Assignment #6

Lindsay Brock, Gustavo Esparza, and Brian Schetzsle

10/10/2019

1 Bootstrap Linear Models

```r
Albumin <- read.csv("Albumin.csv", header = TRUE)
data = cbind(Albumin[,1], Albumin[,2:4])
names(data) = c("Y", "X1", "X2", "X3")
attach(data)

#Cross-Validation
n = length(X1)
test.n = n/10#1/10 of that bc of k fold
index = sample(1:n, n, replace=F)
og = rep(0, n)
m1 = og
m2 = og
sst = og
method1_coeff = matrix(0, 1000, dim(data)[2])
method2_coeff = method1_coeff

for(j in 1:10){
  testingRows = index[(1 + (j-1)*test.n):(test.n + (j-1)*test.n)]
  test = data[testingRows, ]
  training = data[-testingRows, ]

  #original model
  model = lm(Y~X1+X2+X3, data = training)
  Coef = model$coef
  og[testingRows] = (predict.lm(model, newdata = test[,-1]) - test[,1])

  #method one
  Ei = model$residuals
  for(i in 1:1000){
    Eisamp = sample(Ei, length(Ei), replace = T)
    bs_Y = training[,1] + Eisamp
    model_bs = lm(bs_Y~X1+X2+X3, data = training)
    method1_coeff[i,] = model_bs$coefficients
  }
  model$coefficients = apply(method1_coeff, 2, mean)
  m1[testingRows] = (predict.lm(model, newdata = test[,-1]) - test[,1])

  #method two
  n2 = dim(training)[1]
  for(i in 1:1000){
    indx = sample(1:n2, n2, replace = T)
    bs_data = training[indx, ]
    model_bs = lm(Y~X1+X2+X3, data = bs_data)
    method2_coeff[i,] = model_bs$coefficients
  }
  model$coefficients = apply(method2_coeff, 2, mean)
  m2[testingRows] = (predict.lm(model, newdata = test[,-1]) - test[,1])

  sst[testingRows] = (test[,1] - mean(test[,1]))
}
SSE = rbind(sum(og^2), sum(m1^2), sum(m2^2))

SST = sum(sst^2)
#R^2 for all three models
r2 = 1-SSE/SST
```

In this problem our goal is to fit a linear model to our data (we will call this the original model) and compare it with models created via bootstrapping the residuals and bootstrapping indices. To begin we will fit a simple linear model to a data set called Albumin, which has one response and three predictors, and produces the coefficients in the table below.

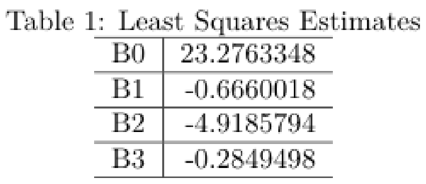


Table 1: Least Squares Estimates

B0	23.2763348
B1	-0.6660018
B2	-4.9185794
B3	-0.2849498

Next, we will fit our model created by bootstrapping the residuals from the original model. First, the residuals from the original model are sampled from, with replacement. They are then added to the response variable to create a new bootstrapped response variable. Then, a new model is fit using this b.s. response and the original explanatory variables which will in turn generate new bootstrapped coefficients. This is done 10,000 times to create 10,000 sets of b.s. coefficients. Taking the mean of each b.s. coefficient will give us the final set of coefficients to use for prediction (in the table below). We will call this Method 1.

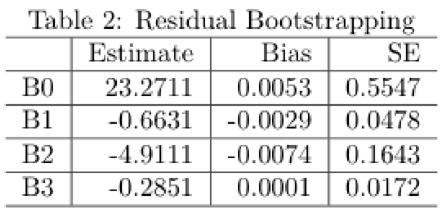


Table 2: Residual Bootstrapping

	Estimate	Bias	SE
B0	23.2711	0.0053	0.5547
B1	-0.6631	-0.0029	0.0478
B2	-4.9111	-0.0074	0.1643
B3	-0.2851	0.0001	0.0172

Our final model will be created by simply bootstrapping the indices of the original data. First, we will sample from the indices of the data set, or from the values of 1 to the total number of observations, with replacement. This means an index value can show up more than once which will give us the variability in the data set we need for bootstrapping. Next, we will shuffle the rows in the data set according to the index values which will create a new b.s. data set (that does not contain exactly the same data, in the same order as the original). We will then fit a model using the new b.s. data which will in turn provide new model coefficients. Again, this will be done 10,000 times to create 10,000 sets of b.s. coefficients. Taking the mean of each b.s. coefficient will give us the final set of coefficients to use for prediction (in the table below). We will call this Method 2.

Table 3: Index Bootstrapping

	Estimate	Bias	SE
B0	23.2827	-0.0064	0.5821
B1	-0.6656	-0.0004	0.0438
B2	-4.9231	0.0045	0.1625
B3	-0.2852	0.0003	0.0183

Finally, we come to the comparison. After comparing the resulting coefficients from each Method with the original model, it seems that they are extremely similar although slightly smaller in value. The b.s. standard errors also seem to be comparable between the two Methods. To give a complete comparison, cross-validation will need to be implemented to compare the r^2 values to see which model fits best. This is done by splitting the data into training (90%) and testing (10%) sets. All three models where fit with the training data to find their respective coefficients (Method 1 & 2 used bootstrap on the training data), then the testing data was used to make predictions. The test response was subtracted from test predictions to calculate the r^2 values. This was done 10 times for 10-fold CV. Looking at the values in the table below, they all seem to explain a similar amount of variance while Method 1 seems to do slightly better than the rest, providing better model prediction.

Table 4: R2

Least Squares	0.8503159
Residual BS	0.8504188
Index BS	0.8502252

2 Chapter 16, Sparse Modeling and the Lasso

With Emphasis on Algorithm 16.4, Least-Angle Regression (LAR)

Chapter 16 addresses a problem with many contemporary data sets; they can have a large number of predictors, sometimes more than the number of observations (p>n). This situation precludes a unique solution in a linear regression setting and even when a solution is possible, the variance of the coefficients due to covariance in the predictors can make the solution undesirable. This is the motivation behind choosing an optimal subset of predictors to include in a model, essentially adding some bias to the coefficients for a big payoff in the reduction of variance.

We covered three primitive methods for subset selection at the beginning of this class: best-subset selection, forward stepwise selection and backward stepwise selection. Best-subset selection is the brute-force method and is guaranteed to find the optimal subset of parameters with respect to whatever evaluation method is used (AIC, MSE, etc.) but is prohibitively computationally intensive for more than a few predictors. The other two methods are faster but do not guarantee optimality. An alternative method adds a penalty term, $\lambda \sum ||\beta||_1$, to the regression model and then uses convex optimization to find the coefficients β to minimize the squared residual in the presence of this penalty. If this penalty is the ℓ_1-norm, as it is here, then this method is called lasso.

The process of fitting the lasso model is dependent on the value of λ, the tuning parameter in the penalty. If $\lambda = 0$ you get the same β as from ordinary least squares. If λ is very large, you get a constant model where $\beta_0 = \bar{Y}$. Any $\lambda > 0$ will yield a unique solution $\hat{\beta}$, so it's useful to find the relationship between λ and β so you can easily find the λ that will lead to β that minimizes some objective function, like cross-validated error.

In a linear regression setting, finding the relationship between λ and β is facilited by realizing that the covariance of some predictors with the residuals at some value of λ are equal in absolute value to $n\lambda$ (those predictors in the *active set*) and the covariance of the rest are different and less than $n\lambda$. This leads us to conclude that $\hat{\beta}(\lambda)$, the optimal β as a function of λ, is linear for small changes in λ and piecewise-linear across all λ. The covariance between the predictors and the residuals changes when a new predictor is added to the active set; that's where the junctures happen in $\hat{\beta}(\lambda)$ that make it not completely linear.

The Least-Angle Regression is an algorithm that utilizes this piece-wise linear structure of $\hat{\beta}(\lambda)$. It's pretty complicated, so we will attempt to give a general idea for how it works in a few sentences:

Start by finding the standardized predictor that has the highest correlation with the residuals of the constant model $y - \bar{y}$. Note this correlation as λ_0 and add it to the *active set* of predictors that will determine the model (and residuals) at each subsequent step. Then, iteratively fit a model with only predictors in the active set and move the coefficients β in a particular direction δ until some predictor not in the active set has the same correlation to the residuals as the predictors in the active set; add that predictor, note its correlation, and repeat until all predictors have been added to the active set. Now you will have a sequence of λs and their corresponding βs.

The direction δ is where "Least-Angle" in the algorithm's name comes from. It is defined as

$$\delta = \frac{1}{\lambda_{k-1}} (X_A^T X_A)^{-1} X_A^T r_{k-1}$$

where λ_{k-1} is the covariance of the most recently added predictor with the most recent set of residuals r_{k-1} and X_A is the active set of predictors. At each step, the predictor that is not yet included in the active set but has the biggest contribution to make to the model is identified, then the coefficients are iteratively moved toward the new least squares fitted values.

Some problems can happen if a non-zero coefficient crosses 0 as it is being updated. In such a case, that coefficient's predictor should be dropped from the active set. This happens when predictors are highly correlated.

3 SVD: Singular Value Decomposition

Singular Value Decomposition is a way to turn a matrix into the product of three matrices. If we have a matrix M, which for us in statistics is typically data and has dimensions $n \times p$, its singular value decomposition is $U\Sigma V^T$, where U is a unitary matrix of dimension $n \times p$, Σ is a square diagonal matrix of dimension $p \times p$ and V is another unitary matrix of dimension $p \times p$. A unitary matrix has the property that if it is multiplied by its transpose, you get an appropriately dimensioned identity matrix, so $UU^T = I$ and $VV^T = I$. Singular value decomposition can vastly simplify calculations on data because the unitary matrices turn into identities and Σ is full of zeros. SVD also plays a role in principal component analysis because it gives us quick access to the eigenvectors and eigenvalues of our data which are the components and the proportion of the variance in the data they explain respectively.

Below is an example of a singular value decomposition from Math 537 Multivariate Statistics

Let $A = \begin{bmatrix} 1 & 1 \\ 2 & -2 \\ 2 & 2 \end{bmatrix}$. Perform Singular Value Decomposition of A into $\Gamma\Lambda\Delta^T$

First find the eigenvalues and eigenvectors of A^TA. The squareroot of the eigenvalues go on the diagonal of Λ

$$A^TA = \begin{bmatrix} 9 & 1 \\ 1 & 9 \end{bmatrix} \qquad \implies \qquad \lambda_1 = 10, \lambda_2 = 8$$

$$\Lambda = \begin{bmatrix} \sqrt{10} & 0 \\ 0 & \sqrt{8} \end{bmatrix}$$

Δ contains the corresponding normalized eigenvectors of A^TA

$$\Delta = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

$$\implies \qquad \Delta^T = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix}$$

Γ is found by solving $A\Delta = \Gamma\Lambda$

$$A\Delta = \begin{bmatrix} \frac{2}{\sqrt{2}} & 0 \\ 0 & -\frac{4}{\sqrt{2}} \\ \frac{4}{\sqrt{2}} & 0 \end{bmatrix} = \Gamma \begin{bmatrix} \sqrt{10} & 0 \\ 0 & \sqrt{8} \end{bmatrix}$$

$$\implies \Gamma = \begin{bmatrix} \frac{1}{\sqrt{5}} & 0 \\ 0 & -1 \\ \frac{2}{\sqrt{5}} & 0 \end{bmatrix}$$

4 ISLR 10.2

This section will focus on Unsupervised Learning, where we are only concerned with a set of features $X_1, X_2, \ldots, X_P$ measured on n observations. More-so, we are not considering a response variable Y and, as a result, not interested in making predictions. The primary interest for Unsupervised learning is exploring the related subgroups of variables and data.

We will discuss *Principal Component Analysis* as one method for exploring the relationships found within our data. Principal Component Analysis, commonly referred to as PCA, is useful when dealing with a data set that contains a large set of correlated predictors. Our ultimate goal is to reduce the amount of variables by summarizing the data with a smaller amount of *representative* variables that still explain most of the variability in the original set. Here, it is important to note that *most* of the variability will be retained. We cannot retain all of the variability, since that would require keeping the entire predictor space and would prove the PCA to be null. The previously mentioned representative variables are indeed constructed from the correlated variables in the original data set. These variables are explained by *directions* in the feature space (or predictor space) along which the original data is highly variable.

PCA refers to both the computation/construction of the principal components and the exploratory use of these components to understand the data. We will further explore how these principal components are derived and how they are subsequently visualized.

Provided a predictor space of size p: $X_1, \ldots, X_P$ where we wish to perform an exploratory data analysis, we are initially interested in how these variables are related to one another. Keeping in mind that there is no Response measure such as R^2 to critique our variables, we are solely concerned with the actual variability within our variables. In a basic analysis, we are able to take two variables at a time and construct a scatter plot in order to explore any trends or relationships. Given a large predictor space, this can obviously prove to be an inefficient method. Most obviously, there are far too many plots to review for large predictor spaces. More importantly, we are merely analyzing the relationship between two variables at a time when we should ultimately be concerned with the overall interactions among the entire predictor space. This is the essence of *PCA*: we find a low-dimensional representation of a data set that contains as much as possible of the variation.

Going further, the underlying theory for PCA is that each of our n observations from the data set reside in a p-dimensional space where they are fully defined, but this space may contain many dimensions that are not useful or significant in terms of variation. Therefore, we begin a search for the most significant dimensions for all of our data, where each dimension in PCA is a linear combination of the p features (predictors).

Now, we will dive into the actual calculations for these *primary* dimensions, now referred to as *Components*. The first principal component is the *normalized* linear combination of features defined by

$$Z_1 = \phi_{11}X_1 + \phi_{21}X_2 + \cdots + \phi_{p1}X_p$$

that has the largest variance.

There are a few observations to be made regarding this first principal component definition. First, the term *normalized* refers to our ϕ_{1i} values (referred to as **Loadings**) whose squared values sum to 1. The normalized ϕ values provide an easy interpretation of which predictors are given importance in our principal component. These Loadings are weights that define the amount of contribution for the associated feature, X_i. In practice, the predictor space is often standardized with mean 0 and variance 1 so that the PCA process is able to correctly estimate the variance between features that may not be on the same scale. Second, we should place emphasis on the fact that the first principal component will consist of the linear combination that has the largest variance. Each subsequent principal component will be the linear combination that has the next largest variance.

Now, we can return to discussing how the Principal Component is found. Given our $n \times p$ data set X, we know that we are looking for the linear combination

$$Z_1 = \phi_{11}X_1 + \phi_{21}X_2 + \cdots + \phi_{p1}X_p$$

that has the largest sample variance and subject to the constraint $\sum_{j=1}^{p} \phi_{ji}^2 = 1$. Finding the largest sample variance leads to the following maximization:

$$\max_{\phi_{11},\ldots,\phi_{p1}} \left\{ \frac{1}{n} \sum_{i=1}^{n} \Big(\sum_{j=1}^{p} \phi_{j1} x_{ij} \Big)^2 \right\} \text{ subject to } \sum_{j=1}^{p} \phi_{j1}^2 = 1$$

We have previously mentioned that the x predictors have been standardized and therefore average to zero. Provided this fact and the structure for Z_{i1} (referred to as the *score*), we are then left with the following simplified maximization:

$$\max_{\phi_{11},\ldots,\phi_{p1}} \left\{ \frac{1}{n} \sum_{i=1}^{n} z_{i1}^2 \right\}$$

Although we have reduced the maximization equation, we still require eigen decomposition in order to solve for the principal component. Eigen decomposition is not deeply discussed in this chapter, but **Singular Value Decomposition** (summarized in a previous exercise) can be briefly described as the deconstruction of a matrix representing all of our linear combinations Z_{ij} into separate matrices representing orthogonal eigenvalues/eigenvectors that provide information regarding the optimal **loadings** for each principal component.

Having solved for the First Principal Component, we can now interpret the results in a geometric manner. Given our loading vector $\phi_1 = \{\phi_{11}, \phi_{21}, \ldots, \phi_{p1}\}$, we are able to define a direction in our feature space which the data vary the most (it follows that each subsequent loading vector ϕ_2 and so on will define the direction with the next largest variation). Projecting our data points $x_1, \ldots, x_n$ onto the direction of our loading vector ϕ_1 will produce the actual score Z_1. Each score can be represented as a coordinate on the plane that gives information regarding how the particular principal components interact with the given observation. This will be further illustrated in a plot below.

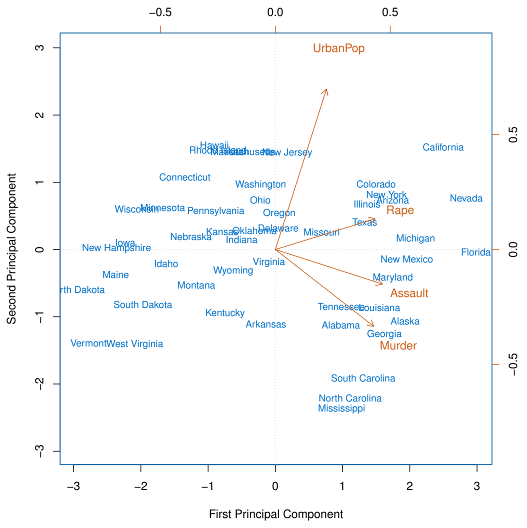
As the first principal component has been determined, we are able to repeat the process for each subsequent principal component (with a new search for the next maximized variance). It is important to note that the second principal component will need to be uncorrelated with the first principal component, as there should be no overlap of information or variance between any of the components. This leads to adding the mathematical requirement that Z_2 be orthogonal to Z_1. This process then continues with each new orthogonal constriction being added for each new principal component.

Once all principal components have been defined, we can plot them against each other in order to produce the desired low-dimensional view of the data. We will display this graphical view with the following useful example:

Example:

Consider the USArrests data set. For each of the 50 states in the United States, the data set contains the number of arrests per 100, 000 residents for each of three crimes: Assault, Murder, and Rape. We also record UrbanPop (the percent of the population in each state living in urban areas). The principal component score vectors have length n = 50, and the principal component loading vectors have length p = 4.

The following page displays a *biplot* that displays the findings from the PCA:



We can see that each of the 50 states are plotted along with the vectors defining the four features. Their length and direction corresponds to their contribution to the first and second principal component. We can see that the three variables *Rape, Assault and Murder* are all contributing mostly to our first Principal Component (defined along the x-axis) and the variable *UrbanPop* is contributing mostly to the second Principal Component (defined along the y-axis).

Two main conclusions can be made from this information. First and foremost, we can infer that the three predictors associated with the first component are correlated with one another. This makes sense when considering the fact that each of the three variables relate to violent crimes. Furthermore, this tells us that the first principal component itself is a component that is determined by violent crime, whereas the second principal component can almost entirely be determined by the *UrbanPop* variable. When considering the benefits of PCA, we have just seen a Predictor Space potentially reduce to 2 dimensions rather than the initial 4 dimensions.

Now looking beyond the predictors, we can focus in on our $n = 50$ states. Having each of them plotted onto our first two principal components, we are able to observe how each state is correlated to the first component relating to violent crimes and the the second component relating to the Urban Population. For example, *Mississippi* is located on the bottom right of our plot, implying that the state has a relatively high violent crime rate (first component) but a low urban population (second component). This is extremely useful when attempting to define a relationship between our observations and the predictor space for each observation.

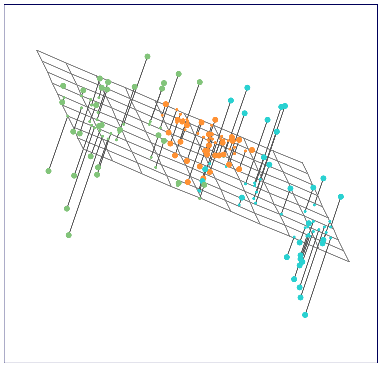
Another Interpretation of PCA

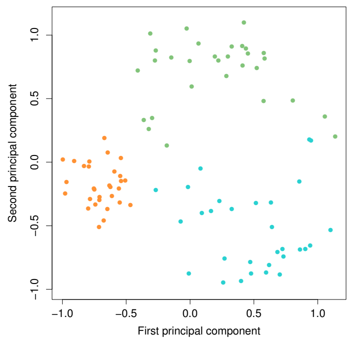
We have described principal components in terms of vectors with direction and magnitude along the p-dimensional predictor space, but we can also discuss PCA in terms of linear surfaces that are close to our n *observations*. More explicitly, the first principal component loading vector is the line in p-dimensional space that is closest to the n observations. This measure of "closeness" uses the average squared Euclidean distance (IE computing the square root of the sum of the squares of the differences between the loading vector and the observed values), which provides an easy interpretation for how distant our loading vector is to the observations (a useful illustration of this distance is provided below). We are strictly considering the first principal component, thus we have a single dimension of the data that lies as close as possible to all of the data points, thus providing a good summary of the data.

Of course, we can extend the component line with minimal observation distance to also include multiple components, thus resulting in a *plane* that is closest to the n observations. Together the first M principal component score vectors and the first M principal component loading vectors provide the best M-dimensional approximation to the ith observation x_{ij}. More specifically for standardized data, we have

$$x_{ij} \approx \sum_{m=1}^{M} z_{im}\phi_{im}$$

This implies that the actual observations can be approximated closely by a plane of principal components, where the approximation gets better as more principal components are added. When $M = min(n-1, p)$, we are no longer approximating but rather giving the exact observation x_{ij} values (as we have essentially loaded all of our original information from the feature space). It is evident that the objective for PCA is to minimize the size of M while maintaining the approximation to be as close to the observed values. Now, we will present two plots that illustrate PCA in terms of distance from observations:





The two plots above represent ninety observations simulated in three dimensions. On the left, we observe the first two principal component directions which span the plane that best fits the data. It minimizes the euclidean distance from each point to the plane. The plane created from these two principal components is the best attempt to account for all of the data.

On the right, we can observe the first two principal component score vectors with the coordinates of the projection of the 90 observations onto the plane. We can see that the observations are evenly distributed within the plane (no instance of excessive clustering), thus enforcing the desired maximized variance.

This illustration is only considering two principal components, yet still appear to minimize distance and account for a large amount of variance. An objective to consider for PCA is deciding on the number of Principal Components that will be chosen to represent our feature space.

More on PCA

It has already been mentioned that in order to perform PCA, the variables should be standardized (centered to have a mean of zero and standard deviation of one). This continues to be necessary because the results of PCA are dependent upon whether the variables have been individually scaled so that they are measured in the same units, where each variable would need to be multiplied by a different constant to make them comparable. Continuing with the *USArrests* data, the text provides an example of why this scaling is so essential. *Murder, Rape*, and *Assault* are given as the number of occurrences/100,000 people. *UrbanPop* is the percentage of the state's population that lives in an urban area. Not only is there a major difference in that some variables are counts while another is a percentage, the variances of each of these variables are quite different, 18.97, 87.73, 6945.16, and 209.5, respectively. If we left the variables unscaled and performed PCA, since *Assualt* and *UrbanPop* have such a high variance compared to the others, the first and second PCA loading vectors would have large loadings for these variables when maybe they shouldn't. This scenario would work in the opposite way if the variances were extremely small, the loadings would also be small.

Therefore scaling them to be more comparable to one another would give us more accurate loading results. If the variables are already measured in the same units, this scaling step may not always be necessary.

Further discussing the loading vectors, even though each vector is unique, flipping it's sign does not have any effect on the interpretation of the value. When using different software packages to perform PCA, the loading vector values should be the same while the signs may not. This is no cause for alarm. The reasoning is the vector specifies a direction in a p-dimensional space, it being positive or negative does not matter as long as the direction is correct and does not change. Score vectors follow the same rule and line of reasoning as the variance of Z does not change if it is positive or negative.

The section continues with an important question regarding the variance explained by PCA, that is how much of the variance in the data is not contained in the first few principal components? In other words, how much information is lost by projecting the observations onto the first principal components? We may be lowering the number of variables by using PCA, but at what cost. Using the *proportion of variance explained* (PVE), we are able to see just how much variance of the original data is explained by each new loading. To start we can find the *total variance* within the data that has already been scaled and is calculated as,

$$\sum_{j=1}^{p} Var(X_j) = \sum_{j=1}^{p} \frac{1}{n} \sum_{i=1}^{n} x_{ij}^2$$

Next, we find the variance explained by the m^{th} principal component using,

$$\frac{1}{n} \sum_{i=1}^{n} z_{im}^2 = \frac{1}{n} \sum_{i=1}^{n} \left(\sum_{j=1}^{p} \phi_{jm} x_{ij} \right)^2$$

Setting these as a ratio of the variance explained by the m^{th} principal component over the *total variance* we have the PVE equation below.

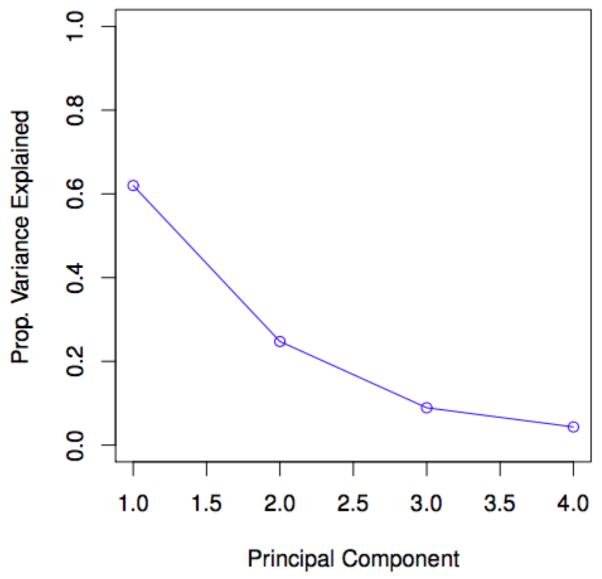
$$\frac{\sum_{i=1}^{n} \left(\sum_{j=1}^{p} \phi_{jm} x_{ij} \right)^2}{\sum_{j=1}^{p} \sum_{i=1}^{n} x_{ij}^2}$$

Note: The value produced will be positive and between 0 and 1.

Finding the cumulative PVE of the first M principal components (PCs) may also be informative to understand how much of the original variance is explained by multiple PCs. This can be done by simply summing over the M PC's PVE. After PCA there will be either *n-1* or p PCs, whichever is smaller, and all of their PVEs will sum to 1, explaining all of the original variance.

The discussion of the PCs in this context brings us to the next question, how do we decide on the appropriate number of PCs? We know we do not want to use all of them or we could have simply stuck with the original data. The goal of PCA is to select the smallest number of PCs so that we may still have an adequate understanding of the original data, just with fewer variables. But to decide on just how many that is, there is no one clear cut method and those used are often subjective or depend on the specific area of application and specific data set.

One way to attempt this is by using a *scree plot* as seen in the plot below. We want to select the least amount of PCs that explain the highest amount of the variance from the original data. This is visualized by the scree plot. The x-axis denotes the PCs and the y-axis the amount of variance that PC explains. Looking for the *elbow*, or the point at which the proportion of variance explained by each increasing PC stops decreasing as quickly and levels off, we can see this happens sometime after the 2nd PC. The first 2 PCs explain 60% and 20% of the variance and after that only 10% and 5%, which is not very much. In this case we might use the first 2 PCs.



Another method might be to directly look at the first few principal components to see if there are any noticeable patterns in the data. If the first few PCs produce nothing interesting, it is safe to say the remaining PCs will not either. Similarly, if the first few components produce noticeable patterns it might be worth looking into the following PCs until nothing else interesting is found. As PCA is a tool most often used for exploratory analysis, this subjective approach seems to also fit into that category well.

One last method is discussed that is less subjective and can be used in a supervised analysis such as principal components regression. In this method, we use cross-validation or a similar approach on the number of PC score vectors to be used in regression to find the number that produces the best fit. A method we are very familiar with at this point.

The section concludes with a quick discussion of the other uses that PCs might have. Many statistical methods like regression, classification, and clustering, can be modified to use the user selected number of PC score vectors found through PCA (each PC as a column in the new data set) as variables instead of simply using the original data. This often leads to less *noise* (discussed in the previous assignment's summary), concentrating the *signal* information in the first few principal components.

5 ISLR Book Problems

6.8

In this exercise, we will generate simulated data, and will then use this data to perform best subset selection.

A

Use the rnorm() function to generate a predictor X of length n = 100, as well as a noise vector ϵ of length n = 100.

Here is the generated data:

```r
set.seed(533)
x = rnorm(100) #Predictor
e = rnorm(100) #Noise
```

B

Generate a response vector Y of length n = 100 according to the model

$$Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3 + \epsilon, \text{ where } \beta_0, \beta_1, \beta_2, \text{ and } \beta_3 \text{ are constants of your choice}$$

Using our generated data, here is the 3rd degree polynomial with $\beta_0 = 1$, $\beta_1 = 11$, $\beta_2 = 21$ and $\beta_3 = 31$:

```r
y=1+11*x+21*x^2+31*x^3+e
```

C

Use the regsubsets() function to perform best subset selection in order to choose the best model containing the predictors $X, X^2, ..., X^{10}$. What is the best model obtained according to Cp, BIC, and adjusted R^2? Show some plots to provide evidence for your answer, and report the coefficients of the best model obtained

First, we will create a model of ten degrees:

```r
ten= poly(x,10,raw = T)
```

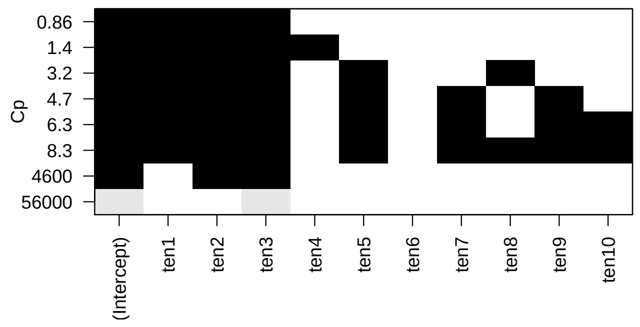
Now, we can perform a Best Subset Selection:

```r
bestsubset =regsubsets(y~ten,data=data.frame(x,y),method = 'exhaustive')
reg.summary = summary(bestsubset)
```

Considering C_p, BIC and adjusted R^2, we are able to display their corresponding values for variations of the full 10^{th} degree model as follows:

C_p

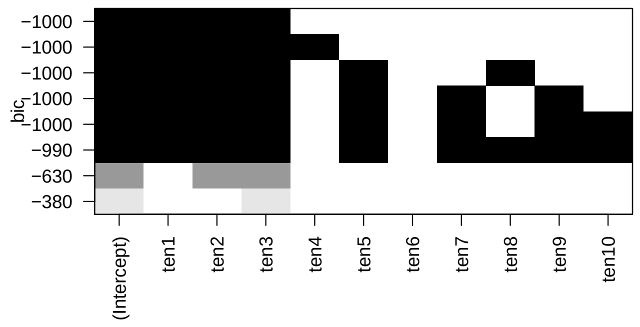
```r
plot(bestsubset,scale="Cp")
```



Here, we are seeking the model with the lowest C_p value. We can see that a third degree model produces the lowest C_p value, which corresponds to our original generation of y.

BIC:

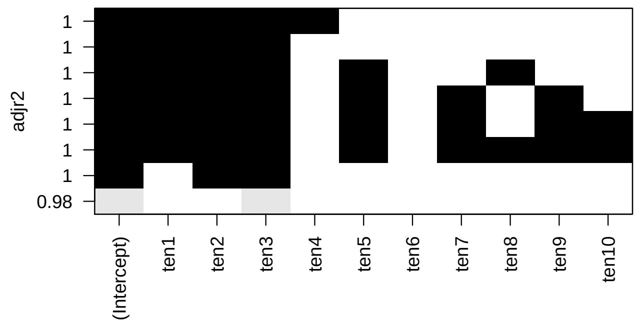
```
plot(bestsubset,scale="bic")
```



Here, we are looking for the model that contains the highest (in magnitude) BIC values. Once again, it appears that the 3rd degree polynomial is.

adj R^2:

```
plot(bestsubset,scale="adjr2")
```



From these plots, it is now apparent that the model defined by

$$Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3$$

is the best subset for our defined response Y. Having established a 3rd degree polynomial as being optimal, we will extract the respective coefficients and state the final model:

```
best_subsets_coef =coef(bestsubset ,3)
```

Y = 1.0105582 + 10.8419996X + 20.9616608X^2 31.0408136X^3

We can observe that the estimated coefficients via best subset are strikingly close to the original β values used to generate the data.

D

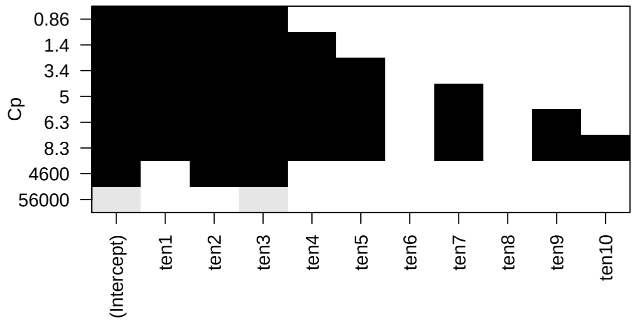
Repeat (c), using forward stepwise selection and also using back- wards stepwise selection. How does your answer compare to the results in (c)?

Forward Stepwise Selection

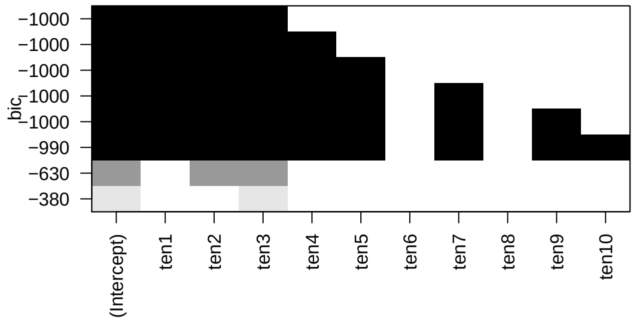
```
forward=regsubsets(y~ten,data=data.frame(x,y),method = 'forward')
forward.summary = summary(forward)
```

Considering the ranking of C_p, BIC and adjusted R^2, we are able to display the selected variables for the best model with a given number of predictors. We will now show the mentioned plots:

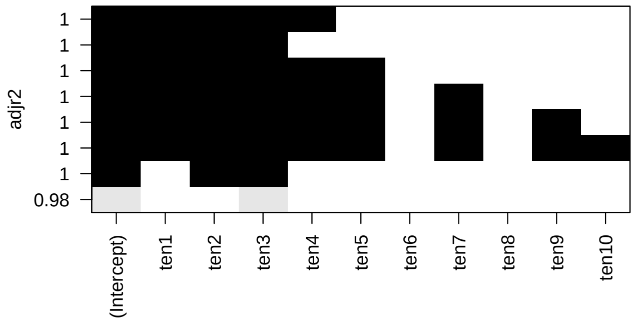
```
plot(forward,scale="Cp")
```



```
plot(forward,scale="bic")
```



```
plot(forward,scale="adjr2")
```



Considering all three of our measures of fit, we can see that the Forward Selection method is in agreement with the Best Subset method, thus selecting the 3^{rd} degree model.

From these plots, it is now apparent that the model defined by

$$Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3$$

is the best subset for our defined response Y.

As before, here are the coefficients for our Forward Selection Model:

```r
forward_coef = coef(forward ,3)
```

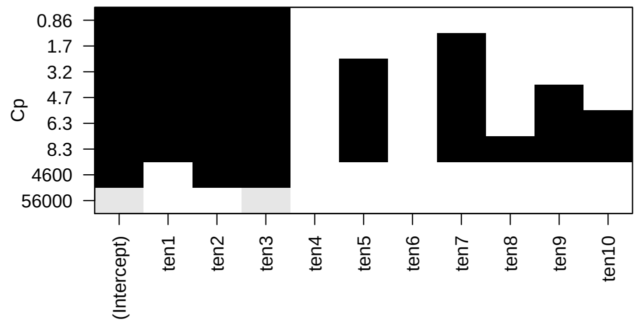
Y = 1.0105582 + 10.8419996X + 20.9616608X^2 + 31.0408136X^3

Backwards Stepwise Selection

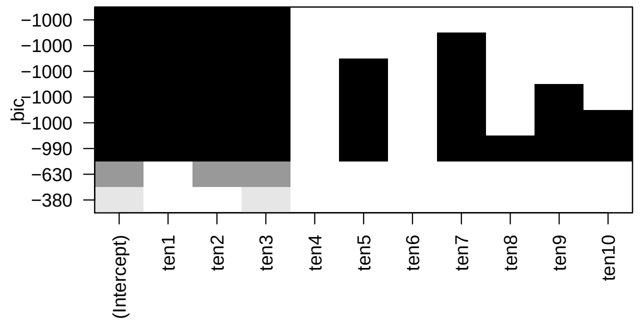
```r
backwards=regsubsets(y~ten,data=data.frame(x,y),method = 'backward')
backwards.summary = summary(backwards)
```

Considering the ranking of C_p, BIC and adjusted R^2, we are able to display the selected variables for the best model with a given number of predictors. We will now show the mentioned plots:

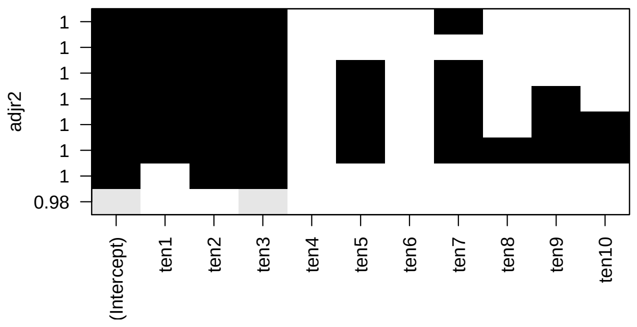
```r
plot(backwards,scale="Cp")
```



```r
plot(backwards,scale="bic")
```



```r
plot(backwards,scale="adjr2")
```



Considering all three of our measures of fit, we can see that the Backwards Selection method is in agreement with the Best Subset method.

From these plots, it is now apparent that the model defined by

$$Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3$$

is the best subset for our defined response Y.

As before, here are the coefficients for our Forward Selection Model:

```r
backward_coef = coef(backwards ,3)
```

Y = 1.0105582 + 10.8419996X + 20.9616608X^2 + 31.0408136X^3

E

Now fit a lasso model to the simulated data, again using $X, X2, ..., X10$ as predictors. Use cross-validation to select the optimal value of λ. Create plots of the cross-validation error as a function of λ. Report the resulting coefficient estimates, and discuss the results obtained.

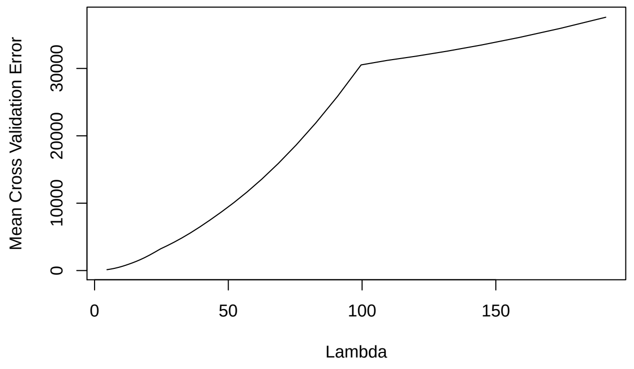
We will fit a lasso model to the 10ht degree polynomial and select the cross-validation option in order to find the optimal λ value:

```r
y.lasso.cv=cv.glmnet(ten,y)
opt_lambda = y.lasso.cv$lambda.min
```

The optimal $\lambda = 4.6249697$

Here is a plot displaying the cross-validation error as a function of λ:

```r
plot(data.frame(y.lasso.cv$lambda,y.lasso.cv$cvm),xlab='Lambda',
     ylab='Mean Cross Validation Error',type='l')
```



From the graph, we can see that the Mean Cross Validated Error increases dramatically as we move away from our optimal λ. As expected, cross-validated error is minimized at our optimal λ.

Now, we will report the coefficients for a model when using LASSO with the cross validated optimal λ:

```
predict(y.lasso.cv,s = y.lasso.cv$lambda.min,type='coefficients')
```

```
11 x 1 sparse Matrix of class "dgCMatrix"
                     1
(Intercept)  4.464722
1            6.821750
2           18.455261
3           31.189616
4            .
5            .
6            .
7            .
8            .
9            .
10           .
```

We can see that LASSO gives weight to the first four β estimates (similar to our subset methods) and gives no weight to the remaining predictors. We can also observe that the coefficient estimates are not as similar to the originally generated data, when compared to the subset selection methods.

F

Now generate a response vector Y according to the model $Y = \beta_0 + \beta_7 X^7 + \epsilon$, and perform best subset selection and the lasso. Discuss the results obtained.

We will generate a response vector Y according to the model

$$Y = \beta_0 + \beta_7 X^7 + \epsilon$$

and perform the best subset selection and LASSO.

Once again, here is our generated data with $\beta_1 = 1$, $\beta_7 = 7$:

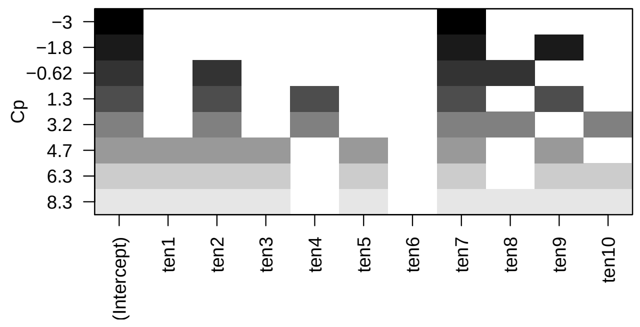
```
Y2=1+7*x^7+e
```

Using our new data, we can perform Best Subset selection:

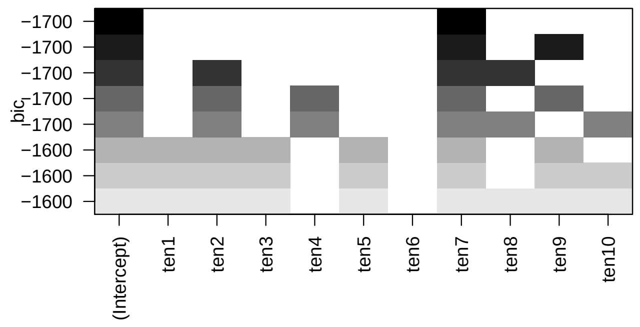
```
bestsubset=regsubsets(Y2~ten,data=data.frame(x,y),method = 'exhaustive')
```

Considering C_p, BIC and adjusted R^2, we are able to display their corresponding values for variations of the full 10th degree model as follows:

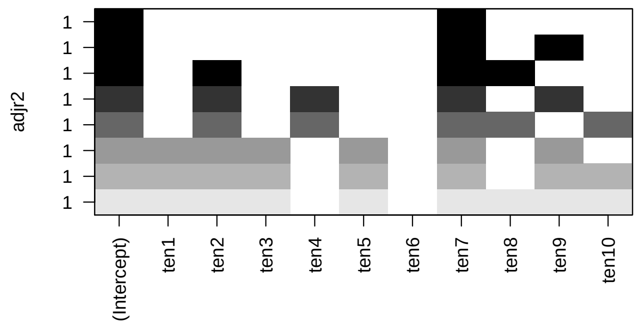
```
plot(bestsubset,scale="Cp")
```



```
plot(bestsubset,scale="bic")
```



```
plot(bestsubset,scale="adjr2")
```



From these plots, it is now apparent that the objective model (including only β_0 and β_7) is indeed selected when using any of our measures of fit.

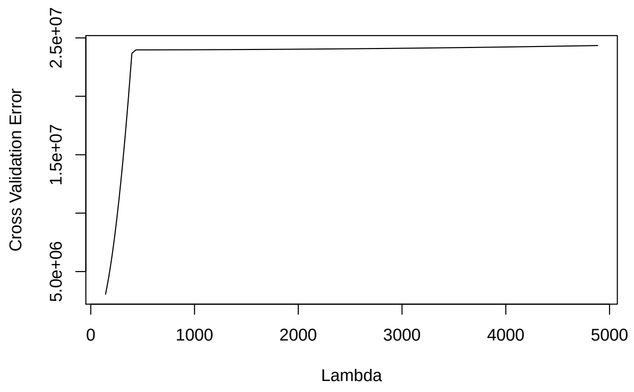
LASSO

```
y2.lasso.cv=cv.glmnet(ten,Y2)
opt_lambda = y2.lasso.cv$lambda.min
```

The optimal $\lambda = 142.4145382$

Here is a plot displaying the cross-validation error as a function of λ:

```
plot(data.frame(y2.lasso.cv$lambda,y2.lasso.cv$cvm),
xlab='Lambda',ylab='Cross Validation Error',type='l')
```



```r
predict(y.lasso.cv,s = y2.lasso.cv$lambda.min,type='coefficients')
```

```
11 x 1 sparse Matrix of class "dgCMatrix"
                    1
(Intercept) 39.533616
1             .
2             .
3            9.264693
4             .
5             .
6             .
7             .
8             .
9             .
10            .
```

In this instance, X_3 is selected as the best predictor, as opposed to the generated X_7 from our response Y. Since the models using X^3 and X^7 are *odd* functions, Lasso may have found X^3 to be a predictor that best estimates X^7 and reduces MSE.

6.9

In this exercise, we will predict the number of applications received using the other variables in the College data set.

A

```r
college = College
```

Split the data set into a training set and a test set.

```r
set.seed(533)
n=nrow(College)
train=sample(1:n,n/2)
test=-train
y.test=College[test,2]
```

B

Fit a linear model using least squares on the training set, and report the test error obtained.

Here is the least squares linear model using the training set:

```
fit =lm(Apps~.,data=College[train,])
#summary(fit)
```

Now, we will compute the test error by using our least squares training model to make predictions for the test data. We will then compute the mean squared error for our test data:

```
pred=predict(fit,newdata =College[test,])
error = (pred-College[test,2])^2
test_error = round(mean(error))
```

The MSE obtained is 1272298

C

Fit a ridge regression model on the training set, with λ chosen by cross validation. Report the test error obtained

First, we will create a grid for the train and test data:

```
grid = 10^seq(4, -2, length=100)
train.mat = model.matrix(Apps~., data=College[train,])
test.mat = model.matrix(Apps~., data=College[test,])
```

Now, using our training data, we will perform a cross-validated ridge regression in order to find the optimal λ:

```
ridge.cv=cv.glmnet(x=train.mat,y=College[train,2],alpha=0)
cv_lambda = ridge.cv$lambda.min
```

The optimal λ found via cross validation = 426.

Now, we will use our optimal training ridge regression and make predictions from our test data:

```
ridge.fit=glmnet(x=train.mat,y=College[train,"Apps"],alpha=0,lambda = ridge.cv$lambda.min)
ridge.pred=predict(ridge.fit,newx=test.mat )
```

Using these predictions, we are able to compute MSE in order to represent the test error obtained:

```
MSE_ridge = round(mean((ridge.pred-y.test)^2))
```

The MSE obtained via Ridge regression is 1075256

From our Ridge regression, we can report the coefficients obtained when using the Cross Validated λ:

```
ridge.coef=predict(ridge.fit,type='coefficients',s=ridge.cv$lambda.min)
ridge.coef
```

```
19 x 1 sparse Matrix of class "dgCMatrix"
                        1
(Intercept) -1.453888e+03
(Intercept)  .
PrivateYes  -5.296351e+02
Accept       1.084006e+00
Enroll       2.543016e-01
Top10perc    2.310837e+01
Top25perc    4.511081e+00
F.Undergrad  5.881810e-02
```

```
P.Undergrad  4.362714e-02
Outstate    -3.259400e-02
Room.Board   2.044799e-01
Books        3.056859e-01
Personal     2.681017e-02
PhD         -7.244786e+00
Terminal    -6.026523e-01
S.F.Ratio    1.420417e+01
perc.alumni -9.833154e+00
Expend       5.238084e-02
Grad.Rate    9.435253e+00
```

We can observe that there are a few predictors that are given minimal weight in the Ridge model, such as **P.Undergrad, Books, Terminal**.

D

Fit a lasso model on the training set, with λ chosen by cross- validation. Report the test error obtained, along with the number of non-zero coefficient estimates.

Using our training data, we will perform a cross-validated ridge regression in order to find the optimal λ:

```r
lasso.cv=cv.glmnet(train.mat,y=College[train,2],alpha=1)
```

The optimal λ found via cross validation = 426.

Now, we will use our optimal training Lasso regression and make predictions from our test data:

```r
lasso.fit=glmnet(x=train.mat,y=College[train,2],alpha=1,lambda = lasso.cv$lambda.min)
lasso.pred=predict(lasso.fit,newx=test.mat )
```

Using these predictions, we are able to compute MSE in order to represent the test error obtained:

```r
MSE_lasso = round(mean((lasso.pred-y.test)^2))
```

The MSE obtained via Lasso regression is 1262360.

Having computed MSE, for all three models, we can compare all three of our Test MSE values in the following table:

	Test MSE
Least Sqaures	1272298
Ridge	1075256
Lasso	1262360

As provided before, here are the coefficients for our Lasso model:

```r
lasso.coef=predict(lasso.fit,type='coefficients',s=lasso.cv$lambda.min)
lasso.coef
```

```
19 x 1 sparse Matrix of class "dgCMatrix"
                       1
(Intercept) -409.36370606
(Intercept)    .
PrivateYes  -336.46184762
Accept         1.75674818
Enroll        -1.41749079
```

```
Top10perc     39.55263040
Top25perc     -4.83064529
F.Undergrad    0.07287822
P.Undergrad    .
Outstate      -0.10363250
Room.Board     0.16837572
Books          .
Personal       0.07308782
PhD           -9.18009374
Terminal       .
S.F.Ratio     15.03947251
perc.alumni    1.96437223
Expend         0.04774238
Grad.Rate      3.85424509
```

We can observe that some of the predictors that had minimal weight in the Ridge regression now hold no weight in our Lasso Regression: **P.Undergrad, Books, Terminal**. This gives us a total of 16 non-zero coefficient estimates.

8.8

In the lab, a classification tree was applied to the Carseats data set after converting Sales into a qualitative response variable. Now we will seek to predict Sales using regression trees and related approaches, treating the response as a quantitative variable.

A

Split the data set into a training set and a test set.

```
set.seed(533)
train=sample(1:nrow(Carseats),200)
Carseats.test=Carseats[-train ,]
```

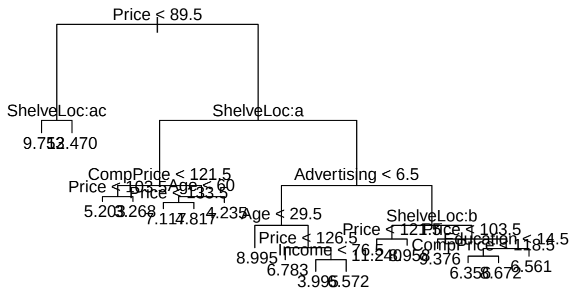
B

Fit a regression tree to the training set. Plot the tree, and interpret the results. What test MSE do you obtain?

```
tree.carseats=tree(Sales~.,data=Carseats,subset = train)
tree.pred=predict(tree.carseats,Carseats.test)
```

We will plot the Regression tree for the training set:

```
plot(tree.carseats)
text(tree.carseats)
```



From our Regression Tree, we can see that Price appears to be the most important predictor, followed by Shelve Location.

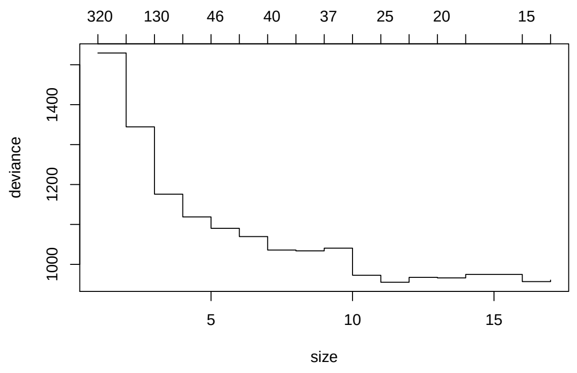
```r
tree_mse = mean((tree.pred-Carseats[-train,'Sales'])^2)
```

The test MSE obtained via regression trees is 5.8249775.

C

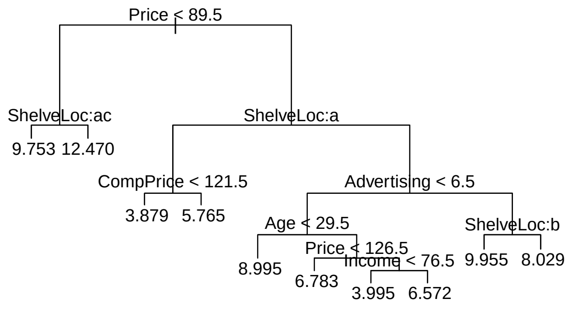
Use cross-validation in order to determine the optimal level of tree complexity. Does pruning the tree improve the test MSE?

```r
tree.carseats.cv=cv.tree(tree.carseats)
plot(tree.carseats.cv)
```



By inspection of the deviance plot, it is apparent that the optimal size (when deviance is minimized) for our tree is approximately 10.

```r
prune.carseats=prune.tree(tree.carseats,best=10)
plot(prune.carseats)
text(prune.carseats)
```



From the pruned tree, we can see that Price and Shelve Location have maintained their respective positions of importance.

```
tree.pred=predict(prune.carseats,Carseats[-train,])
tree_prune_mse = mean((tree.pred-Carseats[-train,'Sales'])^2)
```

The test MSE obtained via pruned regression trees is 6.1434726.

From a comparison of the two MSE values, we can observe that pruning did not improve the test MSE.

D

Use the bagging approach in order to analyze this data. What test MSE do you obtain? Use the importance() function to determine which variables are most important.

For our bagging approach, we will be performing a random forest that includes all of our predictors.

```
all = ncol(Carseats)-1
set.seed(533)
bag.carseats=randomForest(Sales~.,data=Carseats,subset=train,
                          mtry=all,importance=T,ntree=100)

tree.pred=predict(bag.carseats,Carseats[-train,])
bagging_mse= mean((tree.pred-Carseats[-train,'Sales'])^2)
```

The test MSE obtained via pruned regression trees is 6.1434726.

Here is a table that displays the importance function for our variable set:

	%IncMSE	IncNodePurity
CompPrice	8.2597015	132.109295
Income	2.6745948	84.187764
Advertising	8.1118331	134.668063
Population	-1.0274256	70.126465
Price	23.7679345	451.672342
ShelveLoc	17.1367406	325.388014
Age	7.4366211	192.785008
Education	0.8092127	36.490666
Urban	-0.3726536	6.354222
US	0.1482552	6.463867

The first column represents the amount that the RSS is decreased as a result of splits over the given predictor, where a higher value represents a more important predictor. The second column represents the increase in node purity, or how well the tree splits the data, that results from splits over that specific predictor. Again, a higher value shows higher importance.

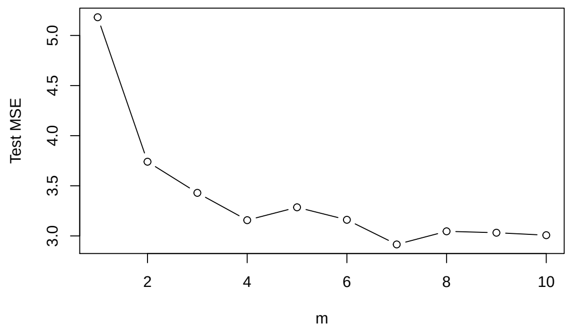
Here, we can see that Price and Shelve Location are clearly the most important variables in our model, with Price carrying slightly more importance.

E

Use random forests to analyze this data. What test MSE do you obtain? Use the importance() function to determine which variables are most important. Describe the effect of m, the number of variables considered at each split, on the error rate obtained.

Bagging was a specific example of Random Forests where all of our predictors were utilized. Now, we will check to see which amount of predictors produces the lowest Test MSE:

```r
set.seed(533)
mse = rep(0,10)
#Find MSE for each m size
for(i in 1:10){
  forest.carseats=randomForest(Sales~.,data=Carseats,subset=train,mtry=i,importance=T,ntree=100)
  tree.pred=predict(forest.carseats,Carseats[-train,])
  mse[i] = mean((tree.pred - Carseats[-train,'Sales'])^2)
}
plot(1:10,mse,type='b',xlab = "m",ylab="Test MSE")
```



In this particular case, the MSE is minimized when mtry (the number of predictors) = 7. By default, randomForest() uses p/3 variables when building a random forest of regression trees.

Here is a table that displays the importance function for our variable set:

	%IncMSE	IncNodePurity
CompPrice	6.6110511	133.126280
Income	3.7621621	80.877199
Advertising	10.3128672	117.910129
Population	0.0161844	83.702723
Price	20.8925104	456.927661
ShelveLoc	21.8803259	348.792467
Age	9.2419381	183.525280
Education	0.9087240	44.124515
Urban	-0.5393639	7.810728
US	0.7101852	11.496831

When performing Random Forests with seven predictors, we can see that Shelve Location is now the most important predictor, with Price closely in second.

8.9

This problem involves the OJ data set which is part of the ISLR package.

A

Create a training set containing a random sample of 800 observations, and a test set containing the remaining observations.

```r
train=sample(1:nrow(OJ),800,replace=F)
OJ.train=OJ[train,]
OJ.test=OJ[-train,]
```

B

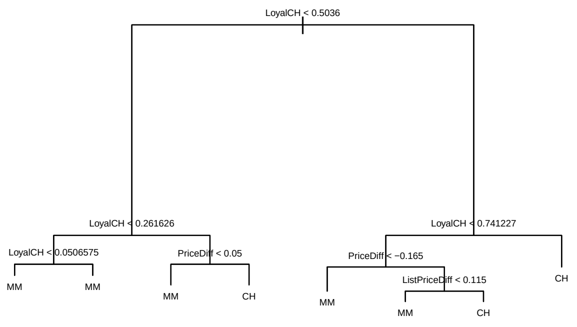
Fit a tree to the training data, with Purchase as the response and the other variables as predictors. Use the summary() function to produce summary statistics about the tree, and describe the results obtained. What is the training error rate? How many terminal nodes does the tree have?

Here is our tree fit to the training data along with a corresponding summary:

```r
OJ.tree=tree(Purchase~.,data=OJ.train)
OJ_summary = summary(OJ.tree)
classification_percent = round(100*OJ_summary$misclass[1]/OJ_summary$misclass[2])
number_nodes=OJ_summary$size
OJ_summary
```

```
Classification tree:
tree(formula = Purchase ~ ., data = OJ.train)
Variables actually used in tree construction:
[1] "LoyalCH"       "PriceDiff"     "ListPriceDiff"
Number of terminal nodes:  8
Residual mean deviance:  0.725 = 574.2 / 792
Misclassification error rate: 0.1525 = 122 / 800
```

From the summary, we can observe the training error rate to be about 15 %. We can also observe the number of terminal nodes to be 8 . This node number can be displayed in the following tree:

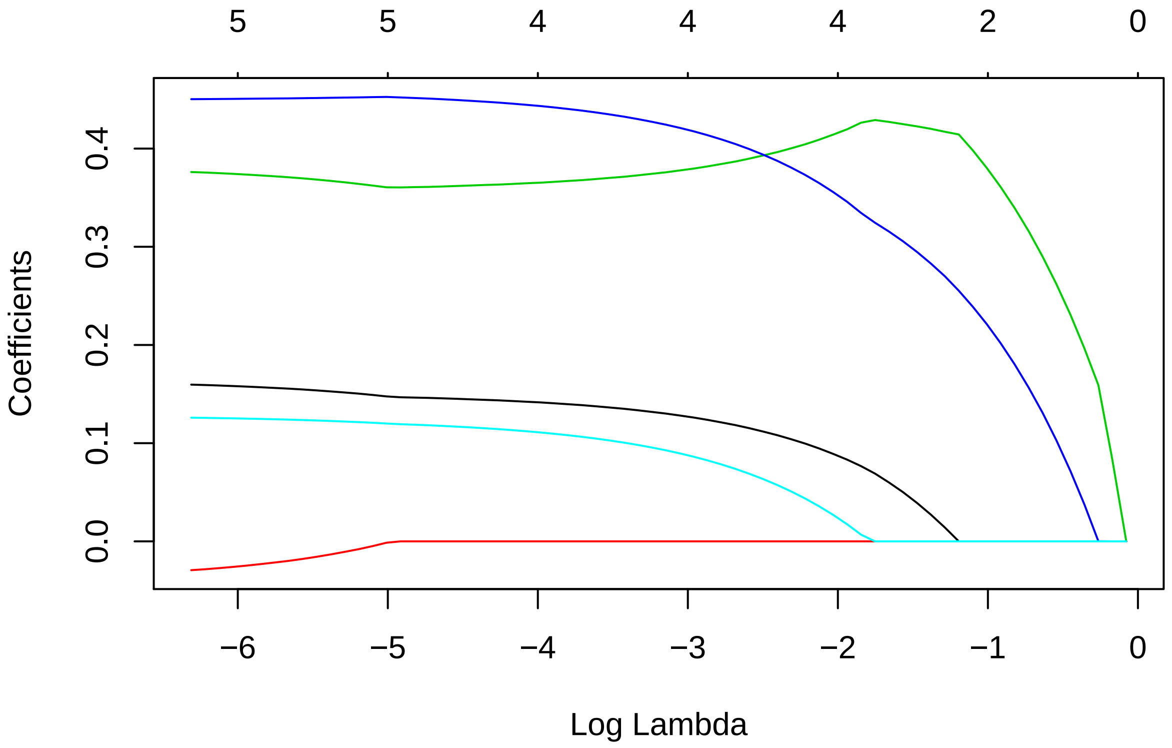


Bonus: Optimal Lasso Lambda

```r
#get the Fullerton Housing data and get rid of any rows that have missing values
data = read.csv("FullertonHousing.csv")
data = na.omit(data)
data = data[!is.na(data$LOT_SIZE),]

#Get your data into predictors and response; standardize them
X = scale(data[,c("BEDS","BATHS","SQUARE_FEET","LOT_SIZE","YEAR_BUILT")])
Y = scale(data[,"PRICE"])
n = dim(X)[1]

#Get an initial sense of how the parameters will change with lambda
model = glmnet(x=X,y=Y,family="gaussian", alpha=1)
plot(model, xvar="lambda")
```



Here is the Cross Validation Code to compute MSE provided a λ value:

```r
#Create your test index to be used for 10-fold cross validation
test.index = vector("list",10)
temp = sample(1:n,n,replace=FALSE)
for (i in 1:9){
  test.index[[i]] = temp[((i-1)*trunc(n/10)+1):(i*trunc(n/10))]
}
test.index[[10]] = temp[(9*trunc(n/10)+1):n]

#Create a function that will take a lambda and return the cross-validated MSE
find_lambda = function(lambda){
  print(lambda)
  MSE = 0
  for (i in 1:10){
    model = glmnet(x=as.matrix(X[-test.index[[i]],]), y=Y[-test.index[[i]]],
                   family="gaussian", alpha=1, lambda=lambda)
    yhat = predict(model, newx=as.matrix(X[test.index[[i]],]), type="response")
    MSE = MSE + sum((Y[test.index[[i]]]-yhat)^2)/n
  }
  return(MSE)
}
```

We will use the optim function to find the λ value that minimized the Cross Validated MSE:

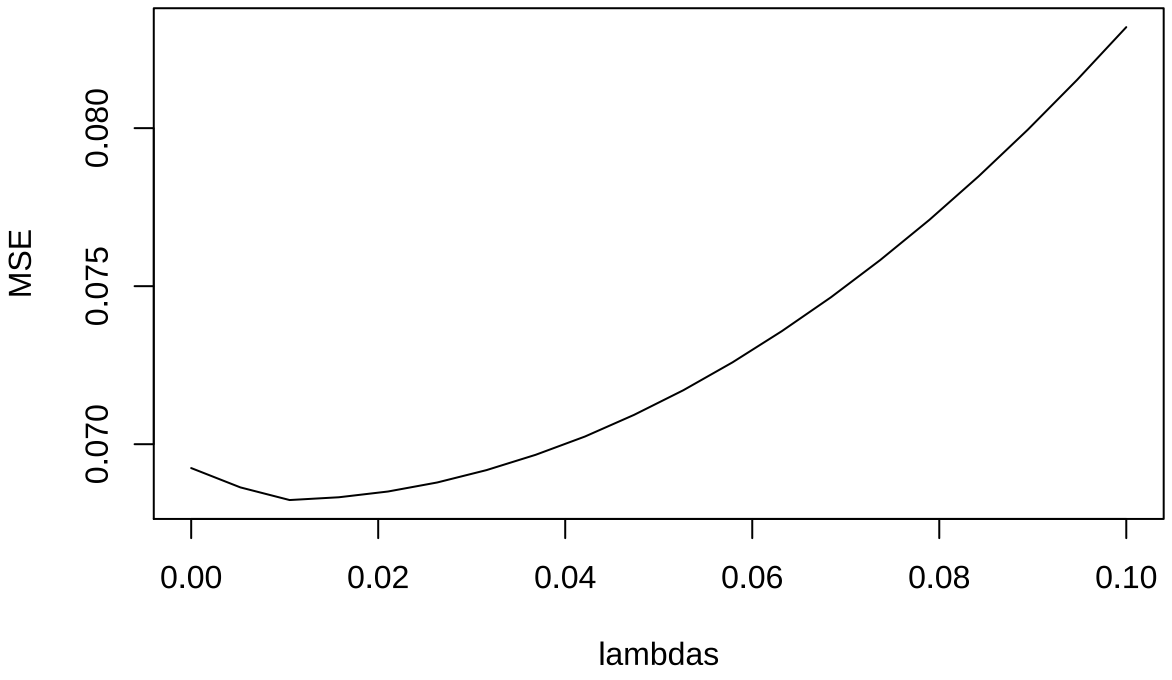
```r
#Get the lambda that minimizes cross-validated MSE using the optim function
my_lambda = optim(par=0, fn=find_lambda, lower=0, method="L-BFGS-B")$par
```

The λ value that minimizes cross-validated MSE using the optim function is 0.0094235.

Here is a plot of various lambdas and their respective MSE:

```r
#Show that my_lambda does indeed minimize the MSE
lambdas = seq(0,0.1,length.out=20)
MSE= lapply(lambdas,find_lambda)
```

```r
plot(lambdas,MSE,type="l")
```



We can observe that our optimized λ does indeed minimize MSE.

As a comparison, we will compute the optimal λ value via the cv.glmnet function:

```r
#Now get the best lambda according to cv.glmnet
glmnet_lambda = cv.glmnet(x=X, y=Y)$lambda.min
```

The glmnet optimal estimate for $\lambda = 0.0185929$. This λ value jumps around quite a bit, but we can compute the smallest λ that is within 1 standard error of the minimum value: 0.0992248.