DSA 6100: Statistical Methods for Data Science and Analytics

Winter 2019

Lab Session 2

1. Best Subset Selection

We apply the best subset selection approach to the Hitters data. We wish to predict a baseball player's Salary on the basis of various statistics associated with performance in the previous year. The Salary variable is missing for some of the players. The sum(is.na(Hitters$Salary)) function can then be used to count all of the missing elements.

```
library (ISLR)
fix(Hitters )
names(Hitters )
sum(is.na(Hitters$Salary))
```

We see that Salary is missing for 59 players. The na.omit() function removes all of the rows that have missing values in any variable. the following function remove the missing values

```
Hitters =na.omit(Hitters )
```

The regsubsets() function (part of the leaps library) performs best subset selection by identifying the best model that contains a given number of predictors, where *best* is quantified using RSS. The summary() command outputs the best set of variables for each model size.

```
>library (leaps)
> regfit .full=regsubsets (Salary∼.,Hitters )
> summary (regfit .full)
```

An asterisk indicates that a given variable is included in the corresponding model. For instance, this output indicates that the best two-variable model contains only Hits and CRBI. By default, regsubsets() only reports results up to the best eight-variable model. But the nvmax option can be used in order to return as many variables as are desired. Here we fit up to a 19-variable model.

```
> regfit .full=regsubsets (Salary∼.,data=Hitters ,nvmax =19)
> reg.summary =summary (regfit .full)
```

The summary() function also returns $R2$, RSS, adjusted $R2$, Cp, and BIC. We can examine these to try to select the *best* overall model.
For instance, we see that the $R2$ statistic increases from 32%, when only one variable is included in the model, to almost 55 %, when all variables are included. As expected, the $R2$ statistic increases monotonically as more variables are included.

```
> reg. summary$rsq
```

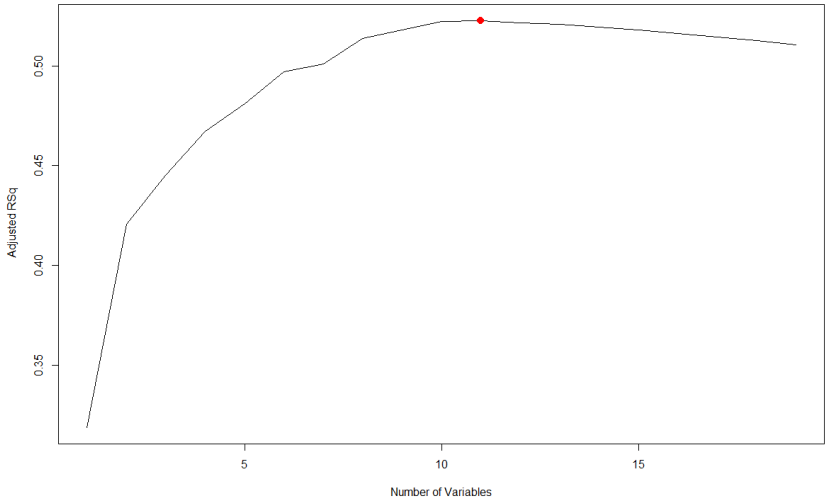
```
[1] 0.321 0.425 0.451 0.475 0.491 0.509 0.514 0.529 0.535
[10] 0.540 0.543 0.544 0.544 0.545 0.545 0.546 0.546 0.546
[19] 0.546
```

Plotting RSS, adjusted $R2$, Cp, and BIC for all of the models at once will help us decide which model to select. Note the type="l" option tells R to connect the plotted points with lines.

```
> par(mfrow =c(2,2))
> plot(reg.summary$rss ,xlab=" Number of Variables ",ylab=" RSS",
type="l")
> plot(reg.summary$adjr2 ,xlab =" Number of Variables ",
ylab=" Adjusted RSq",type="l")
```

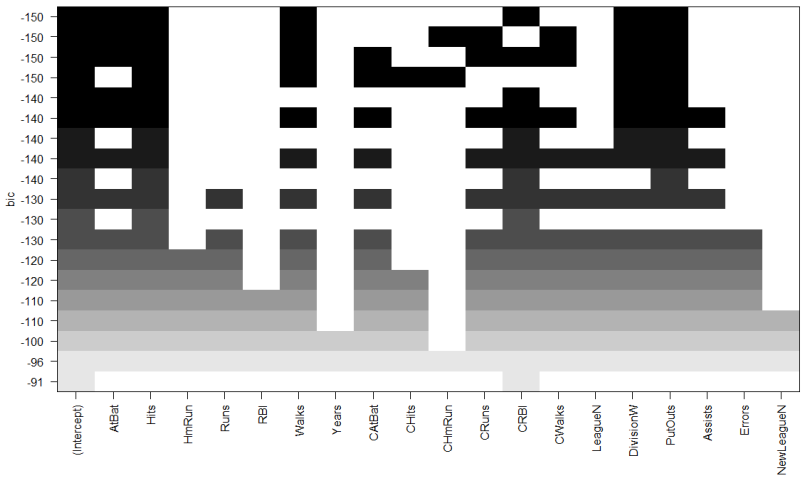
The which.max() function can be used to identify the location of the maximum point of a vector. We will now plot a red dot to indicate the model with the largest adjusted R^2 statistic.

```
>which.max (reg.summary$adjr2)
> points (11, reg.summary$adjr2[11], col ="red",cex =2, pch =20)
```



The regsubsets() function has a built-in plot() command which can be used to display the selected variables for the best model with a given number of predictors, ranked according to the BIC, Cp, adjusted R^2, or AIC.

```
> plot(regfit .full ,scale ="r2")
> plot(regfit .full ,scale =" adjr2 ")
> plot(regfit .full ,scale ="Cp")
> plot(regfit .full ,scale ="bic ")
```



Each row in this graph represents a model; the shaded rectangles in the columns indicate the variables included in the given model. The numbers on the left margin are the values of BIC. The darkness of the shading simply represents the ordering of the BIC values. In the example above, the model with AtBat, Hits, Walks, CRBI, DivisionW and PutOut (and intercept) has the lowst BIC (close to -150).

2. Forward and Backward Stepwise Selection

We can also use the regsubsets() function to perform forward stepwise or backward stepwise selection, using the argument method="forward" or method="backward".

```
> regfit .fwd=regsubsets (Salary∼.,data=Hitters ,nvmax =19, method =" forward ")
> summary (regfit .fwd )
> regfit .bwd=regsubsets (Salary∼.,data=Hitters ,nvmax =19, method =" backward ")
> summary (regfit .bwd )
```

For instance, we see that using forward stepwise selection, the best one variable model contains only CRBI, and the best two-variable model additionally includes Hits. For this data, the best one-variable through six-variable models are each identical for best subset and forward selection.
the best seven-variable models identified by forward stepwise selection, backward stepwise selection, and best subset selection are different

```
> coef(regfit .full ,7)
> coef(regfit .fwd ,7)
>coef(regfit .bwd ,7)
```

3. Ridge Regression

We use the glmnet package in order to perform ridge regression and the lasso. 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 before.

```
> 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.
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)
```

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().

```
> ridge.mod$lambda [50]
```

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 ,]
```

We split the samples into a training set and a test set in order to estimate the test error of ridge regression and the lasso. To do the split, we randomly choose a subset of numbers between 1 and *n*; these can then be used as the indices for the training observations. 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 λ = 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
```

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.

```
> 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 $\lambda = 4$.

4. Lasso

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
```

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
```

5. Principal Components Regression

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. Again, ensure that the missing values have been removed from the data

```
> library (pls)
> set.seed (2)
> pcr.fit=pcr(Salary∼., data=Hitters ,scale=TRUE , validation ="CV")
```

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 )
```

pcr() reports the *root mean squared error*; in order to obtain the usual MSE, we must square this quantity. One can 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 implement partial least squares (PLS) using the plsr() 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 )
```

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] 101417
```