```
     CRuns          CRBI       CWalks     LeagueN   DivisionW
     1.455         0.785       -0.823      43.112    -111.146
   PutOuts       Assists
     0.289         0.269
```

6.6 Lab 2: Ridge Regression and the Lasso

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 this book. 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, as described in Section 6.5.

`glmnet()`

```
> x=model.matrix(Salary~.,Hitters)[,-1]
> y=Hitters$Salary
```

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, quantitative inputs.

6.6.1 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.

```
> library(glmnet)
> grid=10^seq(10,-2,length=100)
> ridge.mod=glmnet(x,y,alpha=0,lambda=grid)
```

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 $\lambda = 10^{-2}$, essentially 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 20×100

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 much 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:

```
> ridge.mod$lambda[50]
[1] 11498
> coef(ridge.mod)[,50]
(Intercept)       AtBat        Hits       HmRun        Runs
    407.356       0.037       0.138       0.525       0.231
        RBI       Walks       Years      CAtBat       CHits
      0.240       0.290       1.108       0.003       0.012
     CHmRun       CRuns        CRBI      CWalks     LeagueN
      0.088       0.023       0.024       0.025       0.085
  DivisionW     PutOuts     Assists      Errors  NewLeagueN
     -6.215       0.016       0.003      -0.021       0.301
> sqrt(sum(coef(ridge.mod)[-1,50]^2))
[1] 6.36
```

In contrast, here are the coefficients when $\lambda = 705$, along with their ℓ_2 norm. Note the much larger ℓ_2 norm of the coefficients associated with this smaller value of λ.

```
> ridge.mod$lambda[60]
[1] 705
> coef(ridge.mod)[,60]
(Intercept)       AtBat        Hits       HmRun        Runs
     54.325       0.112       0.656       1.180       0.938
        RBI       Walks       Years      CAtBat       CHits
      0.847       1.320       2.596       0.011       0.047
     CHmRun       CRuns        CRBI      CWalks     LeagueN
      0.338       0.094       0.098       0.072      13.684
  DivisionW     PutOuts     Assists      Errors  NewLeagueN
    -54.659       0.119       0.016      -0.704       8.612
> sqrt(sum(coef(ridge.mod)[-1,60]^2))
[1] 57.1
```

We can use the `predict()` function for a number of purposes. For instance, we can obtain the ridge regression coefficients for a new value of λ, say 50:

```
> predict(ridge.mod,s=50,type="coefficients")[1:20,]
(Intercept)       AtBat        Hits       HmRun        Runs
     48.766      -0.358       1.969      -1.278       1.146
        RBI       Walks       Years      CAtBat       CHits
      0.804       2.716      -6.218       0.005       0.106
     CHmRun       CRuns        CRBI      CWalks     LeagueN
      0.624       0.221       0.219      -0.150      45.926
  DivisionW     PutOuts     Assists      Errors  NewLeagueN
   -118.201       0.250       0.122      -3.279      -9.497
```

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 data set. 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 in Section 6.5.3. Here we demonstrate the latter approach.

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

```
> set.seed(1)
> train=sample(1:nrow(x), nrow(x)/2)
> test=(-train)
> y.test=y[test]
```

Next we fit a ridge regression model on the training set, and evaluate its MSE on the 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.

```
> ridge.mod=glmnet(x[train,],y[train],alpha=0,lambda=grid,
    thresh=1e-12)
> ridge.pred=predict(ridge.mod,s=4,newx=x[test,])
> mean((ridge.pred-y.test)^2)
[1] 101037
```

The test MSE is 101037. 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:

```
> mean((mean(y[train])-y.test)^2)
[1] 193253
```

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

```
> ridge.pred=predict(ridge.mod,s=1e10,newx=x[test,])
> mean((ridge.pred-y.test)^2)
[1] 193253
```

So fitting a ridge regression model 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 $\lambda = 0$.[7]

[7] In order for `glmnet()` to yield the exact least squares coefficients when $\lambda = 0$, we use the argument `exact=T` when calling the `predict()` function. Otherwise, the `predict()` function will interpolate over the grid of λ values used in fitting the

```
> ridge.pred=predict(ridge.mod,s=0,newx=x[test,],exact=T)
> mean((ridge.pred-y.test)^2)
[1] 114783
> lm(y~x, subset=train)
> predict(ridge.mod,s=0,exact=T,type="coefficients")[1:20,]
```

In general, if 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, though 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. `cv.glmnet()`

```
> set.seed(1)
> cv.out=cv.glmnet(x[train,],y[train],alpha=0)
> plot(cv.out)
> bestlam=cv.out$lambda.min
> bestlam
[1] 212
```

Therefore, we see that the value of λ that results in the smallest cross-validation error is 212. What is the test MSE associated with this value of λ?

```
> ridge.pred=predict(ridge.mod,s=bestlam,newx=x[test,])
> mean((ridge.pred-y.test)^2)
[1] 96016
```

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 data set, using the value of λ chosen by cross-validation, and examine the coefficient estimates.

```
> out=glmnet(x,y,alpha=0)
> predict(out,type="coefficients",s=bestlam)[1:20,]
(Intercept)       AtBat        Hits       HmRun        Runs
     9.8849      0.0314      1.0088      0.1393      1.1132
        RBI       Walks       Years      CAtBat       CHits
     0.8732      1.8041      0.1307      0.0111      0.0649
     CHmRun       CRuns        CRBI      CWalks     LeagueN
     0.4516      0.1290      0.1374      0.0291     27.1823
  DivisionW     PutOuts     Assists      Errors  NewLeagueN
   -91.6341      0.1915      0.0425     -1.8124      7.2121
```

`glmnet()` model, yielding approximate results. When we use `exact=T`, there remains a slight discrepancy in the third decimal place between the output of `glmnet()` when $\lambda = 0$ and the output of `lm()`; this is due to numerical approximation on the part of `glmnet()`.

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

6.6.2 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` data set. 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.

```
> lasso.mod=glmnet(x[train,],y[train],alpha=1,lambda=grid)
> plot(lasso.mod)
```

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.

```
> set.seed(1)
> cv.out=cv.glmnet(x[train,],y[train],alpha=1)
> plot(cv.out)
> bestlam=cv.out$lambda.min
> lasso.pred=predict(lasso.mod,s=bestlam,newx=x[test,])
> mean((lasso.pred-y.test)^2)
[1] 100743
```

This is substantially lower than the test set MSE of the null model and of least squares, and very similar 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 resulting coefficient estimates are sparse. Here we see that 12 of the 19 coefficient estimates are exactly zero. So the lasso model with λ chosen by cross-validation contains only seven variables.

```
> out=glmnet(x,y,alpha=1,lambda=grid)
> lasso.coef=predict(out,type="coefficients",s=bestlam)[1:20,]
> lasso.coef
(Intercept)       AtBat        Hits       HmRun        Runs
     18.539       0.000       1.874       0.000       0.000
        RBI       Walks       Years      CAtBat       CHits
      0.000       2.218       0.000       0.000       0.000
     CHmRun       CRuns        CRBI      CWalks     LeagueN
      0.000       0.207       0.413       0.000       3.267
  DivisionW     PutOuts     Assists      Errors  NewLeagueN
   -103.485       0.220       0.000       0.000       0.000
> lasso.coef[lasso.coef!=0]
(Intercept)        Hits       Walks       CRuns        CRBI
     18.539       1.874       2.218       0.207       0.413
    LeagueN   DivisionW     PutOuts
      3.267    -103.485       0.220
```