: 0.6319   3rd Qu.: 0.53198   3rd Qu.: 0.73559
##  Max.   : 2.75606   Max.   : 2.3301   Max.   : 2.25018   Max.   : 2.13515
##       V17                 V18                V19                V20
##  Min.   :-2.01322   Min.   :-3.4325   Min.   :-3.1944   Min.   :-2.34578
##  1st Qu.:-0.63558   1st Qu.:-0.8642   1st Qu.:-0.5149   1st Qu.:-0.71932
##  Median : 0.02783   Median :-0.1694   Median : 0.2007   Median :-0.07920
##  Mean   : 0.00815   Mean   :-0.2724   Mean   : 0.1156   Mean   :-0.05225
##  3rd Qu.: 0.58884   3rd Qu.: 0.4617   3rd Qu.: 0.9612   3rd Qu.: 0.50403
##  Max.   : 2.56317   Max.   : 2.0789   Max.   : 2.5947   Max.   : 2.51791
```

```
summary(QuickStartExample$y)
```

```
##        V1
##  Min.   :-6.8575
##  1st Qu.:-1.6814
##  Median : 0.7697
##  Mean   : 0.6608
##  3rd Qu.: 2.9038
##  Max.   : 6.2659
```

While this data set is pretty abstract, for our purposes great practice data set to show off some of the key concepts with regularized regression. Before we start working, lets coalesce our variables to a single data frame that we'll work with.

```
x <- QuickStartExample$x
y <- QuickStartExample$y

dat <- data.frame(x, Y = y)
dim(dat); head(dat)
```

```
## [1] 100  21
```

```
##           X1         X2         X3         X4         X5        X6         X7
## 1  0.2738562 -0.0366722  0.8547269  0.9675242  1.4154898 0.5234059  0.5626882
## 2  2.2448169 -0.5460300  0.2340651 -1.3350304  1.3130758 0.5212746 -0.6100346
## 3 -0.1254230 -0.6068782 -0.8539217 -0.1487772 -0.6646828 0.6066164  0.1617207
## 4 -0.5435734  1.1083583 -0.1042480  1.0165262  0.6999042 1.6550164  0.4899635
## 5 -1.4593984 -0.2744945  0.1119060 -0.8517877  0.3152839 1.0507493  1.3863575
## 6  1.0632081 -0.7535232 -1.3825534  1.0762270  0.3700331 1.4987212 -0.3604525
##            X8         X9        X10         X11         X12         X13
## 1  1.11122333  1.6408214  0.6187067  0.99993483 -0.07841916 -0.60332610
## 2 -0.86139651 -0.2704635  0.2300825 -0.10570898  0.16314122  0.76207661
## 3 -0.86272165  0.6042102  1.1939768  0.50125094 -0.94520592  0.39890263
## 4  0.02338209  0.2560304 -0.1273140 -0.06262846  0.64195468  0.07548198
## 5  0.28450104  1.1404976  2.6813460 -1.01473386  0.36704372  1.73759745
## 6 -0.21421571  1.8266483  0.8711225 -0.06372928  1.00507110  0.31520786
##           X14        X15        X16        X17         X18        X19
## 1  0.03323168 -0.7008845  1.1578379  1.4578156  0.77490699 -1.2685177
## 2  0.67812003 -0.5282670 -0.8791548 -0.4729134 -1.11717309 -0.7377321
## 3 -0.76478505  1.2854019  0.6448767  0.1792455  0.04473756  1.1053071
## 4 -1.37846440 -1.0247390 -2.1183615 -0.4695345  0.69779616  0.8656362
## 5 -1.26612706  1.4519995 -0.7894756 -0.9843860 -1.63700993 -0.5829168
## 6  0.53671846 -1.2624227 -1.5848662 -0.6314991 -1.87874350  0.4504287
##          X20          Y
## 1  1.9935800 -1.2748860
## 2 -1.0787929  1.8434251
## 3  0.3040545  0.4592363
## 4 -0.7894895  0.5640407
## 5 -1.5289104  1.8729633
## 6  1.4422643  0.5275317
```

Fitting regression models

We will use the `glmnet()` function from the `glmnet` package to fit our regularized regression models. This function has a lot of different parameters but we'll focus on the following:

- `x` = predictor variable data, as a matrix or dataframe
- `y` = response variable data, as a vector
- `family` = the model family, use "gaussian" for a continuous outcome variable (the OLS equivalent)
- `alpha` = the mixing parameter for the elastic net, 0 for Ridge, 1 for LASSO
- `lamda` = vector of lambda shrinkage penalty parameters
- `nlambda` = the number of shrinkage penalty parameters we want to search over
- `standardize` = whether or not to standardize the data before fitting the model

The mixing parameter, `alpha` determines the relative weight of the ridge and LASSO regression penalties. We'll be using values of either 0 for ridge regression or 1 for LASSO regression. When you use a mixture of these its called an elastic net. Today though, we'll focus on getting a sense of the differences between ridge and LASSO.

Lambda is our shrinkage penalty applied to our coefficients. Remember, if lamda is too large, all coefficients shrink to zero while setting lamda to zero leads to OLS regression. If we want to make our own grid sweep of lambda values we can create a vector and supply it to the `lambda parameter`. That can then be passed to the lambda argument. Alternatively we can let `glmnet()` come up with a set of lambda values, in which case we'll need the `nlambda` paramater to set the size for the range of lambda values to try.

Standardizing our inputs in advance is important as these methods are not scale equivalent like OLS regression. This is especially important when input variables are measured on different scales (e.g. height and weight). Thus, we set `standardize` to `TRUE`.

All of the other parameters are about particular operations of the function and beyond our scope here.

Let's fit a Ridge (L2) regression model by spetting `alpha` as 0, and a LASSO (L1) regression model by setting `aplha` as 1. (The parameters are listed out for our Ridge regression just for illustration. These are all the default values, so they are the same when we fit the LASSO as well.)

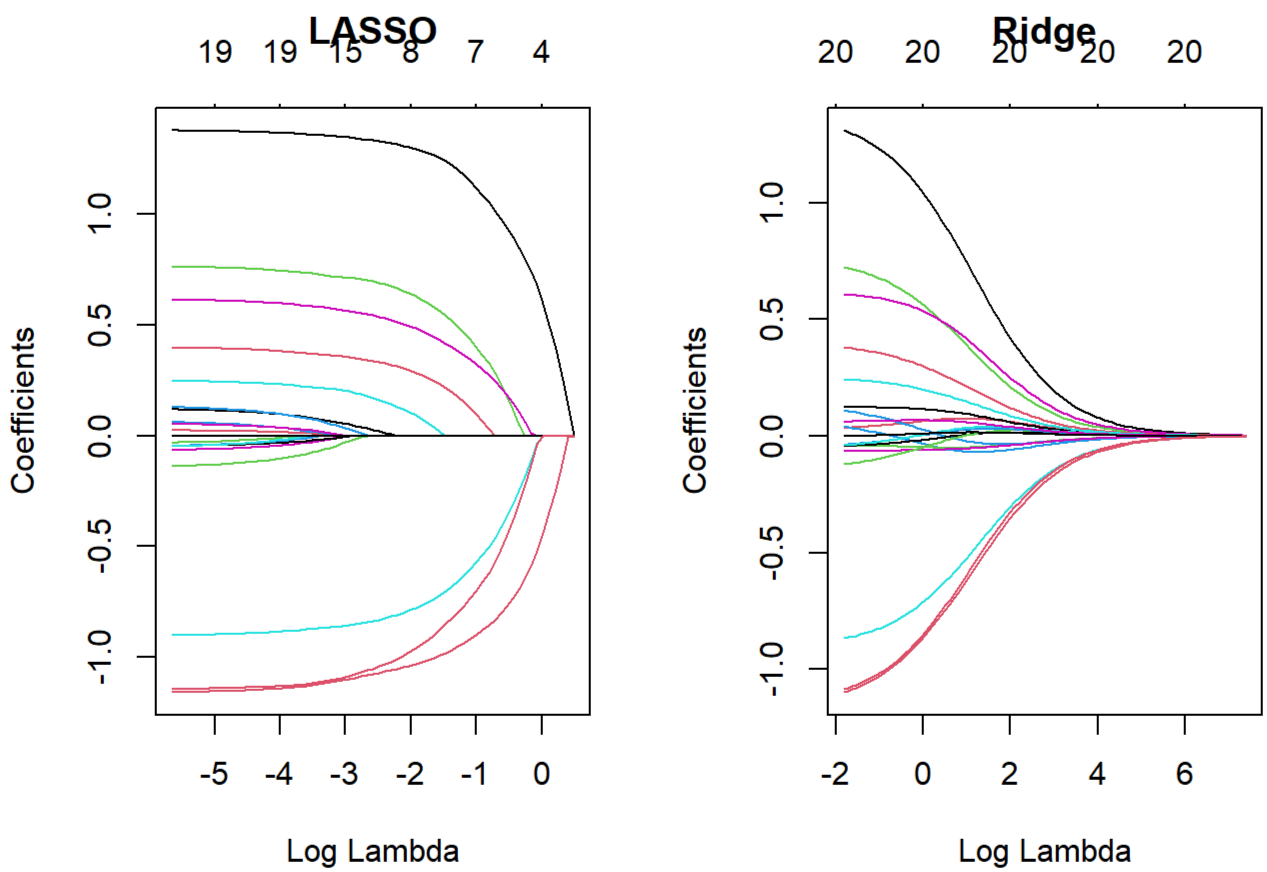
```
#Fit a ridge regression
fit_l2 <- glmnet(x, y,
                  family = "gaussian",
                  alpha = 0,
                  nlambda = 100,
                  lambda = NULL,
                  standardize = TRUE)

#Fit a LASSO by setting alpha to 1
fit_l1 <- glmnet(x, y, alpha = 1)
```

Comparing model coefficients

We can compare the effects of the different regularization penalties by plotting the predictor variable coefficient values against lambda.

```
par(mfrow = c(1,2))
plot(fit_l1, xvar = "lambda", main = "LASSO")
plot(fit_l2, xvar = "lambda", main = "Ridge")
```



We see how as lmanda increases, the L2 norm of the LASSO shrinks coefficient to 0 while the Ridge only approaches 0. Meanwhile, when the lines are flat early on, this is essentially the OLS coefficient values. An important point about both plots the number of coefficients labaled up top. LASSO regression drops coefficients as they reach 0, but Ridge regression always retains coefficients because they never reach 0.

We can also look at these results over the iterations of lambda values.

```
print(fit_l1); print(fit_l2)
```

```
##
## Call:  glmnet(x = x, y = y, alpha = 1)
##
##    Df  %Dev  Lambda
## 1   0  0.00 1.63100
## 2   2  5.53 1.48600
## 3   2 14.59 1.35400
## 4   2 22.11 1.23400
## 5   2 28.36 1.12400
## 6   2 33.54 1.02400
## 7   4 39.04 0.93320
## 8   5 45.60 0.85030
## 9   5 51.54 0.77470
## 10  6 57.35 0.70590
## 11  6 62.55 0.64320
## 12  6 66.87 0.58610
## 13  6 70.46 0.53400
## 14  6 73.44 0.48660
## 15  7 76.21 0.44330
## 16  7 78.57 0.40400
## 17  7 80.53 0.36810
## 18  7 82.15 0.33540
## 19  7 83.50 0.30560
## 20  7 84.62 0.27840
## 21  7 85.55 0.25370
## 22  7 86.33 0.23120
## 23  8 87.06 0.21060
## 24  8 87.69 0.19190
## 25  8 88.21 0.17490
## 26  8 88.65 0.15930
## 27  8 89.01 0.14520
## 28  8 89.31 0.13230
## 29  8 89.56 0.12050
## 30  8 89.76 0.10980
## 31  9 89.94 0.10010
## 32  9 90.10 0.09117
## 33  9 90.23 0.08307
## 34  9 90.34 0.07569
## 35 10 90.43 0.06897
## 36 11 90.53 0.06284
## 37 11 90.62 0.05726
## 38 12 90.70 0.05217
## 39 15 90.78 0.04754
## 40 16 90.86 0.04331
## 41 16 90.93 0.03947
## 42 16 90.98 0.03596
## 43 17 91.03 0.03277
## 44 17 91.07 0.02985
## 45 18 91.11 0.02720
## 46 18 91.14 0.02479
## 47 19 91.17 0.02258
## 48 19 91.20 0.02058
```

```
## 49 19 91.22 0.01875
## 50 19 91.24 0.01708
## 51 19 91.25 0.01557
## 52 19 91.26 0.01418
## 53 19 91.27 0.01292
## 54 19 91.28 0.01178
## 55 19 91.29 0.01073
## 56 19 91.29 0.00978
## 57 19 91.30 0.00891
## 58 19 91.30 0.00812
## 59 19 91.31 0.00739
## 60 19 91.31 0.00674
## 61 19 91.31 0.00614
## 62 20 91.31 0.00559
## 63 20 91.31 0.00510
## 64 20 91.31 0.00464
## 65 20 91.32 0.00423
## 66 20 91.32 0.00386
## 67 20 91.32 0.00351
```

```
##
## Call:  glmnet(x = x, y = y, family = "gaussian", alpha = 0, nlambda = 100,      lambd
a = NULL, standardize = TRUE)
##
##      Df  %Dev  Lambda
## 1   20  0.00 1631.00
## 2   20  0.45 1486.00
## 3   20  0.49 1354.00
## 4   20  0.54 1234.00
## 5   20  0.59 1124.00
## 6   20  0.65 1024.00
## 7   20  0.71  933.20
## 8   20  0.78  850.30
## 9   20  0.86  774.70
## 10  20  0.94  705.90
## 11  20  1.03  643.20
## 12  20  1.13  586.10
## 13  20  1.24  534.00
## 14  20  1.36  486.60
## 15  20  1.49  443.30
## 16  20  1.63  404.00
## 17  20  1.79  368.10
## 18  20  1.96  335.40
## 19  20  2.14  305.60
## 20  20  2.35  278.40
## 21  20  2.57  253.70
## 22  20  2.82  231.20
## 23  20  3.09  210.60
## 24  20  3.38  191.90
## 25  20  3.70  174.90
## 26  20  4.04  159.30
## 27  20  4.42  145.20
## 28  20  4.84  132.30
## 29  20  5.29  120.50
## 30  20  5.77  109.80
## 31  20  6.31  100.10
## 32  20  6.88   91.17
## 33  20  7.51   83.07
## 34  20  8.19   75.69
## 35  20  8.93   68.97
## 36  20  9.72   62.84
## 37  20 10.58   57.26
## 38  20 11.51   52.17
## 39  20 12.50   47.54
## 40  20 13.58   43.31
## 41  20 14.73   39.47
## 42  20 15.96   35.96
## 43  20 17.27   32.77
## 44  20 18.68   29.85
## 45  20 20.17   27.20
## 46  20 21.75   24.79
## 47  20 23.42   22.58
```

```
## 48  20 25.19   20.58
## 49  20 27.04   18.75
## 50  20 28.98   17.08
## 51  20 31.01   15.57
## 52  20 33.12   14.18
## 53  20 35.30   12.92
## 54  20 37.55   11.78
## 55  20 39.86   10.73
## 56  20 42.22    9.78
## 57  20 44.62    8.91
## 58  20 47.05    8.12
## 59  20 49.50    7.39
## 60  20 51.95    6.74
## 61  20 54.39    6.14
## 62  20 56.81    5.59
## 63  20 59.19    5.10
## 64  20 61.52    4.64
## 65  20 63.78    4.23
## 66  20 65.98    3.86
## 67  20 68.09    3.51
## 68  20 70.11    3.20
## 69  20 72.04    2.92
## 70  20 73.85    2.66
## 71  20 75.56    2.42
## 72  20 77.16    2.21
## 73  20 78.64    2.01
## 74  20 80.01    1.83
## 75  20 81.27    1.67
## 76  20 82.42    1.52
## 77  20 83.47    1.39
## 78  20 84.42    1.26
## 79  20 85.27    1.15
## 80  20 86.04    1.05
## 81  20 86.72    0.96
## 82  20 87.32    0.87
## 83  20 87.86    0.79
## 84  20 88.33    0.72
## 85  20 88.75    0.66
## 86  20 89.11    0.60
## 87  20 89.43    0.55
## 88  20 89.70    0.50
## 89  20 89.94    0.45
## 90  20 90.14    0.41
## 91  20 90.32    0.38
## 92  20 90.47    0.34
## 93  20 90.60    0.31
## 94  20 90.71    0.28
## 95  20 90.81    0.26
## 96  20 90.89    0.24
## 97  20 90.96    0.22
## 98  20 91.01    0.20
```

```
## 99  20 91.06    0.18
## 100 20 91.10    0.16
```

We see the L2 converged more quickly while the L1 probably need more lambdas to explain the same amount of variance.

We can extract the coefficients at a specific value of lambda defined as `s` using the `coef()` function

```
coef(fit_l1, s = 0.1)
```

```
## 21 x 1 sparse Matrix of class "dgCMatrix"
##                       s1
## (Intercept)  0.150928072
## V1           1.320597195
## V2           .
## V3           0.675110234
## V4           .
## V5          -0.817411518
## V6           0.521436671
## V7           0.004829335
## V8           0.319415917
## V9           .
## V10          .
## V11          0.142498519
## V12          .
## V13          .
## V14         -1.059978702
## V15          .
## V16          .
## V17          .
## V18          .
## V19          .
## V20         -1.021873704
```

```
coef(fit_l2, s = 0.1)
```

```
## 21 x 1 sparse Matrix of class "dgCMatrix"
##                      s1
## (Intercept)  0.14576689
## V1           1.30922437
## V2           0.03496846
## V3           0.72391240
## V4           0.03882705
## V5          -0.86710569
## V6           0.60697109
## V7           0.12355737
## V8           0.37889309
## V9          -0.03973640
## V10          0.10841981
## V11          0.24189927
## V12         -0.06661643
## V13         -0.04268166
## V14         -1.09804121
## V15         -0.12176667
## V16         -0.03711366
## V17         -0.04019624
## V18          0.06146105
## V19         -0.00179925
## V20         -1.08563245
```

Again we see how the L2 will completely shrink (and select) coefficients

Parameter sweeps using cross-validation

If we want to find an ideal lambda across a range, we can use a built in cross-validation command. To do this, we're going to want to build a test set and training, we'll use 60% of the QuickData dataframe.

We can select our training samples by randomly selecting 60 samples, and designating the remaining observations as testing samples.

```
train_vec <- sort(sample(1:nrow(dat), floor(nrow(dat) * 0.60)))
test_vec <- setdiff(1:nrow(dat), train_vec)
```

We'll also make our own vector of lambda values to sweep.

```
lambda_sweep <- 10 ^ seq(5, -5, length = 500)
```

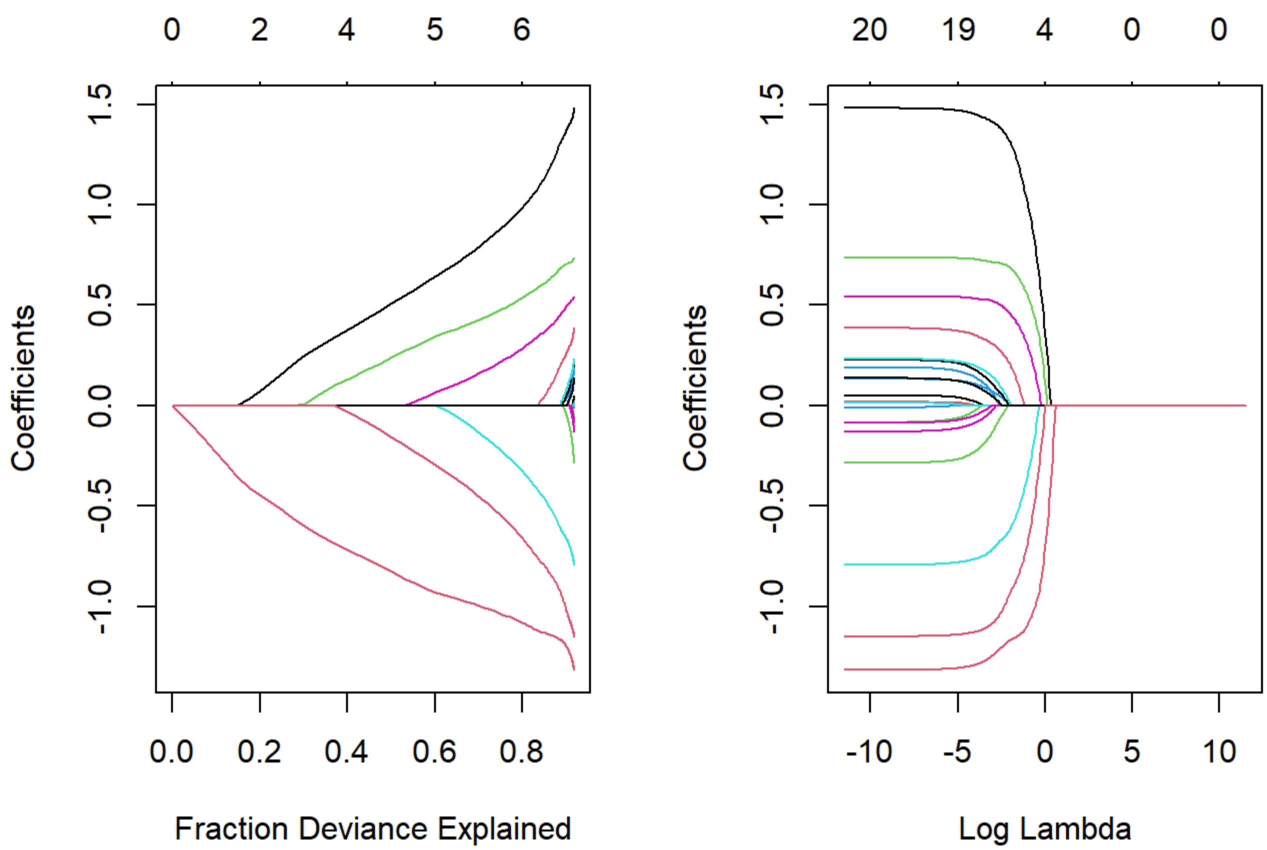
The lambda sweep vector effectively gives a set of lambda's ranging from intercept only (l = 10^5) to basic OLS (l = 0.00001).

LASSO cross validation

Let's start with the full set using LASSO for ease of display, and plot the coefficients against the explained deviance as well as the range of lambda values.

```
# fit LASSO regression using the trianing
fit_l1_sweep <- glmnet(x[train_vec,],
                       y[train_vec],
                       alpha = 1,
                       lambda = lambda_sweep)

par(mfrow = c(1,2))
plot(fit_l1_sweep, xvar = "dev", label = TRUE)
plot(fit_l1_sweep, xvar = "lambda")
```



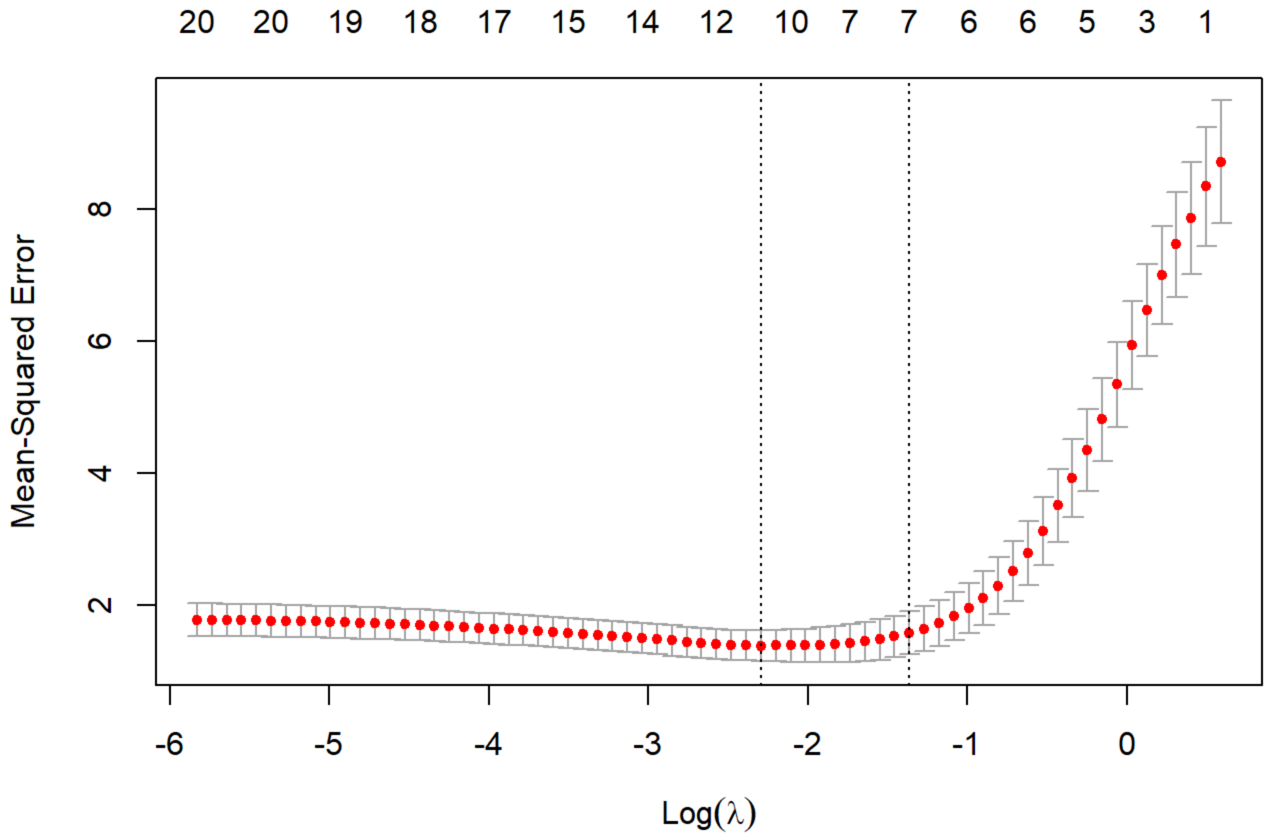
From the left plot, it appears we only need about 7 predictors to explain over 90% of the deviance. From our right plot, our sweep of lambda's mainly shows when lambda essentially becomes too large. But what's the most ideal tuning?

For that we can test with cross validation using the `cv.glmnet()` function. The most important parameter here is `nfolds`, where we set the number of folds for cross validation. For now, we'll leave that at the default 10.

```
fit_l1_cv <- cv.glmnet(x[train_vec,], y[train_vec], alpha = 1)
```

We can plot the MSE as a function of log(lambda) to start to see what the sweet spot is going to be where we have a minimum MSE with the fewest variables. Again, it's a tradeoff of bias and variance

```
plot(fit_l1_cv)
```



We see the best lambda value is going to be around 0.1008 - 0.2555 at log(lambda) of -2.294617 to -1.364533 and corresponds to roughly 9-11 predictors. We can select the lambda with the lowest MSE directly from the CV object by accessing the `lamnda.min` variable.

```
best_l1_lambda <- fit_l1_cv$lambda.min
best_l1_lambda
```

```
## [1] 0.1007692
```

Now that we have our model, we can evaluate how well it makes predictions using our testing data we set aside earlier. To make preditions, we use the `predict()` function. In our `predict()` call we need to specify our model, our lambda value, and our testing data predictor variable data.

```
pred_l1_cv <- predict(fit_l1_sweep,        # model
                      s = best_l1_lambda,  # selected lambda value
                      newx = x[test_vec,]) # testing data predictor observations
```

Finally, we can look at the mean squared error (MSE) of our model outcome predictions to the real values.

```
mean((pred_l1_cv - y[test_vec])^2)
```

```
## [1] 1.007394
```

Comparing OLS, LASSO, and Ridge

Now we can compare against the expected OLS output we can see the similarity with small lambda. Let's start by building out a cross-validated dataset for Ridge to go with our Lasso using the same method as before.

Ridge cross validation

```
# Fit models for our range of lambda values
fit_l2_sweep <- glmnet(x[train_vec,], y[train_vec],
                       alpha = 0, lambda = lambda_sweep)

# Select the best lambda using cross-validation
fit_l2_cv <- cv.glmnet(x[train_vec,], y[train_vec], alpha = 0)
best_l2_lambda <- fit_l2_cv$lambda.min

# Make  predictions
pred_l2_cv <- predict(fit_l2_sweep, s = best_l2_lambda, newx = x[test_vec,])
```

OLS model

Next, let's build our OLS model. We can build this using the basic `lm()` function. We can use the `predict()` function with this model as well, by supplying our model and testing data.

```
fit_ols <- lm(Y ~ ., data = dat[train_vec,])
pred_ols <- predict(fit_ols, newdata = dat[test_vec,])
```

Model coeffiecients

Now that we have all three models, lets put their coefficients side by side to compare. Here we use the `cbind()` function to collumn-wise bind our sets of coefficients togther.

```
coef_results <- cbind(OLS = coef(fit_ols),
                      Ridge_L2 = coef(fit_l2_sweep, s = best_l2_lambda)[,1],
                      LASSO_L1 = coef(fit_l1_sweep, s = best_l1_lambda)[,1])
coef_results
```

```
##                    OLS     Ridge_L2    LASSO_L1
## (Intercept)  0.05821383  0.102880176  0.17014352
## X1           1.48091879  1.322703316  1.35401874
## X2           0.01709023  0.044406668  0.00000000
## X3           0.73784355  0.703911301  0.70065873
## X4           0.13231888  0.091561381  0.02718769
## X5          -0.79405163 -0.717792595 -0.64943556
## X6           0.54314247  0.531302527  0.48226569
## X7           0.22752049  0.198366755  0.03250319
## X8           0.38581490  0.306514059  0.24401196
## X9          -0.08452980 -0.037763656  0.00000000
## X10          0.19059793  0.098247122  0.00000000
## X11          0.23233986  0.186836195  0.06042086
## X12         -0.13028683 -0.110309384  0.00000000
## X13          0.05012370  0.031315284  0.00000000
## X14         -1.31681810 -1.192941934 -1.18867733
## X15         -0.28633018 -0.197201239 -0.02353272
## X16         -0.01020831  0.025689534  0.00000000
## X17          0.01383254  0.001036487  0.00000000
## X18         -0.08699664 -0.061251733  0.00000000
## X19          0.13672434  0.106198897  0.00000000
## X20         -1.14993373 -1.036652851 -0.99059896
```

Again, we see how the Ridge will not zero out a coefficient set but LASSO will.

Prediction mean squared error (MSE)

Now, let's compare the prediction MSE for our three models.

```
mean((pred_ols - y[test_vec])^2)
```

```
## [1] 1.053824
```

```
mean((pred_l2_cv - y[test_vec])^2)
```

```
## [1] 0.9452646
```

```
mean((pred_l1_cv - y[test_vec])^2)
```

```
## [1] 1.007394
```

We can see how the MSE is improved for regularized regression and that LASSO and Ridge largely in agreement.

Similarly, when lambda is prohibitively large we approach an intercept-only model

```
fit_ols_intercept <- lm(Y ~ 1, data=dat[train_vec,])

coef_intercept <- cbind(OLS = coef(fit_ols_intercept),
                        Ridge_L2 = coef(fit_l2_sweep, s = 10^5),
                        Ridge_L1 = coef(fit_l1_sweep, s = 10^5))
coef_intercept
```

```
## 21 x 3 sparse Matrix of class "dgCMatrix"
##                   OLS            s1        s1
## (Intercept) 0.3632427  3.632368e-01 0.3632427
## V1          0.3632427  4.169325e-05 .
## V2          0.3632427  8.956484e-06 .
## V3          0.3632427  3.339678e-05 .
## V4          0.3632427 -7.233409e-06 .
## V5          0.3632427 -2.923502e-05 .
## V6          0.3632427  2.851030e-05 .
## V7          0.3632427  2.734383e-06 .
## V8          0.3632427  4.895044e-06 .
## V9          0.3632427  4.115155e-06 .
## V10         0.3632427 -1.121531e-05 .
## V11         0.3632427  2.956202e-06 .
## V12         0.3632427 -6.091761e-06 .
## V13         0.3632427 -3.564070e-06 .
## V14         0.3632427 -4.777718e-05 .
## V15         0.3632427  6.817468e-06 .
## V16         0.3632427  6.509293e-06 .
## V17         0.3632427 -4.473759e-06 .
## V18         0.3632427  1.454444e-06 .
## V19         0.3632427  3.531605e-06 .
## V20         0.3632427 -3.538384e-05 .
```

We can also see the predictions are just terrible

```
pred_ols_intercept <- predict(fit_ols_intercept, newdata = dat[test_vec,])
mean((pred_ols_intercept - y[test_vec])^2)
```

```
## [1] 8.626318
```

At this stage, we should note this dataset in particular would tend to favor OLS if we added more data to the training set. If our sampling fraction were closer to 90% this would almost entirely favor OLS in terms of the MSE. But this is only because we have so many samples to choose from and a fairly informative dataset. This would not be the case with real world datasets or when our data is especially wide.

We can see what happens with wider datasets, by building models with only the first 10 observations. Now we see how OLS completely collapses but both regularized regressions will find values

```
fit_vif_ols <- lm(Y ~ ., dat[1:10,])
fit_vif_l1 <- glmnet(x[1:10,], y[1:10], alpha=0)
fit_vif_l2 <- glmnet(x[1:10,], y[1:10], alpha=1)


coef_wide <- cbind(OLS = coef(fit_vif_ols),
      Ridge_L2 = coef(fit_vif_l2, s=0.1),
      LASSO_L1 = coef(fit_vif_l1, s=0.1))

coef_wide
```

```
## 21 x 3 sparse Matrix of class "dgCMatrix"
##                     OLS          s1            s1
## (Intercept)   15.833565  0.87232540  0.8530542306
## V1          -159.824673  .          -0.0031086223
## V2            20.364433  .          -0.0162474069
## V3           -64.113457  0.04836284  0.0595702368
## V4            -4.507769 -0.50353361 -0.0626598457
## V5           188.124659 -0.14342495 -0.0759687201
## V6          -119.552068  .          -0.0416955421
## V7          -150.862726 -0.20354188 -0.0604634680
## V8          -122.147841  .           0.0005305579
## V9            63.063623  .          -0.0019866779
## V10                  NA  .           0.0117105106
## V11                  NA -0.41377572 -0.0936635964
## V12                  NA  0.48589410  0.0918983624
## V13                  NA  .           0.0120905264
## V14                  NA  .          -0.0498159451
## V15                  NA  .          -0.0115419195
## V16                  NA  .          -0.0404624536
## V17                  NA  .          -0.0552171118
## V18                  NA  .          -0.0174065814
## V19                  NA  0.13223663  0.0420557018
## V20                  NA -0.19288654 -0.0570310321
```

Conclusions

Now we’ve got a sense for the differences between regularized regression methods and OLS regression. Ridge (L2) regression will shrink coefficient estimates towards zero, but will always retain every variable. LASSO regression (L1) will shrink estimates all the way to zero, effectively removing them from the model.Remember: the goal for these models is not **interpretation** of predictor and outcome variables as with OLS, but **prediction** of outcome variables from new data. Regularized regression methods produce biased coefficient estimates in service of better prediction.