Ridge and Lasso Regression

Laura Cline

24/10/2021

Contents


```
library(glmnet) #glmnet() function for ridge/lasso
```

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

```
## Loaded glmnet 4.1-2
```

```
library(ISLR) # Hitters data
library(pls) #pcr() function for principal components regression
```

```
##
## Attaching package: 'pls'
```

```
## The following object is masked from 'package:stats':
##
##     loadings
```

We will use the `glmnet` package in order to perform ridge regression and the lasso. The main function in this package is `glmnet()`, which can be used to fit ridge regression models, lasso models, and more. This function has slightly different syntax from other model-fitting functions that we have encountered thus far. In particular, we must pass in an `x` matrix as well as a `y` vector, and we do not use the `y ~ x` syntax. We will now perform ridge regression and the lasso in order to predict `Salary` on the `Hitters` data. Before proceeding, ensure that the missing values have been removed from the data.

```
# Prepare the Hitters data as done in the previous lab
hitters <- Hitters
hitters <- na.omit(hitters)
attach(hitters)
colnames(hitters) <- tolower(colnames(hitters))
```

The `model.matrix()` function is particularly useful for creating `x`; not only does it produce a matrix corresponding to the 19 predictors but it also automatically transforms any qualitative variables into dummy variables. The latter property is important because `glmnet()` can only take numerical, quanitative inputs.

Ridge Regression

The `glmnet()` function has an alpha argument that determines what type of model is fit. If `alpha=0` then a ridge regression model is fit, and if `alpha=1` then a lasso model is fit. We first fit a ridge regression model.

Before continuing, take a look at the `glmnet()` vignette to get an idea of what we need beforehand.

- `x` - input matrix containing rows of observations
- `y` - response variable matrix
- `lamba` - a user created decreasing sequence of lambda values

```
# Lambda will be a vector of length 100, ranging from 10^10 to 10^-2
x <- model.matrix(salary~., hitters)[,-1]
y <- hitters$salary
grid <-10^seq(10,-2,length=100)
ridge.mod <- glmnet(x,y, alpha=0, lambda=grid)
```

Because we supplied a 100 lambda, we have got a coefficient matrix that is 20x100. We expect that the coefficients of models with larger lambdas will be much smaller than those with smaller lambdas. Remember, the sequence is decreasing so the further along the sequence the larger the coefficients.

By default, the `glmnet()` function performs ridge regression for an automatically selected range of λ values. However, here we have chosen to implement the function over a grid of values ranging from $\lambda = 10^{10}$ to $\lambda = 10^{-2}$, especially covering the full range of scenarios from the null model containing only the intercept, to the least squares fit. As we will see, we can also compute model fits for a particular value of λ that is not one of the original `grid` values. Note that by default, the `glmnet()` function standardizes the variables so that they are on the same scale. To turn off this default setting, use the argument `standardize=FALSE`.

Associated with each value of λ is a vector of ridge regression coefficients, stored in a matrix that can be accessed by `coef()`. In this case, it is a 20x100 matrix, with 20 rows (one for each predictor, plus an intercept) and 100 columns (one for each value of λ).

```
dim(coef(ridge.mod))
```

```
## [1]  20 100
```

We expect the coefficient estimates to be smaller, in terms of l_2 norm, when a large value of λ is used, as compared to when a small value of λ is used. These are the coefficients when $\lambda = 11{,}498$, along with their l_2 norm:

```
# Find the 50th and 100th lambdas, along with the coefficients associated with that model and their l_2
ridge.mod$lambda[50]
```

```
## [1] 11497.57
```

```
coef(ridge.mod)[,50]
```

```
##   (Intercept)         atbat          hits         hmrun          runs
## 407.356050200   0.036957182   0.138180344   0.524629976   0.230701523
##           rbi         walks         years        catbat         chits
##   0.239841459   0.289618741   1.107702929   0.003131815   0.011653637
##        chmrun         cruns          crbi        cwalks       leagueN
##   0.087545670   0.023379882   0.024138320   0.025015421   0.085028114
##     divisionW       putouts       assists        errors    newleagueN
##  -6.215440973   0.016482577   0.002612988  -0.020502690   0.301433531
```

```
sqrt(sum(coef(ridge.mod)[-1,50])^2)
```

```
## [1] 3.08789
```

In contrast, here are the coefficients when $\lambda = 0.01$, along with their l_2 norm. Note that much larger l_2 norm of the coefficients associated with this smaller value of λ.

```
ridge.mod$lambda[100]
```

```
## [1] 0.01
```

```
coef(ridge.mod)[,100]
```

```
##   (Intercept)         atbat          hits         hmrun          runs
##  164.11321606   -1.97386151    7.37772270    3.93660219   -2.19873625
##           rbi         walks         years        catbat         chits
##    -0.91623008    6.20037718   -3.71403424   -0.17510063    0.21132772
##        chmrun         cruns          crbi        cwalks       leagueN
##     0.05629004    1.36605490    0.70965516   -0.79582173   63.40493257
##     divisionW       putouts       assists        errors    newleagueN
## -117.08243713    0.28202541    0.37318482   -3.42400281  -25.99081928
```

```
sqrt(sum(coef(ridge.mod)[-1,100])^2)
```

```
## [1] 72.35287
```

```
# Just for fun, let's practice writing a function to extract this information for any lambda
shrinkage_coef <- function(glmnet_mod, ld) {
  coef_names <- names(coef(glmnet_mod)[,50])
  return(list(print(paste("The Lambda value is:",glmnet_mod$lambda[ld])),
         print(paste("The coefficient for",coef_names,"is:", coef(glmnet_mod)[,ld])),
         print(paste("The l_2 norm of these coefficients is:", sqrt(sum(coef(glmnet_mod)[-1,ld])^2)))
         ))
}
```

We now split the samples into a training set and a test set in order to estimate the test error of ridge regression and the lasso. There are two common ways to randomly split a dataset. The first is to produce a random vector of `TRUE`, `FALSE` elements and select the observations corresponding to `TRUE` for the training data. The second is to randomly choose a subset of numbers between 1 and n; these can then be used as the indices for the training observations. The two approaches work equally well. We used the former method above. Here we demonstrate the latter approach.

We first set a random seed so that the results obtained will be reproducible.

```
# Now that we've got a practice run down, let's split our sample to do some testing
set.seed(1)
train <- sample(1:nrow(x), nrow(x)/2)
test <- (-train)
y.test <- y[test]
```

There are two common ways to randomly split a dataset. You can produce a random logical vector (`TRUE`/`FALSE`) and select observations corresponding to `TRUE` for the training data. Alternatively, randomly choose a subset of numbers between 1 and n, which can then be used as the indices for the training data. We do (and have been doing) the former in previous labs; this lab makes use of the latter.

Next we fit a ridge regression model on the training set, and evaluate its MSE on th test set, using $\lambda = 4$. Note the use of the `predict()` function again. This time we get predictions for a test set, by replacing `type="coefficients"` with the `newx` argument.

```
# Fit a ridge regression on the training set and evaluate its MSE on the test, using lambda = 4
ridge_mod <- glmnet(x[train,], y[train], alpha=0, lambda=grid, thresh=1e-12)
ridge_pred <- predict(ridge_mod, s=4, newx=x[test,]) #s option sets the lambda value, newx specifies ne
mean((ridge_pred - y.test)^2) #MSE
```

```
## [1] 142199.2
```

The test MSE is 142199. Note that if we had instead simply fit a model with just an intercept, we would have predicted each test observation using the mean of the training observations. In that case, we could compute the test set MSE like this:

```
# Compare to the test MSE when lambda is extremely large (coefficients are approximately 0)
ridge_pred <- predict(ridge_mod, s=1e10, newx=x[test,])
mean((ridge_pred - y.test)^2)
```

```
## [1] 224669.8
```

We could also get the same result by fitting a ridge regression model with a *very* large value of λ. Bote that `1e10` means 10^{10}.

```
# Finally, let's check if the ridge regression gives us better results than the least squares option (l
ridge_pred <- predict(ridge_mod, s=0, newx=x[test,]) #the exact option allow us to specify that lambda
mean((ridge_pred - y.test)^2)
```

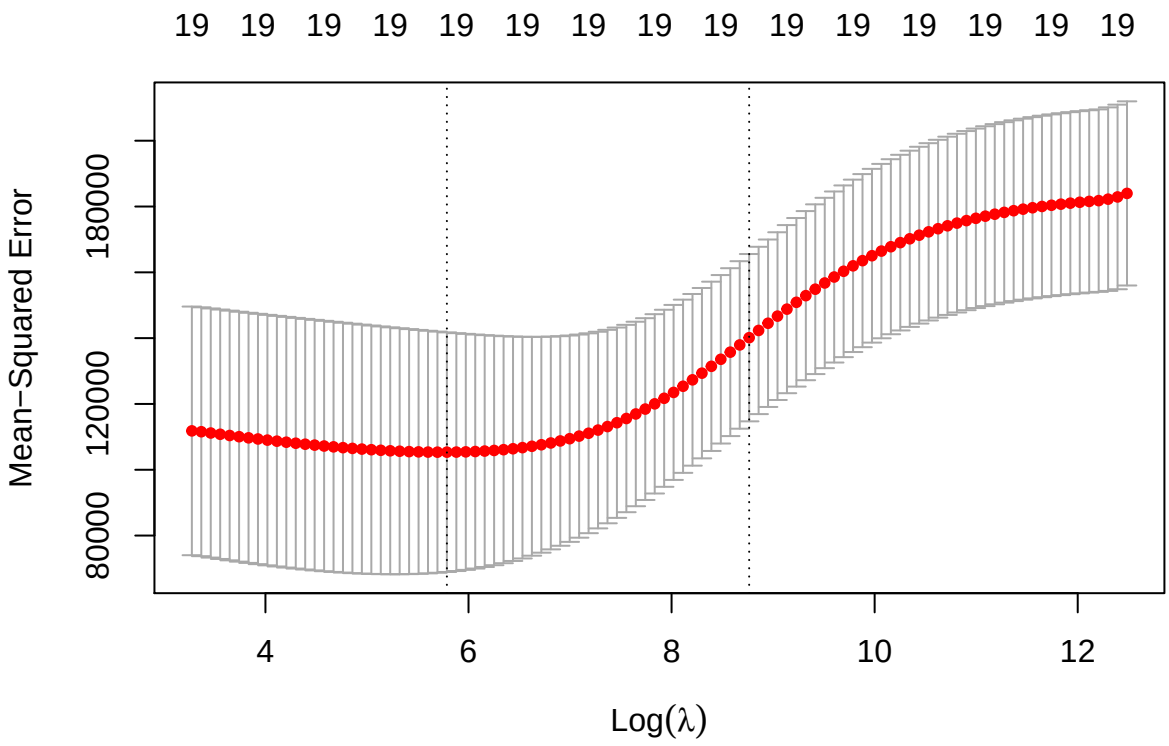
```
## [1] 167789.8
```

So fitting a ridge regression with $\lambda = 4$ leads to a much lower test MSE than fitting a model with just an intercept. We now check whether there is any benefit to performing ridge regression with $\lambda = 4$ instead of just performing least squares regression. Recall that least squares is simply ridge regression with $\lambda = 0$.

In general, we want to fit a unpenalized least squares model, then we should use the `lm()` function, since that function provides more useful outputs, such as standard errors and p-values for the coefficients.

In general, instead of arbitrarily choosing $\lambda = 4$, it would be better to use cross-validation to choose the tuning parameter λ. We can do this using the built-in cross-validation function, `cv.glmnet()`. By default, the function performs ten-fold cross validation, through this can be changed using the argument `nfolds`. Note that we set a random seed first so our results will be reproducible, since the choice of the cross-validation folds is random.

```
# Instead of arbitrarily choosing a lambda value a prior, it is better to use cross-validation to find
# This can be done using cv.glmnet(), which conducts 10-fold CV and can be increased to n-folds with th
set.seed(1)
cv_out <- cv.glmnet(x[train,], y[train], alpha=0)
plot(cv_out)
```




```
best_lam <- cv_out$lambda.min
best_lam
```

```
## [1] 326.0828
```

Therefore, we see that the value of λ that results in the smallest cross-validation 326. What is the test MSE associated with the value of λ?

```
# What is the test MSE associated with this lambda?
ridge_pred <- predict(ridge_mod, s=best_lam, newx=x[test,])
mean((ridge_pred - y.test)^2)
```

```
## [1] 139856.6
```

This represents a further improvement over the test MSE that we got using $\lambda = 4$. Finally, we refit our ridge regression model on the full dataset, using the value of λ chosen by cross-validation, and examine the coefficient estimates.

```
# Finally we can run ridge regression on the entire dataset, now that we have found the best value for
out <- glmnet(x, y, alpha=0)
predict(out, type="coefficients", s=best_lam)[1:20,]
```

```
##  (Intercept)        atbat         hits        hmrun         runs          rbi
##  15.44383135   0.07715547   0.85911581   0.60103107   1.06369007   0.87936105
##        walks        years       catbat        chits       chmrun        cruns
##   1.62444616   1.35254780   0.01134999   0.05746654   0.40680157   0.11456224
##         crbi       cwalks      leagueN    divisionW      putouts      assists
##   0.12116504   0.05299202  22.09143189 -79.04032637   0.16619903   0.02941950
##       errors   newleagueN
##  -1.36092945   9.12487767
```

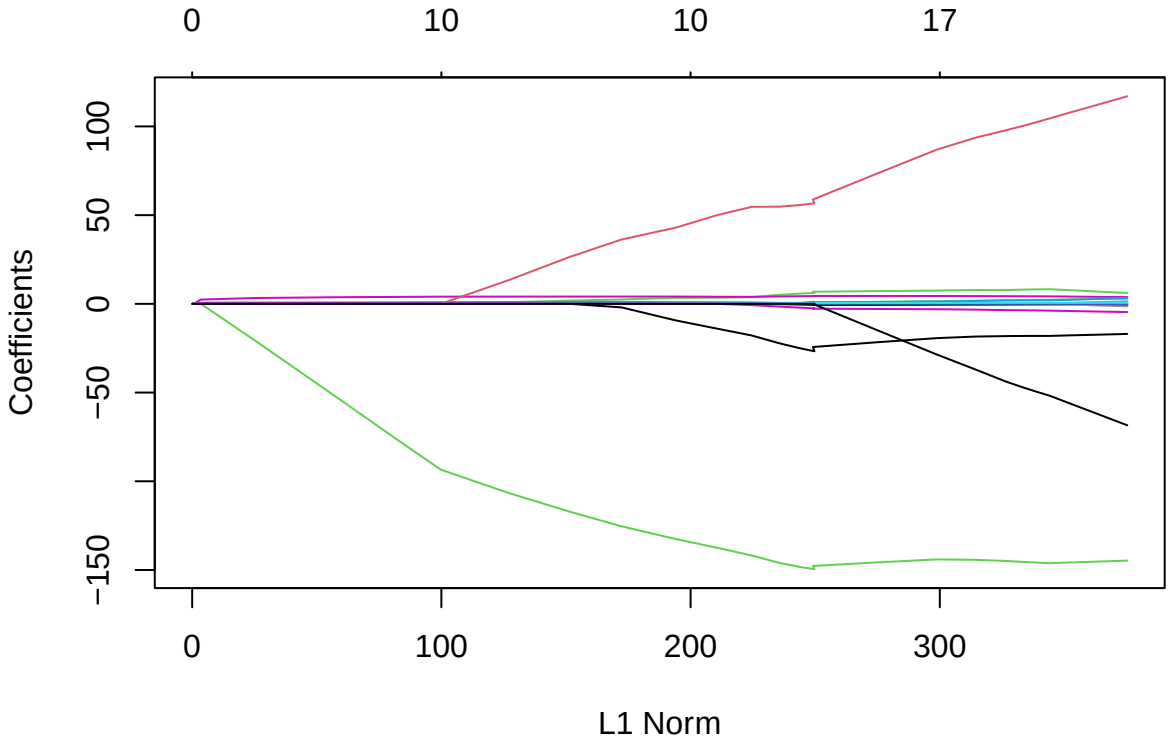
As expexted, none of the coefficients are zero - ridge regression does not perform variable selection!

The Lasso

We saw that ridge regression with a wise choice of λ can outperform least squares as well as the null model on the `Hitters` dataset. We now ask whether the lasso can yield either a more accurate or a more interpretable model than ridge regression. In order to fit a lasso model, we once again use the `glmnet()` function; however, this time we use the argument `alpha=1`. Other than that change, we proceed just as we did in fitting a ridge model.

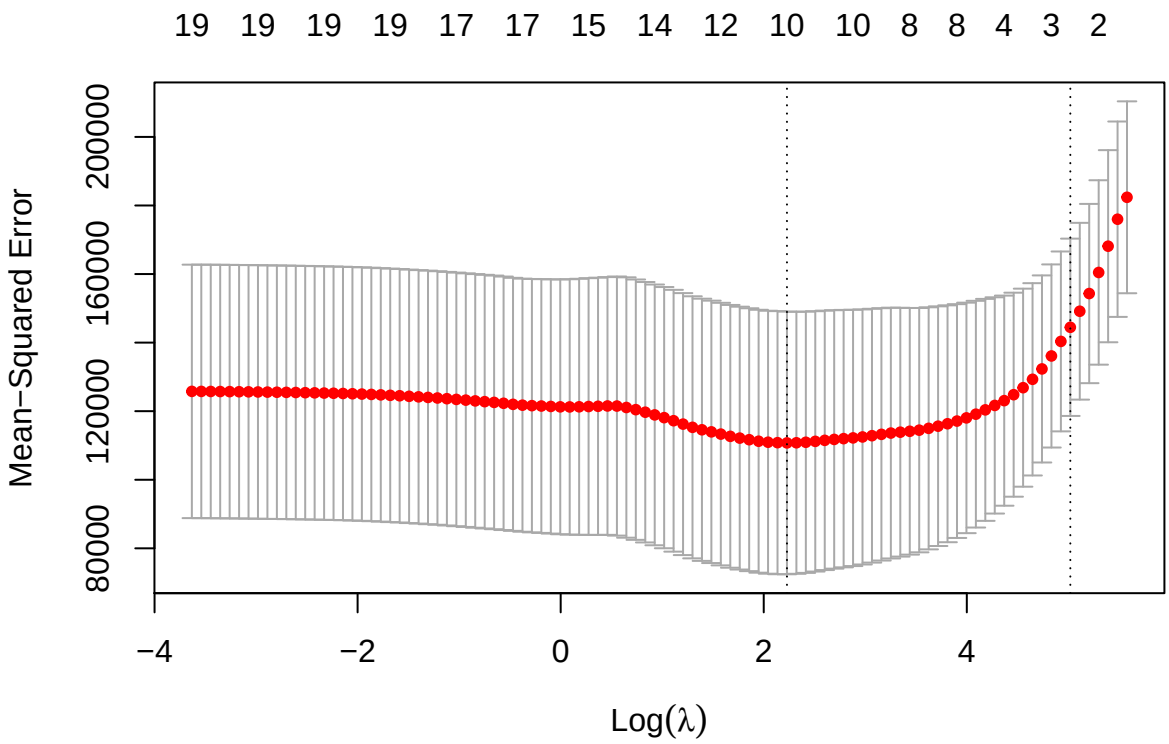
```
# Fit the lasso model and observe how some of the coefficients are exactly 0
lasso_mod <- glmnet(x[train,], y[train], alpha=1, lambda=grid)
plot(lasso_mod)
```

```
## Warning in regularize.values(x, y, ties, missing(ties), na.rm = na.rm):
## collapsing to unique 'x' values
```



We can see from the coefficient plot that depending on the choice of tuning parameter, some of the coefficients will be exactly equal to zero. We now perform cross-validation and compute the associated test error.

```
# Perform CV and compute test errors
set.seed(1)
cv_out <- cv.glmnet(x[train,], y[train], alpha=1)
plot(cv_out)
```




```
best_lam <- cv_out$lambda.min
lasso_pred <- predict(lasso_mod, s=best_lam, newx=x[test,])
mean((lasso_pred - y.test)^2)
```

```
## [1] 143673.6
```

This is substantially lower than the test set MSE of the null model and of least squares, and very similiar to the test MSE of ridge regression with λ chosen by cross-validation.

However, the lasso has a substantial advantage over ridge regression in that the result coefficient estimates are sparse. Here we see that 8 of the 19 coefficient estimates are exactly zero. So the lasso model with λ chosen by cross-validation contains only seven variables.

```
# Fit the lasso over the entire dataset
out <- glmnet(x, y, alpha=1, lambda=grid)
lasso_coef <- predict(out, type="coefficients", s=best_lam)[1:20,]
lasso_coef
```

```
##   (Intercept)         atbat          hits         hmrun          runs
##    1.27479059   -0.05497143    2.18034583    0.00000000    0.00000000
##           rbi         walks         years        catbat         chits
##    0.00000000    2.29192406   -0.33806109    0.00000000    0.00000000
##        chmrun         cruns          crbi        cwalks       leagueN
##    0.02825013    0.21628385    0.41712537    0.00000000   20.28615023
##     divisionW       putouts       assists        errors    newleagueN
## -116.16755870    0.23752385    0.00000000   -0.85629148    0.00000000
```

```
lasso_coef[lasso_coef!=0]
```

```
##   (Intercept)         atbat          hits         walks         years
##    1.27479059   -0.05497143    2.18034583    2.29192406   -0.33806109
##        chmrun         cruns          crbi       leagueN     divisionW
##    0.02825013    0.21628385    0.41712537   20.28615023 -116.16755870
```

```
##       putouts        errors
##    0.23752385   -0.85629148
```

The test MSE for the lasso is very similar to the ridge regression, but it does have a minor advantage: 8 of the 19 coefficients in the lasso model are exactly zero.

Principal Components Regression (PCR)

Principal components regression (PCR) can be performed using the `pcr()` function, which is part of the `pls` library. We now apply PCR to the `Hitters` data, in order to predict `salary`. Afain, ensure that the missing values have been removed from the data.

```
set.seed(2)
pcr.fit <- pcr(salary ~ ., data=hitters, scale=TRUE, validation="CV")
```

The syntax for the `pcr()` function is similiar to that for `lm()`, with a few additional options. Setting `scale=TRUE` has the effect of *standardizing* each predictor, prior to generating the principal components, so that the scale on which each variable is measured will not have an effct. Setting `validation="CV"` causes `pcr()` to compute the ten-fold cross-validation error for each possible value of M, the number of principal components used. The resulting fit can be examined using `summary()`.

```
summary(pcr.fit)
```

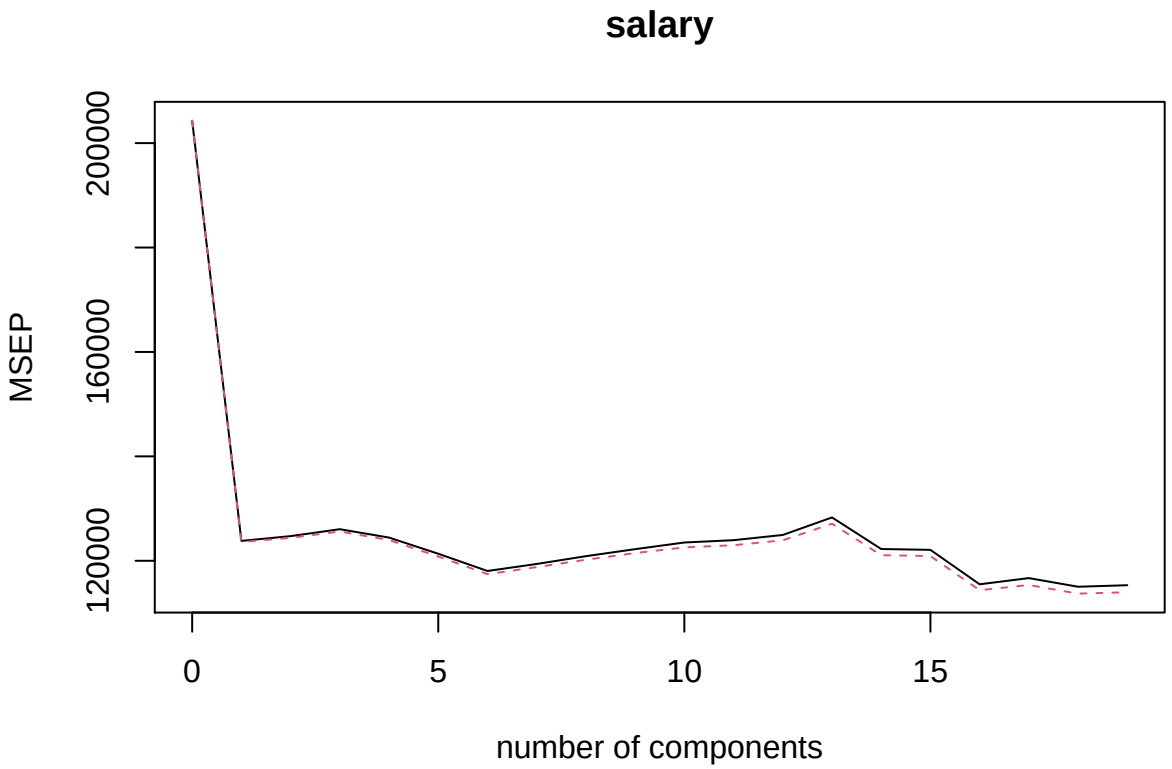
```
## Data:    X dimension: 263 19
##  Y dimension: 263 1
## Fit method: svdpc
## Number of components considered: 19
##
## VALIDATION: RMSEP
## Cross-validated using 10 random segments.
##        (Intercept)  1 comps  2 comps  3 comps  4 comps  5 comps  6 comps
## CV             452    351.9    353.2    355.0    352.8    348.4    343.6
## adjCV          452    351.6    352.7    354.4    352.1    347.6    342.7
##        7 comps  8 comps  9 comps  10 comps  11 comps  12 comps  13 comps
## CV       345.5    347.7    349.6     351.4     352.1     353.5     358.2
## adjCV    344.7    346.7    348.5     350.1     350.7     352.0     356.5
##        14 comps  15 comps  16 comps  17 comps  18 comps  19 comps
## CV        349.7     349.4     339.9     341.6     339.2     339.6
## adjCV     348.0     347.7     338.2     339.7     337.2     337.6
##
## TRAINING: % variance explained
##         1 comps  2 comps  3 comps  4 comps  5 comps  6 comps  7 comps  8 comps
## X         38.31    60.16    70.84    79.03    84.29    88.63    92.26    94.96
## salary    40.63    41.58    42.17    43.22    44.90    46.48    46.69    46.75
##         9 comps  10 comps  11 comps  12 comps  13 comps  14 comps  15 comps
## X         96.28     97.26     97.98     98.65     99.15     99.47     99.75
## salary    46.86     47.76     47.82     47.85     48.10     50.40     50.55
##         16 comps  17 comps  18 comps  19 comps
## X          99.89     99.97     99.99    100.00
## salary     53.01     53.85     54.61     54.61
```

The CV score is provided for each possible number of components, ranging from M = 0 onwards. Note that `pcr()` reports the *root mean squared error*; in order to obtain the usual MSE, we must square this quantity. For instance, a root mean squared error of 352.8 corresponds to an MSE of $352.8^2 = 124{,}468$.

One cal also plot the cross-validation scores using the `validationplot()` function. Using `val.type="MSEP"`

will cause the cross-validation MSE to be plotted.

```
validationplot(pcr.fit, val.type="MSEP")
```

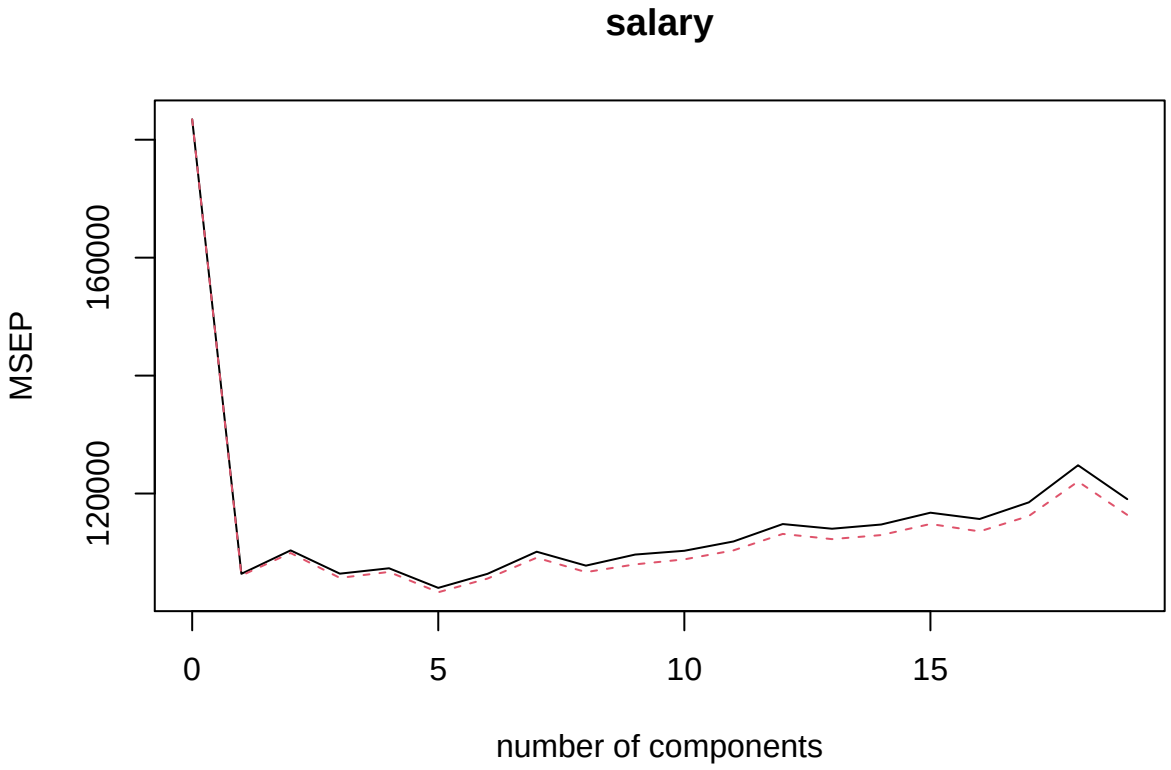


We see that the smallest cross-validation error occurs when M = 16 components are used. This is barely fewer than M = 19, which amounts to simply performing least squares, because when all of the components are used in PCR no dimension reduction occurs. However, from the plot we also see that the cross-validation error is rougly the same when only one component is included in the model. This suggests that a model that uses just a small number of components might suffice.

The `summary()` function also provides the *percentage of variance explained* in the predictors and in the response using different numbers of components. Briefly, we can think of this as the amount of information about the predictors or the response that is captured using M principal components. For example, setting M = 1 only captures 38.31% of all the variance, or information, in the predictors. In contrast, using M = 6 increases the value to 88.63%. If we were to use all M = p = 19 components, this would increase to 100%.

We now perform PCR on the training data and evaluate its test set performance.

```
set.seed(1)
pcr.fit = pcr(salary ~ ., data=hitters, subset=train, scale=TRUE, validation ="CV")
validationplot(pcr.fit,val.type="MSEP")
```



Now we find that the lowest cross-validation errors occurs when M = 5 components are used. We compute the test MSE as follows:

```
pcr.pred = predict(pcr.fit, x[test,], ncomp=7)
mean((pcr.pred = y.test)^2)
```

```
## [1] 516156.5
```

The test set MSE is competitive with the results obtained using ridge regression and the lasso. However, as a result of the way PCR is implemented, the final model is more difficult to interpret because it does not perform any kind of variable selection or even directly produce coefficient estimates.

Finally, we fit PCR on the full dataset, using M = 7, the number of components identified by cross-validation.

```
pcr.fit = pcr(y~x, scale=TRUE, ncomp=7)
summary(pcr.fit)
```

```
## Data:    X dimension: 263 19
##  Y dimension: 263 1
## Fit method: svdpc
## Number of components considered: 7
## TRAINING: % variance explained
##    1 comps  2 comps  3 comps  4 comps  5 comps  6 comps  7 comps
## X    38.31    60.16    70.84    79.03    84.29    88.63    92.26
## y    40.63    41.58    42.17    43.22    44.90    46.48    46.69
```

Partial Least Squares (PLS)

We implement partial least squares (PLS) using the `plar()` function, also in the `pls` library. The syntax is just like that of the `pcr()` function.

```
set.seed(1)
pls.fit = plsr(salary ~ ., data=hitters, subset=train, scale=TRUE, validation="CV")
summary(pls.fit)
```

```
## Data:    X dimension: 131 19
##  Y dimension: 131 1
## Fit method: kernelpls
## Number of components considered: 19
##
## VALIDATION: RMSEP
## Cross-validated using 10 random segments.
##        (Intercept)  1 comps  2 comps  3 comps  4 comps  5 comps  6 comps
## CV           428.3    325.5    329.9    328.8    339.0    338.9    340.1
## adjCV        428.3    325.0    328.2    327.2    336.6    336.1    336.6
##        7 comps  8 comps  9 comps  10 comps  11 comps  12 comps  13 comps
## CV       339.0    347.1    346.4     343.4     341.5     345.4     356.4
## adjCV    336.2    343.4    342.8     340.2     338.3     341.8     351.1
##        14 comps  15 comps  16 comps  17 comps  18 comps  19 comps
## CV        348.4     349.1     350.0     344.2     344.5     345.0
## adjCV     344.2     345.0     345.9     340.4     340.6     341.1
##
## TRAINING: % variance explained
##         1 comps  2 comps  3 comps  4 comps  5 comps  6 comps  7 comps  8 comps
## X         39.13    48.80    60.09    75.07    78.58    81.12    88.21    90.71
## salary    46.36    50.72    52.23    53.03    54.07    54.77    55.05    55.66
##         9 comps  10 comps  11 comps  12 comps  13 comps  14 comps  15 comps
## X         93.17     96.05     97.08     97.61     97.97     98.70     99.12
## salary    55.95     56.12     56.47     56.68     57.37     57.76     58.08
##         16 comps  17 comps  18 comps  19 comps
## X          99.61     99.70     99.95    100.00
## salary     58.17     58.49     58.56     58.62
```

The lowest cross-validation error occurs when only M = 2 partial least squares directions are used. We now evaluate the corresponding test set MSE.

```
pls.pred = predict(pls.fit, x[test,], ncomp=2)
mean((pls.pred - y.test)^2)
```

```
## [1] 145367.7
```

The test MSE is comparable to, but slightly higher than, the test MSE obtained using ridge regression, the lasso, and PCR.

Finally, we perform PLS using the full dataset, using M = 2, the number of components identified by cross-validation.

```
pls.fit = plsr(salary ~ ., data=hitters, scale=TRUE, ncomp=2)
summary(pls.fit)
```

```
## Data:    X dimension: 263 19
##  Y dimension: 263 1
## Fit method: kernelpls
## Number of components considered: 2
## TRAINING: % variance explained
##         1 comps  2 comps
## X         38.08    51.03
## salary    43.05    46.40
```

Notice that the percentage of variance in `Salary` that the two-component PLS fit explains, 46.40% is almost as much as that explained using the final seven-component model PCR fit, 46.69%. This is because PCR only attempts to maximize the amount of variance explained in the predictors, while PLS searches for directions that explain variance in both the predictors and the response.