

Variable Selection and Dimensionality Reduction

Contents

1	Getting Set Up	1
1.1	Setting chunk options	1
1.2	Installing Packages	1
2	Subset Selection Methods	1
2.1	Best subset selection	1
2.2	Forward and Backward Stepwise Selection	8
2.3	Ridge Regression	11
3	PCR and PLS Regression	15
3.1	Principal Components Regression	15
3.2	Partial Least Squares	18

1 Getting Set Up

1.1 Setting chunk options

```
knitr::opts_chunk$set(warning=FALSE, message=FALSE)
knitr::purl("variable-selection.Rmd")
```

1.2 Installing Packages

```
install.packages('ISLR')
install.packages('leaps')
install.packages('glmnet')
install.packages('pls')
```

2 Subset Selection Methods

2.1 Best subset selection

```
library(ISLR)
library(leaps)
attach(Hitters)

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 syntax is the same as for `lm()`.

```
regfit.full = regsubsets(Salary~., Hitters)
summary(regfit.full)
```

```
## Subset selection object
## Call: regsubsets.formula(Salary ~ ., Hitters)
## 19 Variables (and intercept)
##           Forced in Forced out
## AtBat      FALSE      FALSE
## Hits       FALSE      FALSE
## HmRun       FALSE      FALSE
## Runs        FALSE      FALSE
## RBI         FALSE      FALSE
## Walks       FALSE      FALSE
## Years       FALSE      FALSE
## CatBat      FALSE      FALSE
## CHits       FALSE      FALSE
## CHmRun      FALSE      FALSE
## CRuns       FALSE      FALSE
## CRBI        FALSE      FALSE
## CWalks      FALSE      FALSE
## LeagueN     FALSE      FALSE
## DivisionW   FALSE      FALSE
## PutOuts     FALSE      FALSE
## Assists     FALSE      FALSE
## Errors      FALSE      FALSE
## NewLeagueN  FALSE      FALSE
## 1 subsets of each size up to 8
## Selection Algorithm: exhaustive
##           AtBat Hits HmRun Runs RBI Walks Years CatBat CHits CHmRun CRuns CRBI
## 1 ( 1 ) " " " " " " " " " " " " " " " " " " " " " "
## 2 ( 1 ) " " "*" " " " " " " " " " " " " " " " "
## 3 ( 1 ) " " "*" " " " " " " " " " " " " " " " "
## 4 ( 1 ) " " "*" " " " " " " " " " " " " " " " "
## 5 ( 1 ) "*" "*" " " " " " " " " " " " " " " " "
## 6 ( 1 ) "*" "*" " " " " " " "*" " " " " " " " "
## 7 ( 1 ) " " "*" " " " " " " "*" " " "*" "*" " " "
## 8 ( 1 ) "*" "*" " " " " " " "*" " " " " "*" "*" " "
##           CWalks LeagueN DivisionW PutOuts Assists Errors NewLeagueN
## 1 ( 1 ) " " " " " " " " " " " "
## 2 ( 1 ) " " " " " " " " " " " "
## 3 ( 1 ) " " " " " " "*" " " "
## 4 ( 1 ) " " " " "*" "*" " " " "
```

```
## 5 ( 1 ) " " " " "*" "*" " " " " " "
## 6 ( 1 ) " " " " "*" "*" " " " " " "
## 7 ( 1 ) " " " " "*" "*" " " " " " "
## 8 ( 1 ) "*" " " "*" "*" " " " " " "
```

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)
names(reg.summary)
```

```
## [1] "which" "rsq" "rss" "adjr2" "cp" "bic" "outmat" "obj"
```

```
reg.summary$rsq
```

```
## [1] 0.3214501 0.4252237 0.4514294 0.4754067 0.4908036 0.5087146 0.5141227
## [8] 0.5285569 0.5346124 0.5404950 0.5426153 0.5436302 0.5444570 0.5452164
## [15] 0.5454692 0.5457656 0.5459518 0.5460945 0.5461159
```

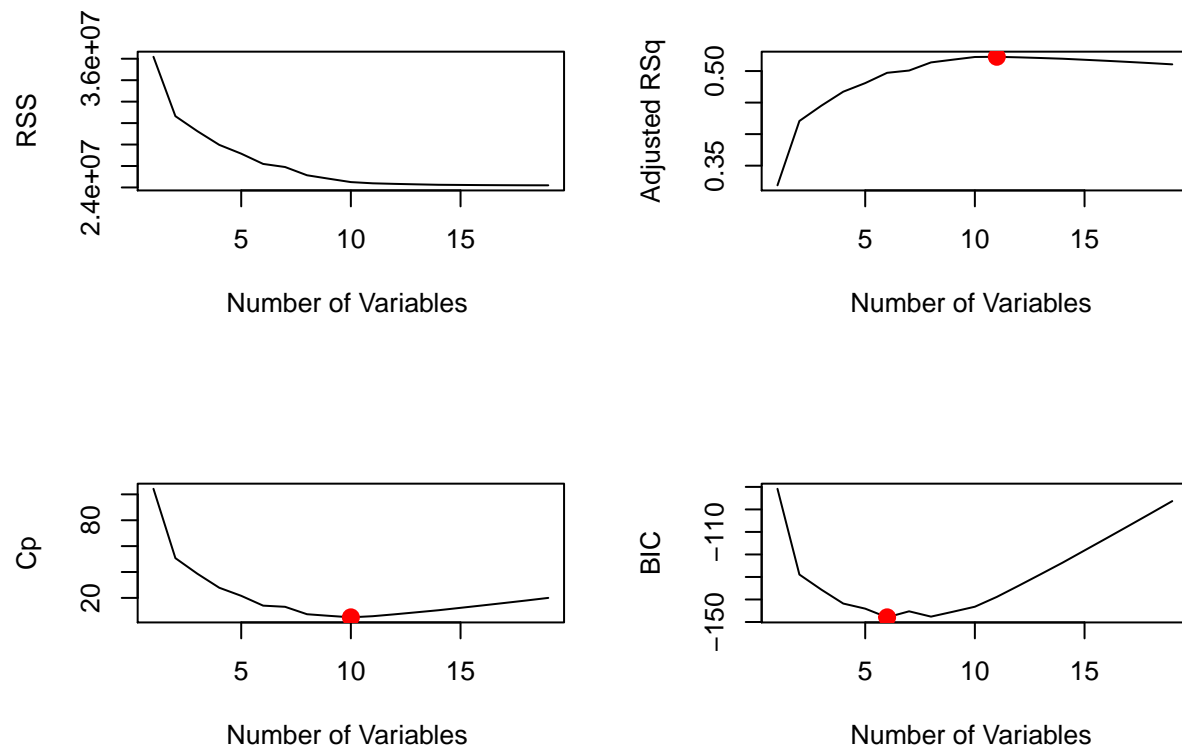
Plotting RSS, adjusted R2, Cp, and BIC for all of the models at once will help us decide which model to select. Note the `type="l"` option tells R to connect the plotted points with lines.

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

plot(reg.summary$adjr2, xlab="Number of Variables ", ylab="Adjusted RSq", type="l")
max.adj2 = which.max(reg.summary$adjr2)
points(max.adj2, reg.summary$adjr2[max.adj2], col="red", cex=2, pch =20)

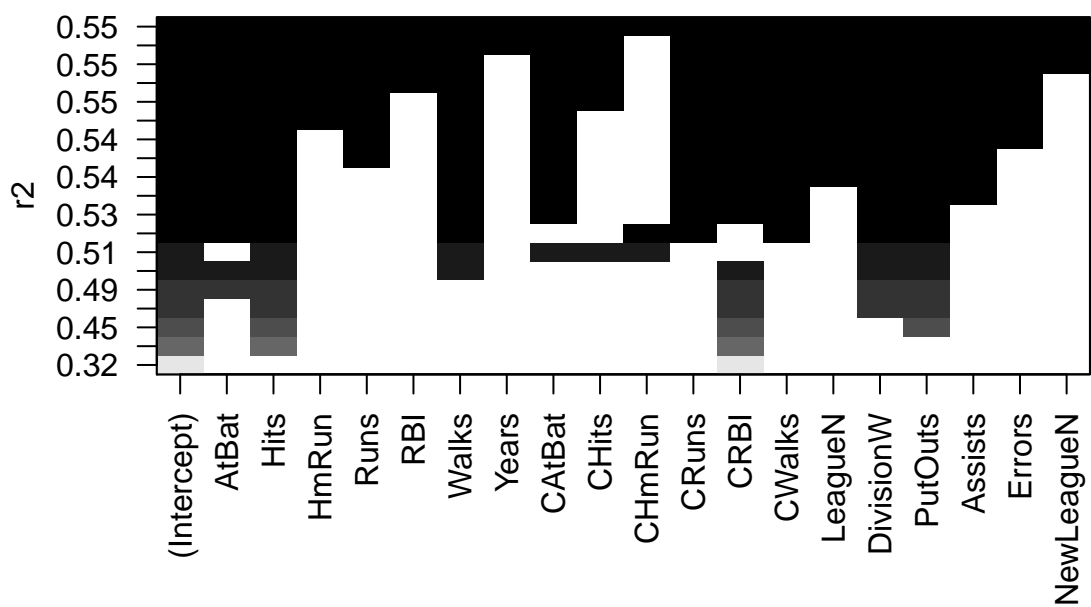
plot(reg.summary$cp, xlab="Number of Variables ", ylab="Cp", type="l")
min.cp = which.min(reg.summary$cp)
points(min.cp, reg.summary$cp[min.cp], col="red", cex=2, pch =20)

plot(reg.summary$bic, xlab="Number of Variables ", ylab="BIC", type="l")
min.bic = which.min(reg.summary$bic)
points(min.bic, reg.summary$bic[min.bic], col="red", cex=2, pch =20)
```

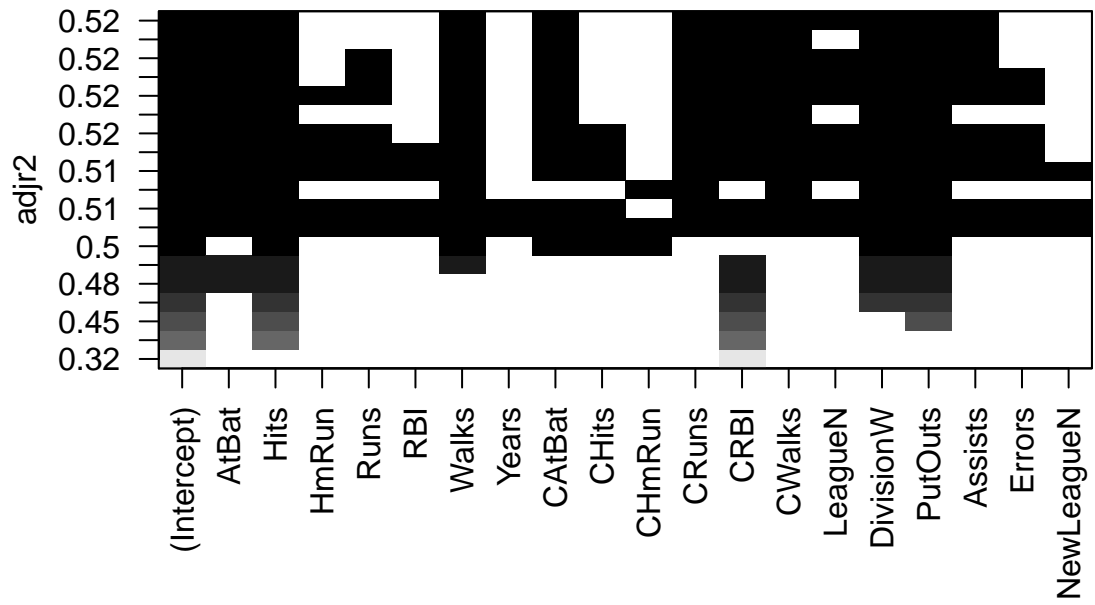


The `regsubsets()` function has a built-in `plot()` command which can be used to display the selected variables for the best model with a given number of predictors, ranked according to the BIC, Cp, adjusted R2, or AIC.

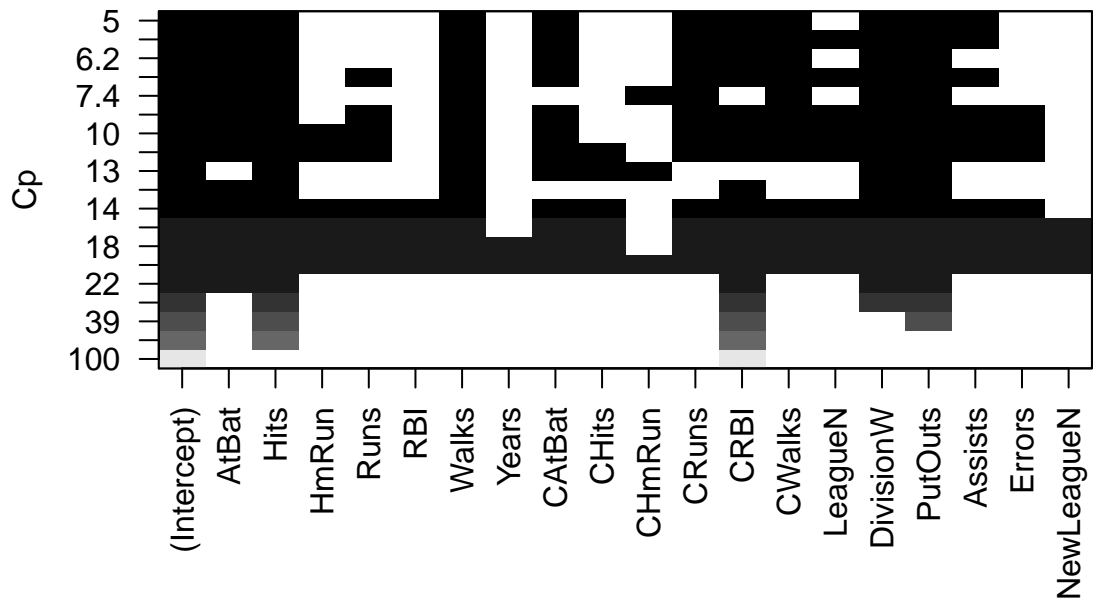
```
plot(regfit.full, scale="r2")
```



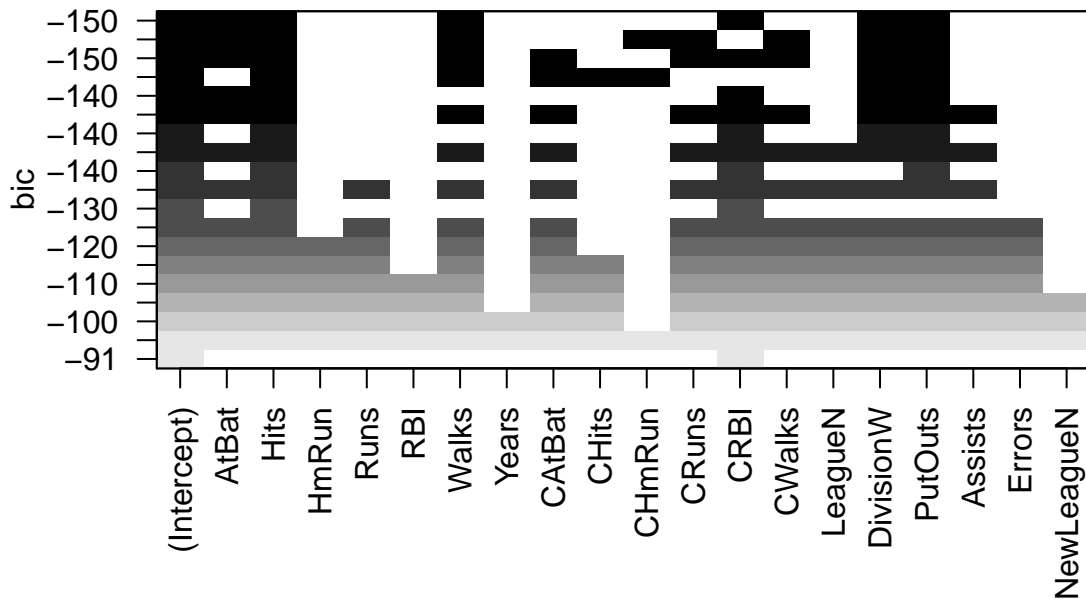
```
plot(regfit.full, scale="adjr2")
```



```
plot(regfit.full, scale="Cp")
```



```
plot(regfit.full, scale="bic")
```



The top row of each plot contains a black square for each variable selected according to the optimal model associated with that statistic. For instance, we see that several models share a BIC close to -150 . However, the model with the lowest BIC is the six-variable model that contains only AtBat, Hits, Walks, CRBI, DivisionW, and PutOuts. We can use the `coef()` function to see the coefficient estimates associated with this model.

```
coef(regfit.full, 6)
```

```
## (Intercept)      AtBat      Hits      Walks      CRBI      DivisionW
##  91.5117981   -1.8685892    7.6043976    3.6976468    0.6430169   -122.9515338
##      PutOuts
##    0.2643076
```

2.2 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)
```

```
## Subset selection object
## Call: regsubsets.formula(Salary ~ ., data = Hitters, nvmax = 19, method = "forward")
## 19 Variables (and intercept)
```



```

##          Forced in Forced out
## AtBat      FALSE      FALSE
## Hits       FALSE      FALSE
## HmRun      FALSE      FALSE
## Runs       FALSE      FALSE
## RBI        FALSE      FALSE
## Walks      FALSE      FALSE
## Years      FALSE      FALSE
## CAtBat     FALSE      FALSE
## CHits      FALSE      FALSE
## CHmRun     FALSE      FALSE
## CRuns      FALSE      FALSE
## CRBI       FALSE      FALSE
## CWalks     FALSE      FALSE
## LeagueN    FALSE      FALSE
## DivisionW  FALSE      FALSE
## PutOuts    FALSE      FALSE
## Assists    FALSE      FALSE
## Errors     FALSE      FALSE
## NewLeagueN FALSE      FALSE
## 1 subsets of each size up to 19
## Selection Algorithm: forward
##          AtBat Hits HmRun Runs RBI Walks Years CAtBat CHits CHmRun CRuns CRBI
## 1 ( 1 ) " " " " " " " " " " " " " " " " " " " "
## 2 ( 1 ) " " "*" " " " " " " " " " " " " " " "*"
## 3 ( 1 ) " " "*" " " " " " " " " " " " " " " "*"
## 4 ( 1 ) " " "*" " " " " " " " " " " " " " " "*"
## 5 ( 1 ) "*" "*" " " " " " " " " " " " " " " "*"
## 6 ( 1 ) "*" "*" " " " " " " "*" " " " " " " " " "*"
## 7 ( 1 ) "*" "*" " " " " " " "*" " " " " " " " " "*"
## 8 ( 1 ) "*" "*" " " " " " " "*" " " " " " " "*" "*"
## 9 ( 1 ) "*" "*" " " " " " " "*" " " "*" " " " " "*" "*"
## 10 ( 1 ) "*" "*" " " " " " " "*" " " "*" " " " " "*" "*"
## 11 ( 1 ) "*" "*" " " " " " " "*" " " "*" " " " " "*" "*"
## 12 ( 1 ) "*" "*" " " "*" " " "*" " " "*" " " " " " "*" "*"
## 13 ( 1 ) "*" "*" " " "*" " " "*" " " "*" " " " " " "*" "*"
## 14 ( 1 ) "*" "*" "*" "*" " " "*" " " "*" " " " " " "*" "*"
## 15 ( 1 ) "*" "*" "*" "*" " " "*" " " "*" "*" " " " " "*" "*"
## 16 ( 1 ) "*" "*" "*" "*" "*" "*" " " "*" "*" " " " " "*" "*"
## 17 ( 1 ) "*" "*" "*" "*" "*" "*" " " "*" "*" " " " " "*" "*"
## 18 ( 1 ) "*" "*" "*" "*" "*" "*" "*" "*" "*" " " " " "*" "*"
## 19 ( 1 ) "*" "*" "*" "*" "*" "*" "*" "*" "*" "*" " " " " "*" "*"
##          CWalks LeagueN DivisionW PutOuts Assists Errors NewLeagueN
## 1 ( 1 ) " " " " " " " " " " " "
## 2 ( 1 ) " " " " " " " " " " " "
## 3 ( 1 ) " " " " " " "*" " " " " "
## 4 ( 1 ) " " " " "*" "*" " " " " "
## 5 ( 1 ) " " " " "*" "*" " " " " "
## 6 ( 1 ) " " " " "*" "*" " " " " "
## 7 ( 1 ) "*" " " "*" "*" " " " " "
## 8 ( 1 ) "*" " " "*" "*" " " " " "
## 9 ( 1 ) "*" " " "*" "*" " " " " "
## 10 ( 1 ) "*" " " "*" "*" "*" " " " "
## 11 ( 1 ) "*" "*" "*" "*" "*" " " " "

```

```
## 12 ( 1 ) "*" "*" "*" "*" "*" " " " "
## 13 ( 1 ) "*" "*" "*" "*" "*" "*" " "
## 14 ( 1 ) "*" "*" "*" "*" "*" "*" " "
## 15 ( 1 ) "*" "*" "*" "*" "*" "*" " "
## 16 ( 1 ) "*" "*" "*" "*" "*" "*" " "
## 17 ( 1 ) "*" "*" "*" "*" "*" "*" "*"
## 18 ( 1 ) "*" "*" "*" "*" "*" "*" "*"
## 19 ( 1 ) "*" "*" "*" "*" "*" "*" "*"

```

```
regfit.bwd = regsubsets(Salary~., data=Hitters, nvmax=19, method="backward")
summary(regfit.bwd)
```

```
## Subset selection object
## Call: regsubsets.formula(Salary ~ ., data = Hitters, nvmax = 19, method = "backward")
## 19 Variables (and intercept)
##           Forced in Forced out
## AtBat      FALSE      FALSE
## Hits       FALSE      FALSE
## HmRun       FALSE      FALSE
## Runs        FALSE      FALSE
## RBI         FALSE      FALSE
## Walks       FALSE      FALSE
## Years       FALSE      FALSE
## CAtBat      FALSE      FALSE
## CHits       FALSE      FALSE
## CHmRun      FALSE      FALSE
## CRuns       FALSE      FALSE
## CRBI        FALSE      FALSE
## CWalks      FALSE      FALSE
## LeagueN     FALSE      FALSE
## DivisionW   FALSE      FALSE
## PutOuts     FALSE      FALSE
## Assists     FALSE      FALSE
## Errors      FALSE      FALSE
## NewLeagueN  FALSE      FALSE
## 1 subsets of each size up to 19
## Selection Algorithm: backward
##           AtBat Hits HmRun Runs RBI Walks Years CAtBat CHits CHmRun CRuns CRBI
## 1 ( 1 ) " " " " " " " " " " " " " " " " "*" " "
## 2 ( 1 ) " " "*" " " " " " " " " " " " " "*" " "
## 3 ( 1 ) " " "*" " " " " " " " " " " " " "*" " "
## 4 ( 1 ) "*" "*" " " " " " " " " " " " " "*" " "
## 5 ( 1 ) "*" "*" " " " " " " "*" " " " " " "*" " "
## 6 ( 1 ) "*" "*" " " " " " " "*" " " " " " "*" " "
## 7 ( 1 ) "*" "*" " " " " " " "*" " " " " " "*" " "
## 8 ( 1 ) "*" "*" " " " " " " "*" " " " " " "*" "*"
## 9 ( 1 ) "*" "*" " " " " " " "*" " " "*" " " "*" "*"
## 10 ( 1 ) "*" "*" " " " " " " "*" " " "*" " " "*" "*"
## 11 ( 1 ) "*" "*" " " " " " " "*" " " "*" " " "*" "*"
## 12 ( 1 ) "*" "*" " " "*" " " "*" " " "*" " " " "*" "*"
## 13 ( 1 ) "*" "*" " " "*" " " "*" " " "*" " " " "*" "*"
## 14 ( 1 ) "*" "*" "*" "*" " " "*" " " "*" " " " "*" "*"
## 15 ( 1 ) "*" "*" "*" "*" " " "*" " " "*" "*" " "*" "*"
## 16 ( 1 ) "*" "*" "*" "*" "*" "*" " " "*" "*" " "*" "*"

```

```
## 17 ( 1 ) "*" "*" "*" "*" "*" "*" " " "*" "*" " " "*" "*"
## 18 ( 1 ) "*" "*" "*" "*" "*" "*" "*" "*" "*" " " "*" "*"
## 19 ( 1 ) "*" "*" "*" "*" "*" "*" "*" "*" "*" "*" "*" "*"
##           CWalks LeagueN DivisionW PutOuts Assists Errors NewLeagueN
## 1 ( 1 ) " " " " " " " " " " " "
## 2 ( 1 ) " " " " " " " " " " " "
## 3 ( 1 ) " " " " " " "*" " " " " "
## 4 ( 1 ) " " " " " " "*" " " " " "
## 5 ( 1 ) " " " " " " "*" " " " " "
## 6 ( 1 ) " " " " "*" "*" " " " " "
## 7 ( 1 ) "*" " " "*" "*" " " " " "
## 8 ( 1 ) "*" " " "*" "*" " " " " "
## 9 ( 1 ) "*" " " "*" "*" " " " " "
## 10 ( 1 ) "*" " " "*" "*" "*" " " " "
## 11 ( 1 ) "*" "*" "*" "*" "*" " " " "
## 12 ( 1 ) "*" "*" "*" "*" "*" " " " "
## 13 ( 1 ) "*" "*" "*" "*" "*" "*" " " "
## 14 ( 1 ) "*" "*" "*" "*" "*" "*" " " "
## 15 ( 1 ) "*" "*" "*" "*" "*" "*" " " "
## 16 ( 1 ) "*" "*" "*" "*" "*" "*" " " "
## 17 ( 1 ) "*" "*" "*" "*" "*" "*" "*"
## 18 ( 1 ) "*" "*" "*" "*" "*" "*" "*"
## 19 ( 1 ) "*" "*" "*" "*" "*" "*" "*"

```

#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 the ridge regression models, lasso models, and more.

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.

```
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.

2.3 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 values. However, here we have chosen to implement the function over a grid of values ranging from 10^{10} to 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.

Associated with each value of `lambda` is a vector of ridge regression coefficients, stored in a matrix that can be accessed by `coef()`.

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

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

We expect the coefficient estimates to be much smaller, in terms of 2 norm, when a large value of `lambda` is used, as compared to when a small value of `lambda` is used. These are the coefficients when `lambda = 11,498`, along with their 2 norm:

```
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	CAatBat	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] 6.360612
```

In contrast, here are the coefficients when `lambda = 705`, along with their 2 norm. Note the much larger 2 norm of the coefficients associated with this smaller value of `lambda`.

```
ridge.mod$lambda[60]
```

```
## [1] 705.4802
```

```
coef(ridge.mod)[,60]
```

##	(Intercept)	AtBat	Hits	HmRun	Runs	RBI
##	54.32519950	0.11211115	0.65622409	1.17980910	0.93769713	0.84718546
##	Walks	Years	CAatBat	CHits	CHmRun	CRuns
##	1.31987948	2.59640425	0.01083413	0.04674557	0.33777318	0.09355528
##	CRBI	CWalks	LeagueN	DivisionW	PutOuts	Assists
##	0.09780402	0.07189612	13.68370191	-54.65877750	0.11852289	0.01606037
##	Errors	NewLeagueN				
##	-0.70358655	8.61181213				

```
sqrt(sum(coef(ridge.mod)[-1,60]^2))
```

```
## [1] 57.11001
```

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 s , say 50:

```
predict(ridge.mod, s=50, type="coefficients")[1:20,]
```

```
##      (Intercept)      AtBat      Hits      HmRun      Runs
## 4.876610e+01 -3.580999e-01 1.969359e+00 -1.278248e+00 1.145892e+00
##           RBI      Walks      Years      CAtBat      CHits
## 8.038292e-01 2.716186e+00 -6.218319e+00 5.447837e-03 1.064895e-01
##      CHmRun      CRuns      CRBI      CWalks      LeagueN
## 6.244860e-01 2.214985e-01 2.186914e-01 -1.500245e-01 4.592589e+01
##      DivisionW      PutOuts      Assists      Errors      NewLeagueN
## -1.182011e+02 2.502322e-01 1.215665e-01 -3.278600e+00 -9.496680e+00
```

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.

```
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 $s = 4$. 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.

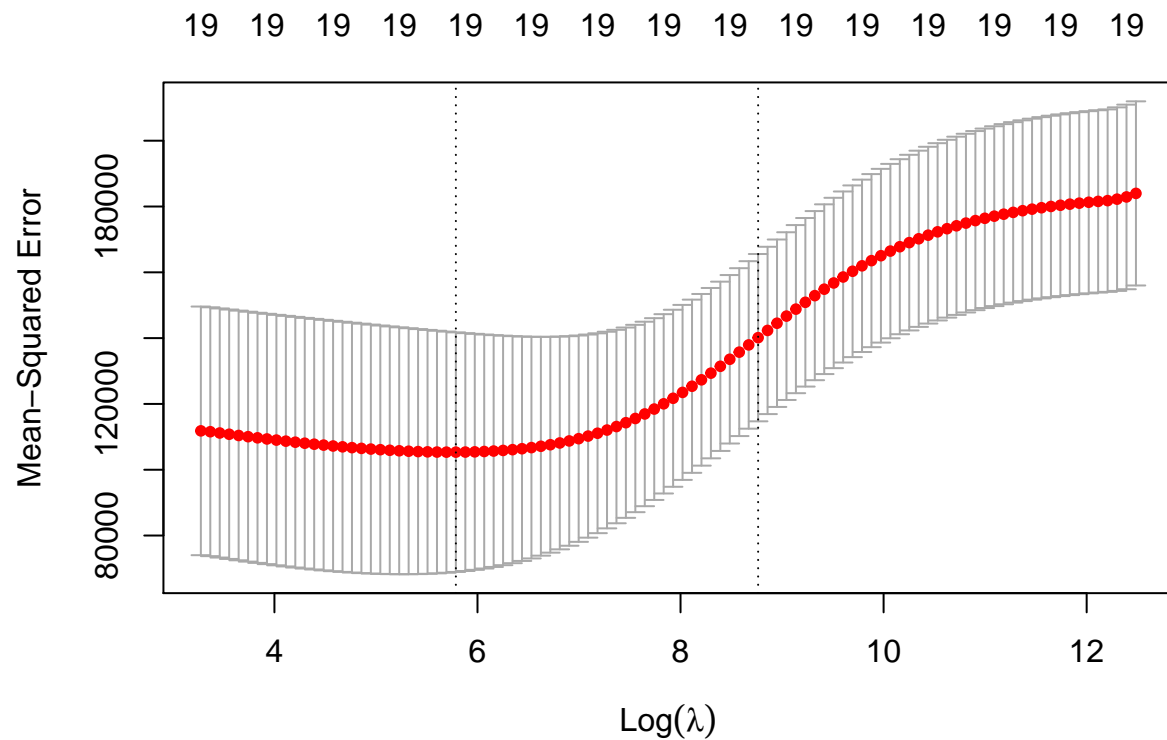
```
ridge.mod = glmnet(x[train,], y[train], alpha=0, lambda=grid, thresh=1e-12)
ridge.pred = predict(ridge.mod, s=4, newx=x[test,])
mean((ridge.pred-y.test)^2)
```

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

It is also possible to fit a least squares model with $\lambda = 0$. 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 s , it would be better to use cross-validation to choose the tuning parameter s . 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.

```
set.seed(1)
cv.out = cv.glmnet(x[train,], y[train], alpha=0)
plot(cv.out)
```



```
bestlam = cv.out$lambda.min
bestlam
```

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

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] 139856.6
```

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      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
```

3 PCR and PLS Regression

3.1 Principal Components Regression

Principal components regression (PCR) can be performed using the `pcr()` function, which is part of the `pls` library.

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

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 effect. 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()`.

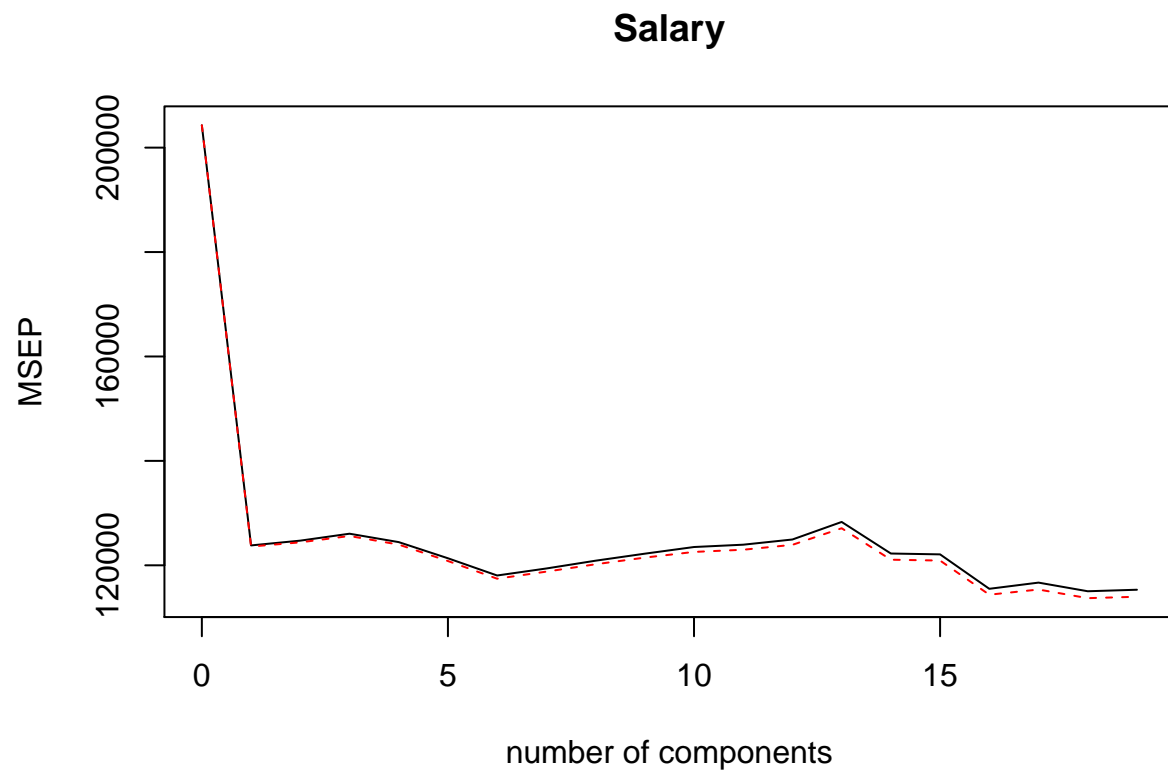
```
summary(pcr.fit)
```

```
## 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
```

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 $352.8^2 = 124,468$. The `summary()` function also provides the percentage of variance explained in the predictors and in the response using different numbers of components.

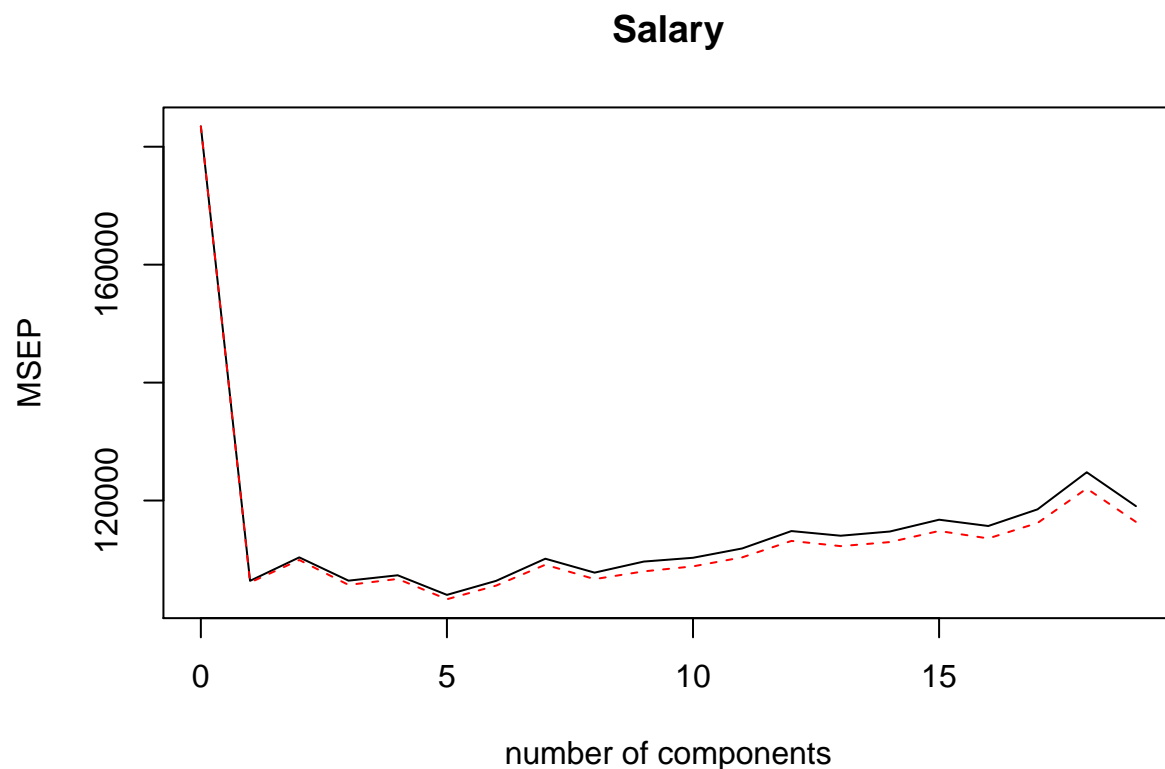
One can also plot the cross-validation scores using the `validationplot()` function. Using `val.type="MSEP"` will cause the cross-validation MSE to be plotted.

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



We 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 error occurs when $M = 5$ components are used. We compute the test MSE as follows.

```
pcr.pred = predict(pcr.fit, x[test,], ncomp=5)
mean((pcr.pred - y.test)^2)
```

```
## [1] 142811.8
```

This 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 data set, using $M = 5$, the number of components identified by cross-validation.

```
pcr.fit = pcr(y~x, scale=TRUE, ncomp=5)
summary(pcr.fit)
```

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

3.2 Partial Least Squares

We implement partial least squares (PLS) using the `plsr()` function, also in the `pls` library.

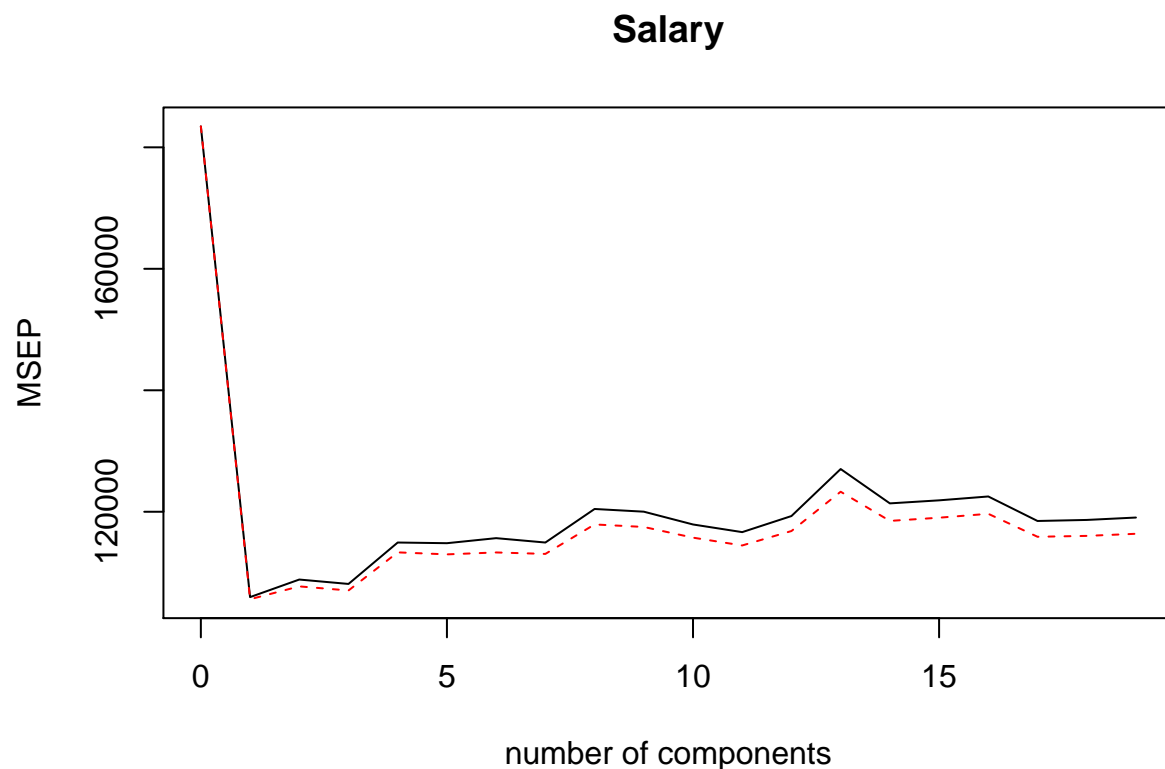
```
set.seed(1)
pls.fit = plsr(Salary~., data=Hitters, subset=train, scale=TRUE, validation="CV")
summary(pls.fit)
```



```
## 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
```



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



The lowest cross-validation error occurs when only $M = 1$ partial least squares directions are used. We now evaluate the corresponding test set MSE.

```
pls.pred=predict(pls.fit, x[test,], ncomp=1)
mean((pls.pred-y.test)^2)
```

```
## [1] 151995.3
```

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

Finally, we perform PLS using the full data set, using $M = 1$, the number of components identified by cross-validation.

```
pls.fit = plsrf(Salary~., data=Hitters, scale=TRUE, ncomp=1)
summary(pls.fit)
```

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

Notice that the percentage of variance in Salary that the one-component PLS fit explains is almost as much as that explained using the final five-component model PCR fit. 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.