Machine Learning Using R

Basic

Mohammad Wasiq
DATA SCIENCE

Basic Machine Learning

Mohammad Wasiq

Table of Contents

1 Machine Learning with Statology

2 Introduction to Machine Learning

The field of machine learning contains a massive set of algorithms that can be used for understanding data. These algorithms can be classified into one of two categories:
1. **Supervised Learning Algorithms :** Involves building a model to estimate or predict an output based on one or more inputs.
2. **Unsupervised Learning Algorithms :** Involves finding structure and relationships from inputs. There is no "supervising" output.

2.1 Supervised Learning Algorithms

A **Supervised Learning Algorithms** can be used when we have one or more *explanatory variables* $(X_1, X_2, X_3, \cdots, X_p)$ and a *response variable* (Y) and we would like to find some function that describes the relationship between the explanatory variables and the response variable:

$$Y = f(X) + \epsilon$$

where, **f** represents systematic information that **X** provides about Y and where **ε** is a random error term independent of X with a mean of zero.

Thee are two main types of *Supervised Learning algorithms :*
1. **Regression :** The output variable is continuous. (e.g. weight, height, time, etc.)

2. **Classification :** The output variable is categorical. (e.g. male or female, pass or fail, benign or malignant, etc.)

There are two main reasons that we use supervised learning algorithms:

A. **Prediction :** We often use a set of explanatory variables to predict the value of some response variable . (e.g. using square footage and number of bedrooms to predict home price)

B. **Inference :** We may be interested in understanding the way that a response variable is affected as the value of the explanatory variables change . (e.g. how much does home price increase, on average, when the number of bedrooms increases by one ?)

Depending on whether our goal is inference or prediction (or a mix of both), we may use different methods for estimating the function f. For example, linear models offer easier interpretation but non-linear models that are difficult to interpret may offer more accurate prediction.

Hee is list of most commonly used supervised learning algorithms : -
Linear Regression - **Logistic Regression** - **Linear Discriminant Analysis** - **Quadratic Discriminant Analysis** - **Decision Trees** - **Naive Bayes** - **Support Vector Machines** - **Neural Networks**

2.2 Unsupervised Learning Algorithms

An **Unsupervised Learning Algorithm** can be used when we have a list of variables $(X_1, X_2, X_3, \cdots, X_p)$ and we would simply like to find underlying structure or patterns within the data.

There are two main types of unsupervised learning algorithms:

1. **Clustering :** Using these types of algorithms, we attempt to find "clusters" of observations in a dataset that are similar to each other. This is often used in retail when a company would like to identify clusters of customers who have similar shopping habits so that they can create specific marketing strategies that target certain clusters of customers.

2. **Association :** Using these types of algorithms, we attempt to find "rules" that can be used to draw associations. For example, retailers may develop an association algorithm that says "if a customer buys product X they are highly likely to also buy product Y."

Here is a list of the most commonly used unsupervised learning algorithms : -
Principal Component Analysis , **K-Means Clustering** , **K-Medoids Clustering** , **Hierarchical Clustering** , **Apriori Algorithm**

2.2.0.1 Regression

Regression : The response variable is continuous. For example, the response variable could be: - Weight - Height - Price - Time - Total units In each case, a regression model seeks to predict a continuous quantity.

Regression Example :
i. Suppose we have a dataset that contains three variables for 100 different houses: square footage, number of bathrooms, and selling price.

ii. We could fit a regression model that uses square footage and number of bathrooms as explanatory variables and selling price as the response variable.

iii. We could then use this model to predict the selling price of a house, based on its square footage and number of bathrooms.

iv. This is an example of a regression model because the response variable (selling price) is continuous.

The most common way to measure the **accuracy** of a **regression model** is by calculating the **Root Mean Square Error (RMSE)** , a metric that tells us how far apart our predicted values are from our observed values in a model, on average.
It is calculated as:

$$RMSE = \sqrt{\frac{\sum(P_i - O_i)}{n}}$$

where, $\sum$ is a fancy symbol that means "sum", P_i is the predicted value for the ith observation , O_i is the observed value for the ith observation n is the sample size

The *smaller* the *RMSE* , the better a regression model is able to fit the data .

2.2.0.2 Classification

Classification : The response variable is categorical.

For example, the response variable could take on the following values : - Male or Female - Pass or Fail - Low, Medium, or High In each case, a classification model seeks to predict some class label.

Classification Example : 1. Suppose we have a dataset that contains three variables for 100 different college basketball players: average points per game, division level, and whether or not they got drafted into the NBA. 2. We could fit a classification model that uses average points per game and division level as explanatory variables and "drafted" as the response variable. 3. We could then use this model to predict whether or not a given player will get drafted into the NBA based on their average points per game and division level. 4. This is an example of a classification model because the response variable ("drafted") is categorical. That is, it can only take on values in two different categories: "Drafted" or "Not drafted."

The most common way to measure the **accuracy** of a *classification model* is by simply calculating the *percentage of correct classifications* the model makes :

$$Accuracy = \frac{Correction\ Classifications}{Total\ Attempted\ Classifications} \times 100$$

For example, if a model correctly identifies whether or not a player will get drafted into the NBA 88 times out of 100 possible times then the accuracy of the model is :
$Accuracy = \frac{88}{100} \times 100$

The **higher** the accuracy, the better a classification model is able to predict outcomes.

Similarities Between Regression and Classification : Regression and classification algorithms are similar in the following ways: - Both are supervised learning algorithms, i.e. they both involve a response variable. - Both use one or more explanatory variables to build models to predict some response. - Both can be used to understand how changes in the values of explanatory variables affect the values of a response variable.

Differences Between Regression and Classification : Regression and classification algorithms are different in the following ways: - Regression algorithms seek to predict a continuous quantity and classification algorithms seek to predict a class label. - The way we measure the accuracy of regression and classification models differs.

Converting Regression into Classification : It's worth noting that a regression problem can be converted into a classification problem by simply *discretizing* the response variable into buckets.

For example, suppose we have a dataset that contains three variables: square footage, number of bathrooms, and selling price. We could build a regression model using square footage and number of bathrooms to predict selling price. However, we could discretize selling price into three different classes:
$80k$–160k : "Low selling price"
$161k$–240k : "Medium selling price "
$241k$–320k : "High selling price"
We could then use square footage and number of bathrooms as explanatory variables to predict which class (low, medium or high) that a given house selling price will fall in. This would be an example of a classification model since we're attempting to place each house in a class.

2.2.1 Bias-Variance Tradeoff in Machine Learning

To evaluate the performance of a model on a dataset, we need to measure how well the model predictions match the observed data.

For regression models, the most commonly used metric is the *mean squared error (MSE)* , which is calculated as :

$$MSE = \frac{1}{n}\sum\left(y_i - f(X_i)\right)^2$$

where, n : Total number of observations , y_i : The response value of the ith observation
$f(X_i)$: The predicted response value of the ith observation
The closer the model predictions are to the observations, the smaller the MSE will be. However, we only care about **test MSE** – the MSE when our model is applied to unseen data. This is because we only care about how the model will perform on unseen data, not existing data.
For example : it's nice if a model that predicts stock market prices has a low MSE on historical data, but we really want to be able to use the model to accurately forecast future data.

It turns out that the test MSE can always be decomposed into two parts :
1. ***The Variance :*** Refers to the amount by which our function f would change if we estimated it using a different training set.
2. ***The Bias :*** Refers to the error that is introduced by approximating a real-life problem, which may be extremely complicated, by a much simpler model.

Written in mathematical terms :

$$Test\ MSE = Var\left(\hat{f}(x_0)\right) + \left[Bias\left(\hat{f}(x_0)\right)\right]^2 + Var(\epsilon)$$
$$Test\ MSE = Variance + Bias^2 + Irreducible\ \ error$$

The third term, the irreducible error, is the error that cannot be reduced by any model simply because there always exists some noise in the relationship between the set of explanatory variables and the *response variable*.

Models that have **high bias** tend to have **low variance**.
For example, linear regression models tend to have high bias (assumes a simple linear relationship between explanatory variables and response variable) and low variance (model estimates won't change much from one sample to the next).

However, models that have **low bias** tend to have **high variance**.
For example, complex non-linear models tend to have low bias (does not assume a certain relationship between explanatory variables and response variable) with high variance (model estimates can change a lot from one training sample to the next).

2.2.1.1 The Bias-Variance Tradeoff

The **Bias-Variance Tradeoff** refers to the tradeoff that takes place when we choose to lower bias which typically increases variance, or lower variance which typically increases bias.

The total error decreases as the complexity of a model increases but only up to a certain point. Past a certain point, variance begins to increase and total error also begins to increase.

In practice, we only care about minimizing the total error of a model, not necessarily minimizing the variance or bias. It turns out that the way to minimize the total error is to strike the right balance between variance and bias.

In other words, we want a model that is complex enough to capture the true relationship between the explanatory variables and the response variable, but not overly complex such that it finds patterns that don't really exist.

When a model is too complex, it **overfits** the data. This happens because it works too hard to find patterns in the training data that are just caused by random chance. This type of model is likely to perform poorly on unseen data.

But when a model is too simple, it **underfits** the data. This happens because it assumes the true relationship between the explanatory variables and the response variable is more simple than it actually is.

The way to pick optimal models in machine learning is to strike the balance between bias and variance such that we can minimize the test error of the model on future unseen data.

In practice, the most common way to minimize test MSE is to use cross-validation.

3 Linear Regression

3.1 Simple Linear Regression (SLR)

Simple linear regression is a statistical method you can use to understand the relationship between two variables, x and y. One variable, **x**, is known as the predictor variable. The other variable, **y**, is known as the response variable.

3.1.1 SLR using R

For example, suppose we have the following dataset with the weight and height of seven individuals :

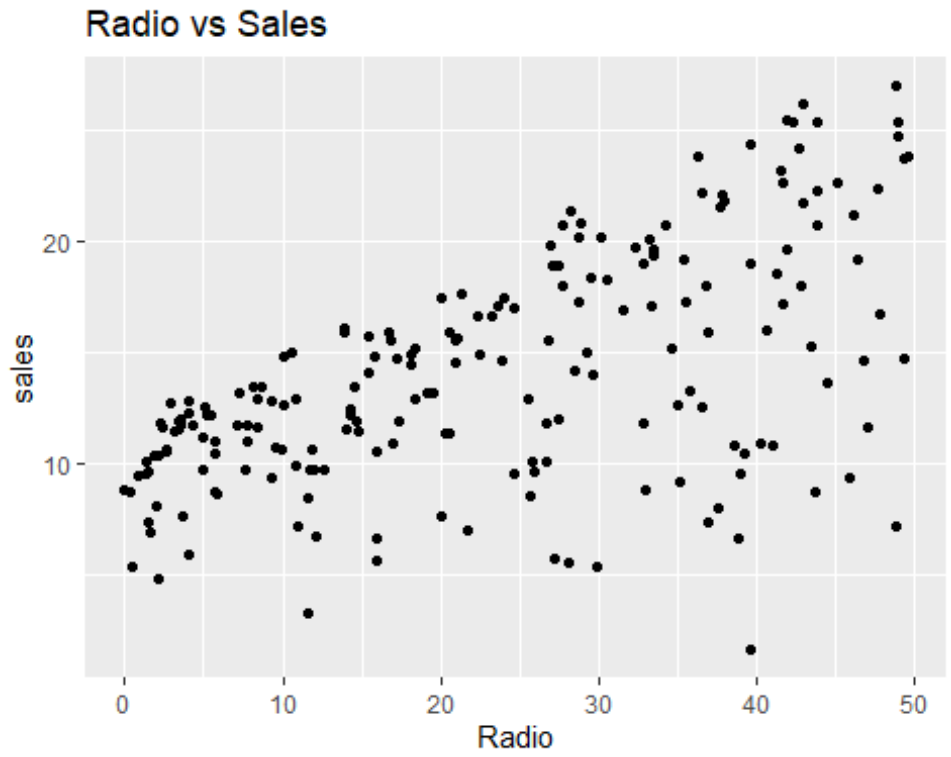
```
library(tidyverse)
df = read_csv("E:\\AMU M.Sc     (Data Sciene)\\D - 1003 RA & PM\\Source Files
by Wasiq\\Advertising.csv")

df %>% head()
```

```
## # A tibble: 6 x 5
##    ...1    TV radio newspaper sales
##   <dbl> <dbl> <dbl>     <dbl> <dbl>
## 1     1 230.   37.8      69.2  22.1
## 2     2  44.5  39.3      45.1  10.4
## 3     3  17.2  45.9      69.3   9.3
## 4     4 152.   41.3      58.5  18.5
## 5     5 181.   10.8      58.4  12.9
## 6     6   8.7  48.9      75     7.2
```

Let weight be the predictor variable and let height be the response variable.

If we graph these two variables using a *scatterplot*, with weight on the x-axis and height on the y-axis, here's what it would look like :

```
ggplot(df, aes(radio, sales)) +
  geom_point() +
  labs(x  = "Radio", y = "sales", title = "Radio vs Sales")
```



Suppose we're interested in understanding the relationship between weight and height. From the scatterplot we can clearly see that as weight increases, height tends to increase as well, but to actually *quantify* this relationship between weight and height, we need to use linear regression.

Using linear regression, we can find the line that best "fits" our data. This line is known as the **least squares regression line** and it can be used to help us understand the relationships between weight and height. Usually you would use software like Microsoft Excel, SPSS, or a graphing calculator to actually find the equation for this line.

The formula for the line of best fit is written as:

$$y = \beta_0 + \beta_1 x + \epsilon \quad \Rightarrow \quad \hat{y} = \beta_0 + \beta_1 x$$

where $\hat{y}$ is the predicted value of the response variable, β_0 is the y-intercept, β_1 is the regression coefficient, and x is the value of the predictor variable.

Model : $Sales = \beta_0 + \beta_1 \times Radio + \epsilon$

```
x = df$radio
y = df$sales
model = lm(y ~ x, data = df)
model %>% summary()

##
## Call:
## lm(formula = y ~ x, data = df)
```

```
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -15.7305  -2.1324   0.7707   2.7775   8.1810
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  9.31164    0.56290  16.542   <2e-16 ***
## x            0.20250    0.02041   9.921   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.275 on 198 degrees of freedom
## Multiple R-squared:  0.332,  Adjusted R-squared:  0.3287
## F-statistic: 98.42 on 1 and 198 DF,  p-value: < 2.2e-16
```

Fittted Model : $\widehat{Sales} = 9.31 + 0.2025 \times Radio \;\; R^2 : \; 0.33$

3.2 Multiple Linear Regression (MLR)

If we have **p** predictor variables, then a Multiple Linear Regression Model takes the form :

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p + \epsilon$$

where, Y : The Response Variable X_j : The j^{th} Predictor Variable β_j : The average effect on Y of a one unit increase in X_j, holding all other predictors fixed ϵ : The Error term

The values for $\beta_0, \beta_1, \beta_2, \cdots, \beta_p$ are chosen using the least square method, which minimizes the **Sum of Squared Residuals (RSS)** :

$$RSS = \sum \left(Y_i - \widehat{Y}_i\right)^2$$

where, $\sum$: A greek symbol that means sum , Y_i : The actual response value for the ith observation , $\widehat{Y}_i$: The predicted response value based on the multiple linear regression model

3.2.1 MLR Using R

Our Model : $Sales = \beta_0 + \beta_1 \times radio + \beta_2 \times newspaper + \beta_3 \times TV + \epsilon$

```
model = lm(sales ~ radio + newspaper + TV , data =  df)
model %>% summary()
```

```
##
## Call:
## lm(formula = sales ~ radio + newspaper + TV, data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -8.8277 -0.8908  0.2418  1.1893  2.8292
##
```

```
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  2.938889   0.311908   9.422   <2e-16 ***
## radio        0.188530   0.008611  21.893   <2e-16 ***
## newspaper   -0.001037   0.005871  -0.177     0.86
## TV           0.045765   0.001395  32.809   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.686 on 196 degrees of freedom
## Multiple R-squared:  0.8972, Adjusted R-squared:  0.8956
## F-statistic: 570.3 on 3 and 196 DF,  p-value: < 2.2e-16
```

Our Fitted Model :

$\widehat{Sales} = 2.939 + 0.1885 \times radio + (-)0.0010 \times newspaper + 0.0458 \times TV$

R^2 : 0.896

4 Classification

4.1 Logistic Regression

When we want to understand the relationship between one or more predictor variables and a continuous response variable, we often use **linear regression**.

However, when the *response variable* is *categorical* we can instead use *logistic regression*.

Logistic Regression is a type of classification algorithm because it attempts to "classify" observations from a dataset into distinct categories.

Assumptions of Logistic Regression

Logistic regression uses the following assumptions :

1. **The response variable is binary :** It is assumed that the response variable can only take on two possible outcomes.
2. **The observations are independent :** It is assumed that the observations in the dataset are independent of each other. That is, the observations should not come from repeated measurements of the same individual or be related to each other in any way.
3. **There is no severe multicollinearity among predictor variables :** It is assumed that none of the predictor variables are highly correlated with each other.
4. **There are no extreme outliers :** It is assumed that there are no extreme outliers or influential observations in the dataset.
5. **There is a linear relationship between the predictor variables and the logit of the response variable :** This assumption can be tested using a Box-Tidwell test.
6. **The sample size is sufficiently large :** As a rule of thumb, you should have a minimum of 10 cases with the least frequent outcome for each explanatory variable.

For example, if you have 3 explanatory variables and the expected probability of the least frequent outcome is 0.20, then you should have a sample size of at least (10*3) / 0.20 = 150.

Here are a few examples of when we might use logistic regression: - We want to use credit score and bank balance to predict whether or not a given customer will default on a loan. (Response variable = “Default” or “No default”) - We want to use average rebounds per game and average points per game to predict whether or not a given basketball player will get drafted into the NBA (Response variable = “Drafted” or “Not Drafted”) - We want to use square footage and number of bathrooms to predict whether or not a house in a certain city will be listed at a selling price of 200k or more. (Response variable = “Yes” or “No”)

Notice that the response variable in each of these examples can only take on one of two values. Contrast this with linear regression in which the response variable takes on some continuous value.

The Logistic Regression Equation : Logistic regression uses a method known as maximum likelihood estimation (details will not be covered here) to find an equation of the following form:

$$log(\frac{p(x)}{1-p(x)}) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p$$

where, X_j : The j^{th} predictor variable β_j : The coefficient estimate for the jth predictor variable The formula on the right side of the equation predicts the **log odds** of the response variable taking on a value of 1. Thus, when we fit a logistic regression model we can use the following equation to calculate the probability that a given observation takes on a value of 1:

$$p(x) = \frac{e^{\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p}}{1 + e^{\beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p}}$$

We then use some probability threshold to classify the observation as either 1 or 0.

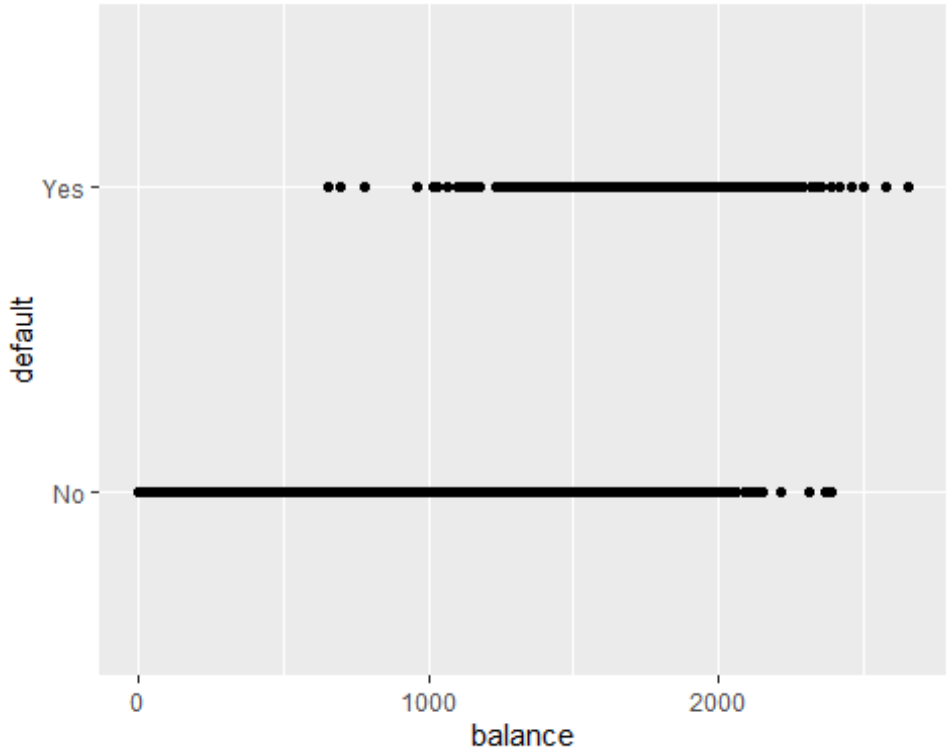
For example, we might say that observations with a probability greater than or equal to 0.5 will be classified as “1” and all other observations will be classified as “0.”

4.1.1 Logistic Regression Using R

```
library(ISLR2)
data("Default")
df = Default
df %>% head()
```

```
##   default student   balance    income
## 1      No      No  729.5265 44361.625
## 2      No     Yes  817.1804 12106.135
## 3      No      No 1073.5492 31767.139
## 4      No      No  529.2506 35704.494
## 5      No      No  785.6559 38463.496
## 6      No     Yes  919.5885  7491.559
```

```
ggplot(df, aes(x = balance, y = default)) +
  geom_point()
```



Our Model : $default = \frac{e^{\beta_0+\beta_1\times balance}}{1+e^{\beta_0+\beta_1\times balance}}$

```
# Fitting the Logistic Model :

log_m <- glm(default ~ balance , data = Default , family = binomial)
log_m

##
## Call:  glm(formula = default ~ balance, family = binomial, data = Default)
##
## Coefficients:
## (Intercept)      balance
##  -10.651331     0.005499
##
## Degrees of Freedom: 9999 Total (i.e. Null);  9998 Residual
## Null Deviance:       2921
## Residual Deviance: 1596  AIC: 1600

# Summary of Model
summary(log_m)

##
## Call:
## glm(formula = default ~ balance, family = binomial, data = Default)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
```

```
## -2.2697  -0.1465  -0.0589  -0.0221   3.7589
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept) -1.065e+01  3.612e-01  -29.49   <2e-16 ***
## balance      5.499e-03  2.204e-04   24.95   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2920.6  on 9999  degrees of freedom
## Residual deviance: 1596.5  on 9998  degrees of freedom
## AIC: 1600.5
##
## Number of Fisher Scoring iterations: 8
```

Fitted Model : $\hat{p}(default) = \frac{e^{-10.65+0.0055(balance)}}{1+e^{-10.65+0.0055(balance)}}$

Interpretation : A one-unit increase in **balance** is associated with an increase in the log odds of **default** by **0.0055** units. **OR** $\hat{p}(default) = \frac{e^{-10.65+0.0055\times1000}}{1+e^{-10.65+0.0055\times1000}} = 0.00576$
1000 unit increase in **balance** is associated with an increase in the log odds of **default** by **5** units with probability **0.00576** .

4.2 Multiple Logistic Regression Using R

$$p(X) = \frac{e^{\beta_0+\beta_1X_1+\ldots+\beta_pX_p}}{1+e^{\beta_0+\beta_1X_1+\ldots+\beta_pX_p}}$$

Model : $p(default) = \frac{e^{\beta_0+\beta_1(balance)+\beta_2(income)+\beta_3(student)}}{1+e^{\beta_0+\beta_1(balance)+\beta_2(income)+\beta_3(student)}}$

```
# Fitting the Logistic Model :

log_bis <- glm(default ~ balance + income + student , data = Default , family
= binomial)
# Summary of Model
summary(log_bis)

##
## Call:
## glm(formula = default ~ balance + income + student, family = binomial,
##     data = Default)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.4691  -0.1418  -0.0557  -0.0203   3.7383
##
## Coefficients:
```

```
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept) -1.087e+01  4.923e-01 -22.080  < 2e-16 ***
## balance      5.737e-03  2.319e-04  24.738  < 2e-16 ***
## income       3.033e-06  8.203e-06   0.370  0.71152
## studentYes  -6.468e-01  2.363e-01  -2.738  0.00619 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 2920.6  on 9999  degrees of freedom
## Residual deviance: 1571.5  on 9996  degrees of freedom
## AIC: 1579.5
##
## Number of Fisher Scoring iterations: 8
```

Fitted Model : $p(default) = \frac{e^{-10.869+0.0057(balance)+0.0030(income)-0.6468(student)}}{1+e^{-10.869+0.0057(balance)+0.0030(income)-0.6468(student)}}$

The negative coefficient for *student* in the multiple logistic regression indicates that for a fixed value of *balance* and *income* , a student is less likely to default than a non-student.

A student with a credit card balance of *$1500* and an income of *$40000* has an estimated probability of default of $\hat{p}(X) = \frac{e^{-10.869+0.0057\times1500+0.0030\times40-0.6468\times1}}{1+e^{-10.869+0.0057\times1500+0.0030\times40-0.6468\times1}} = 0.058$

A non-student with the same balance and income has an estimated probability of default of $\hat{p}(X) = \frac{e^{-10.869+0.0057\times1500+0.0030\times40-0.6468\times0}}{1+e^{-10.869+0.0057\times1500+0.0030\times40-0.6468\times0}} = 0.105$

we multiply the income coefficient estimate by *40*, rather than by 40,000, because in that table the model was fit with income measured in units of *$1, 000*

4.3 Linear Discriminant Analysis (LDA)

When we have a set of predictor variables and we'd like to classify a *response variable* into one of two classes, we typically use *logistic regression*.

For example, we may use logistic regression in the following scenario : - We want to use credit score and bank balance to predict whether or not a given customer will default on a loan. (Response variable = "Default" or "No default") However, when a response variable has more than two possible classes then we typically prefer to use a method known as *linear discriminant analysis*, often referred to as LDA.

For example, we may use LDA in the following scenario: We want to use points per game and rebounds per game to predict whether a given high school basketball player will get accepted into one of three schools: Division 1, Division 2, or Division 3. Although LDA and logistic regression models are both used for classification, it turns out that LDA is far more stable than logistic regression when it comes to making predictions for multiple classes and is therefore the preferred algorithm to use when the response variable can take on more than two classes.

LDA also performs better when sample sizes are small compared to logistic regression, which makes it a preferred method to use when you're unable to gather large samples.

How to Build LDA Models : LDA makes the following assumptions about a given dataset:

1. The values of each predictor variable are *normally distributed*. That is, if we made a histogram to visualize the distribution of values for a given predictor, it would roughly have a "bell shape."

2. Each predictor variable has the *same variance*. This is almost never the case in real-world data, so we typically scale each variable to have the same mean and variance before actually fitting a LDA model.

Once these assumptions are met, LDA then estimates the following values : -
μ_l : The meaan of all training observations from the k^{th} class .
σ^2 : The weighted average of the samples variances for each of the k classes .
π_k : The proportion of the training oservations that belong to the k^{th} class.

LDA then plugs thses numbers into the following formula and assigns each observation $X = x$ to the class for which the formula produces the largest value :

$$D_k(x) = x \times \frac{\mu_k}{\sigma^2} - \frac{\mu_k^2}{\sigma_2^2} + log\,(\pi_k)$$

Note that *LDA* has *linear* in its name because the value produced by the function above comes from a result of *linear functions of x*.

How to Prepare Data for LDA :

Make sure your data meets the following requirements before applying a LDA model to it :

1. **The response variable is categorical :** LDA models are designed to be used for classification problems, i.e. when the response variable can be placed into classes or categories.

2. **The predictor variables follow a normal distribution :** First, check that each predictor variable is roughly normally distributed. If this is not the case, you may choose to first *transform the data* to make the distribution more normal.

3. **Each predictor variable has the same variance :** As mentioned earlier, LDA assumes that each predictor variable has the same variance. Since this is rarely the case in practice, it's a good idea to scale each variable in the dataset such that it has a mean of 0 and a standard deviation of 1.

4. **Account for extreme outliers :** Be sure to check for extreme outliers in the dataset before applying LDA. Typically you can check for outliers visually by simply using *boxplots* or *scatterplots*.

Examples of Using Linear Discriminant Analysis :
LDA models are applied in a wide variety of fields in real life. Some examples include :

1. **Marketing :** Retail companies often use LDA to classify shoppers into one of several categories. For example, they may build an LDA model to predict whether or not a given shopper will be a low spender, medium spender, or high spender using predictor variables like income, total annual spending, and household size.
2. **Medical :** Hospitals and medical research teams often use LDA to predict whether or not a given group of abnormal cells is likely to lead to a mild, moderate, or severe illness.
3. **Product Development :** Companies may build LDA models to predict whether a certain consumer will use their product daily, weekly, monthly, or yearly based on a variety of predictor variables like gender, annual income, and frequency of similar product usage.
4. **Ecology :** Researchers may build LDA models to predict whether or not a given coral reef will have an overall health of good, moderate, bad, or endangered based on a variety of predictor variables like size, yearly contamination, and age.

4.3.1 LDA Using R

Linear Discriminant Analysis

Steps to perform LDA in R :
Step - 1 Load require libaries

```
library(MASS)
library(tidyverse)
```

Step - 2 Load the Data

```
data("iris")

# Attach iris data to make it easy to work
iris %>% attach()

## The following objects are masked from . (pos = 3):
##
##     Petal.Length, Petal.Width, Sepal.Length, Sepal.Width, Species

## The following objects are masked from . (pos = 4):
##
##     Petal.Length, Petal.Width, Sepal.Length, Sepal.Width, Species

## The following objects are masked from . (pos = 5):
##
##     Petal.Length, Petal.Width, Sepal.Length, Sepal.Width, Species

## The following objects are masked from . (pos = 6):
##
##     Petal.Length, Petal.Width, Sepal.Length, Sepal.Width, Species
```

```
## The following objects are masked from . (pos = 22):
##
##     Petal.Length, Petal.Width, Sepal.Length, Sepal.Width, Species

## The following objects are masked from . (pos = 23):
##
##     Petal.Length, Petal.Width, Sepal.Length, Sepal.Width, Species
```

```
# Structure of Data
iris %>% str
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1
1 1 1 1 ...
```

For this example we'll build a linear discriminant analysis model to classify which species a given flower belongs to.
We'll use the following predictor variables in the model: - Sepal.length - Sepal.Width - Petal.Length - Petal.Width
And we'll use them to predict the response variable Species, which takes on the following three potential classes: - setosa - versicolor - virginica

Step - 3 Scale the Data using `scale()` One of the key assumptions of linear discriminant analysis is that each of the predictor variables have the same variance. An easy way to assure that this assumption is met is to scale each variable such that it has a mean of 0 and a standard deviation of 1.

```
#scale each predictor variable (i.e. first 4 columns)
scaled_iris <- scale(iris[1:4])

scaled_iris %>% head()
```

```
##      Sepal.Length Sepal.Width Petal.Length Petal.Width
## [1,]   -0.8976739  1.01560199    -1.335752   -1.311052
## [2,]   -1.1392005 -0.13153881    -1.335752   -1.311052
## [3,]   -1.3807271  0.32731751    -1.392399   -1.311052
## [4,]   -1.5014904  0.09788935    -1.279104   -1.311052
## [5,]   -1.0184372  1.24503015    -1.335752   -1.311052
## [6,]   -0.5353840  1.93331463    -1.165809   -1.048667
```

We can use the `apply()` function to verify that each predictor variable now has a mean of 0 and a standard deviation of 1 :

```
# Mean of each Predictor VAriable
apply(scaled_iris, 2, mean)
```

```
##  Sepal.Length   Sepal.Width  Petal.Length   Petal.Width
## -4.484318e-16  2.034094e-16 -2.895326e-17 -3.663049e-17
```

```
# SD of each Predictor VAriable
apply(scaled_iris, 2, sd)
```

```
## Sepal.Length  Sepal.Width Petal.Length  Petal.Width
##            1            1            1            1
```

Step - 4 Create Training and Test Samples
Next, we'll split the dataset into a training set to train the model on and a testing set to test the model on :

```
#Use 70% of dataset as training set and remaining 30% as testing set
sample <- sample(c(TRUE, FALSE), nrow(iris), replace=TRUE, prob=c(0.7,0.3))

# train Data
train <- iris[sample, ]

# Test Data
test <- iris[!sample, ]
```

Step - 5 Fit the LDA Model
Next, we'll use the **`lda()`** function from the MASS package to fit the LDA model to our data :

```
#fit LDA model
model <- lda(Species~., data=train)

#view model output
model
```

```
## Call:
## lda(Species ~ ., data = train)
##
## Prior probabilities of groups:
##     setosa versicolor  virginica
##  0.3301887  0.3396226  0.3301887
##
## Group means:
##            Sepal.Length Sepal.Width Petal.Length Petal.Width
## setosa         4.940000    3.405714     1.445714   0.2428571
## versicolor     5.908333    2.766667     4.202778   1.3027778
## virginica      6.634286    2.925714     5.565714   2.0428571
##
## Coefficients of linear discriminants:
##                     LD1        LD2
## Sepal.Length  0.5329061  0.2942187
## Sepal.Width   2.1256776  1.9325511
```

```
## Petal.Length -1.9624296 -1.1434715
## Petal.Width  -3.5611214  3.0035717
##
## Proportion of trace:
##   LD1   LD2
## 0.993 0.007
```

Here is how to interpret the output of the model :

Prior probabilities of group : These represent the proportions of each Species in the training set. For example, 30.2% of all observations in the training set were of species virginica.

Group means : These display the mean values for each predictor variable for each species.

Coefficients of Linear Discriminants : These display the linear combination of predictor variables that are used to form the decision rule of the LDA model. For example: - LD1: 0.840*Sepal.Length + 1.320*Sepal.Width – 2.236*Petal.Length – 2.793*Petal.Width - LD2: -0.454*Sepal.Length - 1.849*Sepal.Width + 1.069*Petal.Length - 2.754*Petal.Width

Proportion of Trace : These display the percentage separation achieved by each linear discriminant function.

Step - 6 Use the Model to Make Predictions
Once we've fit the model using our training data, we can use it to make predictions on our test data :

```
#use LDA model to make predictions on test data
predicted <- predict(model, test)

names(predicted)

## [1] "class"     "posterior" "x"
```

This returns a list with three variables:

class : The predicted class *posterior* : The posterior probability that an observation belongs to each class *x* : The linear discriminants

We can quickly view each of these results for the first six observations in our test dataset :

```
#view predicted class for first six observations in test set
head(predicted$class)

## [1] setosa setosa setosa setosa setosa setosa
## Levels: setosa versicolor virginica

#view posterior probabilities for first six observations in test set
head(predicted$posterior)

##    setosa   versicolor    virginica
## 2       1 1.368345e-19 3.862237e-43
```

```
## 4       1 5.821808e-19 3.138592e-42
## 5       1 7.504010e-26 1.959362e-51
## 8       1 6.042884e-23 1.505230e-47
## 11      1 6.272356e-27 7.218899e-53
## 16      1 8.088751e-32 1.435061e-58
```

```
#view linear discriminants for first six observations in test set
head(predicted$x)
```

```
##          LD1         LD2
## 2   7.589072 -0.64800788
## 4   7.445525 -0.65736554
## 5   8.917770  0.54094463
## 8   8.296391  0.04008727
## 11  9.147257  0.73754007
## 16 10.082879  2.77930577
```

We can use the following code to see what percentage of observations the LDA model correctly predicted the Species for :

```
#find accuracy of model
mean(predicted$class==test$Species)
```

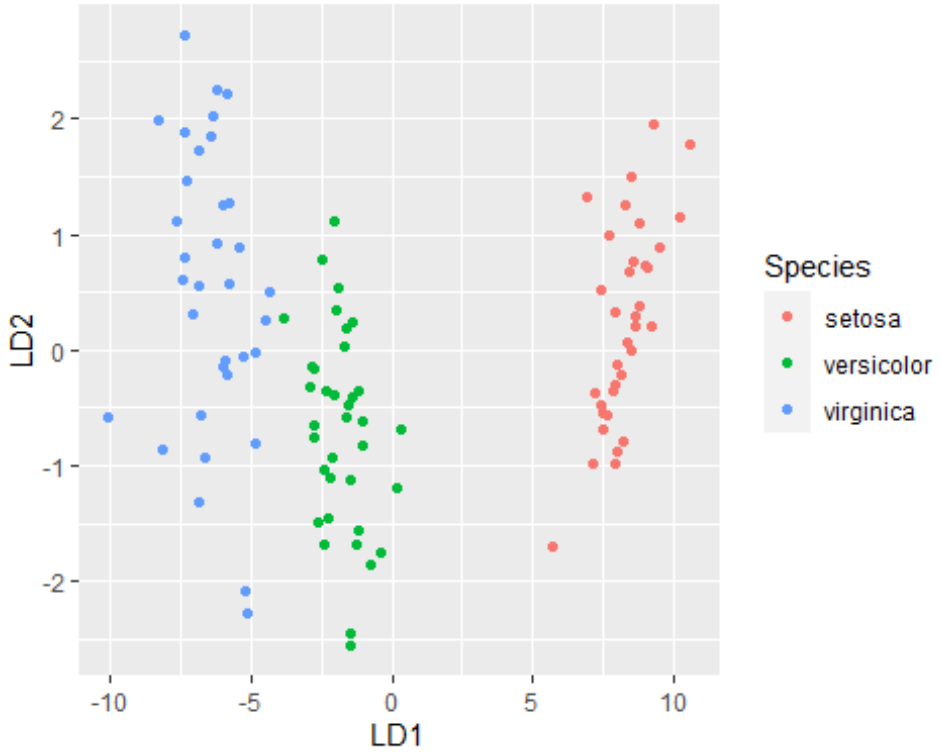
```
## [1] 0.9545455
```

It turns out that the model correctly predicted the Species for **100%** of the observations in our test dataset.

In the real-world an LDA model will rarely predict every class outcome correctly, but this iris dataset is simply built in a way that machine learning algorithms tend to perform very well on it.

Step - 7 Visualize the Results
Lastly, we can create an LDA plot to view the linear discriminants of the model and visualize how well it separated the three different species in our dataset:

```
#define data to plot
lda_plot <- cbind(train, predict(model)$x)

#create plot
ggplot(lda_plot, aes(LD1, LD2)) +
  geom_point(aes(color = Species))
```



4.4 Quadratic Discriminant Analysis (QDA)

When we have a set of predictor variables and we'd like to classify a *response variable* into one of *two classes*, we typically use *logistic regression*.

However, when* a response variable* has *more than two possible classes* then we typically use *linear discriminant analysis*, often referred to as LDA.

LDA assumes that
(1) observations from each class are normally distributed
(2) observations from each class share the same covariance matrix. Using these assumptions, LDA then finds the following values :
μ_k : The mean of all training observations from the kth class.
σ^2 : The weighted average of the sample variances for each of the k classes.
π_k : The proportion of the training observations that belong to the k^{th} class. LDA then plugs these numbers into the following formula and assigns each observation $X = x$ to the class for which the formula produces the largest value :

$$D_k(x) = x \times \frac{\mu_k}{\sigma^2} + \frac{\mu_k^2}{2\sigma^2} + log(\pi_k)$$

LDA has linear in its name because the value produced by the function above comes from a result of linear functions of x.

An *extension* of linear discriminant analysis *(LDA)* is **Quadratic Discriminant Analysis**, often referred to as QDA.

This method is similar to LDA and also assumes that the observations from each class are normally distributed, but it does not assume that each class shares the same covariance matrix. Instead, QDA assumes that each class has its own covariance matrix.

That is, it assumes that an observation from the kth class is of the form $\boldsymbol{X \sim N(\mu_k , \Sigma_k)}$

Using this *assumption, QDA* then finds the following values :
μ_k : The mean of all training observations from the k^{th} class.
Σ_k : The covariance matrix of the k^{th} class.
π_k : The proportion of the training observations that belong to the k^{th} class.

QDA then plugs these numbers into the following formula and assigns each observation $X = x$ to the class for which the formula produces the largest value :

$$D_k(x) = -\frac{1}{2}(x-\mu_k)^T\Sigma_k^{-1}(x-\mu_k) - \frac{1}{2}log|\Sigma_k| + log(\pi_k)$$

Note that QDA has *quadratic* in its name because the value produced by the function above comes from a result of quadratic functions of x.

LDA vs. QDA : When to Use One vs. the Other
The main difference between LDA and QDA is that LDA assumes each class shares a covariance matrix, which makes it a much less flexible classifier than QDA.

This inherently means it has low variance – that is, it will perform similarly on different training datasets. The drawback is that if the assumption that the K classes have the same covariance is untrue, then LDA can suffer from *high bias*.

QDA is generally preferred to LDA in the following situations :
(1) The training set is large.
(2) It's unlikely that the *K* classes share a common covariance matrix.

When these conditions hold, QDA tends to perform better since it is more flexible and can provide a better fit to the data.

How to Prepare Data for QDA :
Make sure your data meets the following requirements before applying a QDA model to it :

1. **The response variable is categorical :** QDA models are designed to be used for classification problems, i.e. when the response variable can be placed into classes or categories.

2. **The observations in each class follow a normal distribution :** First, check that each the distribution of values in each class is roughly normally distributed. If this is not the case, you may choose to first transform the data to make the distribution more normal.

3. **Account for extreme outlier :**. Be sure to check for extreme outliers in the dataset before applying LDA. Typically you can check for outliers visually by simply using *boxplots* or *scatterplots*.

4.4.1 QDA Using R

Quadratic Discriminant Analysis

Step-1 : Load Necessary Libraries

```
library(MASS)
library(tidyverse)
```

Step-2 : Load the Data

```
# attach iris dataset to make it easy to work with
iris %>% attach()

## The following objects are masked from . (pos = 3):
##
##     Petal.Length, Petal.Width, Sepal.Length, Sepal.Width, Species

## The following objects are masked from . (pos = 4):
##
##     Petal.Length, Petal.Width, Sepal.Length, Sepal.Width, Species

## The following objects are masked from . (pos = 5):
##
##     Petal.Length, Petal.Width, Sepal.Length, Sepal.Width, Species

## The following objects are masked from . (pos = 6):
##
##     Petal.Length, Petal.Width, Sepal.Length, Sepal.Width, Species

## The following objects are masked from . (pos = 7):
##
##     Petal.Length, Petal.Width, Sepal.Length, Sepal.Width, Species

## The following objects are masked from . (pos = 23):
##
##     Petal.Length, Petal.Width, Sepal.Length, Sepal.Width, Species

## The following objects are masked from . (pos = 24):
##
##     Petal.Length, Petal.Width, Sepal.Length, Sepal.Width, Species

# view structure of dataset
iris %>% str()

## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1
1 1 1 1 ...
```

For this example we'll build a quadratic discriminant analysis model to classify which species a given flower belongs to.

We'll use the following predictor variables in the model : - Sepal.length - Sepal.Width - Petal.Length - Petal.Width
And we'll use them to predict the response variable Species, which takes on the following three potential classes: - setosa - versicolor - virginica

Step-3 : Create Training and Test Samples
Next, we'll split the dataset into a training set to train the model on and a testing set to test the model on:

```
# make this example reproducible
set.seed(1)

# Use 70% of dataset as training set and remaining 30% as testing set
sample <- sample(c(TRUE, FALSE), nrow(iris), replace=TRUE, prob=c(0.7,0.3))

# Train Data
train <- iris[sample, ]

# Test Data
test <- iris[!sample, ]
```

Step-4 : Fit the QDA Model
Next, we'll use the **`qda()`** function from the MASS package to fit the QDA model to our data :

```
# fit QDA model
model <- qda(Species~., data=train)

# view model output
model

## Call:
## qda(Species ~ ., data = train)
##
## Prior probabilities of groups:
##     setosa versicolor  virginica
##  0.3207547  0.3207547  0.3584906
##
## Group means:
##            Sepal.Length Sepal.Width Petal.Length Petal.Width
## setosa         4.982353    3.411765     1.482353   0.2411765
## versicolor     5.994118    2.794118     4.358824   1.3676471
## virginica      6.636842    2.973684     5.592105   2.0552632
```

Here is how to interpret the output of the model:

Prior Probabilities of Group : These represent the proportions of each Species in the training set.
For example, 35.8% of all observations in the training set were of species virginica.

Group Means : These display the mean values for each predictor variable for each species.

Step-5 : Use the Model to Make Predictions
Once we've fit the model using our training data, we can use it to make predictions on our test data :

```
#use QDA model to make predictions on test data
predicted <- predict(model, test)

names(predicted)

## [1] "class"     "posterior"
```

class : The predicted class
posterior : The posterior probability that an observation belongs to each class

We can quickly view each of these results for the first six observations in our test dataset:

```
# view predicted class for first six observations in test set
head(predicted$class)

## [1] setosa setosa setosa setosa setosa setosa
## Levels: setosa versicolor virginica

# view posterior probabilities for first six observations in test set
head(predicted$posterior)

##    setosa   versicolor    virginica
## 4       1 7.224770e-20 1.642236e-29
## 6       1 6.209196e-26 8.550911e-38
## 7       1 1.248337e-21 8.132700e-32
## 15      1 2.319705e-35 5.094803e-50
## 17      1 1.396840e-29 9.586504e-43
## 18      1 7.581165e-25 8.611321e-37
```

Step-6 : Evaluate the Model
We can use the following code to see what percentage of observations the QDA model correctly predicted the Species for :

```
# Find Accuracy of Model
mean(predicted$class == test$Species)

## [1] 1
```

It turns out that the model correctly predicted the Species for 100% of the observations in our test dataset.

In the real-world an QDA model will rarely predict every class outcome correctly, but this iris dataset is simply built in a way that machine learning algorithms tend to perform very well on it.

5 How to Assess Model Fit

5.1 Overfitting

In the field of machine learning, we often build models so that we can make accurate predictions about some phenomenon.

For example, suppose we want to build a *regression model* that uses the predictor variable hours spent studying to predict the response variable ACT score for students in high school.

To build this model, we'll collect data about hours spent studying and the corresponding ACT Score for hundreds of students in a certain school district.

Then we'll use this data to train a model that can make predictions about the score a given student will receive based on their total hours studied.

To assess how useful the model is, we can measure how well the model predictions match the observed data. One of the most commonly used metrics for doing so is the **Mean Squared Error** (MSE), which is calculated as :

$$MSE = \frac{1}{n}\sum\left(y_i - f(x_i)\right)^2$$

where, n : Total number of observations , y_i : The response value of the i^{th} observation
$f(x_i)$: The predicted response value of the i^{th} observation

The closer the model predictions are to the observations, the smaller the MSE will be.

However, one of the biggest mistakes made in machine learning is optimizing models to reduce **training MSE** – i.e. how closely the model predictions match up with the data that we used to train the model.

When a model focuses too much on reducing training MSE, it often works too hard to find patterns in the training data that are just caused by random chance. Then when the model is applied to unseen data, it performs poorly.

This phenomenon is known as overfitting. It occurs when we "fit" a model too closely to the training data and we thus end up building a model that isn't useful for making predictions about new data.

5.2 Leave-One-Out Cross-Validation (LOOCV)

To evaluate the performance of a model on a dataset, we need to measure how well the predictions made by the model match the observed data.

The most common way to measure this is by using the mean squared error (MSE), which is calculated as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - f(x_i)\right)^2$$

where, n : Total number of observations y_i : The response value of the ith observation
$f(x_i)$: The predicted response value of the ith observation

The closer the model predictions are to the observations, the smaller the MSE will be.

In practice, we use the following process to calculate the MSE of a given model : - Split a dataset into a training set and a testing set. - Build the model using only data from the training set. - Use the model to make predictions on the testing set and measure the MSE – this is know as the **test MSE**.

The test MSE gives us an idea of how well a model will perform on data it hasn't previously seen, i.e. data that wasn't used to "train" the model.

However, the drawback of using only one testing set is that the test MSE can vary greatly depending on which observations were used in the training and testing sets.

It's possible that if we use a different set of observations for the training set and the testing set that our test MSE could turn out to be much larger or smaller.

One way to avoid this problem is to fit a model several times using a different training and testing set each time, then calculating the test MSE to be the average of all of the test MSE's.

This general method is known as cross-validation and a specific form of it is known as **leave-one-out cross-validation**.

LOOCV
Leave-one-out cross-validation uses the following approach to evaluate a model: - Split a dataset into a training set and a testing set, using all but one observation as part of the training set. Note that we only leave one observation "out" from the training set. This is where the method gets the name "leave-one-out" cross-validation. - Build the model using only data from the training set. - Use the model to predict the response value of the one observation left out of the model and calculate the MSE. - Repeat the process n times.

Lastly, we repeat this process n times (where n is the total number of observations in the dataset), leaving out a different observation from the training set each time.

We then calculate the test MSE to be the average of all of the test MSE's:

$$Test\ MSE = \frac{1}{n}\sum_{i=1}^{n} MSE_i$$

where:, n : The total number of observations in the dataset
MSE_i : The test MSE during the ith time of fitting the model.

Pros & Cons of LOOCV

Leave-one-out cross-validation offers the following pros : -
It provides a much less biased measure of test MSE compared to using a single test set because we repeatedly fit a model to a dataset that contains n-1 observations.
It tends not to overestimate the test MSE compared to using a single test set.

However, leave-one-out cross-validation comes with the following cons:

- It can be a time-consuming process to use when n is large.
- It can also be time-consuming if a model is particularly complex and takes a long time to fit to a dataset.
- It can be computationally expensive.

Fortunately, modern computing has become so efficient in most fields that LOOCV is a much more reasonable method to use compared to many years ago.

Note that **LOOCV** can be *used* in both **regression** and **classification settings** as well. For regression problems, it calculates the test MSE to be the mean squared difference between predictions and observations while in classification problems it calculates the test MSE to be the percentage of observations correctly classified during the n repeated model fittings.

5.2.1 LOOCV Using R

Leave-One-Out Cross-Validation

```
library(tidyverse)
data("trees")
df <- trees

# Load caret library
library(caret)

#specify the cross-validation method
ctrl <- trainControl(method = "LOOCV")

#fit a regression model and use LOOCV to evaluate performance
model <- train(Volume ~ Height + Girth, data = df, method = "lm", trControl =
ctrl)

#view summary of LOOCV
print(model)

## Linear Regression
##
## 31 samples
##  2 predictor
##
## No pre-processing
## Resampling: Leave-One-Out Cross-Validation
## Summary of sample sizes: 30, 30, 30, 30, 30, 30, ...
```

```
## Resampling results:
##
##   RMSE    Rsquared   MAE
##   4.2612  0.9306206  3.355245
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

Here is how to interpret the output:

- 31 different samples were used to build 10 models. Each model used 2 predictor variables.
- No pre-processing occured. That is, we didn't scale the data in any way before fitting the models.
- The resampling method we used to generate the 31 samples was - Leave-One-Out Cross Validation.
- The sample size for each training set was 30.
- **RMSE :** The root mean squared error. This measures the average difference between the predictions made by the model and the actual observations. The lower the RMSE, the more closely a model can predict the actual observations.
- **Rsquared :** This is a measure of the correlation between the predictions made by the model and the actual observations. The higher the R-squared, the more closely a model can predict the actual observations.
- **MAE :** The mean absolute error. This is the average absolute difference between the predictions made by the model and the actual observations. The lower the MAE, the more closely a model can predict the actual observations.

Each of the three metrics provided in the output (RMSE, R-squared, and MAE) give us an idea of how well the model performed on previously unseen data.

In practice we typically fit several different models and compare the three metrics provided by the output seen here to decide which model produces the lowest test error rates and is therefore the best model to use.

5.3 K-Fold Validation

To evaluate the performance of some model on a dataset, we need to measure how well the predictions made by the model match the observed data.

The most common way to measure this is by using the mean squared error (MSE), which is calculated as :

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - f(x_i)\right)^2$$

where, n : Total number of observations , y_i : The response value of the ith observation
$f(x_i)$: The predicted response value of the ith observation
The closer the model predictions are to the observations, the smaller the MSE will be.

In practice, we use the following process to calculate the MSE of a given model :
1. Split a dataset into a training set and a testing set.
2. Build the model using only data from the training set.
3. Use the model to make predictions on the testing set and measure the test MSE.

The test MSE gives us an idea of how well a model will perform on data it hasn't previously seen. However, the drawback of using only one testing set is that the test MSE can vary greatly depending on which observations were used in the training and testing sets.

One way to avoid this problem is to fit a model several times using a different training and testing set each time, then calculating the test MSE to be the average of all of the test MSE's.

This general method is known as cross-validation and a specific form of it is known as k-fold cross-validation.

K-Fold Cross-Validation
K-fold cross-validation uses the following approach to evaluate a model : Step 1: Randomly divide a dataset into k groups, or "folds", of roughly equal size. Step 2: Choose one of the folds to be the holdout set. Fit the model on the remaining k-1 folds. Calculate the test MSE on the observations in the fold that was held out. Step 3: Repeat this process k times, using a different set each time as the holdout set. Step 4: Calculate the overall test MSE to be the average of the k test MSE's.

$$Test\ MSE = \frac{1}{k}\sum_{i=1}^{n} MSE_i$$

where, k : Number of folds , MSE_i : Test MSE on the ith iteration

How to Choose K

In general, the more folds we use in k-fold cross-validation the lower the bias of the test MSE but the higher the variance. Conversely, the fewer folds we use the higher the bias but the lower the variance. This is a classic example of the **bias-variance tradeoff** in machine learning.

In practice, we typically choose to use between 5 and 10 folds. As noted in An Introduction to Statistical Learning, this number of folds has been shown to offer an optimal balance between bias and variance and thus provide reliable estimates of test MSE: - To summarize, there is a bias-variance trade-off associated with the choice of k in k-fold cross-validation. - Typically, given these considerations, one performs k-fold cross-validation using k = 5 or k = 10, as these values have been shown empirically to yield test error rate estimates that suffer neither from excessively high bias nor from very high variance.

Advantages of K-Fold Cross-Validation
When we split a dataset into just one training set and one testing set, the test MSE calculated on the observations in the testing set can vary greatly depending on which observations were used in the training and testing sets.

By using k-fold cross-validation, we're able to use calculate the test MSE using several different variations of training and testing sets. This makes it much more likely for us to obtain an unbiased estimate of the test MSE.

K-fold cross-validation also offers a computational advantage over leave-one-out cross-validation **(LOOCV)** because it only has to fit a model k times as opposed to n times.

For models that take a long time to fit, k-fold cross-validation can compute the test MSE much quicker than LOOCV and in many cases the test MSE calculated by each approach will be quite similar if you use a sufficient number of folds.

Extensions of K-Fold Cross-Validation There are several extensions of k-fold cross-validation, including:

1. **Repeated K-fold Cross-Validation :** This is where k-fold cross-validation is simply repeated *n* times. Each time the training and testing sets are shuffled, so this further reduces the bias in the estimate of test MSE although this takes longer to perform than ordinary k-fold cross-validation.

2. **Leave-One-Out Cross-Validation :** This is a special case of k-fold cross-validation in which *k*=*n*. You can read more about this method here.

3. **Stratified K-Fold Cross-Validation :** This is a version of k-fold cross-validation in which the dataset is rearranged in such a way that each fold is representative of the whole. As noted by Kohavi, this method tends to offer a better tradeoff between bias and variance compared to ordinary k-fold cross-validation.

4. **Nested Cross-Validation :** This is where k-fold cross validation is performed within each fold of cross-validation. This is often used to perform hyperparameter tuning during model evaluation.

5.3.1 K-Fold Cross Validation Using R

```
df <- trees

# specify the cross-validation method from caret
# k = 5
ctrl <- trainControl(method = "cv", number = 5)

# fit a regression model and use k-fold CV to evaluate performance
model <- train(Volume ~ Height + Girth, data = df, method = "lm", trControl =
ctrl)

# view summary of k-fold CV
print(model)

## Linear Regression
##
## 31 samples
##  2 predictor
##
```

```
## No pre-processing
## Resampling: Cross-Validated (5 fold)
## Summary of sample sizes: 25, 23, 25, 25, 26
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   3.933816  0.9619364  3.166978
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

Here is how to interpret the output:

- No pre-processing occured. That is, we didn't *scale the data* in any way before fitting the models.
- The resampling method we used to evaluate the model was cross-validation with 5 folds.
- The sample size for each training set was 8.
- **RMSE** : The root mean squared error. This measures the average difference between the predictions made by the model and the actual observations. The lower the RMSE, the more closely a model can predict the actual observations.
- **Rsquared** : This is a measure of the correlation between the predictions made by the model and the actual observations. The higher the R-squared, the more closely a model can predict the actual observations.
- **MAE** : The mean absolute error. This is the average absolute difference between the predictions made by the model and the actual observations. The lower the MAE, the more closely a model can predict the actual observations.

Each of the three metrics provided in the output (RMSE, R-squared, and MAE) give us an idea of how well the model performed on previously unseen data.

In practice we typically fit several different models and compare the three metrics provided by the output seen here to decide which model produces the lowest test error rates and is therefore the best model to use.

We can use the following code to examine the final model fit :

```
# View final Model
model$finalModel
```

```
##
## Call:
## lm(formula = .outcome ~ ., data = dat)
##
## Coefficients:
## (Intercept)       Height        Girth
##    -57.9877       0.3393       4.7082
```

The final model turns out to be :

$$Volue = -57.89 + 0.34 \times Height + 4.70 \times Girth$$

We can use the following code to view the model predictions made for each fold :

```
# View predictions for each fold
model$resample
```

```
##       RMSE  Rsquared      MAE Resample
## 1 5.507771 0.9528857 4.340838    Fold1
## 2 4.560770 0.9715747 3.930646    Fold2
## 3 3.051449 0.9683287 2.511209    Fold3
## 4 2.669287 0.9651855 1.925083    Fold4
## 5 3.879803 0.9517074 3.127114    Fold5
```

Note that in this example we chose to use k=5 folds, but you can choose however many folds we'd like. In practice, we typically choose between 5 and 10 folds because this turns out to be the optimal number of folds that produce reliable test error rates.

6 Model Selection

6.1 Best Subset Selection

In the field of machine learning, we're often interested in building models using a set of predictor variables and a *response variable*. Our goal is to build a model that can effectively use the predictor variables to predict the value of the response variable.

Given a set of p total predictor variables, there are many models that we could potentially build. One method that we can use to pick the *best model* is known as **best subset selection** and it works as follows :

1. Let M_0 denote the null model, which contains no predictor variables.
2. For $k = 1,2,\dots,p$

- fit all $\binom{p}{k}$ models that contain exactly k predictors.
- Pick the best among these $\binom{p}{k}$ models and call it M_k. Define "best" as the model with the highest R^2 or equivalently the lowest RSS.

3. Select a single best model from among $M_0 \dots M_p$ using **cross-validation prediction error**, C_p, BIC, AIC or $adjusted\ R^2 \Rightarrow R^2_{adj.}$.

Note that for a set of p predictor variables, there are 2^p possible models.

Example of Best Subset Selection : Suppose we have a dataset with $p = 3$ predictor variables and one response variable y. To perform best subset selection with this dataset, we would fit the following $2^p = 23 = 8$ models :

- A model with no predictors
- A model with predictor x_1

- A model with predictor x_2
- A model with predictor x_3
- A model with predictors x_1, x_2
- A model with predictors x_1, x_3
- A model with predictors x_2, x_3
- A model with predictors x_1, x_2, x_3

Next, we'd choose the model with the highest R^2 among each set of models with k predictors.
For example, we might end up choosing: -
A model with predictor x_2
A model with predictors x_1, x_2
A model with predictors x_1, x_2, x_3

Next, we'd perform *cross-validation* and choose the best model to be the one that results in the lowest **prediction error**, C_p, BIC, AIC or **adjusted** $R^2 = R^2_{adj.}$.

For example, we might end up choosing the following model as the "best" model because it produced the lowest cross-validated prediction error: - A model with predictors x_1, x_2

Criteria for Choosing the "Best" Model

The last step of best subset selection involves choosing the model with the **lowest prediction error, lowest Cp, lowest BIC, lowest AIC, or highest adjusted** $R^2 = R^2_{adj.}$.

Here are the formulas used to calculate each of these metrics :

$C_p: \frac{(RSS+2d\hat{\sigma})}{n}$

$AIC: \frac{(RSS+2d\widehat{\sigma^2})}{n\widehat{\sigma^2}}$

$BIC: \frac{(RSS+log(n)d\widehat{\sigma^2})}{n}$

$R^2_{adj.}: 1 - \frac{\frac{RSS}{(n-d-1)}}{\frac{TSS}{(n-1)}}$

where, d : The number of predictors , n : Total observations , $\hat{\sigma}$: Estimate of the variance of the error associate with each response measurement in a regression model
RSS : Residual sum of squares of the regression model , TSS : Total sum of squares of the regression model

Pros & Cons of Best Subset Selection

Best subset selection offers the following pros :

- It's a straightforward approach to understand and interpret.
- It allows us to identify the best possible model since we consider all combinations of predictor variables.

However, this method comes with the following cons :

- It can be computationally intense. For a set of p predictor variables, there are 2^p possible models. For example, with 10 predictor variables there are $2^{10} = 1{,}000$ possible models to consider.
- Because it considers such a large number of models, it could potentially find a model that performs well on training data but not on future data. This could result in *overfitting.*

Conclusion

While best subset selection is straightforward to implement and understand, it can be unfeasible if you're working with a dataset that has a large number of predictors and it could potentially lead to overfitting.

An alternative to this method is known as **stepwise selection**, which is more computationally efficient.

6.2 Stepwise Selection

In the field of machine learning, our goal is to build a model that can effectively use a set of predictor variables to predict the value of some *response variable*.

Given a set of p total predictor variables, there are many models that we could potentially build. One method that we can use to pick the best model is known as **best subset selection**, which attempts to choose the best model from all possible models that could be built with the set of predictors.

Unfortunately this method suffers from two drawbacks : - It can be computationally intense. For a set of p predictor variables, there are 2^p possible models. For example, with 10 predictor variables there are $2^{10} = 1{,}000$ possible models to consider. - Because it considers such a large number of models, it could potentially find a model that performs well on training data but not on future data. This could result in *overfitting*.

An alternative to best subset selection is known as **stepwise selection**, which compares a much more restricted set of models.

There are two types of stepwise selection methods: forward stepwise selection and backward stepwise selection.

Forward Stepwise Selection

Forward stepwise selection works as follows :
1. Let M_0 denote the null model, which contains no predictor variables.
2. For $k = 0,2,\ldots,p-1$: - Fit all $p-k$ models that augment the predictors in Mk with one additional predictor variable. - Pick the best among these $p-k$ models and call it M_{k+1}. Define "best" as the model with the highest R^2 or equivalently the lowest RSS.
3. Select a single best model from among $M_0 \ldots M_p$ using *cross-validation prediction error, Cp, BIC, AIC, or adjusted* $R^2 = R^2_{adj}$.

Backward Stepwise Selection

Backward stepwise selection works as follows :
1. Let M_p denote the full model, which contains all p predictor variables.
2. For $k = p, p-1, \ldots, 1$: - Fit all k models that contain all but one of the predictors in M_k, for a total of $k-1$ predictor variables. - Pick the best among these k models and call it M_{k-1}. Define "best" as the model with the highest R^2 or equivalently the lowest RSS.
3. Select a single best model from among $M_0 \ldots M_p$ using *cross-validation prediction error, Cp, BIC, AIC, or adjusted* R^2.

Criteria for Choosing the "Best" Model

The last step of both forward and backward stepwise selection involves choosing the model with the *lowest prediction error, lowest Cp, lowest BIC, lowest AIC, or highest adjusted* R^2.

Here are the formulas used to calculate each of these metrics :

$C_p : \frac{(RSS + 2d\hat{\sigma})}{n}$ $\qquad$ $AIC : \frac{(RSS + 2d\widehat{\sigma^2})}{n\widehat{\sigma^2}}$

$BIC : \frac{(RSS + log(n)d\widehat{\sigma^2})}{n}$ $\qquad$ $R^2_{adj.} : 1 - \frac{\frac{RSS}{(n-d-1)}}{\frac{TSS}{(n-1)}}$

where, d : The number of predictors , n : Total observations , $\hat{\sigma}$: Estimate of the variance of the error associate with each response measurement in a regression model
RSS : Residual sum of squares of the regression model
TSS : Total sum of squares of the regression model

Pros & Cons of Stepwise Selection

Stepwise selection offers the following **benefit** : -
It is more computationally efficient than best subset selection.
Given p predictor variables, best subset selection must fit 2^p models.

Conversely, stepwise selection only has to fit $\frac{1+p(p+1)}{2}$ models.
For $p = 10$ predictor variables, best subset selection must fit 1,000 models while stepwise selection only has to fit 56 models.

However, stepwise selection has the following potential **drawback** :
It is not guaranteed to find the best possible model out of all 2^p potential models.
For example, suppose we have a dataset with $p = 3$ predictors.
The best possible one-predictor model may contain x_1 and the best possible two-predictor model may instead contain x_1 and x_2.

In this case, forward stepwise selection will fail to select the best possible two-predictor model because M_1 will contain x_1, so M_2 must also contain x_1 along with some other variable.

7 Regularization

7.1 Ridge Regression

In ordinary multiple linear regression, we use a set of p predictor variables and a response variable to fit a model of the form :

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p + \epsilon$$

where, Y : The response variable X_j : The j^{th} predictor variable β_j : The average effect on Y of a one unit. ϵ : The error term

The values for $\beta_0, \beta_1, \beta_2, \ldots, \beta_p$ are chosen using the **least square method** , which minimizes the sum of squared residuals *(RSS)*

$$RSS = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

where, $\sum$: A greek symbol that means *sum*. y_i : The actual response value for the i^{th} observation. $\hat{y}_i$: The predicted response value based on the multiple linear regression model.

However, when the predictor variables are highly correlated then **multicollinearity** can become a problem. This can cause the coefficient estimates of the model to be unreliable and have high variance.

One way to get around this issue without completely removing some predictor variables from the model is to use a method known as ridge regression, which instead seeks to minimize the following :

$$RSS + \lambda \Sigma \beta_j^2$$

where, j ranges from $1 \; to \; p$ and $\lambda \geq 0$.

This second term in the equation is known as a **shrinkage penalty**.

When $\lambda = 0$, this penalty term has no effect and ridge regression produces the same coefficient estimates as least squares. However, as λ approaches infinity, the shrinkage penalty becomes more influential and the ridge regression coefficient estimates approach zero.

In general, the predictor variables that are least influential in the model will shrink towards zero the fastest.

Why Use Ridge Regression ?
The advantage of ridge regression compared to least squares regression lies in the *bias-variance tradeoff*.

Recall that mean squared error (MSE) is a metric we can use to measure the accuracy of a given model and it is calculated as :

$$MSE = Var\left(\hat{f}(x_0)\right) + \left[Bias\left(\hat{f}(x_0)\right)\right]^2 + Var(\epsilon)$$

$$MSE = Variance + Bias^2 + Irreducible\ error$$

The basic idea of ridge regression is to introduce a little bias so that the variance can be substantially reduced, which leads to a lower overall MSE.

Ridge regression test MSE reduction

This means the model fit by ridge regression will produce smaller test errors than the model fit by least squares regression.

Steps to Perform Ridge Regression in Practice
The following steps can be used to perform ridge regression :

Step 1: Calculate the correlation matrix and VIF values for the predictor variables.

First, we should produce a correlation matrix and calculate the **VIF** (variance inflation factor) values for each predictor variable.

If we detect high correlation between predictor variables and high VIF values (some texts define a "high" VIF value as 5 while others use 10) then ridge regression is likely appropriate to use.

However, if there is no multicollinearity present in the data then there may be no need to perform ridge regression in the first place. Instead, we can perform ordinary least squares regression.

Step 2: Standardize each predictor variable.

Before performing ridge regression, we should scale the data such that each predictor variable has a mean of 0 and a standard deviation of 1. This ensures that no single predictor variable is overly influential when performing ridge regression.

Step 3: Fit the ridge regression model and choose a value for λ.

There is no exact formula we can use to determine which value to use for λ. In practice, there are two common ways that we choose λ:

(1) **Create a Ridge trace plot:** This is a plot that visualizes the values of the coefficient estimates as λ increases towards infinity. Typically we choose λ as the value where most of the coefficient estimates begin to stabilize.

(2) **Calculate the test MSE for each value of λ**

Another way to choose λ is to simply calculate the test MSE of each model with different values of λ and choose λ to be the value that produces the lowest test MSE.

Pros & Cons of Ridge Regression

The biggest **benefit** of ridge regression is its ability to produce a lower test mean squared error (MSE) compared to least squares regression when multicollinearity is present.

However, the biggest **drawback** of ridge regression is its inability to perform variable selection since it includes all predictor variables in the final model. Since some predictors will get shrunken very close to zero, this can make it hard to interpret the results of the model.

In practice, ridge regression has the potential to produce a model that can make better predictions compared to a least squares model but it is often harder to interpret the results of the model.

Depending on whether model interpretation or prediction accuracy is more important to you, you may choose to use ordinary least squares or ridge regression in different scenarios.

7.1.1 Ridge Regression Using R

Step 1 : Load the Data We'll use *hp* as the response variable and the variables as the predictors : *mpg, wt, drat, qsec* from *mtcars* data.

To perform *ridge regression*, we'll use functions from the **glmnet** *package*. This package requires the response variable to be a vector and the set of predictor variables to be of the class *data.matrix*.

The following code shows how to define our data:

```
data("mtcars")

# define response variable
y <- mtcars$hp

# define matrix of predictor variables
x <- data.matrix(mtcars[, c('mpg', 'wt', 'drat', 'qsec')])
```

Step 2 : Fit the Ridge Regression model Next, we'll use the **glmnet()** function to fit the ridge regression model and specify **alpha=0**.

Note that setting alpha equal to 1 is equivalent to using Lasso Regression and setting alpha to some value between 0 and 1 is equivalent to using an elastic net.

Also note that ridge regression requires the data to be standardized such that each predictor variable has a mean of 0 and a standard deviation of 1.

Fortunately $glmnet(\quad)$ automatically performs this standardization for you. If we happened to already standardize the variables, we can specify $standardize = False$.

```
library(glmnet)
```

```
#fit ridge regression model
model <- glmnet(x, y, alpha = 0)

#view summary of model
summary(model)
```

```
##           Length Class     Mode
## a0        100    -none-    numeric
## beta      400    dgCMatrix S4
## df        100    -none-    numeric
## dim         2    -none-    numeric
## lambda    100    -none-    numeric
## dev.ratio 100    -none-    numeric
## nulldev     1    -none-    numeric
## npasses     1    -none-    numeric
## jerr        1    -none-    numeric
## offset      1    -none-    logical
## call        4    -none-    call
## nobs        1    -none-    numeric
```

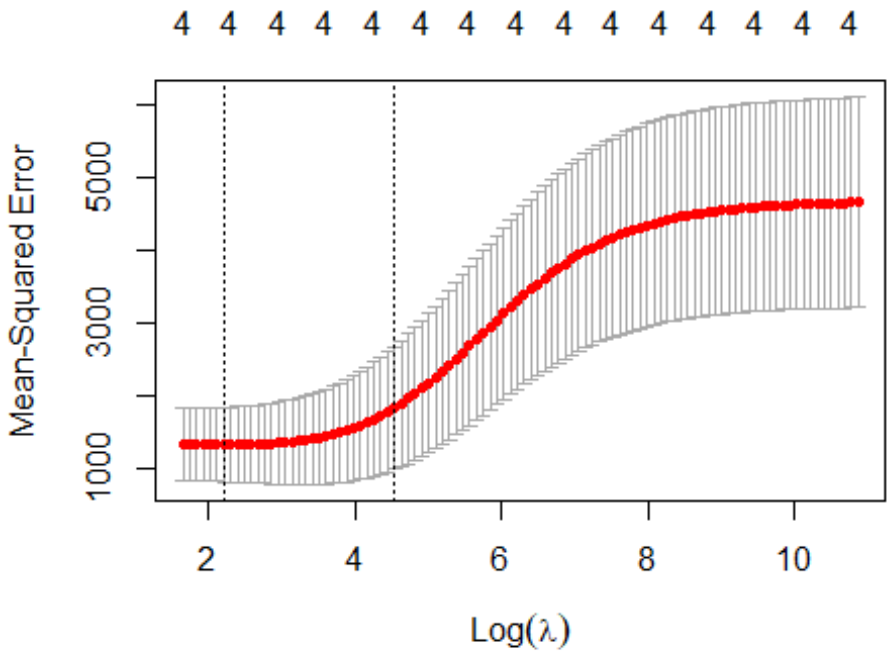
Step 3: Choose an Optimal Value for Lambda Next, we'll identify the lambda value that produces the lowest test mean squared error (MSE) by using k-fold cross-validation.

Fortunately, **glmnet** has the function **cv.glmnet()** that automatically performs k-fold cross validation using *k = 10 folds*.

```
# perform k-fold cross-validation to find optimal lambda value
cv_model <- cv.glmnet(x, y, alpha = 0)

# find optimal lambda value that minimizes test MSE
best_lambda <- cv_model$lambda.min
best_lambda
```

```
## [1] 9.153244
```

```
# produce plot of test MSE by lambda value
plot(cv_model)
```



The lambda value that minimizes the test MSE turns out to be **11.02511**.

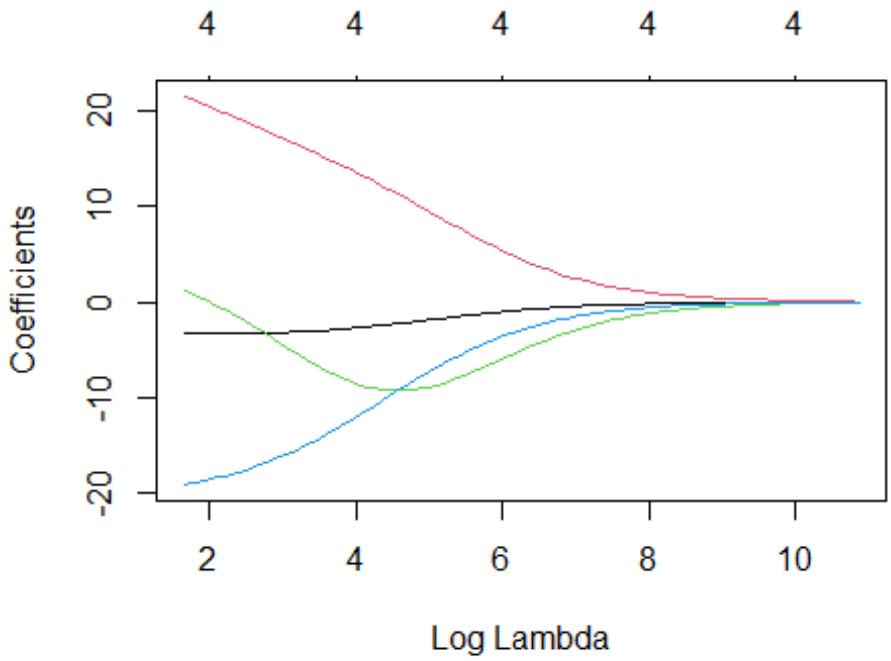
Step 4: Analyze Final Model
Lastly, we can analyze the final model produced by the optimal lambda value.

We can use the following code to obtain the coefficient estimates for this model :

```
# find coefficients of best model
best_model <- glmnet(x, y, alpha = 0, lambda = best_lambda)

coef(best_model)

## 5 x 1 sparse Matrix of class "dgCMatrix"
##                       s0
## (Intercept) 476.3006630
## mpg          -3.3010159
## wt           19.7334277
## drat         -0.8298609
## qsec        -18.1411255
```

We can also produce a Trace plot to visualize how the coefficient estimates changed as a result of increasing lambda :

```
# Produce Ridge Trace plot
plot(model, xvar = "lambda")
```



Lastly, we can calculate the *R-squared* of the model on the training data :

```
# use fitted best model to make predictions
y_predicted <- predict(model, s = best_lambda, newx = x)

# find SST and SSE
sst <- sum((y - mean(y))^2)
sse <- sum((y_predicted - y)^2)

# find R-Squared
rsq <- 1 - sse/sst
rsq

## [1] 0.8009616
```

The R-squared turns out to be **0.7987**. That is, the best model was able to explain 79.87% of the variation in the response values of the training data.

7.2 Lasso Regression

In ordinary multiple linear regression, we use a set of p predictor variables and a response variable to fit a model of the form :

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p + \epsilon$$

where, Y : The response Vairable X_j : The j^{th} predictor variable β_j : The average effect on Y of a one unit increase in X_j, holding all other predictors fixed ϵ : The error term

The values for $\beta_0, \beta_1, \beta_2, \ldots, \beta_p$ are chosen using the **least square method**, which minimizes the sum of squared residuals (RSS) :

$$RSS = \sum_{i=1}^{p} (y_i - \widehat{y_i})^2$$

where, Σ : A greek symbol that means sum y_i : The actual response value for the ith observation $\widehat{y_i}$: The predicted response value based on the multiple linear regression model

However, when the predictor variables are highly correlated then *multicollinearity* can become a problem. This can cause the coefficient estimates of the model to be unreliable and have high variance. That is, when the model is applied to a new set of data it hasn't seen before, it's likely to perform poorly.

One way to get around this issue is to use a method known as **lasso regression**, which instead seeks to minimize the following :

$$RSS + \lambda \Sigma |\beta_j|$$

where j ranges from 1 to p and $\lambda \geq 0$.

This second term in the equation is known as a *shrinkage penalty*.

When $\lambda = 0$, this penalty term has no effect and lasso regression produces the same coefficient estimates as least squares.

However, as λ approaches infinity the shrinkage penalty becomes more influential and the predictor variables that aren't importable in the model get shrunk towards zero and some even get dropped from the model.

Why Use Lasso Regression ?

The advantage of lasso regression compared to least squares regression lies in the *bias-variance tradeoff*.

Recall that mean squared error (MSE) is a metric we can use to measure the accuracy of a given model and it is calculated as :

$$MSE = Var\left(\hat{f}(x_0)\right) + \left[Bias\left(\hat{f}(x_0)\right)\right]^2 + Var(\epsilon)$$

$$MSE = Variance + Bias^2 + Irreducible\ error$$

The basic idea of lasso regression is to introduce a little bias so that the variance can be substantially reduced, which leads to a lower overall MSE.

Lasso Regression vs. Ridge Regression

Lasso regression and ridge regression are both known as *regularization methods* because they both attempt to minimize the sum of squared residuals *(RSS)* along with some penalty term.

In other words, they constrain or regularize the coefficient estimates of the model.

However, the penalty terms they use are a bit different:

Lasso regression attempts to minimize $RSS + \lambda\Sigma|\beta_j|$

Ridge regression attempts to minimize $RSS + \lambda\Sigma\beta_j^2$

When we use ridge regression, the coefficients of each predictor are shrunken towards zero but none of them can go completely to zero.

Conversely, when we use lasso regression it's possible that some of the coefficients could go completely to zero when λ gets sufficiently large.

In technical terms, lasso regression is capable of producing "sparse" models – models that only include a subset of the predictor variables.

This begs the question: Is ridge regression or lasso regression better ? The answer: It depends!

In cases where only a small number of predictor variables are significant, lasso regression tends to perform better because it's able to shrink insignificant variables completely to zero and remove them from the model.

However, when many predictor variables are significant in the model and their coefficients are roughly equal then ridge regression tends to perform better because it keeps all of the predictors in the model.

To determine which model is better at making predictions, we perform *k-fold cross-validation*. Whichever model produces the lowest test mean squared error (MSE) is the preferred model to use.

Steps to Perform Lasso Regression in Practice

The following steps can be used to perform lasso regression:

Step 1: Calculate the correlation matrix and VIF values for the predictor variables.
First, we should produce a *correlation matrix* and calculate the *VIF* (variance inflation factor) values for each predictor variable.
If we detect high correlation between predictor variables and high VIF values (some texts define a "high" VIF value as 5 while others use 10) then lasso regression is likely appropriate to use.

However, if there is no multicollinearity present in the data then there may be no need to perform lasso regression in the first place. Instead, we can perform ordinary least squares regression.

Step 2: Fit the lasso regression model and choose a value for λ.
Once we determine that lasso regression is appropriate to use, we can fit the model (using popular programming languages like R or Python) using the optimal value for λ. To determine the optimal value for λ, we can fit several models using different values for λ and choose λ to be the value that produces the lowest test MSE.

Step 3: Compare lasso regression to ridge regression and ordinary least squares regression.
Lastly, we can compare our lasso regression model to a ridge regression model and least squares regression model to determine which model produces the lowest test MSE by using k-fold cross-validation.
Depending on the relationship between the predictor variables and the response variable, it's entirely possible for one of these three models to outperform the others in different scenarios.

7.2.1 Lasso Regression Using R

Step 1: Load the Data We'll use *hp* as the response variable and the variables as the predictors : *mpg, wt, drat, qsec*

To perform *lasso regression*, we'll use functions from the **glmnet** package. This package requires the *response variable* to be a vector and the set of *predictor variables* to be of the class *data.matrix*.

The following code shows how to define our data:

```
# define response variable
y <- mtcars$hp

# define matrix of predictor variables
x <- data.matrix(mtcars[, c('mpg', 'wt', 'drat', 'qsec')])
```

Step 2: Fit the Lasso Regression Model Next, we'll use the $glmnet(\quad)$ function to fit the lasso regression model and specify $alpha = 1$.

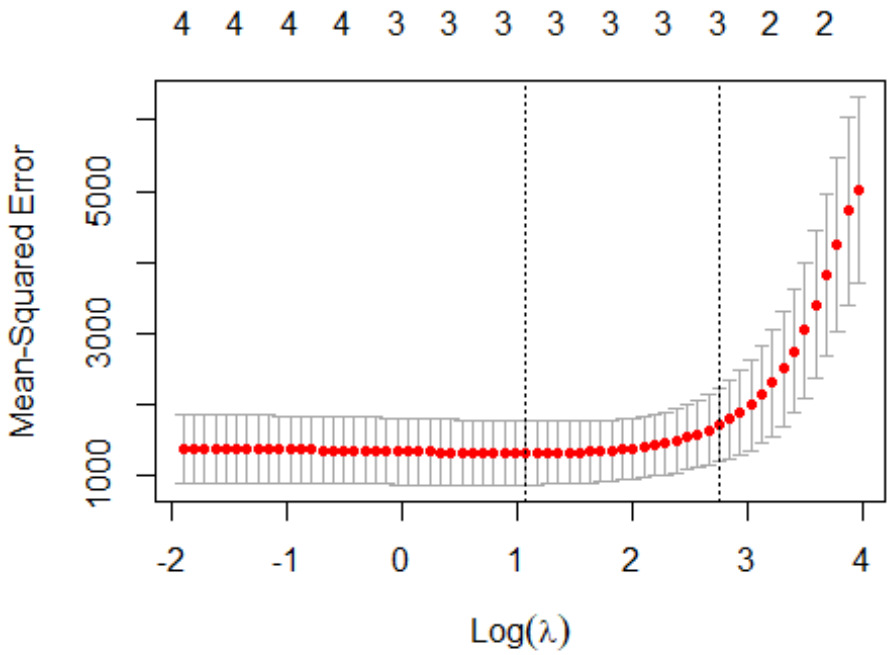
Note that setting alpha equal to 0 is equivalent to using *ridge regression* and setting alpha to some value between 0 and 1 is equivalent to using an elastic net.

To determine what value to use for lambda, we'll perform k-fold cross-validation and identify the lambda value that produces the lowest test mean squared error (MSE).

Note that the function **cv.glmnet()** automatically performs k-fold cross validation using *k = 10* folds.

```
# perform k-fold cross-validation to find optimal lambda value
cv_model <- cv.glmnet(x, y, alpha = 1)

# find optimal lambda value that minimizes test MSE
best_lambda <- cv_model$lambda.min
best_lambda
```

```
## [1] 2.928367

# produce plot of test MSE by lambda value
plot(cv_model)
```



The lambda value that minimizes the test MSE turns out to be 3.53.

Step 3: Analyze Final Model Lastly, we can analyze the final model produced by the optimal lambda value.

We can use the following code to obtain the coefficient estimates for this model :

```
# find coefficients of best model
best_model <- glmnet(x, y, alpha = 1, lambda = best_lambda)
coef(best_model)

## 5 x 1 sparse Matrix of class "dgCMatrix"
##                      s0
## (Intercept) 483.169999
## mpg          -2.981768
## wt           21.029736
## drat          .
## qsec        -19.286215
```

No coefficient is shown for the predictor **drat** because the lasso regression shrunk the coefficient all the way to zero. This means it was completely dropped from the model because it wasn't influential enough.

Note that this is a key difference between *ridge regression and lasso regression.* Ridge regression shrinks all coefficients towards zero, but lasso regression has the potential to remove predictors from the model by shrinking the coefficients completely to zero.

We can also use the final lasso regression model to make predictions on new observations. For example, suppose we have a new car with the following attributes:

- mpg: 24
- wt: 2.5
- drat: 3.5
- qsec: 18.5

The following code shows how to use the fitted lasso regression model to predict the value for hp of this new observation :

```
# define new observation
new = matrix(c(24, 2.5, 3.5, 18.5), nrow=1, ncol=4)

# use lasso regression model to predict response value
predict(best_model, s = best_lambda, newx = new)

##            s1
## [1,] 107.3869
```

Based on the input values, the model predicts this car to have an hp value of 107.97.

Lastly, we can calculate the *R-squared* of the model on the training data :

```
# use fitted best model to make predictions
y_predicted <- predict(best_model, s = best_lambda, newx = x)

# find SST and SSE
sst <- sum((y - mean(y))^2)
sse <- sum((y_predicted - y)^2)

# find R-Squared
rsq <- 1 - sse/sst
rsq

## [1] 0.8032914
```

The R-squared turns out to be **0.8018**. That is, the best model was able to explain 80.18% of the variation in the response values of the training data.

8 Dimension Reduction

8.1 Principal Components Regression(PCR)

One of the most common problems that you'll encounter when building models is *multicollinearity*. This occurs when two or more predictor variables in a dataset are highly correlated.

When this occurs, a given model may be able to fit a training dataset well but it will likely perform poorly on a new dataset it has never seen because it *overfit* the training set.

One way to avoid overfitting is to use some type of **subset selection** method like: 1. Best subset selection 2. Stepwise selection

These methods attempt to remove irrelevant predictors from the model so that only the most important predictors that are capable of predicting the variation in the response variable are left in the final model.

Another way to avoid overfitting is to use some type of **regularization** method like :
1. Ridge Regression
2. Lasso Regression

These methods attempt to constrain or regularize the coefficients of a model to reduce the variance and thus produce models that are able to generalize well to new data.

An entirely different approach to dealing with multicollinearity is known as dimension reduction.

A common method of dimension reduction is know as principal components regression, which works as follows :

1. Suppose a given dataset contains p predictors : $X_1, X_2, \ldots, X_p$
2. Calculate $Z_1, \ldots, Z_M$ to be the M linear combinations of the original p predictors.

- $Z_m = \sum \phi_{jm} X_j$ for some constants $\phi_{1m}, \phi_{2m}, \ldots, \phi_{pm} \quad m = 1, \ldots, M$.
- Z_1 is the linear combination of the predictors that captures the most variance possible.
- Z_2 is the next linear combination of the predictors that captures the most variance while being *orthogonal* (i.e. uncorrelated) to Z_1.
- Z_3 is then the next linear combination of the predictors that captures the most variance while being orthogonal to Z_2.
- And so on.

3. Use the method of least squares to fit a linear regression model using the first M principal components Z_1,..., Z_M$ as predictors.

The phrase **dimension reduction** comes from the fact that this method only has to estimate $M + 1$ coefficients instead of $p + 1$ coefficients, where $M < p$.

In other words, the dimension of the problem has been reduced from $p + 1\ to\ M + 1$.

In many cases where multicollinearity is present in a dataset, principal components regression is able to produce a model that can generalize to new data better than conventional *multiple linear regression*.

Steps to Perform Principal Components Regression
In practice, the following steps are used to perform principal components regression:

1. Standardize the predictors.
First, we typically standardize the data such that each predictor variable has a mean value of 0 and a standard deviation of 1. This prevents one predictor from being overly influential, especially if it's measured in different units (i.e. if X_1 is measured in inches and X_2 is measured in yards).

2. Calculate the principal components and perform linear regression using the principal components as predictors.
Next, we calculate the principal components and use the method of least squares to fit a linear regression model using the first M principal components $Z_1, \ldots, Z_M$ as predictors.

3. Decide how many principal components to keep.
Next, we use *k-fold cross-validation* to find the optimal number of principal components to keep in the model. The "optimal" number of principal components to keep is typically the number that produces the lowest test mean-squared error (MSE).

Pros & Cons of Principal Components Regression

Principal Components Regression (PCR) offers the following *pros* : -
PCR tends to perform well when the first few principal components are able to capture most of the variation in the predictors along with the relationship with the response variable.
PCR can perform well even when the predictor variables are highly correlated because it produces principal components that are orthogonal (i.e. uncorrelated) to each other.
PCR doesn't require you to choose which predictor variables to remove from the model since each principal component uses a linear combination of all of the predictor variables.
PCR can be used when there are more predictor variables than observations, unlike multiple linear regression.

However, PCR comes with one *con* : -
PCR does not consider the response variable when deciding which principal components to keep or drop. Instead, it only considers the magnitude of the variance among the predictor variables captured by the principal components. It's possible that in some cases the principal components with the largest variances aren't actually able to predict the response variable well.

In practice, we fit many different types of models (PCR, Ridge, Lasso, Multiple Linear Regression, etc.) and use k-fold cross-validation to identify the model that produces the lowest test MSE on new data.

In cases where multicollinearity is present in the original dataset (which is often), PCR tends to perform better than ordinary least squares regression. However, it's a good idea to fit several different models so that you can identify the one that generalizes best to unseen data.

8.1.1 PCR Using R

Principal Components Regression

Step 1 : Load Necessary Packages
The easiest way to perform principal components regression in R is by using functions from the **pls** package

```
# load pls package
library(pls)
```

Step 2: Fit PCR Model
For this example, we'll use the built-in R dataset called mtcars which contains data about various types of cars:

```
#view first six rows of mtcars dataset
head(mtcars, 5)
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
```

For this example we'll fit a principal components regression (PCR) model using *hp* as the response variable and the variables as the predictor variables : *mpg, disp, drat, wt, qsec*

The following code shows how to fit the PCR model to this data. Note the following arguments: - $\boldsymbol{scale = TRUE}$ **:** This tells R that each of the predictor variables should be scaled to have a mean of 0 and a standard deviation of 1. This ensures that no predictor variable is overly influential in the model if it happens to be measured in different units. $\boldsymbol{validation = "CV"}$: This tells R to use k-fold cross-validation to evaluate the performance of the model. Note that this uses k=10 folds by default.
Also note that you can specify "LOOCV" instead to perform leave-one-out cross-validation.

```
#make this example reproducible
set.seed(1)

#fit PCR model
model <- pcr(hp~mpg+disp+drat+wt+qsec, data=mtcars,
             scale=TRUE, validation="CV")
```

Step 3: Choose the Number of Principal Components
Once we've fit the model, we need to determine the number of principal components worth

keeping. The way to do so is by looking at the test root mean squared error (test RMSE) calculated by the k-fold cross-validation :

```
# view summary of model fitting
summary(model)
```

```
## Data:    X dimension: 32 5
##  Y dimension: 32 1
## Fit method: svdpc
## Number of components considered: 5
##
## VALIDATION: RMSEP
## Cross-validated using 10 random segments.
##        (Intercept)  1 comps  2 comps  3 comps  4 comps  5 comps
## CV           69.66    43.74    34.58    34.93    36.34    37.40
## adjCV        69.66    43.65    34.30    34.61    35.95    36.95
##
## TRAINING: % variance explained
##     1 comps  2 comps  3 comps  4 comps  5 comps
## X     69.83    89.35    95.88    98.96   100.00
## hp    62.38    81.31    81.96    81.98    82.03
```

There are two tables of interest in the output:

1. VALIDATION : RMSEP

This table tells us the test RMSE calculated by the k-fold cross validation. We can see the following:

- If we only use the intercept term in the model, the test RMSE is 69.66.
- If we add in the first principal component, the test RMSE drops to 44.56.
- If we add in the second principal component, the test RMSE drops to 35.64.

We can see that adding additional principal components actually leads to an increase in test RMSE. Thus, it appears that it would be optimal to only use two principal components in the final model.

2. TRAINING : % variance explained

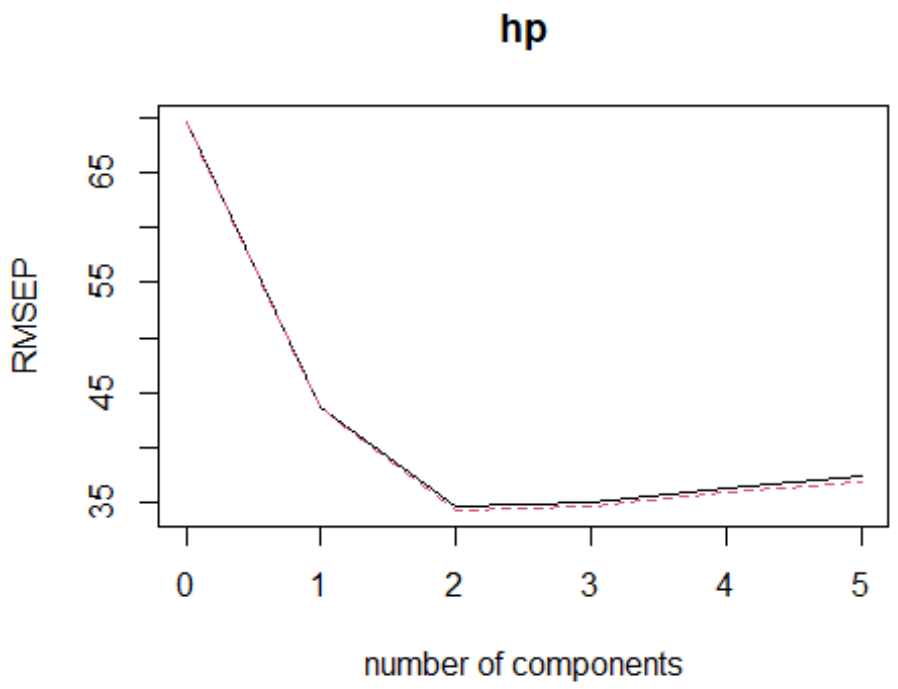
This table tells us the percentage of the variance in the response variable explained by the principal components. We can see the following:

- By using just the first principal component, we can explain 69.83% of the variation in the response variable.
- By adding in the second principal component, we can explain 89.35% of the variation in the response variable.

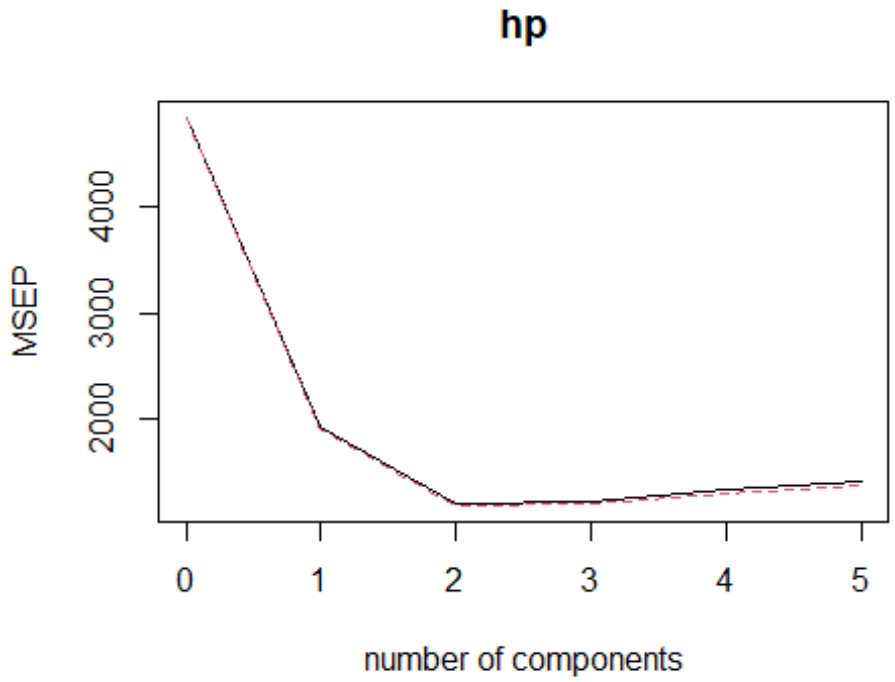
Note that we'll always be able to explain more variance by using more principal components, but we can see that adding in more than two principal components doesn't actually increase the percentage of explained variance by much.

We can also visualize the test RMSE (along with the test MSE and R-squared) based on the number of principal components by using the **validationplot()** function.

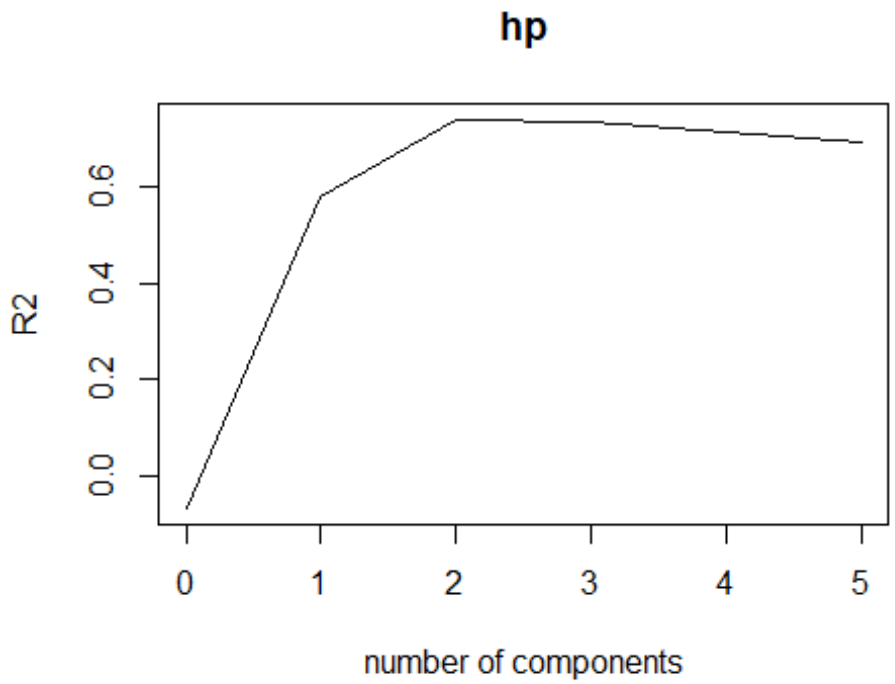
```
# visualize cross-validation plots
validationplot(model)
```




```
validationplot(model, val.type="MSEP")
```




```
validationplot(model, val.type="R2")
```



In each plot we can see that the model fit improves by adding in two principal components, yet it tends to get worse when we add more principal components.

Thus, the optimal model includes just the first two principal components.

Step 4: Use the Final Model to Make Predictions
We can use the final PCR model with two principal components to make predictions on new observations.

```
#define training and testing sets
train <- mtcars[1:25, c("hp", "mpg", "disp", "drat", "wt", "qsec")]

y_test <- mtcars[26:nrow(mtcars), c("hp")]

test <- mtcars[26:nrow(mtcars), c("mpg", "disp", "drat", "wt", "qsec")]

#use model to make predictions on a test set
model <- pcr(hp~mpg+disp+drat+wt+qsec, data=train, scale=TRUE,
validation="CV")

pcr_pred <- predict(model, test, ncomp=2)

#calculate RMSE
sqrt(mean((pcr_pred - y_test)^2))

## [1] 56.86549
```

We can see that the test RMSE turns out to be **56.86549**. This is the average deviation between the predicted value for hp and the observed value for hp for the observations in the testing set.

8.2 Partial Least Squares (PLS)

One of the most common problems that you'll encounter in machine learning is *multicollinearity*. This occurs when two or more predictor variables in a dataset are highly correlated.

When this occurs, a model may be able to fit a training dataset well but it may perform poorly on a new dataset it has never seen because it *overfits* the training set.

One way to get around the problem of multicollinearity is to use *principal components regression (PCR)*, which calculates *M* linear combinations (known as "principal components") of the original p predictor variables and then uses the method of least squares to fit a linear regression model using the principal components as predictors.

The drawback of principal components regression (PCR) is that it does not consider the response variable when calculating the principal components.

Instead, it only considers the magnitude of the variance among the predictor variables captured by the principal components. Because of this, it's possible that in some cases the

principal components with the largest variances aren't actually able to predict the response variable well.

A technique that is related to PCR is known as partial least squares.
Similar to PCR, partial least squares calculates M linear combinations (known as "PLS components") of the original p predictor variables and uses the method of least squares to fit a linear regression model using the PLS components as predictors.

But unlike PCR, partial least squares attempts to find linear combinations that explain the variation in both the response variable and the predictor variables.

Steps to Perform Partial Least Squares

In practice, the following steps are used to perform partial least squares.

1. Standardize the data such that all of the predictor variables and the response variable have a mean of 0 and a standard deviation of 1. This ensures that each variable is measured on the same scale.

2. Calculate $Z_1, \ldots, Z_M$ to be the M linear combinations of the original p predictors.

- $Z_m = \sum \phi_{jm} X_j$ for some constants $\phi_{1m}, \phi_{2m}, \ldots, \phi_{pm} \quad m = 1,2, \ldots, M$
- To calculate Z_1, set ϕ_{j1} equal to the coefficient from the simple linear regression of Y onto X_j is the linear combination of the predictors that captures the most variance possible.
- To calculate Z_2, regression each variable on Z_1 and take the residuals. Then calculate Z_2 using this orthogonalized data in exactly the same manner that Z_1 was calculated.
- Repeat this process M times to obtain the M PLS components.

3. Use the method of least squares to fit a linear regression model using the PLS components $Z_1, \ldots, Z_M$ as predictors.

4. Lastly, use *k-fold cross-validation* to find the optimal number of PLS components to keep in the model. The "optimal" number of PLS components to keep is typically the number that produces the lowest test mean-squared error (MSE).

Conclusion

In cases where multicollinearity is present in a dataset, partial least squares tends to perform better than ordinary least squares regression. However, it's a good idea to fit several different models so that we can identify the one that generalizes best to unseen data.

In practice, we fit many different types of models **(PLS, PCR, Ridge, Lasso, Multiple Linear Regression, etc.)** to a dataset and use k-fold cross-validation to identify the model that produces the lowest test MSE on new data.

8.2.1 PLS Using R

Partial Least Squares

Step 1 : Step Necessary Package

```
library(pls)
```

Step 2 : Fit Partial Least Squares Model
We'll fit a partial least squares (PLS) model using *hp* as the response variable and the variables as the predictor variables : *mpg, disp, drat, wt, qsec* from *mtcars* data.

The following code shows how to fit the PLS model to this data. Note the following arguments :

- $\boldsymbol{scale = TRUE}$: This tells R that each of the variables in the dataset should be scaled to have a mean of 0 and a standard deviation of 1. This ensures that no predictor variable is overly influential in the model if it happens to be measured in different units.
- $\boldsymbol{validation = "CV"}$: This tells R to use k-fold cross-validation to evaluate the performance of the model. Note that this uses *k=10* folds by default. Also note that you can specify *"LOOCV"* instead to perform leave-one-out cross-validation.

```
# make this example reproducible
set.seed(1)

# fit PCR model
model <- plsr(hp ~ mpg + disp + drat + wt + qsec, data=mtcars,
              scale=TRUE, validation="CV")
```

Step 3: Choose the Number of PLS Components
Once we've fit the model, we need to determine the number of PLS components worth keeping.

The way to do so is by looking at the test root mean squared error (test RMSE) calculated by the k-fold cross-validation :

```
# View summary of model fitting
summary(model)
```

```
## Data:    X dimension: 32 5
##  Y dimension: 32 1
## Fit method: kernelpls
## Number of components considered: 5
##
## VALIDATION: RMSEP
## Cross-validated using 10 random segments.
##        (Intercept)  1 comps  2 comps  3 comps  4 comps  5 comps
## CV           69.66    38.81    34.94    36.08    37.27    37.40
## adjCV        69.66    38.69    34.64    35.72    36.82    36.95
##
## TRAINING: % variance explained
##     1 comps  2 comps  3 comps  4 comps  5 comps
## X     68.66    89.27    95.82    97.94   100.00
## hp    71.84    81.74    82.00    82.02    82.03
```

There are two tables of interest in the output:

1. VALIDATION : RMSEP

This table tells us the test RMSE calculated by the k-fold cross validation. We can see the following:

- If we only use the intercept term in the model, the test RMSE is 69.66.
- If we add in the first PLS component, the test RMSE drops to 40.57.
- If we add in the second PLS component, the test RMSE drops to 35.48.

We can see that adding additional PLS components actually leads to an increase in test RMSE. Thus, it appears that it would be optimal to only use two PLS components in the final model.

2. TRAINING: % variance explained

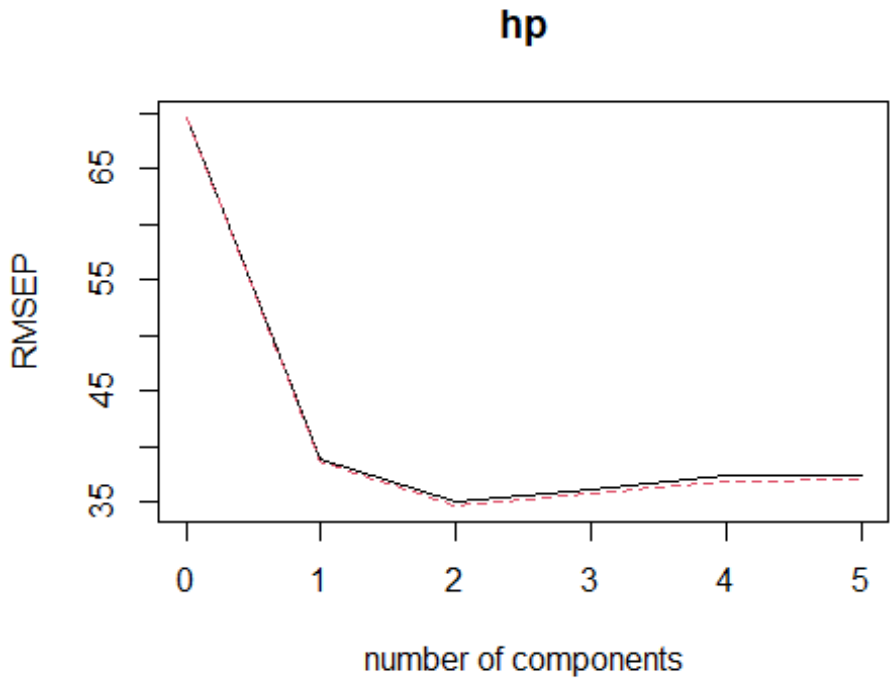
This table tells us the percentage of the variance in the response variable explained by the PLS components. We can see the following:

- By using just the first PLS component, we can explain 68.66% of the variation in the response variable.
- By adding in the second PLS component, we can explain 89.27% of the variation in the response variable.

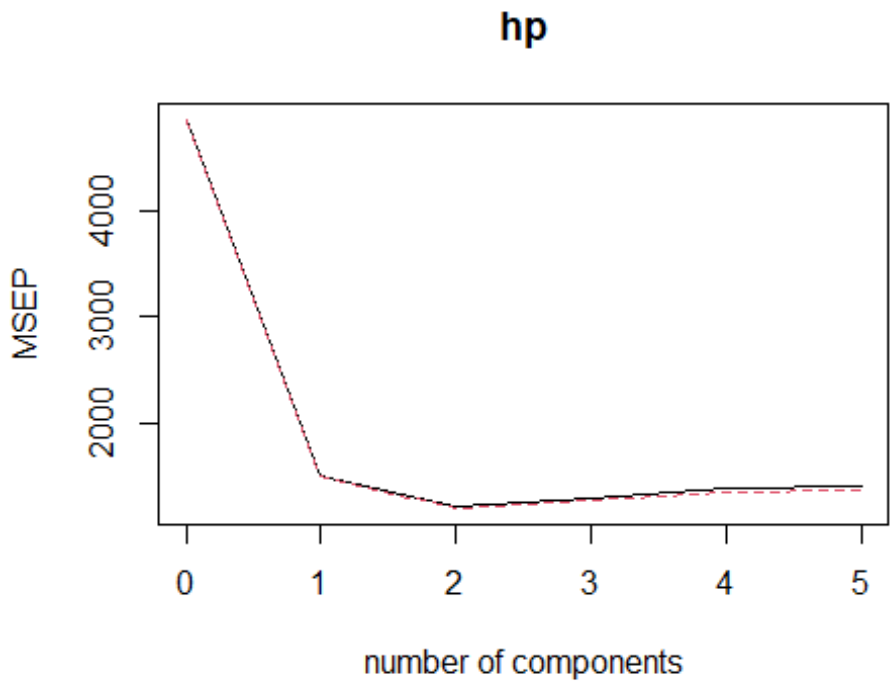
Note that we'll always be able to explain more variance by using more PLS components, but we can see that adding in more than two PLS components doesn't actually increase the percentage of explained variance by much.

We can also visualize the test RMSE (along with the test MSE and R-squared) based on the number of PLS components by using the **validationplot()** function.

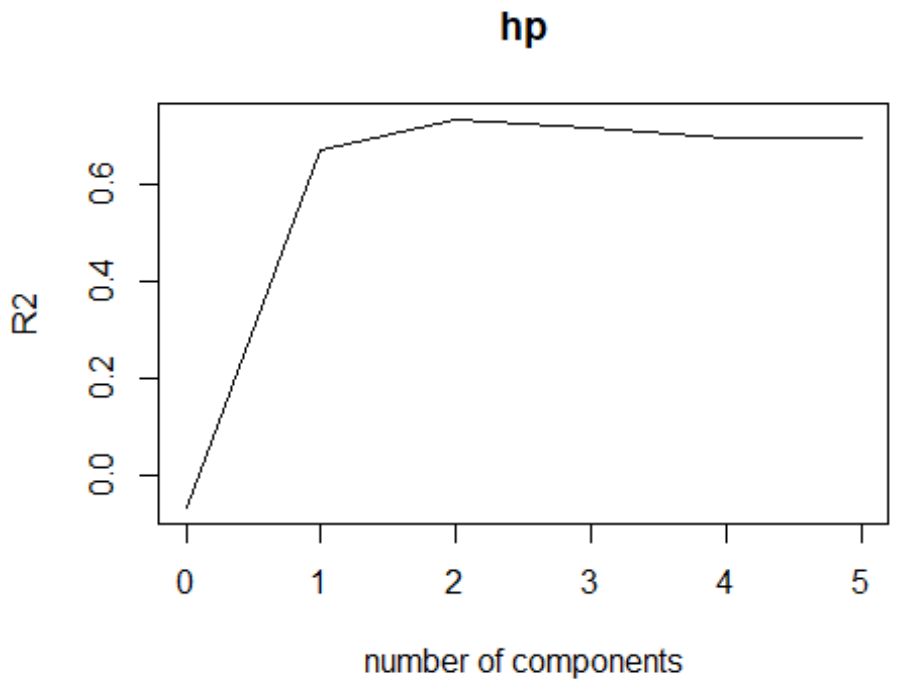
```
# visualize cross-validation plots
validationplot(model)
```




```
validationplot(model, val.type="MSEP")
```




```
validationplot(model, val.type="R2")
```



In each plot we can see that the model fit improves by adding in two PLS components, yet it tends to get worse when we add more PLS components.

Thus, the optimal model includes just the first two PLS components.

Step 4: Use the Final Model to Make Predictions
We can use the final model with two PLS components to make predictions on new observations.

The following code shows how to split the original dataset into a training and testing set and use the final model with two PLS components to make predictions on the testing set.

```
# define training and testing sets
train <- mtcars[1:25, c("hp", "mpg", "disp", "drat", "wt", "qsec")]

y_test <- mtcars[26:nrow(mtcars), c("hp")]

test <- mtcars[26:nrow(mtcars), c("mpg", "disp", "drat", "wt", "qsec")]

# use model to make predictions on a test set
model <- plsr(hp~mpg+disp+drat+wt+qsec, data=train, scale=TRUE,
validation="CV")

pcr_pred <- predict(model, test, ncomp=2)

# calculate RMSE
sqrt(mean((pcr_pred - y_test)^2))
```

```
## [1] 54.89609
```

We can see that the test RMSE turns out to be **54.896**. This is the average deviation between the predicted value for hp and the observed value for hp for the observations in the testing set.

Note that an equivalent principal components regression model with two principal components produced a test RMSE of **56.865**. Thus, the PLS model slightly outperformed the PCR model for this dataset.

9 Advanced Regression Models

9.1 Polynomial Regression

When we have a dataset with *one predictor variable and one response variable*, we often use *simple linear regression* to quantify the relationship between the two variables.

However, simple linear regression *(SLR)* assumes that the relationship between the predictor and response variable is linear. Written in mathematical notation, SLR assumes that the relationship takes the form :

$$Y = \beta_0 + \beta_1 X + \epsilon$$

But in practice the relationship between the two variables can actually be nonlinear and attempting to use linear regression can result in a poorly fit model.

One way to account for a nonlinear relationship between the predictor and response variable is to use **polynomial regression**, which takes the form :

$$Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \cdots + \beta_h X^h + \epsilon$$

In this equation, h is referred to as the degree of the polynomial.

As we increase the value for h, the model is able to fit nonlinear relationships better, but in practice we rarely choose h to be greater than 3 or 4. Beyond this point, the model becomes too flexible and *overfits the data*.

Technical Notes

- Although polynomial regression can fit nonlinear data, it is still considered to be a form of linear regression because it is linear in the coefficients $\beta_1, \beta_2, \cdots, \beta_h$
- Polynomial regression can be used for multiple predictor variables as well but this creates interaction terms in the model, which can make the model extremely complex if more than a few predictor variables are used.

When to Use Polynomial Regression

We use polynomial regression when the relationship between a predictor and response variable is nonlinear.

There are *three* common ways to detect a nonlinear relationship:

1. Create a Scatterplot
The easiest way to detect a nonlinear relationship is to create a *scatterplot* of the response vs. predictor variable.

2. Create a residuals vs. fitted plot
Another way to detect nonlinearity is to fit a simple linear regression model to the data and then produce a *residuals vs. fitted values plot*.

If the residuals of the plot are roughly evenly distributed around zero with no clear pattern, then simple linear regression is likely sufficient.

However, if the residuals display a nonlinear pattern in the plot then this is a sign that the relationship between the predictor and the response is likely nonlinear.

3. Calculate the R^2 of the model
The R^2 value of a regression model tells us the percentage of the variation in the response variable that can be explained by the predictor variable(s).

If we fit a simple linear regression model to a dataset and the R^2 value of the model is quite low, this could be an indication that the relationship between the predictor and response variable is more complex than just a simple linear relationship.

This could be a sign that you may need to try polynomial regression instead.

How to Choose the Degree of the Polynomial

A polynomial regression model takes the following form:

$$Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \cdots + \beta_h X^h + \epsilon$$

In this equation, h is the degree of the polynomial.

But how do we choose a value for h ?

In practice, we fit several different models with different values of h and perform *k-fold cross-validation* to determine which model produces the lowest test mean squared error (MSE).

For example, we may fit the following models to a given dataset :

- $Y = \beta_0 + \beta_1 X$
- $Y = \beta_0 + \beta_1 X + \beta_2 X^2$
- $Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3$
- $Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3 + \beta_4 X^4$

We can then use k-fold cross-validation to calculate the test MSE of each model, which will tell us how well each model performs on data it hasn't seen before.

The Bias-Variance Tradeoff of Polynomial Regression

There exists a *bias-variance tradeoff* when using polynomial regression. As we increase the degree of the polynomial, the bias decreases (as the model becomes more flexible) but the variance increases.

As with all machine learning models, we must find an optimal tradeoff between bias and variance.

In most cases it helps to increase the degree of the polynomial to an extent, but beyond a certain value the model begins to fit the noise of the data and the test MSE begins to decrease.

To ensure that we fit a model that is flexible but not too flexible, we use k-fold cross-validation to find the model that produces the lowest test MSE.

9.1.1 Polynomial Regression Using R

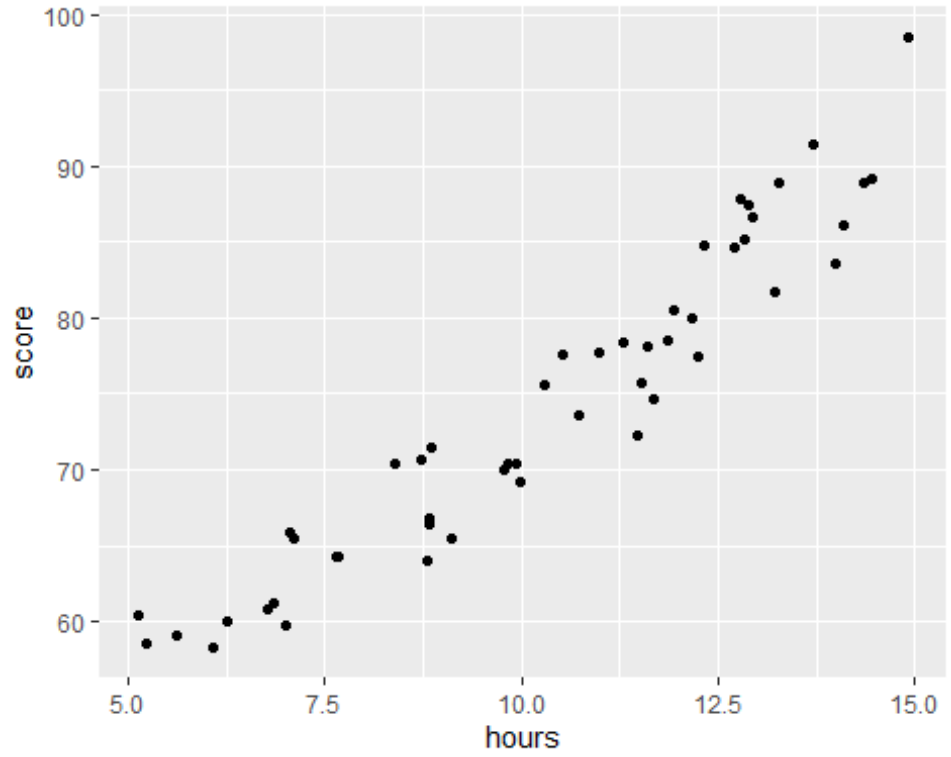
Step 1 : Create / Load the Data
We'll create a dataset that contains the number of hours studied and final exam score for a class of 50 students:

```
# make this example reproducible
set.seed(1)

# create dataset
df <- data.frame(hours = runif(50, 5, 15), score=50)

df$score = df$score + df$hours^3/150 + df$hours*runif(50, 1, 2)

# view first six rows of data
head(df)
```

```
##       hours    score
## 1  7.655087 64.30191
## 2  8.721239 70.65430
## 3 10.728534 73.66114
## 4 14.082078 86.14630
## 5  7.016819 59.81595
## 6 13.983897 83.60510
```

Step 2 : Visualize the Data
Before we fit a regression model to the data, let's first create a scatterplot to visualize the relationship between hours studied and exam score:

```
ggplot(df, aes(x = hours, y = score)) +
  geom_point()
```



We can see that the data exhibits a bit of a quadratic relationship, which indicates that polynomial regression could fit the data better than simple linear regression.

Step 3 : Fit the Polynomial Regression Models
Next, we'll fit five different polynomial regression models with degrees $h = 1 \ldots 5$ and use k-fold cross-validation with $k = 10$ folds to calculate the test MSE for each model :

```
# randomly shuffle data
df.shuffled <- df[sample(nrow(df)),]

# define number of folds to use for k-fold cross-validation
K <- 10

# define degree of polynomials to fit
degree <- 5

# create k equal-sized folds
folds <- cut(seq(1,nrow(df.shuffled)),breaks=K,labels=FALSE)

# create object to hold MSE's of models
mse <- matrix(data=NA,nrow=K,ncol=degree)

# Perform K-fold cross validation
for(i in 1:K){

    # define training and testing data
```

```
    testIndexes <- which(folds==i,arr.ind=TRUE)
    testData <- df.shuffled[testIndexes, ]
    trainData <- df.shuffled[-testIndexes, ]

    # use k-fold cv to evaluate models
    for (j in 1:degree){
        fit.train <- lm(score ~ poly(hours,j), data=trainData)
        fit.test <- predict(fit.train, newdata=testData)
        mse[i,j] <- mean((fit.test-testData$score)^2)
    }
}

# find MSE for each degree
colMeans(mse)

## [1]  9.886172  8.589655  9.439514 10.214915 12.745574
```

From the output we can see the test MSE for each model :

- Test MSE with degree $h = 1$: 9.80
- Test MSE with degree $h = 2$: 8.75
- Test MSE with degree $h = 3$: 9.60
- Test MSE with degree $h = 4$: 10.59
- Test MSE with degree $h = 5$: 13.55

The model with the lowest test MSE turned out to be the polynomial regression model with degree $h = 2$.

This matches our intuition from the original scatterplot : A quadratic regression model fits the data best.

Step 4 : Analyze the Final Model Lastly, we can obtain the coefficients of the best performing model :

```
# fit best model
best <- lm(score ~ poly(hours,2, raw=T), data=df)

# view summary of best model
summary(best)

##
## Call:
## lm(formula = score ~ poly(hours, 2, raw = T), data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -5.6589 -2.0770 -0.4599  2.5923  4.5122
##
## Coefficients:
##                          Estimate Std. Error t value Pr(>|t|)
```

```
## (Intercept)                54.00526    5.52855   9.768 6.78e-13 ***
## poly(hours, 2, raw = T)1 -0.07904    1.15413  -0.068  0.94569
## poly(hours, 2, raw = T)2  0.18596    0.05724   3.249  0.00214 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 2.8 on 47 degrees of freedom
## Multiple R-squared:   0.93,  Adjusted R-squared:  0.927
## F-statistic: 312.1 on 2 and 47 DF,  p-value: < 2.2e-16
```

From the output we can see that the final fitted model is :

$$Score = 54.00526 - .07904 * (hours) + .18596 * (hours)^2$$

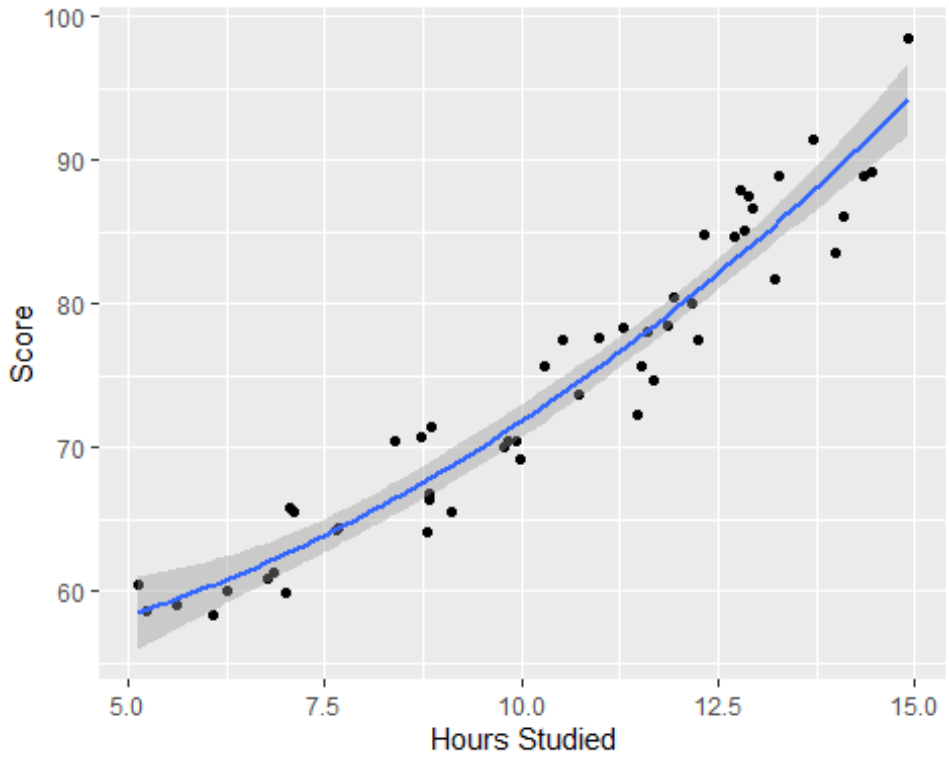
We can use this equation to estimate the score that a student will receive based on the number of hours they studied.

For example, a student who studies for 10 hours is expected to receive a score of **71.81**:

$$Score = 54.00526 - .07904 * (10) + .18596 * (10)^2 = 71.81$$

We can also plot the fitted model to see how well it fits the raw data :

```
ggplot(df, aes(x=hours, y=score)) +
          geom_point() +
          stat_smooth(method='lm', formula = y ~ poly(x,2), size = 1) +
          xlab('Hours Studied') +
          ylab('Score')
```



9.2 Multivariate Adaptive Regression Splines (MARS)

When the relationship between a set of predictor variables and a *response variable* is linear, we can often use *linear regression*, which assumes that the relationship between a given predictor variable and a response variable takes the form :

$$Y = \beta_0 + \beta_1 X + \epsilon$$

But in practice the relationship between the variables can actually be nonlinear and attempting to use linear regression can result in a poorly fit model.

One way to account for a nonlinear relationship between the predictor and response variable is to use *polynomial regression*, which takes the form :

$$Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \cdots + \beta_h X^h + \epsilon$$

In this equation, h is referred to as the "degree" of the polynomial. As we increase the value for h, the model becomes more flexible and is able to fit nonlinear data.

However, polynomial regression has a couple drawbacks:

1. Polynomial regression can easily *overfit* a dataset if the degree, h, is chosen to be too large. In practice, h is rarely larger than 3 or 4 because beyond this point it simply fits the noise of a training set and does not generalize well to unseen data.

2. Polynomial regression imposes a global function on the entire dataset, which is not always accurate.

An alternative to polynomial regression is multivariate adaptive regression splines.

The Basic Idea

Multivariate adaptive regression splines work as follows:

1. Divide a dataset into k pieces First, we divide a dataset into k different pieces. The points where we divide the dataset are known as *knots*.

We identify the knots by assessing each point for each predictor as a potential knot and creating a linear regression model using the candidate features. The point that is able to reduce the most error in the model is deemed to be the knot.

Once we've identified the first knot, we then repeat the process to find additional knots. You can find as many knots as you think is reasonable to start.

2. Fit a regression function to each piece to form a hinge function Once we've chosen the knots and fit a regression model to each piece of the dataset, we're left with something known as a *hinge function*, denoted as $h(x - a)$, where a is the *cutpoint value(s)*.

For example, the hinge function for a model with one knot may be as follows :

- $y = \beta_0 + \beta_1(4.3 - x)\ ;\ if x < 4.3$
- $y = \beta_0 + \beta_1(x - 4.3)\ ;\ if x > 4.$

In this case, it was determined that choosing 4.3 to be the cutpoint value was able to reduce the error the most out of all possible cutpoints values. We then fit a different regression model to the values less than *4.3* compared to values greater than* 4.3*.

A hinge function with two knots may be as follows:

- $y = \beta_0 + \beta_1(4.3 - x)$; $if x < 4.3$
- $y = \beta_0 + \beta_1$ (x-4.3) ; if x > 4.3 & x < 6.7
- $y = \beta_0 + \beta_1(6.7 - x)$; $if x > 6.7$

In this case, it was determined that choosing 4.3 and 6.7 as the cutpoint values was able to reduce the error the most out of all possible cutpoint values. We then fit one regression model to the values less than *4.3*, another regression model to values between *4.3* and *6.7*, and another regression model to the values greater than *4.3*.

3. Choose k based on k-fold cross-validation
Lastly, once we've fit several different models using a different number of knots for each model, we can perform k-fold cross-validation to identify the model that produces the lowest test mean squared error (MSE).

The model with the lowest test MSE is chosen to be the model that generalizes best to new data.

Pros & Cons

Multivariate adaptive regression splines come with the following pros and cons:

Pros :

It can be used for both regression and classification p roblems. - It works well on large datasets.
It offers quick computation. - It does not require you to standardize the predictor variables.

Cons :

- It tends to not perform as well as non-linear methods like random forests and gradient boosting machines.

9.2.1 MARS Using R

Multivariate Adaptive Regression Splines

Multivariate adaptive regression splines (MARS) can be used to model nonlinear relationships between a set of *predictor variables and a response variable*.

This method works as follows:
1. Divide a dataset into *k* pieces.
2. Fit a regression model to each piece. 3. Use k-fold cross-validation to choose a value for *k*.

This tutorial provides a step-by-step example of how to fit a MARS model to a dataset in R.

Step 1: Load Necessary Packages
For this example we'll use the **Wage** dataset from the *ISLR* package, which contains the annual wages for 3,000 individuals along with a variety of predictor variables like age, education, race, and more.

Before we fit a MARS model to the data, we'll load the necessary packages:

```
library(ISLR2)     # contains Wage dataset
library(dplyr)     # data wrangling
library(ggplot2)   # plotting
library(earth)     # fitting MARS models
library(caret)     # tuning model parameters
```

Step 2 : View Data Next, we'll view the first six rows of the dataset we're working with :

```
data("Wage")

# head of wage data
Wage %>% head(5)

##        year age           maritl     race       education
region       jobclass          health health_ins  logwage
## 231655 2006  18 1. Never Married 1. White    1. < HS Grad 2. Middle
Atlantic  1. Industrial      1. <=Good      2. No 4.318063
## 86582  2004  24 1. Never Married 1. White 4. College Grad 2. Middle
Atlantic 2. Information 2. >=Very Good      2. No 4.255273
## 161300 2003  45       2. Married 1. White 3. Some College 2. Middle
Atlantic  1. Industrial      1. <=Good     1. Yes 4.875061
## 155159 2003  43       2. Married 3. Asian 4. College Grad 2. Middle
Atlantic 2. Information 2. >=Very Good     1. Yes 5.041393
## 11443  2005  50      4. Divorced 1. White      2. HS Grad 2. Middle
Atlantic 2. Information      1. <=Good     1. Yes 4.318063
##             wage
## 231655  75.04315
## 86582   70.47602
## 161300 130.98218
## 155159 154.68529
## 11443   75.04315
```

Step 3: Build & Optimize the MARS Model
Next, we'll build the MARS model for this dataset and perform k-fold cross-validation to determine which model produces the lowest test RMSE (root mean squared error).

```
#create a tuning grid
hyper_grid <- expand.grid(
  degree = 1:3,
  nprune = seq(2, 50, length.out = 10) %>% floor()
)

#make this example reproducible
set.seed(1)
```

```
#fit MARS model using k-fold cross-validation
cv_mars <- train(
  x = subset(Wage, select = -c(wage, logwage)),
  y = Wage$wage,
  method = "earth",
  metric = "RMSE",
  trControl = trainControl(method = "cv", number = 10),
  tuneGrid = hyper_grid
)


#display model with lowest test RMSE
cv_mars$results %>%
  filter(nprune == cv_mars$bestTune$nprune, degree ==
cv_mars$bestTune$degree)
```

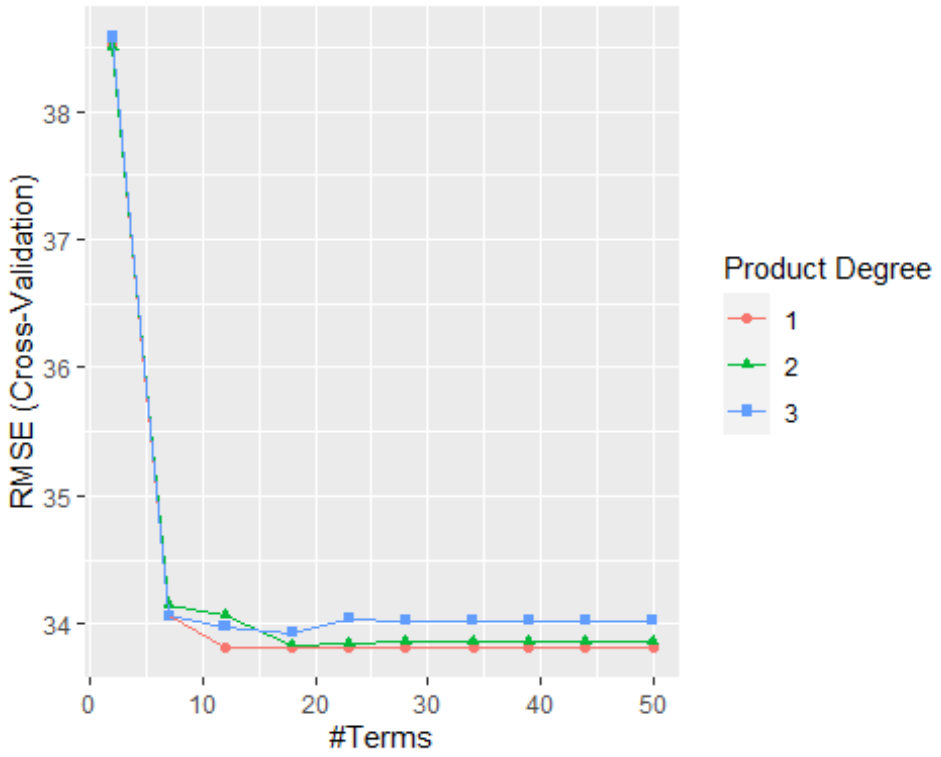
```
##   degree nprune     RMSE  Rsquared      MAE   RMSESD RsquaredSD    MAESD
## 1      1     12 33.81645 0.3431804 22.97108 2.240394 0.03064269 1.455411
```

From the output we can see that the model that produced the lowest test MSE was one with only first-order effects (i.e. no interaction terms) and 12 terms. This model produced a root mean squared error (RMSE) of **33.8164**.

Note : We used method="earth" to specify a MARS model.

We can also create a plot to visualize the test RMSE based on the degree and the number of terms :

```
#display test RMSE by terms and degree
ggplot(cv_mars)
```



In practice we would fit a MARS model along with several other types of models like :

- Multiple Linear Regression
- Polynomial Regression
- Ridge Regression
- Lasso Regression
- Principal Components Regression
- Partial Least Squares

We would then compare each model to determine which one lead to the lowest test error and choose that model as the optimal one to use.

10 Tree - Based Methods

10.1 Classification and Regression Trees (CART)

When the relationship between a set of predictor variables and a *response variable* is linear, methods like *multiple linear regression* can produce accurate predictive models.

However, when the relationship between a set of predictors and a response is highly non-linear and complex then non-linear methods can perform better.

One such example of a non-linear method is *classification and regression trees*, often abbreviated **CART**.

As the name implies, CART models use a set of predictor variables to build decision trees that predict the value of a response variable.

Steps to Build CART Models

We can use the following steps to build a CART model for a given dataset:

Step 1: Use recursive binary splitting to grow a large tree on the training data.

First, we use a *greedy algorithm known as recursive binary splitting* to grow a regression tree using the following method:

- Consider all predictor variables $X_1, X_2, \ldots, X_p$ and all possible values of the cut points for each of the predictors, then choose the predictor and the cut point such that the resulting tree has the lowest RSS (residual standard error).
- For classification trees, we choose the predictor and cut point such that the resulting tree has the lowest misclassification rate.
- Repeat this process, stopping only when each terminal node has less than some minimum number of observations.

This algorithm is *greedy* because at each step of the tree-building process it determines the best split to make based only on that step, rather than looking ahead and picking a split that will lead to a better overall tree in some future step.

Step 2: Apply cost complexity pruning to the large tree to obtain a sequence of best trees, as a function of α.

Once we've grown the large tree, we then need to prune the tree using a method known as cost complexity pruning, which works as follows:

- For each possible tree with T terminal nodes, find the tree that minimizes RSS + $\alpha|T|$.
- Note that as we increase the value of α, trees with more terminal nodes are penalized. This ensures that the tree doesn't become too complex.

This process results in a sequence of best trees for each value of α.

Step 3: Use k-fold cross-validation to choose α.

Once we've found the best tree for each value of α, we can apply k-fold cross-validation to choose the value of α that minimizes the test error.

Step 4: Choose the final model

Lastly, we choose the final model to be the one that corresponds to the chosen value of α.

Pros & Cons of CART Models

CART models offer the following *pros* :

- They are easy to interpret.

- They are easy to explain.
- They are easy to visualize.
- They can be applied to both *regression and classification* problems.

However, CART models come with the following *con* :

- They tend to not have as much predictive accuracy as other non-linear machine learning algorithms. However, by aggregating many decision trees with methods like bagging, boosting, and random forests, their predictive accuracy can be improved.

10.1.1 CART Using R

Classification and Regression Trees

We'll use the **Hitters** dataset from the *ISLR2* package, which contains various information about 263 professional baseball players.

We will use this dataset to build a regression tree that uses the predictor variables *home* runs and *years played* to predict the *Salary* of a given player.

Use the following steps to build this regression tree.

Step 1: Load the necessary packages.

```
library(ISLR) #contains Hitters dataset
library(rpart) #for fitting decision trees
library(rpart.plot) #for plotting decision trees
```

Load the Data

```
data("Hitters")
Hitters %>% names()
```

```
##  [1] "AtBat"     "Hits"      "HmRun"     "Runs"      "RBI"       "Walks"
"Years"     "CAtBat"    "CHits"     "CHmRun"
## [11] "CRuns"     "CRBI"      "CWalks"    "League"    "Division"  "PutOuts"
"Assists"   "Errors"    "Salary"    "NewLeague"
```

Step 2: Build the initial regression tree

First, we'll build a large initial regression tree. We can ensure that the tree is large by using a small value for **cp** which stands for **"complexity parameter"**.

This means we will perform new splits on the regression tree as long as the overall R-squared of the model increases by at least the value specified by *cp*.

We'll then use the **printcp()** function to print the results of the model :

```
# build the initial tree
tree <- rpart(Salary ~ Years + HmRun, data=Hitters,
              control=rpart.control(cp=.0001))
```

```
# view results
printcp(tree)

##
## Regression tree:
## rpart(formula = Salary ~ Years + HmRun, data = Hitters, control =
rpart.control(cp = 1e-04))
##
## Variables actually used in tree construction:
## [1] HmRun Years
##
## Root node error: 53319113/263 = 202734
##
## n=263 (59 observations deleted due to missingness)
##
##            CP nsplit rel error  xerror    xstd
## 1  0.24674996      0   1.00000 1.00503 0.13824
## 2  0.10806932      1   0.75325 0.76468 0.12823
## 3  0.01865610      2   0.64518 0.68960 0.12232
## 4  0.01761100      3   0.62652 0.70566 0.11979
## 5  0.01747617      4   0.60891 0.70351 0.11964
## 6  0.01038188      5   0.59144 0.69105 0.12013
## 7  0.01038065      6   0.58106 0.70474 0.12077
## 8  0.00731045      8   0.56029 0.70146 0.11990
## 9  0.00714883      9   0.55298 0.69998 0.12161
## 10 0.00708618     10   0.54583 0.70160 0.12157
## 11 0.00516285     12   0.53166 0.70613 0.12105
## 12 0.00445345     13   0.52650 0.71341 0.12013
## 13 0.00406069     14   0.52205 0.70786 0.11981
## 14 0.00264728     15   0.51799 0.70415 0.11985
## 15 0.00196586     16   0.51534 0.70457 0.11987
## 16 0.00016686     17   0.51337 0.70283 0.11793
## 17 0.00010000     18   0.51321 0.70443 0.11792
```

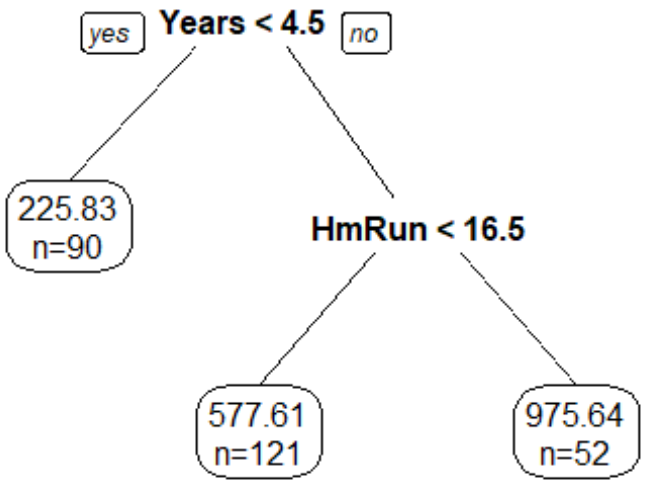
Step 3: Prune the tree

Next, we'll prune the regression tree to find the optimal value to use for cp (the complexity parameter) that leads to the lowest test error.

Note that the optimal value for cp is the one that leads to the lowest **xerror** in the previous output, which represents the error on the observations from the cross-validation data.

```
#  identify best cp value to use
best <- tree$cptable[which.min(tree$cptable[,"xerror"]),"CP"]

#produce a pruned tree based on the best cp value
pruned_tree <- prune(tree, cp=best)

# plot the pruned tree
```

```
prp(pruned_tree,
    faclen=0, # use full names for factor labels
    extra=1,  # display number of obs. for each terminal node
    roundint=F, # don't round to integers in output
    digits=5) # display 5 decimal places in output
```



We can see that the final pruned tree has six terminal nodes. Each terminal node shows the predicted salary of player's in that node along with the number of observations from the original dataset that belong to that note.

For example, we can see that in the original dataset there were 90 players with less than 4.5 years of experience and their average salary was $225.83k.

Step 4: Use the tree to make predictions

We can use the final pruned tree to predict a given player's salary based on their years of experience and average home runs.

For example, a player who has 7 years of experience and 4 average home runs has a predicted salary of $502.81k

We can use the **predict()** function in R to confirm this :

```
# define new player
new <- data.frame(Years=7, HmRun=4)
```

```
# use pruned tree to predict salary of this player
predict(pruned_tree, newdata=new)

##        1
## 577.6061
```

Example 2: Building a Classification Tree in R

For this example, we'll use the **ptitanic** dataset from the **rpart.plot** package, which contains various information about passengers aboard the Titanic.

We will use this dataset to build a classification tree that uses the predictor variables *class, sex, and age* to predict whether or not a given *passenger survived*.

Use the following steps to build this classification tree.

Step 1: Load the necessary packages

First, we'll load the necessary packages for this example:

```
library(rpart) #for fitting decision trees
library(rpart.plot) #for plotting decision trees
```

Load Data

```
data("ptitanic")

head(ptitanic)

##   pclass survived    sex     age sibsp parch
## 1    1st survived female 29.0000     0     0
## 2    1st survived   male  0.9167     1     2
## 3    1st     died female  2.0000     1     2
## 4    1st     died   male 30.0000     1     2
## 5    1st     died female 25.0000     1     2
## 6    1st survived   male 48.0000     0     0
```

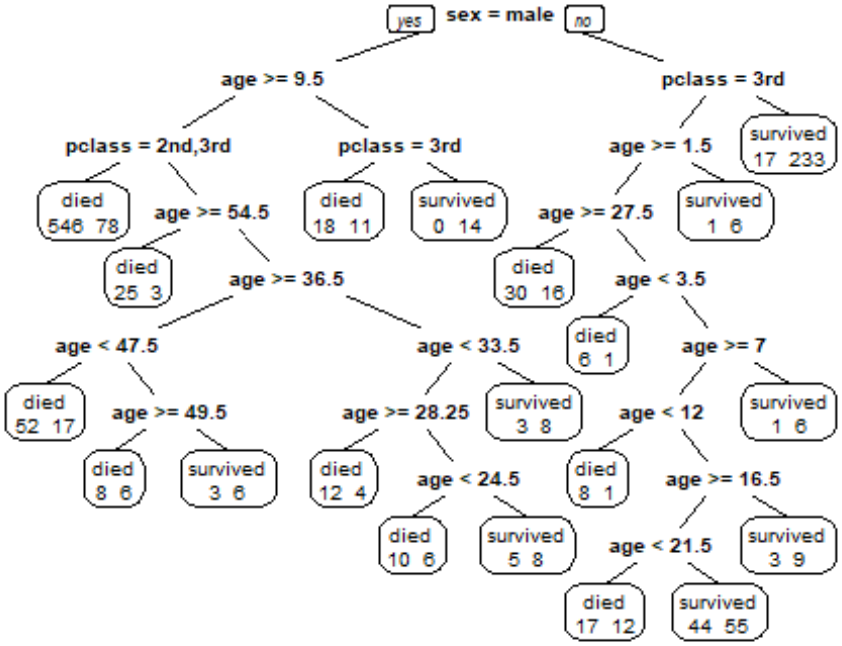
Step 2: Build the initial classification tree

```
# build the initial tree
tree <- rpart(survived~pclass+sex+age, data=ptitanic,
control=rpart.control(cp=.0001))

# view results
printcp(tree)

##
## Classification tree:
## rpart(formula = survived ~ pclass + sex + age, data = ptitanic,
##     control = rpart.control(cp = 1e-04))
##
## Variables actually used in tree construction:
```

```
## [1] age    pclass sex
##
## Root node error: 500/1309 = 0.38197
##
## n= 1309
##
##       CP nsplit rel error xerror     xstd
## 1 0.4240      0     1.000  1.000 0.035158
## 2 0.0140      1     0.576  0.576 0.029976
## 3 0.0095      3     0.548  0.600 0.030413
## 4 0.0070      7     0.510  0.580 0.030050
## 5 0.0050      9     0.496  0.568 0.029825
## 6 0.0025     11     0.486  0.538 0.029238
## 7 0.0020     19     0.464  0.534 0.029157
## 8 0.0001     22     0.458  0.538 0.029238
```

Step 3: Prune the tree

```
# identify best cp value to use
best <- tree$cptable[which.min(tree$cptable[,"xerror"]),"CP"]

# produce a pruned tree based on the best cp value
pruned_tree <- prune(tree, cp=best)

# plot the pruned tree
prp(pruned_tree,
    faclen=0, #use full names for factor labels
    extra=1, #display number of obs. for each terminal node
    roundint=F, #don't round to integers in output
    digits=5) #display 5 decimal places in output
```



We can see that the final pruned tree has 10 terminal nodes. Each terminal node shows the number of passengers that died along with the number that survived.

For example, in the far left node we see that 664 passengers died and **136** survived.

Step 4: Use the tree to make predictions

We can use the final pruned tree to predict the probability that a given passenger will survive based on their class, age, and sex.

For example, a male passenger who is in 1st class and is 8 years old has a survival probability of $11/29 = 37.9\%$.

10.2 Bagging

When the relationship between a set of predictor variables and a *response variable* is linear, we can use methods like *multiple linear regression* to model the relationship between the variables.

However, when the relationship is more complex then we often need to rely on non-linear methods.

One such method is *classification and regression trees* (often abbreviated CART), which use a set of predictor variables to build decision trees that predict the value of a response variable.

However, the downside of CART models is that they tend to suffer from *high variance.* That is, if we split a dataset into two halves and apply a decision tree to both halves, the results could be quite different.

One method that we can use to reduce the variance of CART models is known as **bagging**, sometimes referred to as bootstrap aggregating.

What is Bagging ?

When we create a single decision tree, we only use one training dataset to build the model.

However, *bagging* uses the following method:

1. Take b bootstrapped samples from the original dataset.
- Recall that a bootstrapped sample is a sample of the original dataset in which the observations are taken with replacement.
2. Build a decision tree for each bootstrapped sample.
3. Average the predictions of each tree to come up with a final model.

- For regression trees, we take the average of the prediction made by the B trees.
- For classification trees, we take the most commonly occurring prediction made by the B trees.

Bagging can be used with any machine learning algorithm, but it's particularly useful for decision trees because they inherently have high variance and bagging is able to dramatically reduce the variance, which leads to lower test error.

To apply bagging to decision trees, we grow B individual trees deeply without pruning them. This results in individual trees that have high variance, but low bias. Then when we take the average predictions from these trees we're able to reduce the variance.

In practice, optimal performance typically occurs with 50 to 500 trees, but it's possible to fit thousands of trees to produce a final model.

Just keep in mind that fitting more trees will require more computational power, which may or may not be an issue depending on the size of the dataset.

Out-of-Bag Error Estimation

It turns out that we can calculate the test error of a bagged model without relying on k-fold cross-validation.

The reason is because it can be shown that each bootstrapped sample contains about 2/3 of the observations from the original dataset. The remaining 1/3 of the observations not used to fit the bagged tree are referred to as **out-of-bag (OOB) observations**.

We can predict the value for the ith observation in the original dataset by taking the average prediction from each of the trees in which that observation was OOB.

We can use this approach to make a prediction for all *n* observations in the original dataset and thus calculate an error rate, which is a valid estimate of the test error.

The benefit of using this approach to estimate the test error is that it's much quicker than k-fold cross-validation, especially when the dataset is large.

Understanding the Importance of Predictors

Recall that one of the benefits of decision trees is that they're easy to interpret and visualize.

When we instead use bagging, we're no longer able to interpret or visualize an individual tree since the final bagged model is the resulting of averaging many different trees. We gain prediction accuracy at the expense of interpretability.

However, we can still understand the importance of each predictor variable by calculating the total reduction in RSS (residual sum of squares) due to the split over a given predictor, averaged over all B trees. The larger the value, the more important the predictor.

Variable importance plot for bagging model Example of a variable importance plot.

Similarly, for classification models we can calculate the total reduction in the Gini Index due to the split over a given predictor, averaged over all B trees. The larger the value, the more important the predictor.

So, although we can't exactly interpret a final bagged model we can still get an idea of how important each predictor variable is when predicting the response.

Going Beyond Bagging

The benefit of bagging is that it typically offers an improvement in test error rate compared to a single decision tree.

The downside is that the predictions from the collection of bagged trees can be highly correlated if there happens to be a very strong predictor in the dataset.

In this case, most or all of the bagged trees will use this predictor for the first split, which will result in trees that are similar to each other and have highly correlated predictions.

One way to get around this issue is to instead use random forests, which use a similar method as bagging but are able to produce decorrelated trees, which often leads to lower test error rates.

10.2.1 Bagging Using R

Step 1: Load the Necessary Packages

```
library(dplyr)       #for data wrangling
library(e1071)       #for calculating variable importance
library(caret)       #for general model fitting
library(rpart)       #for fitting decision trees
library(ipred)       #for fitting bagged decision trees
```

Step 2: Fit the Bagged Model

For this example, we'll use a built-in R dataset called **airquality** which contains air quality measurements in New York on 153 individual days.

```
# View Structure of Airquality dataset
str(airquality)
```

```
## 'data.frame':    153 obs. of  6 variables:
##  $ Ozone  : int  41 36 12 18 NA 28 23 19 8 NA ...
##  $ Solar.R: int  190 118 149 313 NA NA 299 99 19 194 ...
##  $ Wind   : num  7.4 8 12.6 11.5 14.3 14.9 8.6 13.8 20.1 8.6 ...
##  $ Temp   : int  67 72 74 62 56 66 65 59 61 69 ...
##  $ Month  : int  5 5 5 5 5 5 5 5 5 5 ...
##  $ Day    : int  1 2 3 4 5 6 7 8 9 10 ...
```

The following code shows how to fit a bagged model in R using the **bagging()** function from the *ipred* library.

```
#make this example reproducible
set.seed(1)

#fit the bagged model
bag <- bagging(
  formula = Ozone ~ .,
  data = airquality,
  nbagg = 150,
  coob = TRUE,
  control = rpart.control(minsplit = 2, cp = 0)
)

#display fitted bagged model
bag
```

```
##
## Bagging regression trees with 150 bootstrap replications
##
## Call: bagging.data.frame(formula = Ozone ~ ., data = airquality, nbagg =
150,
##     coob = TRUE, control = rpart.control(minsplit = 2, cp = 0))
##
## Out-of-bag estimate of root mean squared error:  17.4973
```

Note that we chose to use **150** bootstrapped samples to build the bagged model and we specified **coob** to be **TRUE** to obtain the estimated out-of-bag error.

We also used the following specifications in the **rpart.control()** function:

- **minsplit = 2** : This tells the model to only require 2 observations in a node to split.
- **cp = 0** : This is the complexity parameter. By setting it to 0, we don't require the model to be able to improve the overall fit by any amount in order to perform a split.

Essentially these two arguments allow the individual trees to grow extremely deep, which leads to trees with high variance but low bias. Then when we apply bagging we're able to reduce the variance of the final model while keeping the bias low.

From the output of the model we can see that the out-of-bag estimated RMSE is **17.4973**. This is the average difference between the predicted value for Ozone and the actual observed value.

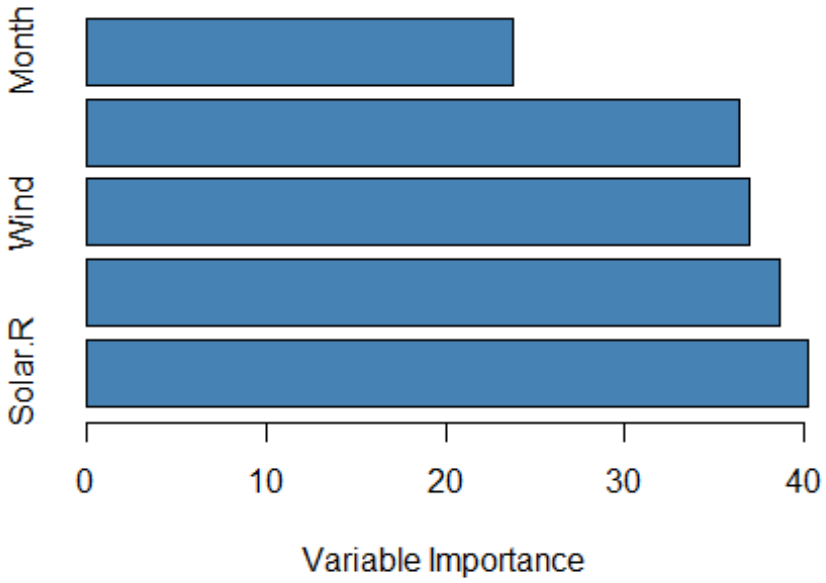
Step 3: Visualize the Importance of the Predictors

Although bagged models tend to provide more accurate predictions compared to individual decision trees, it's difficult to interpret and visualize the results of fitted bagged models.

We can, however, visualize the importance of the predictor variables by calculating the total reduction in RSS (residual sum of squares) due to the split over a given predictor, averaged over all of the trees. The larger the value, the more important the predictor.

The following code shows how to create a variable importance plot for the fitted bagged model, using the **varImp()** function from the **caret** library

```
#calculate variable importance
VI <- data.frame(var=names(airquality[,-1]), imp=varImp(bag))

#sort variable importance descending
VI_plot <- VI[order(VI$Overall, decreasing=TRUE),]

#visualize variable importance with horizontal bar plot
barplot(VI_plot$Overall,
        names.arg=rownames(VI_plot),
        horiz=TRUE,
        col='steelblue',
        xlab='Variable Importance')
```



We can see that *Solar.R* is the most importance predictor variable in the model while Month is the least important.

Step 4: Use the Model to Make Predictions

Lastly, we can use the fitted bagged model to make predictions on new observations.

```
#define new observation
new <- data.frame(Solar.R=150, Wind=8, Temp=70, Month=5, Day=5)

#use fitted bagged model to predict Ozone value of new observation
predict(bag, newdata=new)

## [1] 24.48667
```

Based on the values of the predictor variables, the fitted bagged model predicts that the Ozone value will be **24.487** on this particular day.

10.3 Random Forests

When the relationship between a set of predictor variables and a *response variable* is highly complex, we often use non-linear methods to model the relationship between them.

One such method is *classification and regression trees* (often abbreviated *CART*), which use a set of predictor variables to build decision trees that predict the value of a response variable.

The benefit of decision trees is that they're easy to interpret and visualize. The downsize is that they tend to suffer from *high variance*. That is, if we split a dataset into two halves and apply a decision tree to both halves, the results could be quite different.

One way to reduce the variance of decision trees is to use a method known as *bagging*, which works as follows :

1. Take *b* bootstrapped samples from the original dataset.
2. Build *a* decision tree for each bootstrapped sample.
3. Average the predictions of each tree to come up with a final model.

The benefit of this approach is that a bagged model typically offers an improvement in test error rate compared to a single decision tree.

The downside is that the predictions from the collection of bagged trees can be highly correlated if there happens to be a very strong predictor in the dataset. In this case, most or all of the bagged trees will use this predictor for the first split, which will result in trees that are similar to each other and have highly correlated predictions.

Thus, when we average the predictions of each tree to come up with a final bagged model, it's possible that this model doesn't actually reduce the variance by much compared to a single decision tree.

One way to get around this issue is to use a method known as random forests.

What Are Random Forests ?

Similar to bagging, random forests also take b bootstrapped samples from an original dataset.

However, when building a decision tree for each bootstrapped sample, each time a split in a tree is considered, only a random sample of m predictors is considered as split candidates from the full set of p predictors.

So, here's the full method that random forests use to build a model :

1. Take *b* bootstrapped samples from the original dataset.
2. Build *a* decision tree for each bootstrapped sample.
   - When building the tree, each time a split is considered, only a random sample of *m* predictors is considered as split candidates from the full set of *p* predictors.
3. Average the predictions of each tree to come up with a final model.

By using this method, the collection of trees in a random forest is **decorrelated** compared to the trees produced by bagging.

Thus, when we take the average predictions of each tree to come up with a final model it tends to have less variability and results in a lower test error rate compared to a bagged model.

When using random forests, we typically consider $m = \sqrt{p}$ predictors as split candidates each time we split a decision tree.

For example, if we have p = 16 total predictors in a dataset then we typically only consider m = $\sqrt{16}$ = 4 predictors as potential split candidates at each split.

Technical Note:

It's interesting to note that if we choose $m = p$ (i.e. we consider all predictors as split candidates at each split) then this is equivalent to simply using bagging.

Out-of-Bag Error Estimation

Similar to bagging, we can calculate the test error of a random forest model by using **out-of-bag estimation**.

It can be shown that each bootstrapped sample contains about 2/3 of the observations from the original dataset. The remaining 1/3 of the observations not used to fit the tree are referred to as ***out-of-bag (OOB) observations.**

We can predict the value for the i^{th} observation in the original dataset by taking the average prediction from each of the trees in which that observation was OOB.

We can use this approach to make a prediction for all n observations in the original dataset and thus calculate an error rate, which is a valid estimate of the test error.

The benefit of using this approach to estimate the test error is that it's much quicker than k-fold cross-validation, especially when the dataset is large.

The Pros & Cons of Random Forests

Random forests offer the following *benefits* :

- In most cases, random forests will offer an improvement in accuracy compared to bagged models and especially compared to single decision trees.
- Random forests are robust to outliers.
- No pre-processing is required to use random forests.

However, random forests come with the following potential *drawbacks* :

- They're difficult to interpret.
- They can be computationally intensive (i.e. slow) to build on large datasets.

In practice, data scientists typically use random forests to maximize predictive accuracy so the fact that they're not easily interpretable is usually not an issue.

10.3.1 Random Forest Using R

step 1 : Load the require library

```
library(randomForest)
```

Step 2: Fit the Random Forest Model

For this example, we'll use a built-in R dataset called **airquality** which contains air quality measurements in New York on 153 individual days

```
#view structure of airquality dataset
str(airquality)

#find number of rows with missing values
sum(!complete.cases(airquality))
```

This dataset has 42 rows with missing values, so before we fit a random forest model we'll fill in the missing values in each column with the column medians:

```
# replace NAs with column medians
for(i in 1:ncol(airquality)) {
  airquality[ , i][is.na(airquality[ , i])] <- median(airquality[ , i],
na.rm=TRUE)
}
```

The following code shows how to fit a random forest model in R using the **randomForest()** function from the randomForest package

```
# make this example reproducible
set.seed(1)

#fit the random forest model
model <- randomForest(
  formula = Ozone ~ .,
  data = airquality
)

#display fitted model
model

#find number of trees that produce lowest test MSE
which.min(model$mse)

#find RMSE of best model
sqrt(model$mse[which.min(model$mse)])
```

From the output we can see that the model that produced the lowest test mean squared error (MSE) used *82* trees.

We can also see that the root mean squared error of that model was *17.64*. We can think of this as the average difference between the predicted value for Ozone and the actual observed value.

We can also use the following code to produce a plot of the test MSE based on the number of trees used:

```
#plot the test MSE by number of trees
plot(model)
```

And we can use the **varImpPlot()** function to create a plot that displays the importance of each predictor variable in the final model :

```
#produce variable importance plot
varImpPlot(model)
```

The x-axis displays the average increase in node purity of the regression trees based on splitting on the various predictors displayed on the y-axis.

From the plot we can see that *Wind* is the most important predictor variable, followed closely by *Temp*.

Step 3: Tune the Model

By default, the **randomForest()** function uses 500 trees and (total predictors/3) randomly selected predictors as potential candidates at each split. We can adjust these parameters by using the **tuneRF()** function.

The following code shows how to find the optimal model by using the following specifications:

- **ntreeTry** : The number of trees to build.
- **mtryStart** : The starting number of predictor variables to consider at each split.
- **stepFactor** : The factor to increase by until the out-of-bag estimated error stops improving by a certain amount.
- **improve** : The amount that the out-of-bag error needs to improve by to keep increasing the step factor.

```
model_tuned <- tuneRF(
               x=airquality[,-1], #define predictor variables
               y=airquality$Ozone, #define response variable
               ntreeTry=500,
               mtryStart=4,
               stepFactor=1.5,
               improve=0.01,
               trace=FALSE #don't show real-time progress
               )
```

This function produces the plot, which displays the number of predictors used at each split when building the trees on the x-axis and the out-of-bag estimated error on the y-axis:

We can see that the lowest OOB error is achieved by using 2 randomly chosen predictors at each split when building the trees.

This actually matches the default parameter (total predictors/3 = 6/3 = 2) used by the initial **randomForest()** function.

Step 4: Use the Final Model to Make Predictions

Lastly, we can use the fitted random forest model to make predictions on new observations.

```
#define new observation
new <- data.frame(Solar.R=150, Wind=8, Temp=70, Month=5, Day=5)

#use fitted bagged model to predict Ozone value of new observation
predict(model, newdata=new)
```

Based on the values of the predictor variables, the fitted random forest model predicts that the Ozone value will be **27.19** on this particular day.

10.4 Boosting

Most *supervised machine learning algorithms* are based on using a single predictive model like *linear regression, logistic regression, ridge regression*, etc.

Methods like *bagging* and *random forests*, however, build many different models based on repeated bootstrapped samples of the original dataset. Predictions on new data are made by taking the average of the predictions made by the individual models.

These methods tend to offer an improvement in prediction accuracy over methods that only use a single predictive model because they use the following process:

- First, build individual models that have *high variance and low bias* (e.g. deeply grown *decision trees*).
- Next, take the average of the predictions made by individual models in order to reduce the variance.

Another method that tends to offer even further improvement in predictive accuracy is known as **boosting.**

What is Boosting ?

Boosting is a method that can be used with any type of model, but it is most often used with decision trees.

The idea behind boosting is simple:

1. First, build a weak model.

- A "weak" model is one whose error rate is only slightly better than random guessing.

- In practice, this is typically a decision tree with only one or two splits.

2. Next, build another weak model based on the residuals of the previous model.

- In practice, we use the residuals from the previous model (i.e. the errors in our predictions) to fit a new model that slightly improves upon the overall error rate.

3. Continue this process until k-fold cross-validation tells us to stop.

- In practice, we use k-fold cross-validation to identify when we should stop growing the boosted model. By using this method, we can start with a weak model and keep "boosting" the performance of it by sequentially building new trees that improve upon the performance of the previous tree until we end up with a final model that has high predictive accuracy.

Why Does Boosting Work ?

It turns out that boosting is able to produce some of the most powerful models in all of machine learning.

In many industries, boosted models are used as the go-to models in production because they tend to outperform all other models.

The reason boosted models work so well comes down to understanding a simple idea:

1. First, boosted models build a weak decision tree that has low predictive accuracy. This decision tree is said to have low variance and high bias.

2. As boosted models go through the process of sequentially improving previous decision trees, the overall model is able to slowly reduce the bias at each step without increasing the variance by much.

3. The final fitted model tends to have sufficiently low bias and low variance, which leads to a model that is able to produce low test error rates on new data.

Pros & Cons of Boosting

The obvious benefit of boosting is that it's able to produce models that have high predictive accuracy compared to almost all other types of models.

One potential drawback is that a fitted boosted model is very difficult to interpret. While it may offer tremendous ability to predict the response values of new data, it's difficult to explain the exact process that it uses to do so.

In practice, most data scientists and machine learning practitioners build boosted models because they want to be able to predict the response values of new data accurately. Thus, the fact that boosted models are hard to interpret usually isn't an issue.

Boosting in Practice

In practice there are actually many types of algorithms that are used for boosting, including:

- XGBoost
- AdaBoost
- CatBoost
- LightGBM

Depending on the size of your dataset and the processing power of your machine, one of these methods may be preferable to the other.

11 Unsupervised Learning

11.1 Principal Components Analysis (PCA)

Principal components analysis, often abbreviated **PCA**, is an *unsupervised machine* learning technique that seeks to find principal components – linear combinations of the original predictors – that explain a large portion of the variation in a dataset.

The goal of PCA is to explain most of the variability in a dataset with fewer variables than the original dataset.

For a given dataset with p variables, we could examine the scatterplots of each pairwise combination of variables, but the sheer number of scatterplots can become large very quickly.

For p predictors, there are $p(p-1)/2$ scatterplots.

So, for a dataset with $p = 15$ predictors, there would be *105* different scatterplots!

Fortunately, PCA offers a way to find a low-dimensional representation of a dataset that captures as much of the variation in the data as possible.

If we're able to capture most of the variation in just two dimensions, we could project all of the observations in the original dataset onto a simple scatterplot.

The way we find the principal components is as follows:

Given a dataset with p predictors : $X_1, X_2, \ldots, X_p$, calculate $Z_1, Z_2, \ldots, Z_M$ to be the M linear combinations of the original p predictors where :

- $Z_m = \sum \phi_{jm} X_j$ for some constants $\phi_{1m}, \phi_{2m}, .., \phi_{pm} \quad m = 1,2,..,M$
- Z_1 is the linear combination of the predictors that captures the most variance possible.
- Z_2 is the next linear combination of the predictors that captures the most variance while being *orthogonal* (i.e. uncorrelated) to Z_1.

- Z_3 is then the next linear combination of the predictors that captures the most variance while being orthogonal to Z_2.
- And so on.

In practice, we use the following steps to calculate the linear combinations of the original predictors:

1. Scale each of the variables to have a mean of 0 and a standard deviation of 1.

2. Calculate the covariance matrix for the scaled variables.

3. Calculate the eigenvalues of the covariance matrix.

Using linear algebra, it can be shown that the eigenvector that corresponds to the largest eigenvalue is the first principal component. In other words, this particular combination of the predictors explains the most variance in the data.

The eigenvector corresponding to the second largest eigenvalue is the second principal component, and so on.

11.1.1 PCA Using R

Principal Components Analysis

Step 1 : Load the Data

```
library(tidyverse)

data("USArrests")

USArrests %>% head()
```

```
##            Murder Assault UrbanPop Rape
## Alabama      13.2     236       58 21.2
## Alaska       10.0     263       48 44.5
## Arizona       8.1     294       80 31.0
## Arkansas      8.8     190       50 19.5
## California    9.0     276       91 40.6
## Colorado      7.9     204       78 38.7
```

Step 2: Calculate the Principal Components

After loading the data, we can use the R built-in function **prcomp()** to calculate the *principal components* of the dataset.

Be sure to specify **scale = TRUE** so that each of the variables in the dataset are scaled to have a mean of 0 and a standard deviation of 1 before calculating the principal components.

Also note that eigenvectors in R point in the negative direction by default, so we'll multiply by -1 to reverse the signs.

```
#calculate principal components
results <- prcomp(USArrests, scale = TRUE)

#reverse the signs
results$rotation <- -1*results$rotation

#display principal components
results$rotation
```

```
##                PC1        PC2        PC3         PC4
## Murder   0.5358995 -0.4181809  0.3412327 -0.64922780
## Assault  0.5831836 -0.1879856  0.2681484  0.74340748
## UrbanPop 0.2781909  0.8728062  0.3780158 -0.13387773
## Rape     0.5434321  0.1673186 -0.8177779 -0.08902432
```

We can see that the first principal component (PC1) has high values for Murder, Assault, and Rape which indicates that this principal component describes the most variation in these variables.

We can also see that the second principal component (PC2) has a high value for UrbanPop, which indicates that this principle component places most of its emphasis on urban population.

Note that the principal components scores for each state are stored in results$x. We will also multiply these scores by -1 to reverse the signs:

```
#reverse the signs of the scores
results$x <- -1*results$x

#display the first six scores
head(results$x)
```

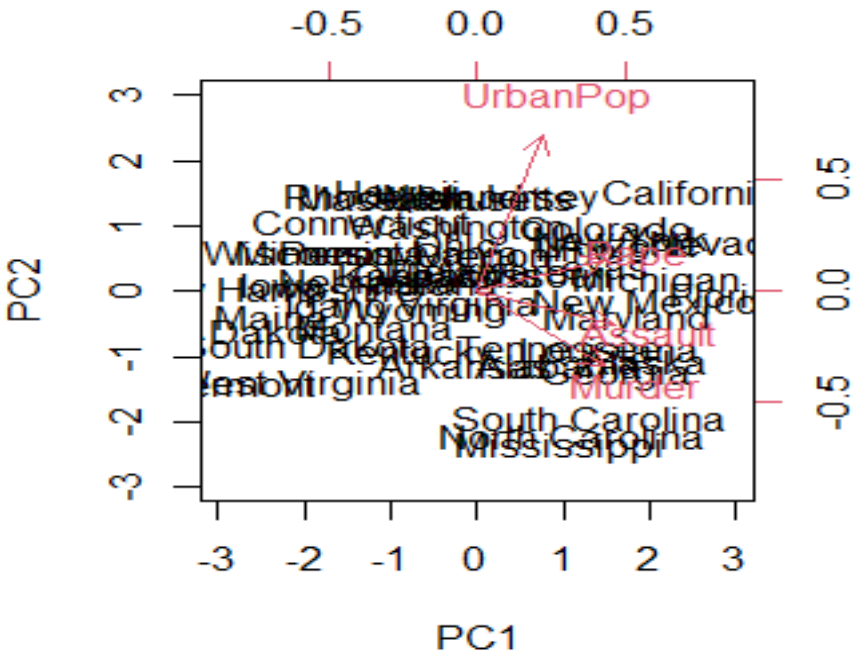
```
##                 PC1        PC2         PC3          PC4
## Alabama     0.9756604 -1.1220012  0.43980366 -0.154696581
## Alaska      1.9305379 -1.0624269 -2.01950027  0.434175454
## Arizona     1.7454429  0.7384595 -0.05423025  0.826264240
## Arkansas   -0.1399989 -1.1085423 -0.11342217  0.180973554
## California  2.4986128  1.5274267 -0.59254100  0.338559240
## Colorado    1.4993407  0.9776297 -1.08400162 -0.001450164
```

Step 3: Visualize the Results with a Biplot

Next, we can create a **biplot** – a plot that projects each of the observations in the dataset onto a scatterplot that uses the first and second principal components as the axes:

Note that **scale = 0** ensures that the arrows in the plot are scaled to represent the loadings.

```
biplot(results, scale = 0)
```



From the plot we can see each of the 50 states represented in a simple two-dimensional space.

The states that are close to each other on the plot have similar data patterns in regards to the variables in the original dataset.

We can also see that the certain states are more highly associated with certain crimes than others. For example, Georgia is the state closest to the variable Murder in the plot.

If we take a look at the states with the highest murder rates in the original dataset, we can see that Georgia is actually at the top of the list:

```
#display states with highest murder rates in original dataset
head(USArrests[order(-USArrests$Murder),])
```

```
##               Murder Assault UrbanPop Rape
## Georgia         17.4     211       60 25.8
## Mississippi     16.1     259       44 17.1
## Florida         15.4     335       80 31.9
## Louisiana       15.4     249       66 22.2
## South Carolina  14.4     279       48 22.5
## Alabama         13.2     236       58 21.2
```

Step 4: Find Variance Explained by Each Principal Component

We can use the following code to calculate the total variance in the original dataset explained by each principal component :

```
#calculate total variance explained by each principal component
results$sdev^2 / sum(results$sdev^2)

## [1] 0.62006039 0.24744129 0.08914080 0.04335752
```

From the results we can observe the following :

- The first principal component explains 62% of the total variance in the dataset.
- The second principal component explains 24.7% of the total variance in the dataset.
- The third principal component explains 8.9% of the total variance in the dataset.
- The fourth principal component explains 4.3% of the total variance in the dataset.

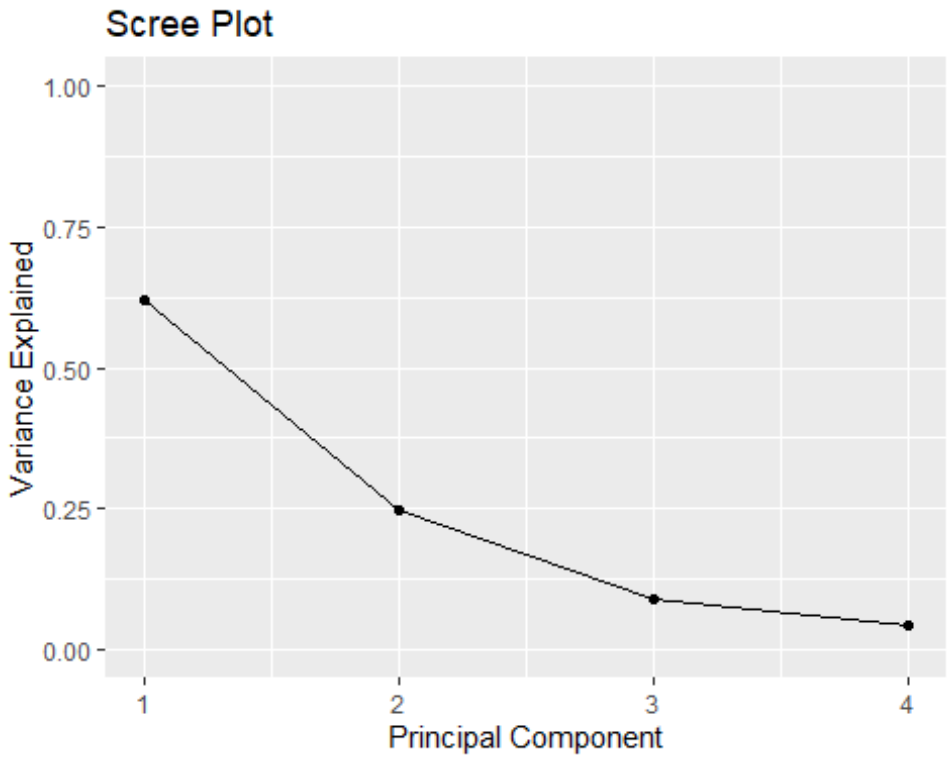
Thus, the first two principal components explain a majority of the total variance in the data.

This is a good sign because the previous biplot projected each of the observations from the original data onto a scatterplot that only took into account the first two principal components.

Thus, it's valid to look at patterns in the biplot to identify states that are similar to each other.

We can also create a **scree plot** – a plot that displays the total variance explained by each principal component – to visualize the results of PCA :

```
#calculate total variance explained by each principal component
var_explained = results$sdev^2 / sum(results$sdev^2)

#create scree plot
qplot(c(1:4), var_explained) +
  geom_line() +
  xlab("Principal Component") +
  ylab("Variance Explained") +
  ggtitle("Scree Plot") +
  ylim(0, 1)
```



Principal Components Analysis in Practice

In practice, PCA is used most often for two reasons:

1. Exploratory Data Analysis (EDA) – We use PCA when we're first exploring a dataset and we want to understand which observations in the data are most similar to each other.

2. Principal Components Regression (PCR) – We can also use PCA to calculate principal components that can then be used in principal components regression. This type of regression is often used when multicollinearity exists between predictors in a dataset

11.2 K-Mean Clustering

Clustering is a technique in machine learning that attempts to find clusters of *observations* within a dataset.

The goal is to find clusters such that the observations within each cluster are quite similar to each other, while observations in different clusters are quite different from each other.

Clustering is a form of *unsupervised learning* because we're simply attempting to find structure within a dataset rather than predicting the value of some *response variable*.

Clustering is often used in marketing when companies have access to information like:

- Household income
- Household size
- Head of household Occupation

- Distance from nearest urban area

When this information is available, clustering can be used to identify households that are similar and may be more likely to purchase certain products or respond better to a certain type of advertising.

One of the most common forms of clustering is known as **k-means clustering**.

What is K-Means Clustering ?

K-means clustering is a technique in which we place each observation in a dataset into one of K clusters.

The end goal is to have K clusters in which the observations within each cluster are quite similar to each other while the observations in different clusters are quite different from each other.

In practice, we use the following steps to perform K-means clustering:

1. Choose a value for K.

- First, we must decide how many clusters we'd like to identify in the data. Often we have to simply test several different values for K and analyze the results to see which number of clusters seems to make the most sense for a given problem.

2. Randomly assign each observation to an initial cluster, from 1 to K.

3. Perform the following procedure until the cluster assignments stop changing.

- For each of the *K* clusters, compute the cluster centroid. This is simply the vector of the *p* feature means for the observations in the kth cluster.
- Assign each observation to the cluster whose centroid is closest. Here, closest is defined using *Euclidean distance*.

11.2.1 K-Means Clustering Using R

Step 1: Load the Necessary Packages First, we'll load two packages that contain several useful functions for k-means clustering in R.

```
library(factoextra)
library(cluster)
```

Step 2: Load and Prep the Data For this example we'll use the *USArrests* dataset built into R, which contains the number of arrests per 100,000 residents in each U.S. state in 1973 for *Murder, Assault*, and *Rape* along with the percentage of the population in each state living in urban areas, *UrbanPop*.

The following code shows how to do the following:

- Load the USArrests dataset
- Remove any rows with missing values

- Scale each variable in the dataset to have a mean of 0 and a standard deviation of 1

```
#load data
df <- USArrests

#remove rows with missing values
df <- na.omit(df)

#scale each variable to have a mean of 0 and sd of 1
df <- scale(df)

#view first six rows of dataset
head(df)
```

```
##                Murder   Assault   UrbanPop         Rape
## Alabama    1.24256408 0.7828393 -0.5209066 -0.003416473
## Alaska     0.50786248 1.1068225 -1.2117642  2.484202941
## Arizona    0.07163341 1.4788032  0.9989801  1.042878388
## Arkansas   0.23234938 0.2308680 -1.0735927 -0.184916602
## California 0.27826823 1.2628144  1.7589234  2.067820292
## Colorado   0.02571456 0.3988593  0.8608085  1.864967207
```

Step 3: Find the Optimal Number of Clusters To perform k-means clustering in R we can use the built-in **kmeans()** function, which uses the following syntax: **kmeans(data, centers, nstart)**

where:

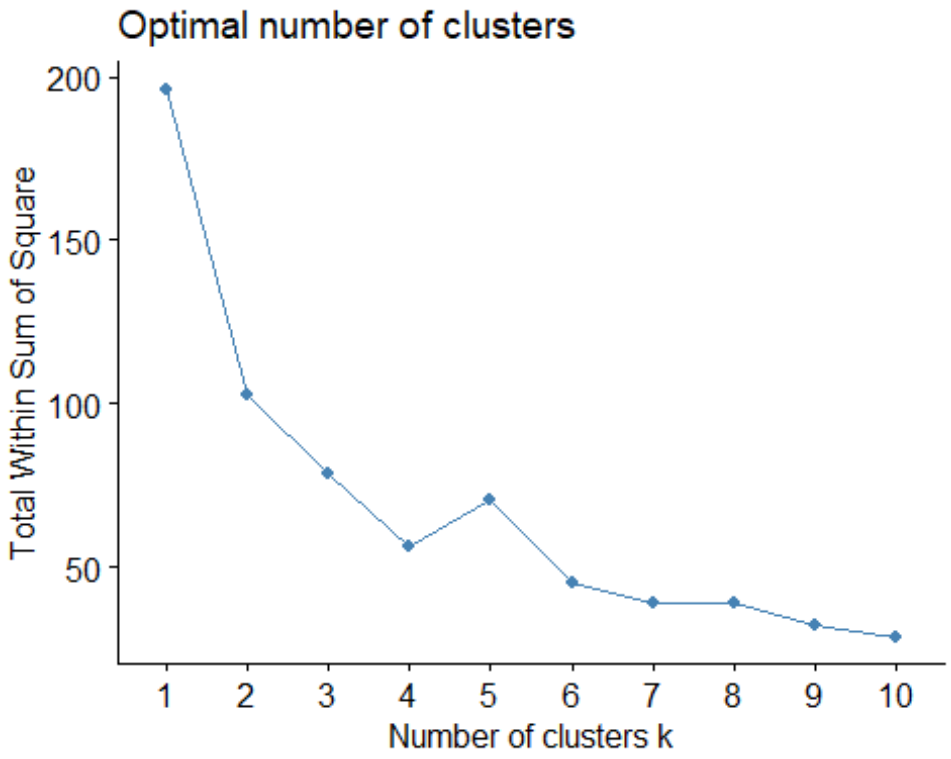
- **data** : Name of the dataset.
- **centers** : The number of clusters, denoted k.
- **nstart** : The number of initial configurations. Because it's possible that different initial starting clusters can lead to different results, it's recommended to use several different initial configurations. The k-means algorithm will find the initial configurations that lead to the smallest within-cluster variation.

Since we don't know beforehand how many clusters is optimal, we'll create two different plots that can help us decide:

1. Number of Clusters vs. the Total Within Sum of Squares

First, we'll use the **fviz_nbclust()** function to create a plot of the number of clusters vs. the total within sum of squares :

```
fviz_nbclust(df, kmeans, method = "wss")
```



Typically when we create this type of plot we look for an "elbow" where the sum of squares begins to "bend" or level off. This is typically the optimal number of clusters.

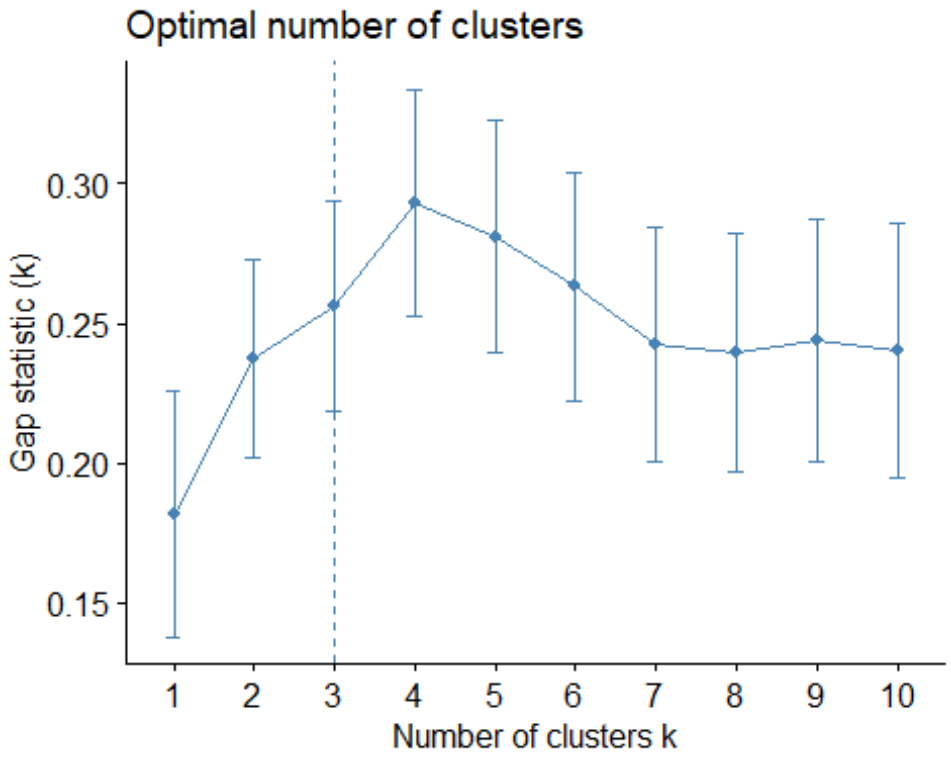
For this plot it appear that there is a bit of an elbow or "bend" at k = 4 clusters.

2. Number of Clusters vs. Gap Statistic

Another way to determine the optimal number of clusters is to use a metric known as the gap statistic, which compares the total intra-cluster variation for different values of k with their expected values for a distribution with no clustering.

We can calculate the gap statistic for each number of clusters using the **clusGap()** function from the cluster package along with a plot of clusters vs. gap statistic using the **fviz_gap_stat()** function :

```
#calculate gap statistic based on number of clusters
gap_stat <- clusGap(df,
                    FUN = kmeans,
                    nstart = 25,
                    K.max = 10,
                    B = 50)

## Clustering k = 1,2,..., K.max (= 10): .. done
## Bootstrapping, b = 1,2,..., B (= 50)  [one "." per sample]:
## ................................................. 50

#plot number of clusters vs. gap statistic
fviz_gap_stat(gap_stat)
```



From the plot we can see that gap statistic is highest at k = 4 clusters, which matches the elbow method we used earlier.

Step 4: Perform K-Means Clustering with Optimal K

Lastly, we can perform k-means clustering on the dataset using the optimal value for k of 4:

```
#make this example reproducible
set.seed(1)

#perform k-means clustering with k = 4 clusters
km <- kmeans(df, centers = 4, nstart = 25)

#view results
km

## K-means clustering with 4 clusters of sizes 13, 13, 16, 8
##
## Cluster means:
##       Murder    Assault   UrbanPop        Rape
## 1 -0.9615407 -1.1066010 -0.9301069 -0.96676331
## 2  0.6950701  1.0394414  0.7226370  1.27693964
## 3 -0.4894375 -0.3826001  0.5758298 -0.26165379
## 4  1.4118898  0.8743346 -0.8145211  0.01927104
##
## Clustering vector:
##        Alabama         Alaska        Arizona       Arkansas     California
```

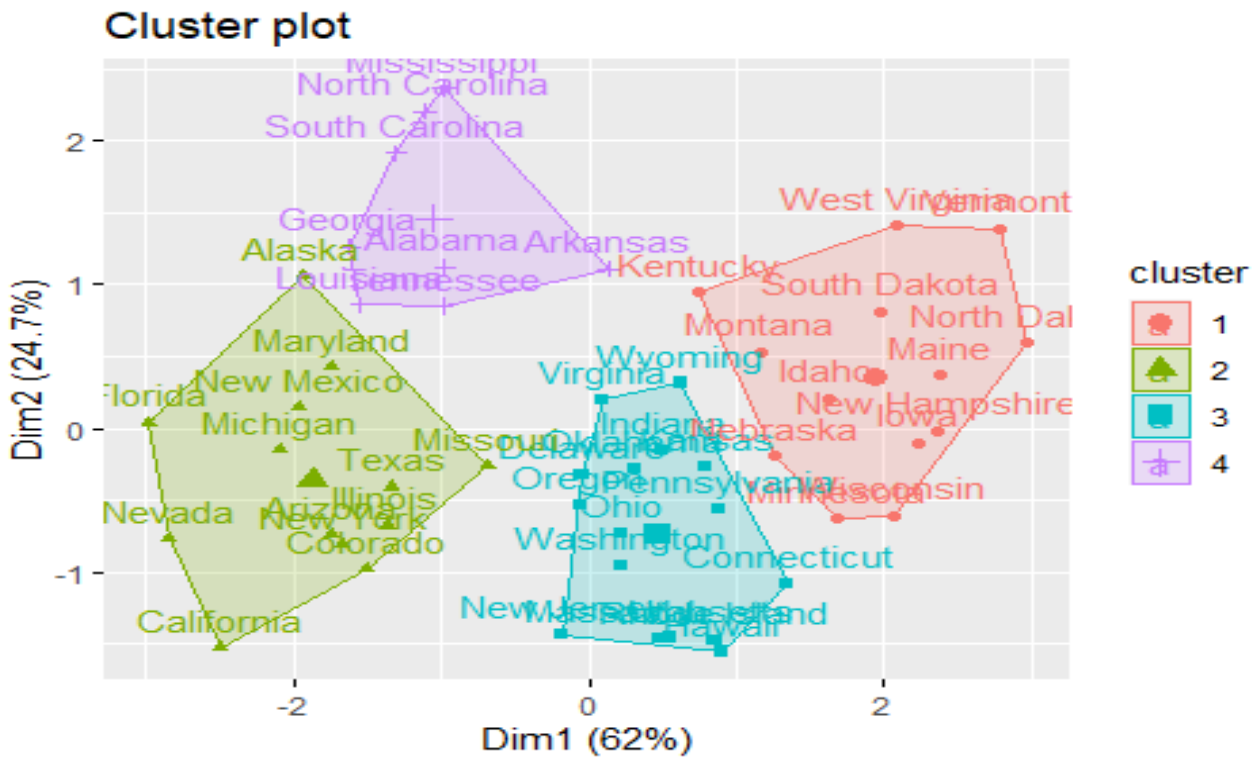
```
Colorado    Connecticut       Delaware
##              4              2              2              4              2
2              3              3
##        Florida        Georgia         Hawaii          Idaho       Illinois
Indiana           Iowa         Kansas
##              2              4              3              1              2
3              1              3
##       Kentucky      Louisiana          Maine       Maryland  Massachusetts
Michigan      Minnesota    Mississippi
##              1              4              1              2              3
2              1              4
##       Missouri        Montana       Nebraska         Nevada  New Hampshire
New Jersey     New Mexico       New York
##              2              1              1              2              1
3              2              2
## North Carolina   North Dakota           Ohio       Oklahoma         Oregon
Pennsylvania   Rhode Island South Carolina
##              4              1              3              3              3
3              3              4
##   South Dakota      Tennessee          Texas           Utah        Vermont
Virginia     Washington  West Virginia
##              1              4              2              3              1
3              3              1
##      Wisconsin        Wyoming
##              1              3
##
## Within cluster sum of squares by cluster:
## [1] 11.952463 19.922437 16.212213  8.316061
##  (between_SS / total_SS =  71.2 %)
##
## Available components:
##
## [1] "cluster"      "centers"      "totss"        "withinss"
"tot.withinss" "betweenss"    "size"         "iter"
## [9] "ifault"
```

From the results we can see that:

- **12** states were assigned to the first cluster
- **20** states were assigned to the second cluster
- **19** states were assigned to the third cluster
- **8** states were assigned to the fourth cluster

We can visualize the clusters on a scatterplot that displays the first two principal components on the axes using the **fivz_cluster()** function:

```
#plot results of final k-means model
fviz_cluster(km, data = df)
```



We can also use the **aggregate()** function to find the mean of the variables in each cluster:

```
#find means of each cluster
aggregate(USArrests, by=list(cluster=km$cluster), mean)

##   cluster   Murder   Assault UrbanPop     Rape
## 1       1  3.60000  78.53846 52.07692 12.17692
## 2       2 10.81538 257.38462 76.00000 33.19231
## 3       3  5.65625 138.87500 73.87500 18.78125
## 4       4 13.93750 243.62500 53.75000 21.41250
```

We interpret this output is as follows:

- The mean number of murders per 100,000 citizens among the states in cluster 1 is 3.6.
- The mean number of assaults per 100,000 citizens among the states in cluster 1 is 78.5.
- The mean percentage of residents living in an urban area among the states in cluster 1 is 52.1%.
- The mean number of rapes per 100,000 citizens among the states in cluster 1 is 12.2.
- And so on.

We can also append the cluster assignments of each state back to the original dataset:

```
#add cluster assigment to original data
final_data <- cbind(USArrests, cluster = km$cluster)
```

```
#view final data
head(final_data)
```

```
##            Murder Assault UrbanPop Rape cluster
## Alabama      13.2     236       58 21.2       4
## Alaska       10.0     263       48 44.5       2
## Arizona       8.1     294       80 31.0       2
## Arkansas      8.8     190       50 19.5       4
## California    9.0     276       91 40.6       2
## Colorado      7.9     204       78 38.7       2
```

Pros & Cons of K-Means Clustering

K-means clustering offers the following **benefits**:

- It is a fast algorithm.
- It can handle large datasets well.

However, it comes with the following potential **drawbacks**:

- It requires us to specify the number of clusters before performing the algorithm.
- It's sensitive to outliers.

Two alternatives to k-means clustering are **k-medoids clustering** and **hierarchical clustering.**

11.3 K-Medoids Clustering

Clustering is a technique in machine learning that attempts to find groups or clusters of observations within a dataset.

The goal is to find clusters such that the observations within each cluster are quite similar to each other, while observations in different clusters are quite different from each other.

Clustering is a form of *unsupervised learning* because we're simply attempting to find structure within a dataset rather than predicting the value of some *response variable*.

Clustering is often used in marketing when companies have access to information like:

- Household income
- Household size
- Head of household Occupation
- Distance from nearest urban area

When this information is available, clustering can be used to identify households that are similar and may be more likely to purchase certain products or respond better to a certain type of advertising.

One of the most common forms of clustering is known as *k-means clustering*.

Unfortunately, this method can be influenced by outliers so an alternative that is often used is k-medoids clustering.

What is K-Medoids Clustering ?

K-medoids clustering is a technique in which we place each observation in a dataset into one of K clusters.

The end goal is to have K clusters in which the observations within each cluster are quite similar to each other while the observations in different clusters are quite different from each other.

In practice, we use the following steps to perform K-means clustering:

1. Choose a value for K

First, we must decide how many clusters we'd like to identify in the data. Often we have to simply test several different values for K and analyze the results to see which number of clusters seems to make the most sense for a given problem.

2. Randomly assign each observation to an initial cluster, from 1 to K

3. **Perform the following procedure until the cluster assignments stop changing**

- For each of the K clusters, compute the cluster centroid. This is the vector of the p feature medians for the observations in the kth cluster.
- Assign each observation to the cluster whose centroid is closest. Here, closest is defined using Euclidean distance.

Technical Note :

- Because k-medoids computes cluster centroids using medians instead of means, it tends to be more robust to outliers compared to k-means.
- In practice, if there are no extreme outliers in the dataset then k-means and k-medoids will produce similar results.

11.3.1 K-Medoids Clustering Using R

Step 1 : Load the require packages

```
library(factoextra)
library(cluster)
```

Step : 2 Load and Prepare the Data We'll use the **USArrests** dataset built into R, which contains the number of arrests per 100,000 residents in each U.S. state in 1973 for *Murder, Assault* and *Rape* along with the percentage of the population in each state living in urban areas, *UrbanPop*.

The following code shows how to do the following:

- Load the USArrests dataset

- Remove any rows with missing values
- Scale each variable in the dataset to have a mean of 0 and a standard deviation of 1

```
#load data
df <- USArrests

#remove rows with missing values
df <- na.omit(df)

#scale each variable to have a mean of 0 and sd of 1
df <- scale(df)

#view first six rows of dataset
head(df)
```

```
##                Murder   Assault   UrbanPop         Rape
## Alabama    1.24256408 0.7828393 -0.5209066 -0.003416473
## Alaska     0.50786248 1.1068225 -1.2117642  2.484202941
## Arizona    0.07163341 1.4788032  0.9989801  1.042878388
## Arkansas   0.23234938 0.2308680 -1.0735927 -0.184916602
## California 0.27826823 1.2628144  1.7589234  2.067820292
## Colorado   0.02571456 0.3988593  0.8608085  1.864967207
```

Step 3 : Find the Optimal Number of Clusters

To perform k-medoids clustering in R we can use the **pam()** function, which stands for "partitioning around medians" and uses the following syntax:

pam(data, k, metric = "euclidean", stand = FALSE)

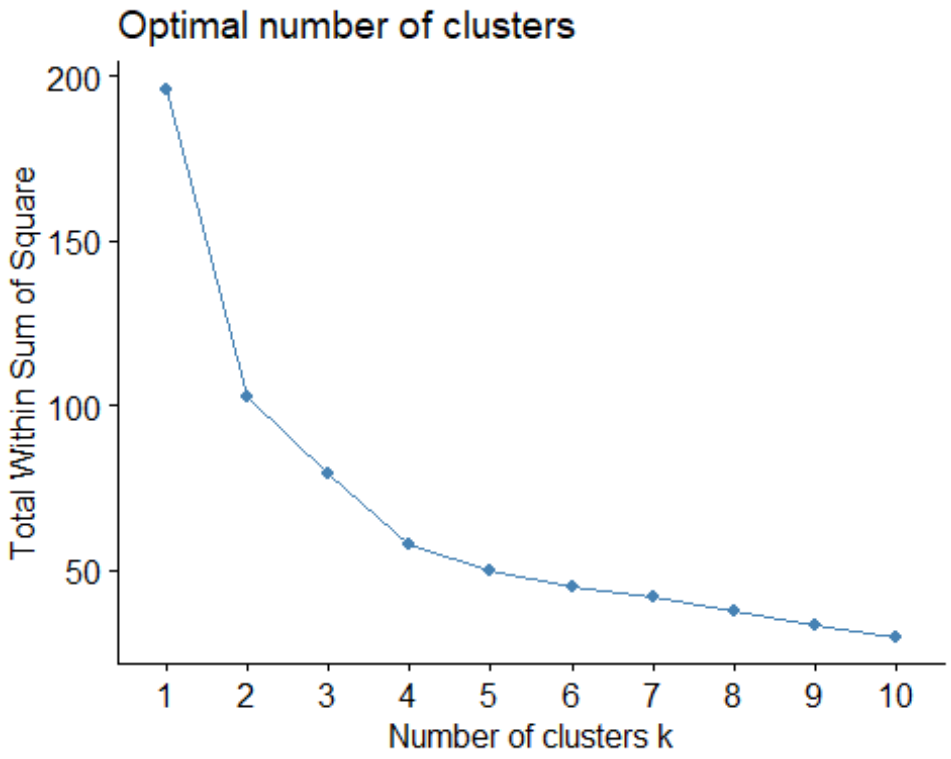
where, **data** : Name of the dataset. **k** : The number of clusters. **metric** : The metric to use to calculate distance. Default is *euclidean* but you could also specify *manhattan*.
stand : Whether or not to standardize each variable in the dataset. Default is FALSE.

Since we don't know beforehand how many clusters is optimal, we'll create two different plots that can help us decide:

1. Number of Clusters vs. the Total Within Sum of Squares

First, we'll use the **fviz_nbclust()** function to create a plot of the number of clusters vs. the total within sum of squares :

```
fviz_nbclust(df, pam, method = "wss")
```



The total within sum of squares will typically always increase as we increase the number of clusters, so when we create this type of plot we look for an "elbow" where the sum of squares begins to "bend" or level off.

The point where the plot bends is typically the optimal number of clusters. Beyond this number, *overfitting* is likely to occur.

For this plot it appear that there is a bit of an elbow or "bend" at k = 4 clusters.

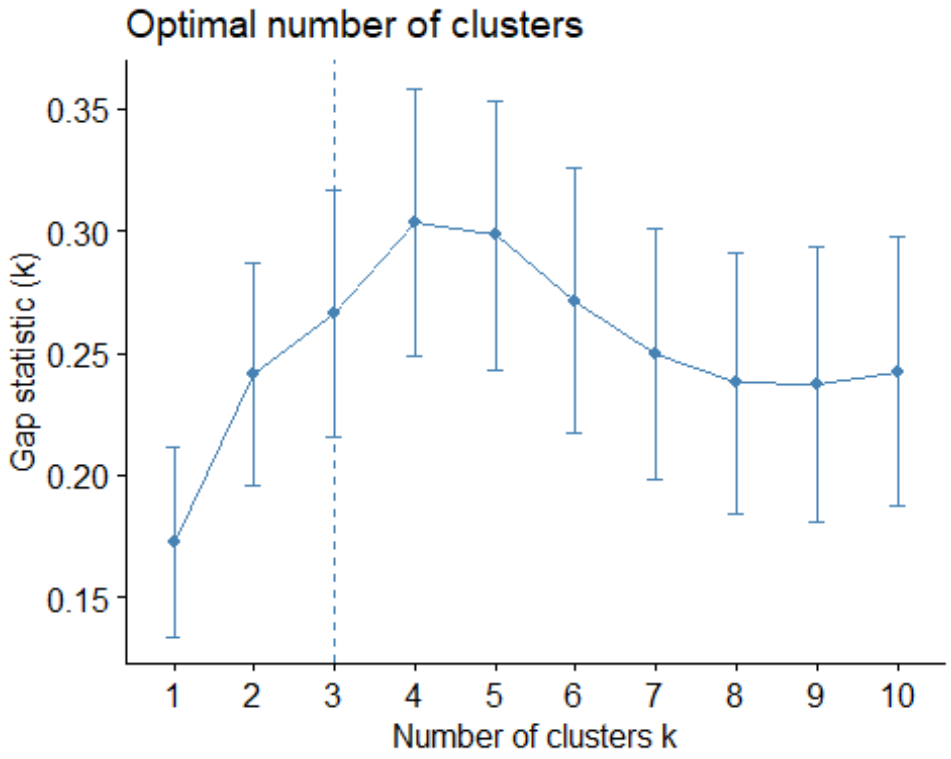
2. Number of Clusters vs. Gap Statistic

Another way to determine the optimal number of clusters is to use a metric known as the gap statistic, which compares the total intra-cluster variation for different values of k with their expected values for a distribution with no clustering.

We can calculate the gap statistic for each number of clusters using the **clusGap()** function from the cluster package along with a plot of clusters vs. gap statistic using the **fviz_gap_stat()** function :

```
#calculate gap statistic based on number of clusters
gap_stat <- clusGap(df,
                    FUN = pam,
                    K.max = 10, #max clusters to consider
                    B = 50) #total bootstrapped iterations
```

```
## Clustering k = 1,2,..., K.max (= 10): .. done
## Bootstrapping, b = 1,2,..., B (= 50)  [one "." per sample]:
## ................................................. 50

#plot number of clusters vs. gap statistic
fviz_gap_stat(gap_stat)
```



From the plot we can see that gap statistic is highest at k = 4 clusters, which matches the elbow method we used earlier

Step 4 : Perform K-Medoids Clustering with Optimal K Lastly, we can perform k-medoids clustering on the dataset using the optimal value for k of 4 :

```
#make this example reproducible
set.seed(1)

#perform k-medoids clustering with k = 4 clusters
kmed <- pam(df, k = 4)

#view results
kmed

## Medoids:
##                ID     Murder    Assault   UrbanPop         Rape
## Alabama         1  1.2425641  0.7828393 -0.5209066 -0.003416473
## Michigan       22  0.9900104  1.0108275  0.5844655  1.480613993
## Oklahoma       36 -0.2727580 -0.2371077  0.1699510 -0.131534211
```

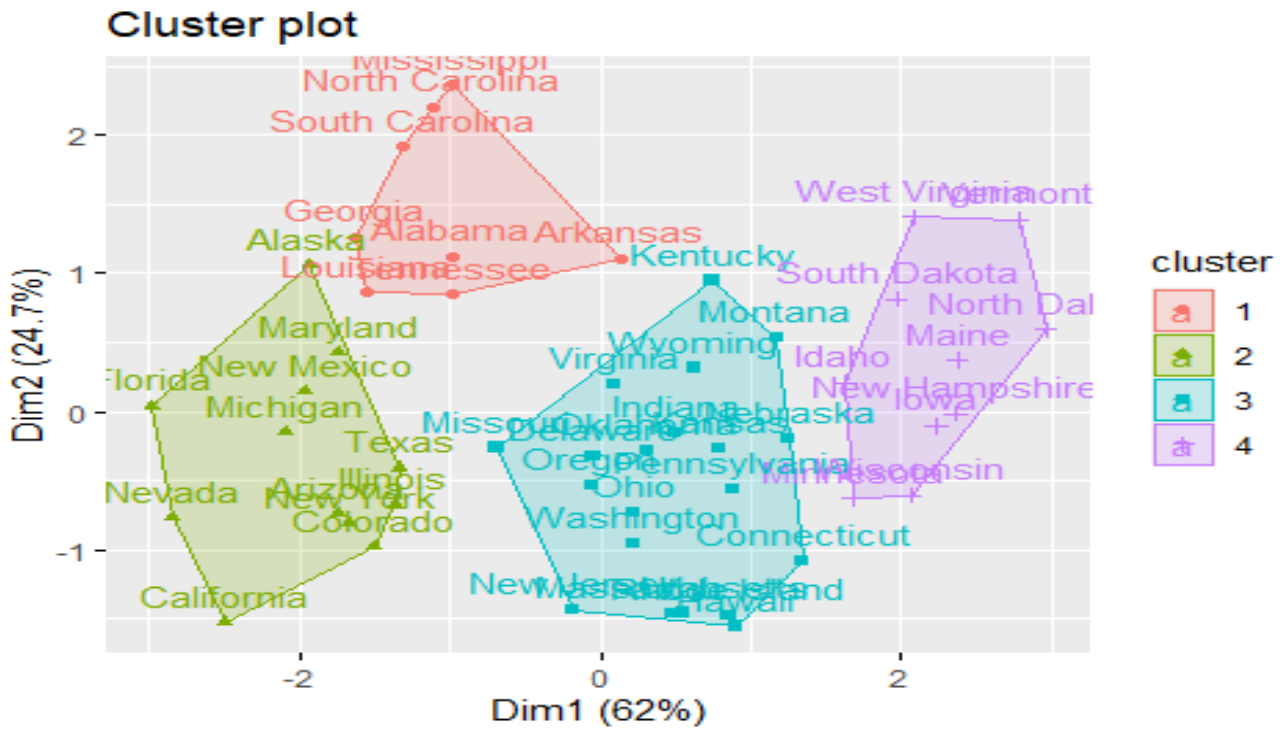
```
## New Hampshire 29 -1.3059321 -1.3650491 -0.6590781 -1.252564419
## Clustering vector:
##        Alabama         Alaska        Arizona       Arkansas     California
Colorado    Connecticut       Delaware
##              1              2              2              1              2
2              3              3
##        Florida        Georgia         Hawaii          Idaho       Illinois
Indiana           Iowa         Kansas
##              2              1              3              4              2
3              4              3
##       Kentucky      Louisiana          Maine       Maryland  Massachusetts
Michigan      Minnesota    Mississippi
##              3              1              4              2              3
2              4              1
##       Missouri        Montana       Nebraska         Nevada  New Hampshire
New Jersey     New Mexico       New York
##              3              3              3              2              4
3              2              2
## North Carolina   North Dakota           Ohio       Oklahoma         Oregon
Pennsylvania   Rhode Island South Carolina
##              1              4              3              3              3
3              3              1
##   South Dakota      Tennessee          Texas           Utah        Vermont
Virginia     Washington  West Virginia
##              4              1              2              3              4
3              3              4
##      Wisconsin        Wyoming
##              4              3
## Objective function:
##    build     swap
## 1.035116 1.027102
##
## Available components:
##  [1] "medoids"    "id.med"     "clustering" "objective"  "isolation"
"clusinfo"   "silinfo"    "diss"       "call"
## [10] "data"
```

Note that the four cluster centroids are actual observations in the dataset. Near the top of the output we can see that the four centroids are the following states :

- Alabama
- Michigan
- Oklahoma
- New Hampshire

We can visualize the clusters on a scatterplot that displays the first two principal components on the axes using the **fivz_cluster()** function :

```
#plot results of final k-medoids model
fviz_cluster(kmed, data = df)
```



We can also append the cluster assignments of each state back to the original dataset:

```
#add cluster assignment to original data
final_data <- cbind(USArrests, cluster = kmed$cluster)

#view final data
head(final_data)
```

```
##            Murder Assault UrbanPop Rape cluster
## Alabama      13.2     236       58 21.2       1
## Alaska       10.0     263       48 44.5       2
## Arizona       8.1     294       80 31.0       2
## Arkansas      8.8     190       50 19.5       1
## California    9.0     276       91 40.6       2
## Colorado      7.9     204       78 38.7       2
```

11.4 Hierarchical Clustering

Clustering is a technique in machine learning that attempts to find groups or clusters of observations within a dataset such that the observations within each cluster are quite similar to each other, while observations in different clusters are quite different from each other.

Clustering is a form of *unsupervised learning* because we're simply attempting to find structure within a dataset rather than predicting the value of some response variable.

Clustering is often used in marketing when companies have access to information like :

- Household income
- Household size
- Head of household Occupation
- Distance from nearest urban area

When this information is available, clustering can be used to identify households that are similar and may be more likely to purchase certain products or respond better to a certain type of advertising.

One of the most common forms of clustering is known as *k-means clustering*. Unfortunately this method requires us to pre-specify the number of clusters K.

An alternative to this method is known as hierarchical clustering, which does not require us to pre-specify the number of clusters to be used and is also able to produce a tree-based representation of the observations known as a dendrogram.

What is Hierarchical Clustering ?

Similar to k-means clustering, the goal of hierarchical clustering is to produce clusters of observations that are quite similar to each other while the observations in different clusters are quite different from each other.

In practice, we use the following steps to perform hierarchical clustering:

1. Calculate the pairwise dissimilarity between each observation in the dataset.

- First, we must choose some distance metric – like the Euclidean distance – and use this metric to compute the dissimilarity between each observation in the dataset.
- For a dataset with n observations, there will be a total of $n(n-1)/2$ pairwise dissimilarities.

2. Fuse observations into clusters

- At each step in the algorithm, fuse together the two observations that are most similar into a single cluster.
- Repeat this procedure until all observations are members of one large cluster. The end result is a tree, which can be plotted as a dendrogram.

To determine how close together two clusters are, we can use a few different methods including:

- **Complete linkage clustering** : Find the max distance between points belonging to two different clusters.
- **Single linkage clustering** : Find the minimum distance between points belonging to two different clusters.
- **Mean linkage clustering** : Find all pairwise distances between points belonging to two different clusters and then calculate the average.

- **Centroid linkage clustering** : Find the centroid of each cluster and calculate the distance between the centroids of two different clusters.
- **Ward's minimum variance method** : Minimize the total Depending on the structure of the dataset, one of these methods may tend to produce better (i.e. more compact) clusters than the other methods.

11.4.1 Hierarchical Clustering Using R

Step 1 : Load Nesessary Package

```
library(factoextra)
library(cluster)
```

**Step 2 :* Load and Prep the Data

We'll use the **USArrests** dataset built into R, which contains the number of arrests per 100,000 residents in each U.S. state in 1973 for *Murder, Assault*, and *Rape* along with the percentage of the population in each state living in urban areas, *UrbanPop*.

The following code shows how to do the following :

- Load the USArrests dataset
- Remove any rows with missing values
- Scale each variable in the dataset to have a mean of 0 and a standard deviation of 1

```
#load data
df <- USArrests

#remove rows with missing values
df <- na.omit(df)

#scale each variable to have a mean of 0 and sd of 1
df <- scale(df)

#view first six rows of dataset
head(df , 3)
```

```
##             Murder   Assault   UrbanPop         Rape
## Alabama 1.24256408 0.7828393 -0.5209066 -0.003416473
## Alaska  0.50786248 1.1068225 -1.2117642  2.484202941
## Arizona 0.07163341 1.4788032  0.9989801  1.042878388
```

Step 3: Find the Linkage Method to Use

To perform hierarchical clustering in R we can use the **agnes()** function from the cluster package, which uses the following syntax :

agnes(data, method)

where, **data** : Name of the dataset. **method** : The method to use to calculate dissimilarity between clusters.

Since we don't know beforehand which method will produce the best clusters, we can write a short function to perform hierarchical clustering using several different methods.

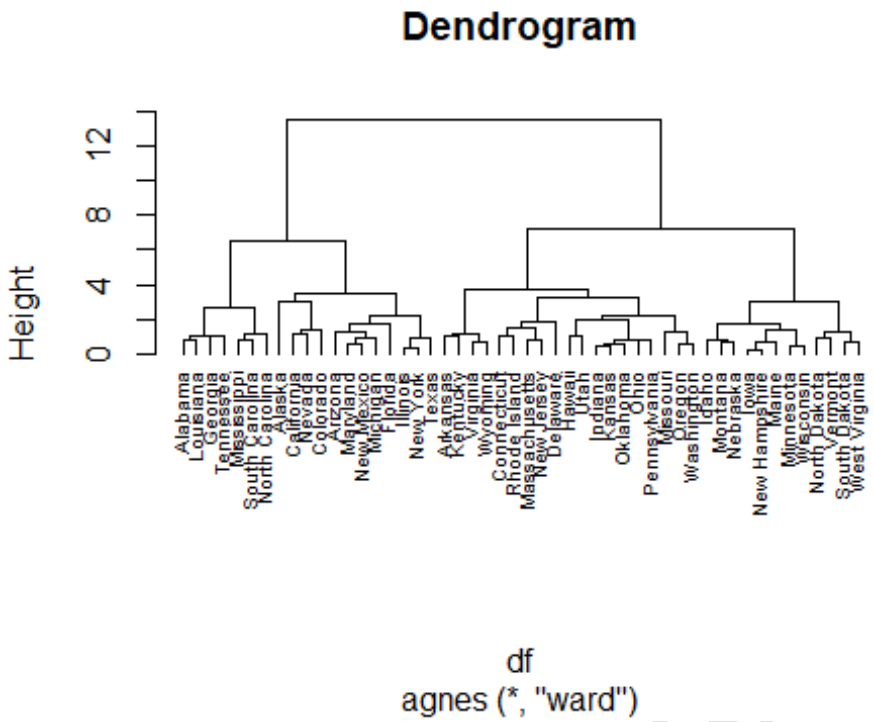
Note that this function calculates the agglomerative coefficient of each method, which is metric that measures the strength of the clusters. The closer this value is to 1, the stronger the clusters.

```
#define linkage methods
m <- c( "average", "single", "complete", "ward")
names(m) <- c( "average", "single", "complete", "ward")

#function to compute agglomerative coefficient
ac <- function(x) {
  agnes(df, method = x)$ac
}

#calculate agglomerative coefficient for each clustering linkage method
sapply(m, ac)
```

```
##   average    single  complete      ward
## 0.7379371 0.6276128 0.8531583 0.9346210
```

We can see that Ward's minimum variance method produces the highest agglomerative coefficient, thus we'll use that as the method for our final hierarchical clustering :

```
#perform hierarchical clustering using Ward's minimum variance
clust <- agnes(df, method = "ward")

#produce dendrogram
pltree(clust, cex = 0.6, hang = -1, main = "Dendrogram")
```



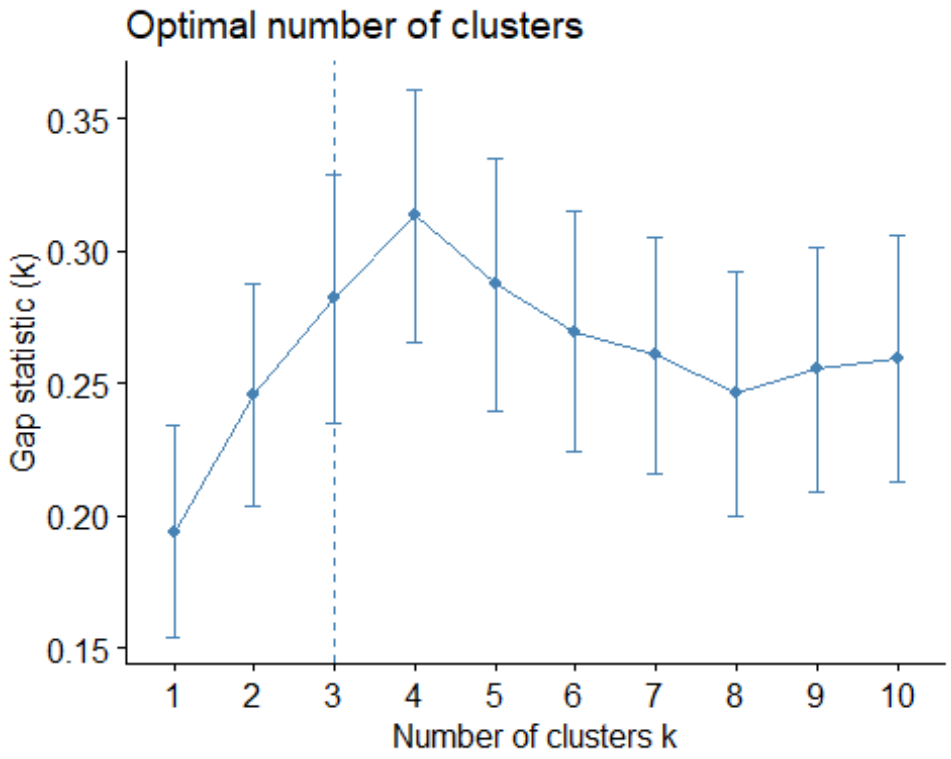
Each leaf at the bottom of the dendrogram represents an observation in the original dataset. As we move up the dendrogram from the bottom, observations that are similar to each other are fused together into a branch.

Step 4 : Determine the Optimal Number of Clusters

To determine how many clusters the observations should be grouped in, we can use a metric known as the **gap statistic**, which compares the total intra-cluster variation for different values of k with their expected values for a distribution with no clustering.

We can calculate the gap statistic for each number of clusters using the **clusGap()** function from the cluster package along with a plot of clusters vs. gap statistic using the **fviz_gap_stat()** function :

```
#calculate gap statistic for each number of clusters (up to 10 clusters)
gap_stat <- clusGap(df, FUN = hcut, nstart = 25, K.max = 10, B = 50)

## Clustering k = 1,2,..., K.max (= 10): .. done
## Bootstrapping, b = 1,2,..., B (= 50)  [one "." per sample]:
## .................................................. 50

#produce plot of clusters vs. gap statistic
fviz_gap_stat(gap_stat)
```



From the plot we can see that the gap statistic is highest at k = 4 clusters. Thus, we'll choose to group our observations into 4 distinct clusters.

Step 5 : Apply Cluster Labels to Original Dataset

To actually add cluster labels to each observation in our dataset, we can use the **cutree()** method to cut the dendrogram into 4 clusters :

```
#compute distance matrix
d <- dist(df, method = "euclidean")

#perform hierarchical clustering using Ward's method
final_clust <- hclust(d, method = "ward.D2" )

#cut the dendrogram into 4 clusters
groups <- cutree(final_clust, k=4)

#find number of observations in each cluster
table(groups)
```

```
## groups
##  1  2  3  4
##  7 12 19 12
```

We can then append the cluster labels of each state back to the original dataset:

```
#append cluster labels to original data
final_data <- cbind(USArrests, cluster = groups)

#display first six rows of final data
head(final_data)

##            Murder Assault UrbanPop Rape cluster
## Alabama      13.2     236       58 21.2       1
## Alaska       10.0     263       48 44.5       2
## Arizona       8.1     294       80 31.0       2
## Arkansas      8.8     190       50 19.5       3
## California    9.0     276       91 40.6       2
## Colorado      7.9     204       78 38.7       2
```

Lastly, we can use the **aggregate()** function to find the mean of the variables in each cluster

```
#find mean values for each cluster
aggregate(final_data, by=list(cluster=final_data$cluster), mean)

##   cluster    Murder  Assault UrbanPop     Rape cluster
## 1       1 14.671429 251.2857 54.28571 21.68571       1
## 2       2 10.966667 264.0000 76.50000 33.60833       2
## 3       3  6.210526 142.0526 71.26316 19.18421       3
## 4       4  3.091667  76.0000 52.08333 11.83333       4
```

We interpret this output is as follows:

- The mean number of murders per 100,000 citizens among the states in cluster 1 is **14.67**.
- The mean number of assaults per 100,000 citizens among the states in cluster 1 is **251.28**.
- The mean percentage of residents living in an urban area among the states in cluster 1 is 54.28%.
- The mean number of rapes per 100,000 citizens among the states in cluster 1 is **21.68**.