Unit 3 Lecture 3: Ridge regression

October 12, 2021

NOTE: This R demo has been updated since it was presented in class on October 12. Through the use of the `glmnetUtils` package, it is no longer necessary to separately construct `X` and `Y` to pass into `cv.glmnet`, or to scale the `X` matrix manually, or to remove the intercept manually. You can use `cv.glmnet` in much the same way you have been using `lm` and `glm`: by specifying a formula and supplying a data frame.

In this R demo, we will learn about the `glmnet` and `glmnetUtils` packages and how to run a cross-validated ridge regression using the `cv.glmnet()` function.

First, let's install the `glmnet` and `glmnetUtils` packages:

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

Next, we load the `glmnetUtils` package:

```
library(glmnetUtils)
```

Let's also source a function called `plot_glmnet` to help us plot our results:

```
source("../../functions/plot_glmnet.R")
```

We will be applying ridge regression to study the effect of 97 socioeconomic factors on violent crimes per capita based on data from 90 communities in Florida:

```
crime_data = read_csv("../../data/CrimeData_FL.csv")
crime_data
```

```
## # A tibble: 90 x 98
##    population household.size race.pctblack race.pctwhite race.pctasian
##         <dbl>          <dbl>         <dbl>         <dbl>         <dbl>
##  1      16023           2.63         13.8           83.9          1.42
##  2      29721           2.34          3.52          95.1          1.03
##  3      10205           2.46          1.06          97.4          1.04
##  4     124773           2.47         29.1           68.2          1.75
##  5      13024           2.25         31.3           67.2          0.5
##  6     280015           2.44         25.0           70.9          1.35
##  7      79443           2.94          3.48          93.1          2.12
##  8      16444           2.57          5.38          91.2          1.96
##  9      46194           2.28         20.1           77.7          0.63
## 10      14044           2.17          0.48          98.3          0.58
## # ... with 80 more rows, and 93 more variables: race.pcthisp <dbl>,
## #   age.pct12to21 <dbl>, age.pct12to29 <dbl>, age.pct16to24 <dbl>,
## #   age.pct65up <dbl>, pct.urban <dbl>, med.income <dbl>, pct.wage.inc <dbl>,
## #   pct.farmself.inc <dbl>, pct.inv.inc <dbl>, pct.socsec.inc <dbl>,
## #   pct.pubasst.inc <dbl>, pct.retire <dbl>, med.family.inc <dbl>,
## #   percap.inc <dbl>, white.percap <dbl>, black.percap <dbl>,
## #   indian.percap <dbl>, asian.percap <dbl>, hisp.percap <dbl>, ...
```

Let's split the data into training and testing, as usual:

```
set.seed(471)
train_samples = sample(1:nrow(crime_data), 0.8*nrow(crime_data))
crime_data_train = crime_data %>% filter(row_number() %in% train_samples)
crime_data_test = crime_data %>% filter(!(row_number() %in% train_samples))
```

Running a cross-validated ridge regression

We call `cv.glmnet` on `crime_data_train`:

```
ridge_fit = cv.glmnet(violentcrimes.perpop ~ .,   # formula notation, as usual
                      alpha = 0,                  # alpha = 0 for ridge
                      nfolds = 10,                # number of folds
                      data = crime_data_train)    # data to run ridge on
```

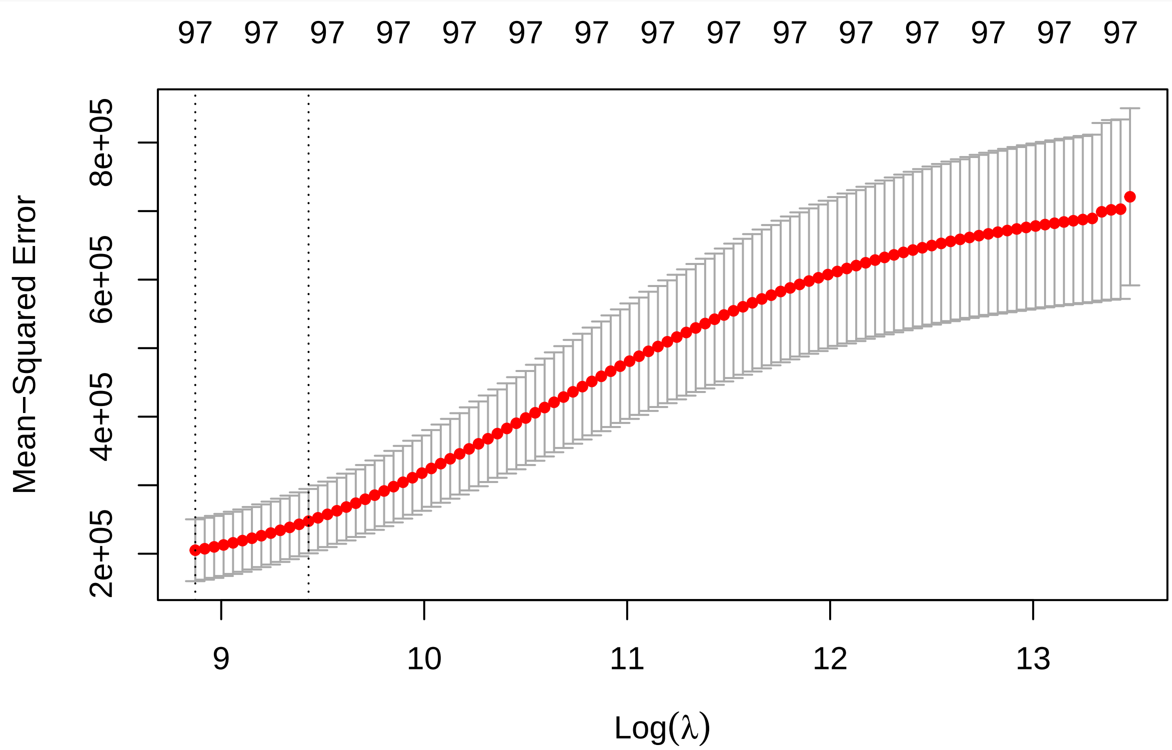
A few things to note:

- the sequence of penalty parameters is automatically chosen for you
- `alpha = 0` means "ridge regression" (we'll discuss other values of alpha next lecture)
- `nfolds` specifies the number of folds for cross-validation
- the columns of the matrix `X` are being standardized for you behind the scenes; there is no need to standardize yourself

Inspecting the results

The `glmnet` package has a very nice `plot` function to produce the CV plot:

```
plot(ridge_fit)
```



The `ridge_fit` object has several fields with information about the fit:

```r
# lambda sequence
head(ridge_fit$lambda)
```

```
## [1] 713062.9 680653.1 649716.3 620185.7 591997.3 565090.1
```

```r
# CV estimates
head(ridge_fit$cvm)
```

```
## [1] 720856.9 702831.8 701779.3 698965.2 689312.4 687654.4
```

```r
# CV standard errors
head(ridge_fit$cvsd)
```

```
## [1] 129230.7 130921.2 131034.4 129625.3 122211.3 121927.5
```

```r
# lambda achieving minimum CV error
ridge_fit$lambda.min
```

```
## [1] 7130.629
```

```r
# lambda based on one-standard-error rule
ridge_fit$lambda.1se
```

```
## [1] 12460.98
```

To get the fitted coefficients at the selected value of lambda:

```r
coef(ridge_fit, s = "lambda.1se") %>% head()
```

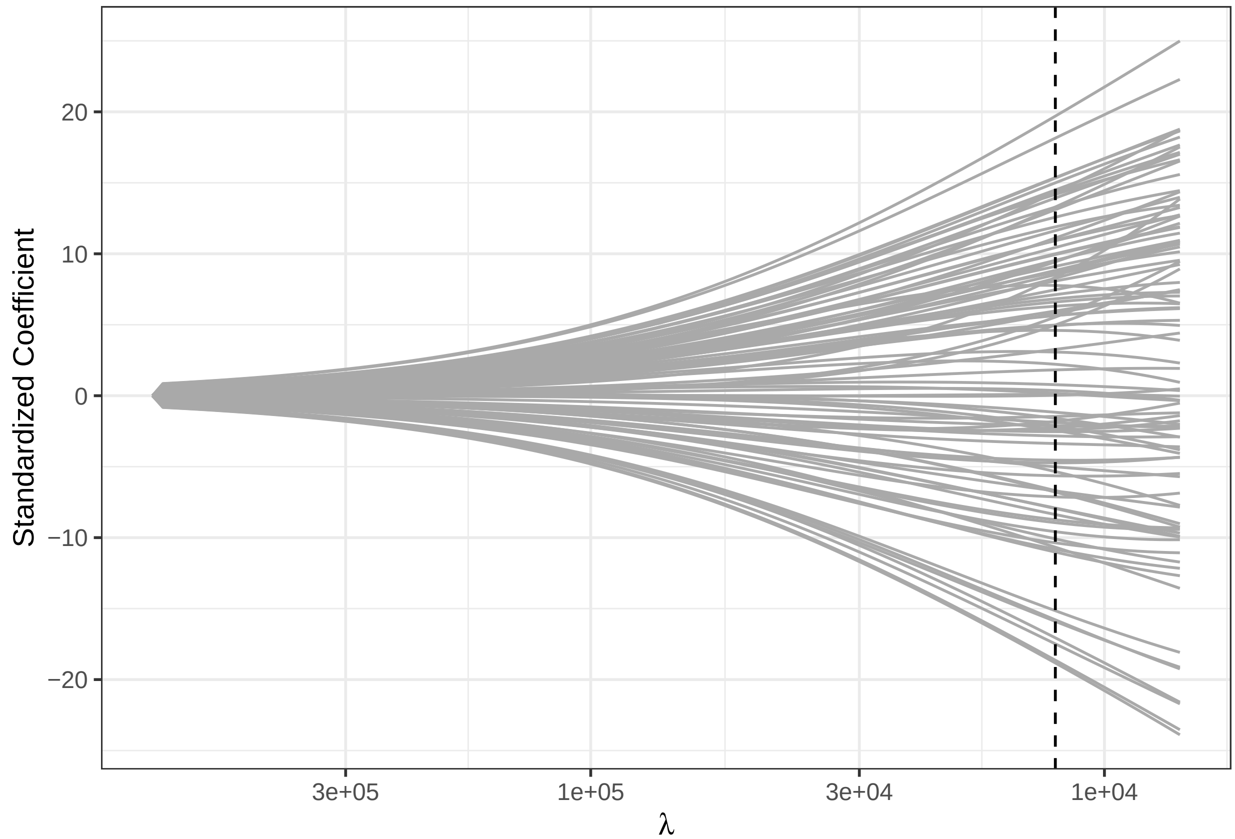
```
## 6 x 1 sparse Matrix of class "dgCMatrix"
##                          s1
## (Intercept)     4.335514e+03
## population      1.020056e-04
## household.size  2.871272e+00
## race.pctblack   1.172688e+00
## race.pctwhite  -1.201426e+00
## race.pctasian  -7.830098e+00
```

```r
coef(ridge_fit, s = "lambda.min") %>% head()
```

```
## 6 x 1 sparse Matrix of class "dgCMatrix"
##                          s1
## (Intercept)     5.226134e+03
## population      1.367863e-04
## household.size  1.337954e+00
## race.pctblack   1.424616e+00
## race.pctwhite  -1.464308e+00
## race.pctasian  -1.131950e+01
```

To visualize the fitted coefficients as a function of lambda, we can make a plot of the coefficients like we saw in class. To do this, we can use the `plot_glmnet` function, which by default shows a dashed line at the lambda value chosen using the one-standard-error rule:

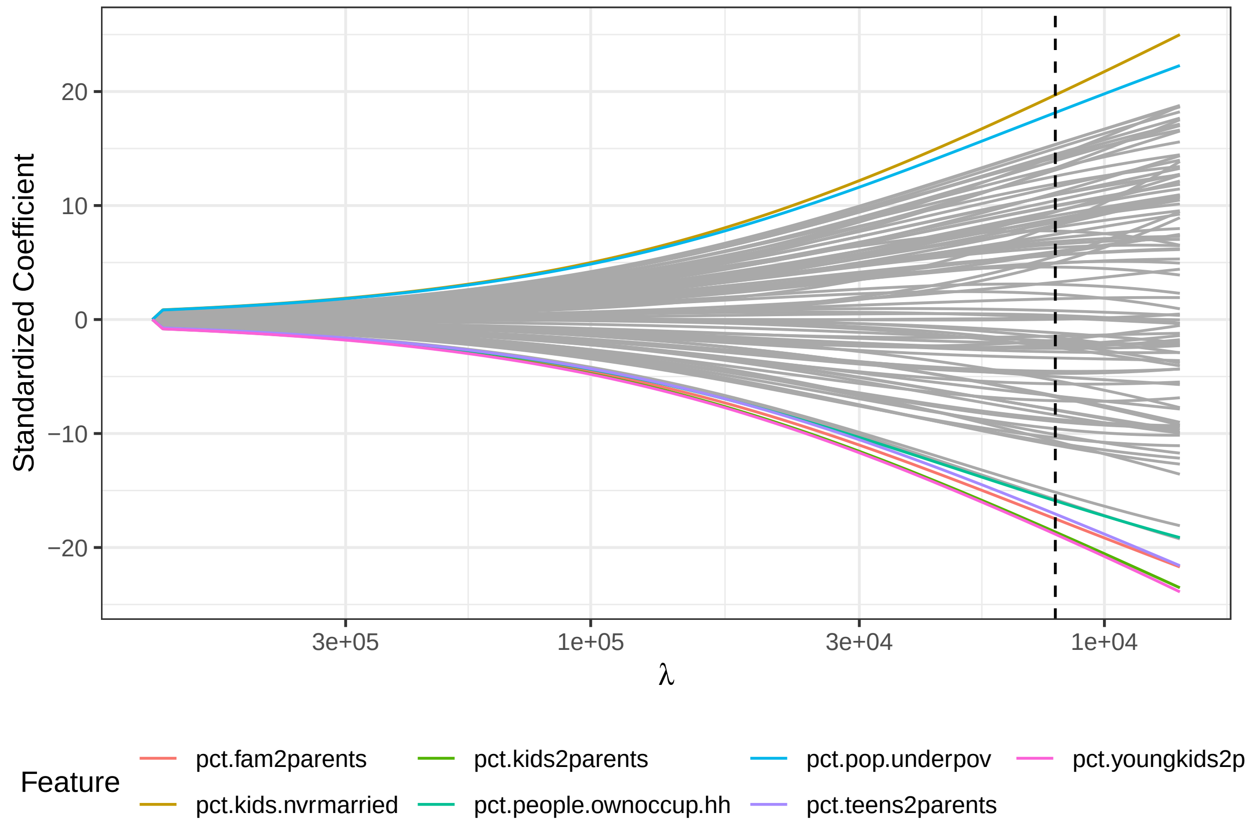
```r
plot_glmnet(ridge_fit, crime_data_train) # NOTE: MUST PASS IN THE DATA AS WELL
```




```r
                                          #  AS THE FIT OBJECT
```

If we want to annotate the features with the top few coefficients, we can use the `features_to_plot` argument:

```r
plot_glmnet(ridge_fit, crime_data_train, features_to_plot = 7)
```



To interpret these coefficient estimates, recall that they are for the *standardized* features.

Making predictions

To make predictions on the test data, we can use the `predict` function (which we've seen before):

```
ridge_predictions = predict(ridge_fit,
                            newdata = crime_data_test,
                            s = "lambda.1se") %>% as.numeric()
ridge_predictions
```

```
##  [1] 1728.4273 1342.5847 1040.6852  771.8934  681.1764  700.3370  981.5614
##  [8]  814.8565  841.2080  749.3758  555.5066 1381.7639 1251.6341 1258.0162
## [15] 1442.3838  710.4379  703.6399  711.3818
```

We can evaluate the root-mean-squared-error as before:

```
RMSE = sqrt(mean(ridge_predictions - crime_data_test$violentcrimes.perpop)^2)
RMSE
```

```
## [1] 175.11
```

Ridge logistic regression

We can also run a ridge-penalized logistic regression. Let's try it out on `default_data`.

```
# load data, convert default to binary
default_data = ISLR2::Default %>%
  as_tibble() %>%
```

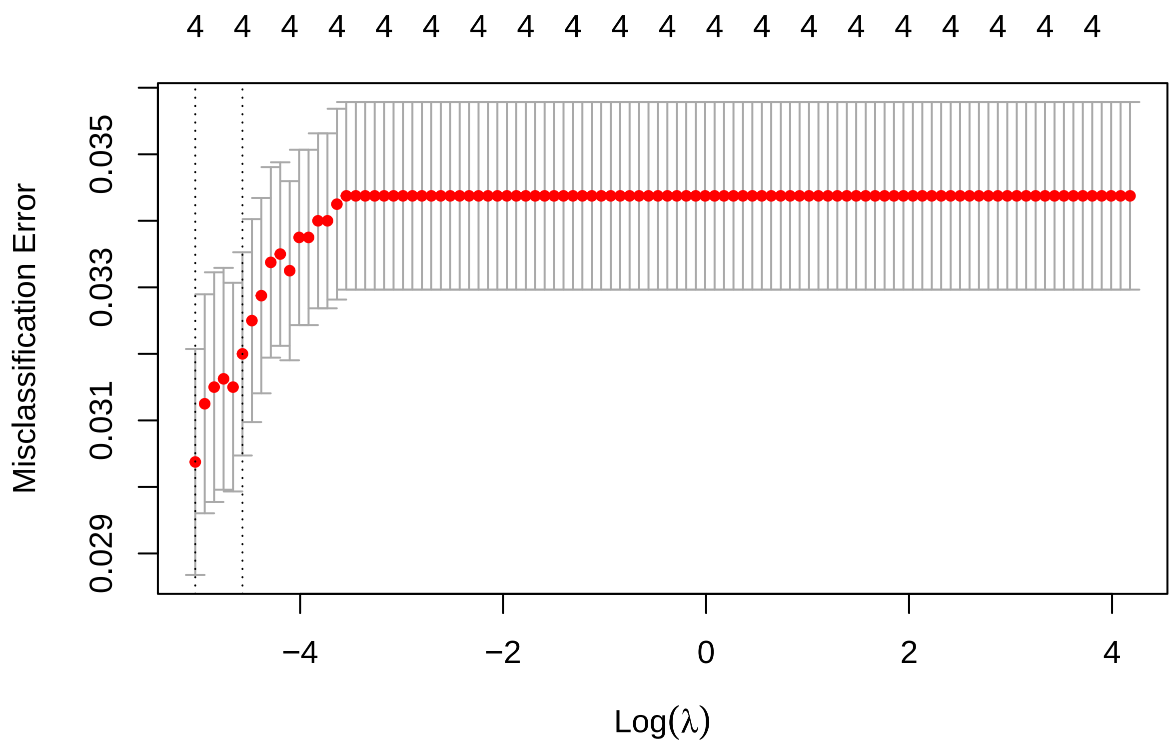
```
  mutate(default = as.numeric(default == "Yes"))
# split into train and test
set.seed(471)
train_samples = sample(1:nrow(default_data), 0.8*nrow(default_data))
default_train = default_data %>% filter(row_number() %in% train_samples)
default_test = default_data %>% filter(!(row_number() %in% train_samples))
```

To run the logistic ridge regression, we call `cv.glmnet` as before, adding the argument `family = binomial` to specify that we want to do a logistic regression and the argument `type.measure = "class`: to specify that we want to use the misclassification error during cross-validation.

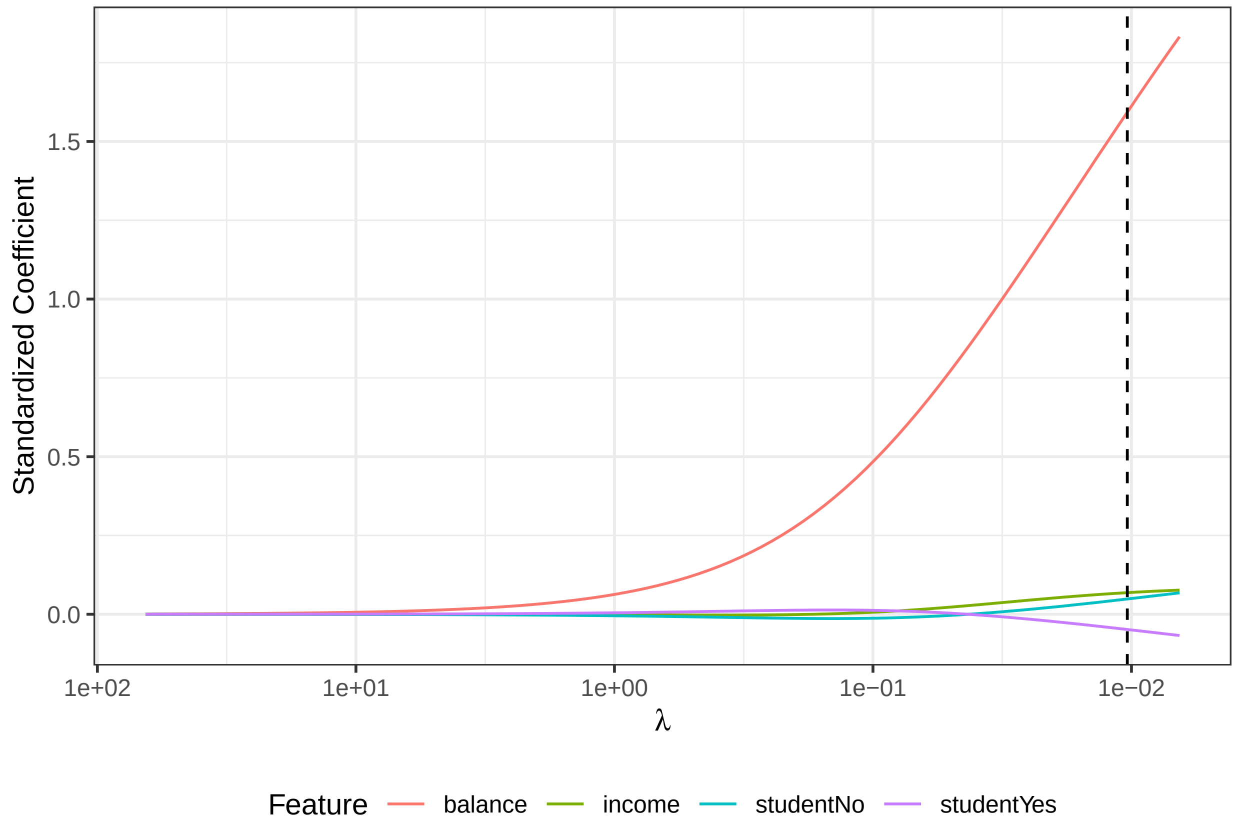
```
ridge_fit = cv.glmnet(default ~ .,              # formula notation, as usual
                      alpha = 0,                # alpha = 0 means ridge
                      nfolds = 10,              # number of CV folds
                      family = "binomial",      # to specify logistic regression
                      type.measure = "class",   # use misclassification error in CV
                      data = default_train)     # train on default_train data
```

We can then take a look at the CV plot and the trace plot as before:

```
plot(ridge_fit)
```




```
plot_glmnet(ridge_fit, default_train, features_to_plot = 4)
```



To predict using the fitted model, we can use the `predict` function again, this time specifying `type = "response"` to get the predictions on the probability scale (as opposed to the log-odds scale).

```
probabilities = predict(ridge_fit,                  # fit object
                        newdata = default_test,     # new data to test on
                        s = "lambda.1se",           # which value of lambda to use
                        type = "response") %>%      # to output probabilities
  as.numeric()                                      # convert to vector
head(probabilities)
```

```
## [1] 0.0006226396 0.0289927850 0.0022106111 0.0034241229 0.0018848093
## [6] 0.0065551004
```

We can threshold the probabilities to get binary predictions as we did with regular logistic regression.