WST 212 - Applications in Data Science

University of Pretoria
Faculty of Natural and Agricultural Sciences
Department of Statistics

Compiled and revised by Priyanka Nagar and Ineke Derks, 2020-2021

©Copyright reserved, University of Pretoria

Contents

Chapter 1

Introduction to R software

R software is a free open source programming language for statistical computing and data visualisation. In this part we will learn some basic coding in R and get comfortable with this new language.

1.1 Packages, libraries, help and syntax

Packages and libraries

Since R is a open source software there are certain functions that require additional packages not included in the base R installation. These packages include functions which help simplify coding various operations in R. To use the functionality provided by a package we first need to install the package and once installed we need to remind R that we would like to use the package (when necessary). To install a package in R you need internet access and the correct name of the package. Please note that R is case sensitive. The function used to install packages is:

```
install.packages("package name")
```

To check whether a package needs to be installed or if you already have installed it use the following function:

```
require("package name")
```

When you want to use some of the functions within these packages you need to remind R that you are using additional packages by specifying :

```
library("package name")
```

If the functions above give an error it means either the name is spelled incorrectly or the package does not exist.

Help

The R help function is extremely valuable if you already know which function you need help with. If you are uncertain about what function to use to perform a specific operation then you will need to consult either your notes or Google. For example, R help is extremely useful for when:

```
help("mean")
```

However, it is not useful when:

```
help("how to calculate the expected value?")
```

Syntax

Please note that R is case sensitive. Coding in R can be tricky as R is case sensitive with function names as well as when you need to specify certain arguments within a function. R is also very sensitive to white spaces and tabs. When coding if you press enter and R automatically inserts a tab or white space do not remove it. The converse also holds, meaning do not include white space of tabs to make your code look fancy or spacey. Brackets and the type of brackets you use mean different things in R so make sure you use them correctly. If you are going to use inverted commas please be consistent. Do not change from a single to double inverted comma in your code.

1.2 Basics

The most basic commands in R are the arithmetic operators, for example:

```
2 + 5
```

```
## [1] 7
```

```
8 - 3
```

```
## [1] 5
```

```
3*4
```

```
## [1] 12
```

```
16/2
```

```
## [1] 8
```

```
3^2
```

```
## [1] 9
```

```
5%%2
```

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

[Remember that the modulo operator returns the reminder after division. So for 5 mod 2 above the results is 1 as 5 divided by 2 has a quotient of 2 and a remainder of 1.]

1.3 Variable assignment

In the example above we used the operators to calculate a value. If we want to use this value again in another formula then it would be easier to assign the operation to a variable. A variable allows you to store a value or an object as any data type. [Remember to use descriptive variable names.] For example: In R we use the assignment operator, <-, to assign a value/object to a variable.

```
x <- 4
y <- 3
x + y
```

```
## [1] 7
```

Or you can assign the sum of $x+y$ to another variable:

```
z <- x + y
```

Now z will store the sum of $x+y$. If we want to view the answer we need to print the variable:

```
print(z)
```

```
## [1] 7
```

1.4 Data type

There are various data types in R, the ones we will work with the most include:

- Decimal values termed as *numerics*.
- Natural numbers termed as *integers*.
- Boolean values (TRUE or FALSE) termed as *logical*.
- Text or string termed as *characters*.
- Datasets or tables termed as *data frames*.
- Categorical variables with limited categories termed as *factors*.

To determine the data type in R we make use of the *class()* function:

```
x <- 2
class(x)
```

```
## [1] "numeric"
```

```r
y <- "Some words"
class(y)
```

```
## [1] "character"
```

```r
z <- TRUE
class(z)
```

```
## [1] "logical"
```

To convert data to a different data type we use the as.new_data_type() function in R.

```r
z <- TRUE
as.numeric(z)
```

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

Note that the logical operators will convert to a *0* or *1* if converted to numeric.

Factors are a little tricky. We use factors when we have categorical data with a limited number of categories. There are two types of categorical variables: a nominal categorical variable and an ordinal categorical variable. A nominal variable is a categorical variable without an implied order. Whereas ordinal variables do have a natural ordering. For example, if you consider a variable that has three temperature indicators (*low*, *medium* and *high*); these categories can be ordered. If you have a list of animals in a specific area for example *elephants*, *donkeys* and *horses* then these categorical variables cannot be ordered in any natural way. If *a* is a factor and we *print(a)* we will see the following result:

```r
print(a)
```

```
## [1] M F F F M M
## Levels: F M
```

```r
class(a)
```

```
## [1] "factor"
```

The levels in the output specify how many categories there are and how the categories are defined.

To determine the categories within a factor variable we can use the *levels()* function:

```r
levels(a)
```

```
## [1] "F" "M"
```

1.5 Logical operators

Logical comparisons work in R, the operators are:

- $<$ less than.

- > greater than.
- <= less than or equal to.
- >= greater than or equal to.
- == equal to.
- != not equal to.
- & the and .
- | the or.

Note that for an equal comparison a double = sign is used. A logical operator will return a Boolean value stating whether the comparison is *TRUE* or *FALSE*, for example:

```
2 > 3
```

```
## [1] FALSE
```

```
4 <= 8
```

```
## [1] TRUE
```

```
7 != 3
```

```
## [1] TRUE
```

```
6 == (3+2)
```

```
## [1] FALSE
```

```
3 < 4 | 7 < 6
```

```
## [1] TRUE
```

```
5 <= 5 & 2 < 3
```

```
## [1] TRUE
```

1.6 Conditional statements

An "if" statement is a conditional statement in programming that performs a function or returns a value if a given condition is proved true. For example, Jonny can only have a chocolate after dinner **if** he eats all his vegetables. The general syntax for an *if* statement is

```
if(condition){
  ...expression...
}
```

Only if the condition evaluates to *TRUE* the expression will execute. To include an expression for when the condition evaluates *FALSE* an *else* statement is needed. The syntax is then:

```
if(condition){
  ...expression TRUE...
} else{
  ...expression FALSE...
}
```

For example:

```
age <- 24

if(age < 35){
  print("You are young.")
} else{
  print("You are not so young.")
}
```

```
## [1] "You are young."
```

```
age <- 50

if(age < 35){
  print("You are young.")
} else{
  print("You are not so young.")
}
```

```
## [1] "You are not so young."
```

Take note that it is important for the *else{ }* to start in the same line as the closing bracket of the *if* statement.

1.7 Functions

In R there are many built in-functions ready to use. Well known functions in R include the *mean()*, *std()*, *levels()*, *names()*, *print()*, *abs()*, etc. There are also additional functions available with add-on packages. Any function in R requires either an argument or arguments to compute the function. Using the *help(function name)* function we can see which arguments need to be defined. We can also create our own functions. For example, the code below creates a function called *quad_eq* which computes value of the quadratic equation for any set of constants a,b and c and any x value:

```
quad_eq <- function(a, b, c, x){
  f <- a*(x^2) + b*x + c
  return(f)
}

quad_eq(2, 2, 2, 1)
```

```
## [1] 6
```

The *quad_eq()* function takes on four arguments namely; *a,b,c* and *x*. When we call the function all these arguments need to be specified in the correct order for the function to execute. For example, on your own copy the *quad_eq* function in R and then try calling the function using the statement *quad_eq(2,3)* and see what happens.

Functions are useful when you need to calculate an expression multiple times for different set of parameter (argument) values.

Some built-in functions that are handy when working with data frames are:

- head() gives us the first 6 rows of the data frame.
- tail() gives us the last 6 rows of the data frame.
- str() gives an overview of the structure of the data frame.
- names() gives the column names of from the data frame.
- summary() gives a five number summary of the data (min, Q1, Q2, Q3 and max).
- dim() gives the dimension of the data frame.
- nrow() gives the number of rows in a data frame.

Chapter 3

Machine Learning

In this chapter, we will focus on machine learning techniques.

3.1 Introduction to machine learning

In this chapter, we will introduce the jargon of machine learning and define some concepts that are crucial when applying any machine learning technique.

In this era of "big data," data are amassed by networks of instruments and computers. There are new opportunities for finding and characterizing patterns using techniques described as data mining, machine learning, data visualization, and so on. The aim of machine learning is to gain an understanding and skills for data wrangling (a process of preparing data for visualization and other modern techniques of statistical interpretation) and using the data to answer statistical questions via modelling and visualization. Doing so inevitably involves the ability to reason statistically and utilize computational and algorithmic capacities. The trick is not mastering programming but rather learning to think in terms of these operations.

What is machine learning? It is the construction and use of algorithms that learn from data. The more data available the better the model can learn. The goal of machine learning is building models for prediction. For example, movie and series recommendations that pop up on your Netflix accounts, shopping basket analysis from take-a-lot to suggest items you might like, decision making for self-driving cars, algorithms that can analyse works of art. The predicted response that we seek can belong to one of two groups:

- Quantitative response - Regression techniques.
- Qualitative (categorical) response - Classification techniques.

The slight difference between machine learning and classical statistics is that in machine learning the model must predict unseen observations whereas, in classical statistics the model must fit, explain or describe data. In other words, in classical statistics the focus is more on model fitting, (building models) and machine learning is all about predictions, (application of models).

Machine learning techniques can be divided into several branches, which we refer to as supervised learning, unsupervised learning, semi-supervised learning and reinforcement learning. In simplified form, if the observations obtained are all labelled then supervised learning techniques will be used. If the observations are not labelled then unsupervised learning techniques will be used and if some of the observations are labelled and others not then we will use semi-supervised learning. In this course, we will focus on supervised and unsupervised learning techniques only.

Supervised learning techniques include linear regression, logistic regression, naive Bayes, linear discriminant analysis, decision trees, k-nearest neighbours, neural networks and support vector machines.

Unsupervised learning techniques include hierarchical clustering, k-means clustering, neural networks and principal component analysis.

The data science process can be viewed as:

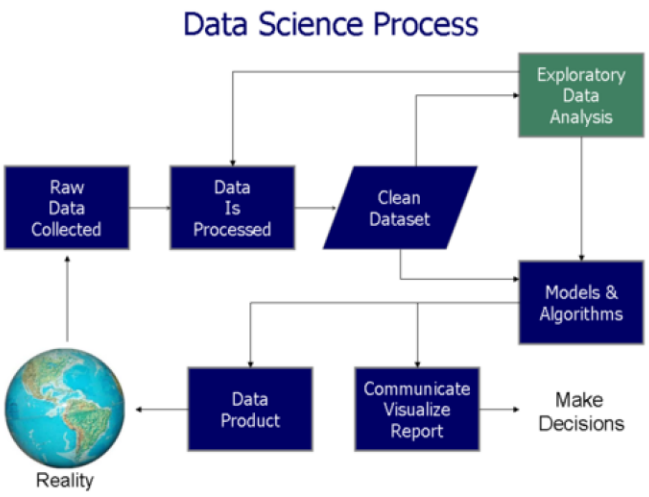
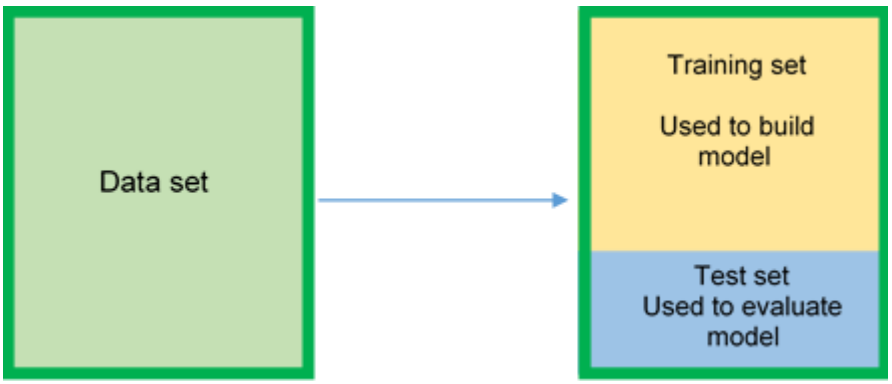


Figure 3.1: An overview of the data science process.

3.1.1 Validation

Since the aim of machine learning is to predict responses, we need to test how well our model will perform on "new" data. Testing the model on the same data used to build the model, results in very optimistic performance and overfitting.

A method used to ensure that models do not overfit is by splitting the data into two parts; a training set and test set. The model is built on the training set and the test set is used to evaluate the performance of the model. The test set can be viewed as the "new" data set for prediction. The norm is usually to split the data into 80% training and 20% test set, however, this could differ depending on the size of the data set. This concept can be viewed as follows:

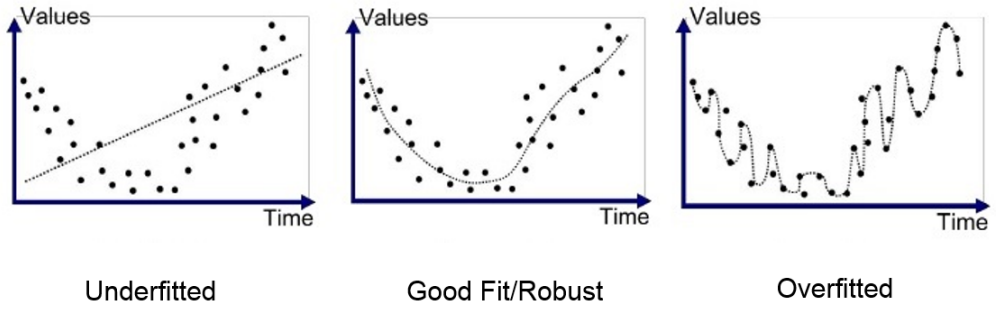


The train/test set is sufficient for small datasets. However, in the case of large data sets (big data) a train/validation/test set is more appropriate. This method is similar to the train/test set but the extra validation set allows for fine-tuning of parameters. Another method is cross fold validation, which is similar to the above but instead the data set is divided into a train/test set multiple times and the evaluated. For this course, we will focus only on a train/test set.

Note that for unsupervised learning techniques, it is not required to split the data into a train and test set since the data is not labelled and thus the correct classification is unknown.

3.1.2 Overfitting

Overfitting is when the model performs extremely well but only for the data used to build the model. In other words, the model learns the details/behaviour of the training data so well that it fits the training set perfectly but fails to fit any additional data or predict responses for any new observations. An example of overfitting is choosing too many clusters for clustering techniques or building too deep forests or trees for decision trees/ random forests.



The problem with overfitting is that it results in high variance but less bias (variance-bias tradeoff). It also results in identification of patterns that only exist in the training set and do not generalise to other data. The more complex the model is, the more data points it will capture, and the lower the bias will be. However, complexity will make the model "move" more to capture the data points, and hence its variance will be larger. Overfitting can be limited by using methods such as cross validation or train/validation/test sets of the data. An underfit model will be less flexible and cannot account for the data.

Overfitting	Underfitting
High variance	High bias
Model too specific	Model to general

3.1.3 Performance measures

There are different ways to determine whether a model is good or not. The accuracy, computation time and interpretability need to be considered when assessing a model. Different machine learning techniques require different performance metrics. These techniques can be separated into three groups; regression, classification and clustering.

For regression techniques, we use the root mean square error (RMSE) to evaluate a model. This measure calculates the distance mean of the predicted outcome and observed outcome. (This measure captures the bias-variance trade-off)

For classification techniques, we use a confusion matrix to obtain the accuracy, precision, recall and specificity measures. A confusion matrix gives the following two-way table when we consider "True" to be the positive outcome.

		Observed	
Prediction		**True**	**False**
	True	True positive (TP)	False positive (FP)
	False	False negative (FN)	True negative (TN)

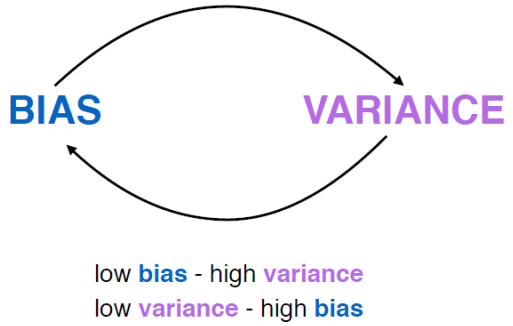
The diagonals provide the number of correctly classified observations (observed=predicted) and the off-diagonal indicate the number of incorrectly classified observations. From the confusion matrix it is possible to calculate the accuracy, precision, recall and specificity as follows:

- Accuracy $= \frac{correctly\ classified\ observations}{total\ number\ of\ observations} = \frac{TP + TN}{TP + FP + FN + TN}$.
- The error of the model $= 1-$ Accuracy.
- Precision $= \frac{True\ positive}{True\ positive + False\ positive} = \frac{TP}{TP + FP}$.
- Recall (sensitivity) $= \frac{True\ positive}{True\ positive + False\ negative} = \frac{TP}{TP + FN}$.
- Specificity $= \frac{True\ negative}{True\ negative + False\ postive} = \frac{TN}{TN + FP}$.

For clustering techniques, since the information is not labelled we use similarity measure. The similarity within clusters, within sum of squares (WSS), and the similarity between cluster, between cluster sum of squares (BSS).

3.1.4 Variance-Bias tradeoff

It is important to understand prediction errors (bias and variance) when working with predictive models. Understanding these errors assist us in building accurate models by avoiding under/over-fitting. There is a tradeoff between a model's ability to minimize bias and variance. Bias is the difference between the average prediction of our model and the correct value which we are trying to predict. Models with high bias pay very little attention to the training data and oversimplify the model. This leads to high errors on the training and test set. Variance is the variability of model prediction for a given data point or a value which tells us spread of our data. Models with high variance pay a lot of attention to training data and do not generalize on other data. This leads to models performing very well on training data but have high error rates on test data.



If the model is too simple and has very few parameters then it may have high bias and low variance. On the other hand, if the model has a large number of parameters then it has a high variance and low bias. Thus, a balance needs to be found. This tradeoff in complexity is why there is a tradeoff between bias and variance.

Before we can jump to model building, there are a few important skills that we need to gain to fully understand our data and identify the technique that will best suit our problem. These skills include data visualisation and data wrangling.

The model building process

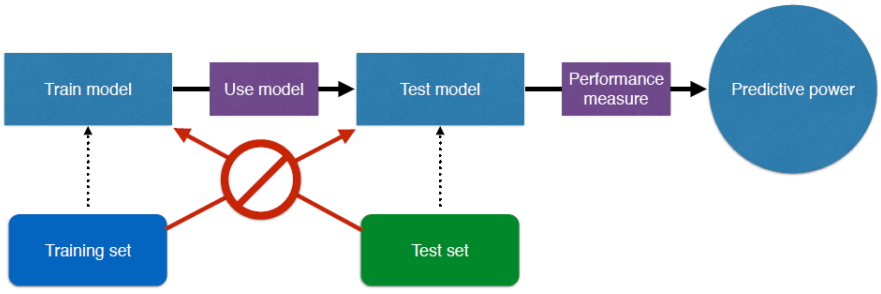


Figure 3.2: The process of model buidling.

3.2 Data Visualisation

In this section, we will focus on well-designed visualisations and how to use them for investigation and effective communication of the data. Visualization is not only an important communication skill but it also allows us to uncover structures in data that cannot be detected from numerical summaries.

" greatest value of a picture is when it forces us to notice what we never expected to see." - Exploratory Data Analysis, Tukey (1977).

When creating a visualization it is crucial that the plot represents the key points of an analysis, hence, the type of plot and variables used should be chosen after careful consideration.

Before plotting the data, we first have to know the data type. Data can be classified into two groups:

- Quantitative (Numeric)
  - Continuous (e.g., amount of rainfall per season).
  - Discrete (e.g., number of siblings).
- Qualitative (Categorical)
  - Nominal (e.g., type of pet, country).
  - Ordinal (e.g., Zomato rating, education level)

Once you have identified the data type, you can decide on the variables you want to plot. If you would like to explore a single variable then you would use a univariate plot, but if you want to compare two variables or see the effect of a certain variable on another then you would make use of a bivariate plot. A basic summary of the univariate and bivariate plot options is given in the tables below.

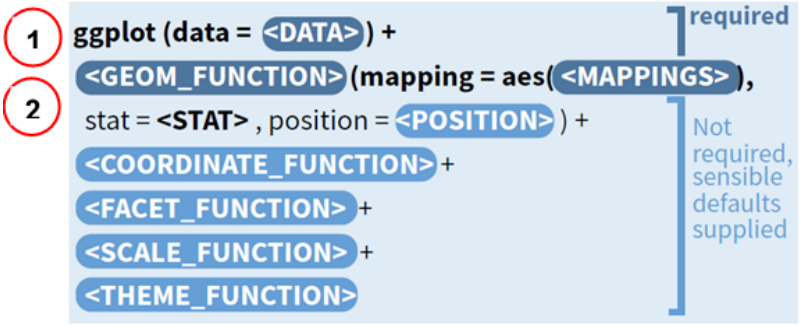
Univariate variable	
Type	**Plot**
Numeric	Few observations Histogram, Density curve Box plot, Violin plot Normal quantile plot, Rug plot, Dot plot (Caution if discrete: density curves and box plots may be misleading)
Categorical – Counts of categories	Dot chart, Bar chart, Pie chart (Caution if ordinal –order of bars, dots, etc. should reflect category order)

Bivariate variable		
	Numeric	**Categorical**
Numeric	Scatter plot, Smooth scatter, Contour plot, Smooth lines and curves	Multiple histograms, density curves
Categorical	Multiple histograms, density curves	Side-by-side bar plot, Overlaid Lines plot, Side-by-side dot chart, Mosaic plot

In R, the "ggplot2" package is very effective for data visualizations. There are base functions in R that plot variables but the "ggplot2" package provides a unifying framework for describing and specifying graphics. It allows for the creation of custom data graphics that support visual display in a purposeful way.

The ggplot2 package is based on the idea that every graph can be built from the same three components: a dataset, a co-ordinate system and geoms (visual marks that represent the data). In the syntax of ggplot2, an aesthetic is an explicit mapping between a variable and the visual cues that represent its values. In ggplot2, a plot is created with the ggplot() command, and any arguments to that function are applied across any subsequent plotting directives. Graphics in ggplot2 are built incrementally by elements. Consider the point plot in Example 3.2 below, the only elements are points, which are plotted using the geom point() function. The arguments to geom point() specify where and how the points are drawn. Here, the two aesthetics (aes()) map the vertical (y) coordinate to the weight variable, and the horizontal (x) coordinate to the height variable. The size argument to geom point() changes the size of all of the symbol/marker. Note that here, every marker is the same size.

The general syntax for plotting a graph is:



In the first line of code the data and the (x,y) co-ordinates are specified. The second line "adds"

on the type of plot. How to build a ggplot:

- Create a plot object with ggplot( ).
- Define the data frame you want to plot.
- Combine components in different ways with the "+" operator to produce a variety of plots.
- Map the aesthetics/features used to represent the data, e.g., x, y locations, color, symbol size.
- Specify what graphics shapes, geoms, to view the data, e.g., point, bars, lines. Add these to the plot with geom_point, geom_bar or geom_line.

The example below shows the step by step procedure to build a plot using ggplot.

Example 3.2

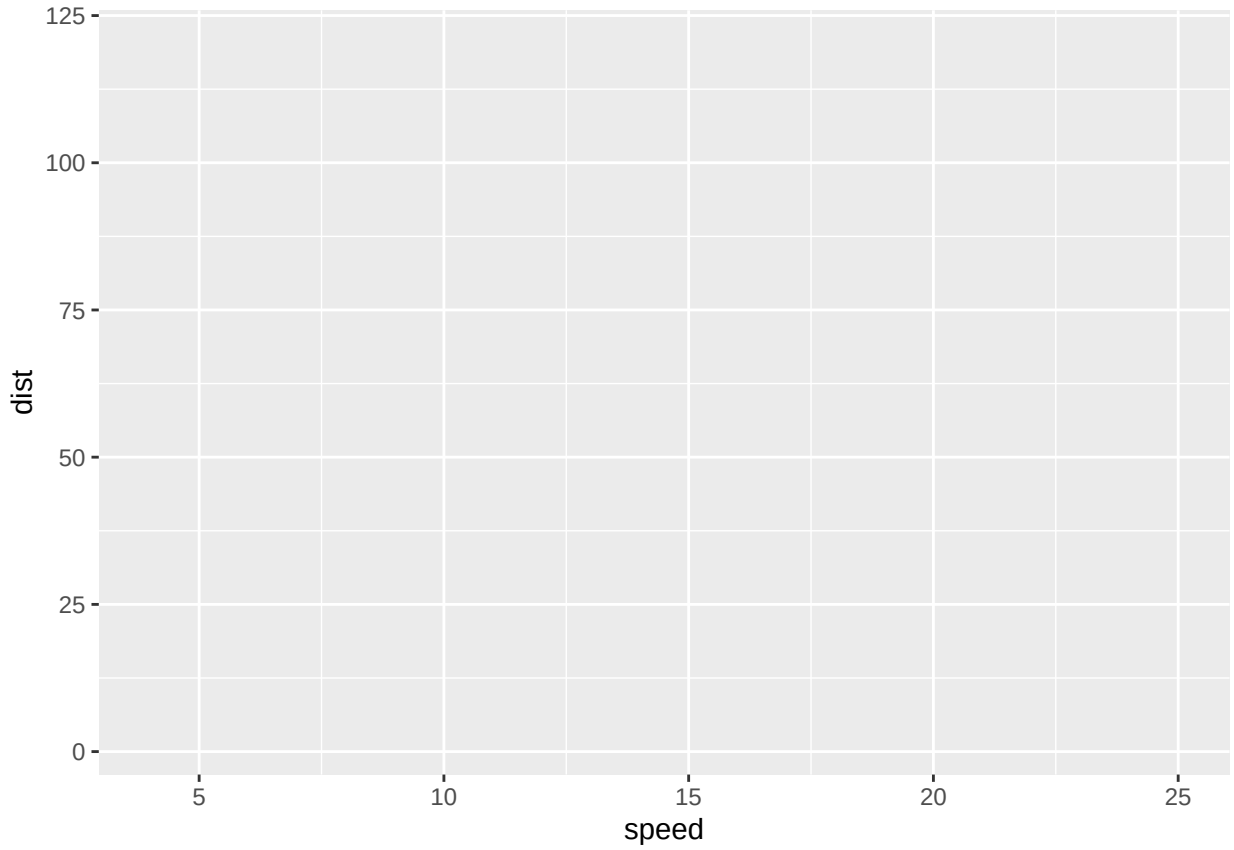
Step 1: Specify the data

```
ggplot(
  data = cars
)
```


Step 2: A mapping from data onto plot aesthetics

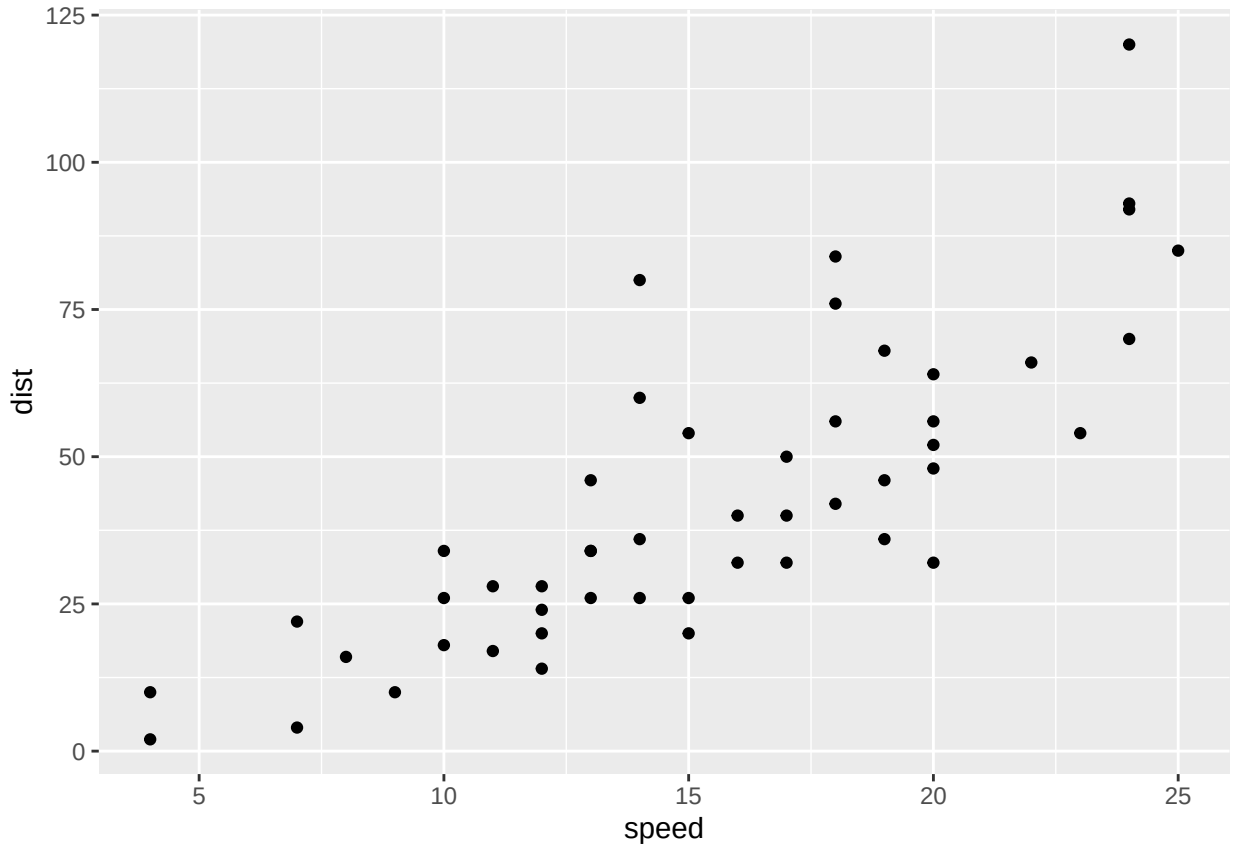
```
ggplot(
  data = cars,
```

```
  mapping = aes(
    x = speed,
    y = dist
  )
)
```



Step 3: Add a plot layer

```
ggplot(
  data = cars,
  mapping = aes(
    x = speed,
    y = dist
  )
) +
  geom_point()
```

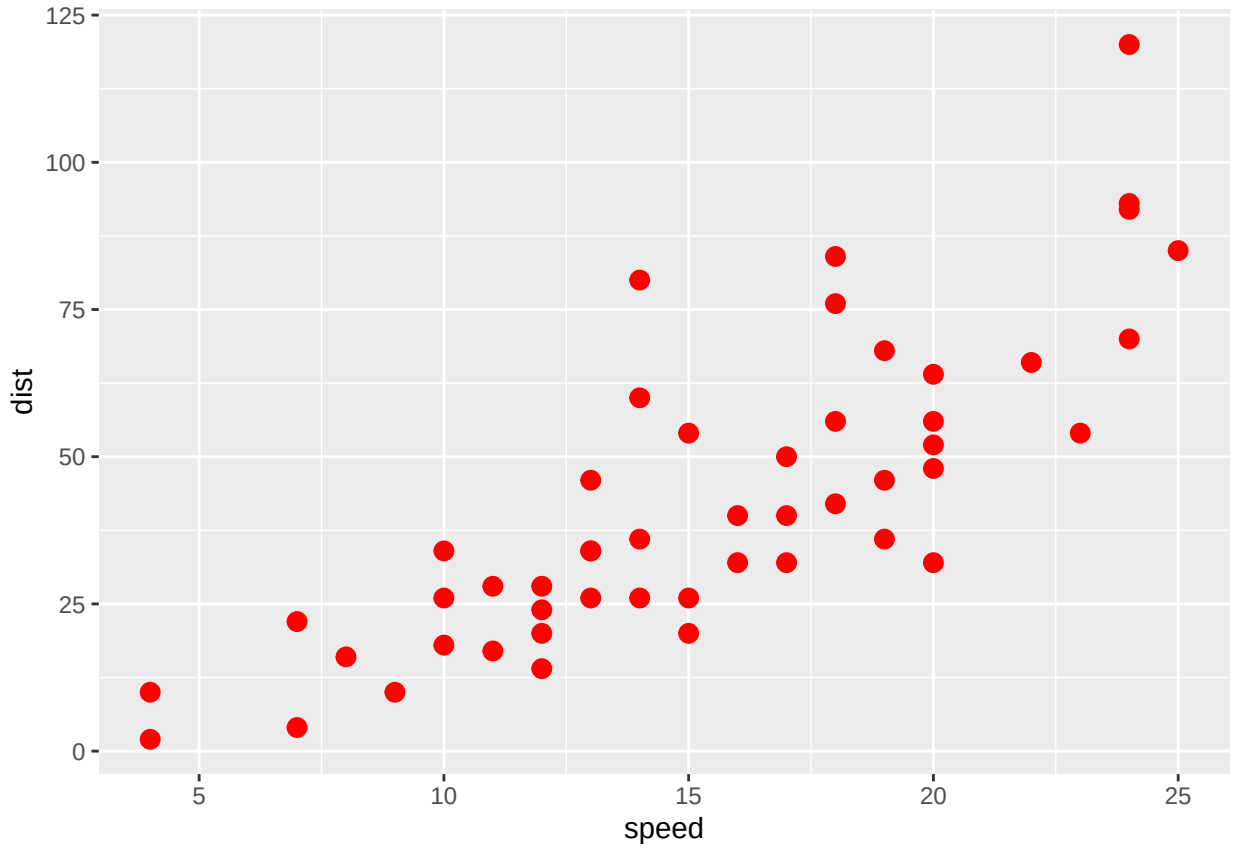


In this step, we can also add layer-specific parameters such as:

- x-axis location.
- y-axis location
- colour of marker
- fill of a marker
- shape of a marker, etc.

For example,

```
ggplot(
  data = cars,
  mapping = aes(
    x = speed,
    y = dist
  )
) +
  geom_point(color="red", size=3)
```

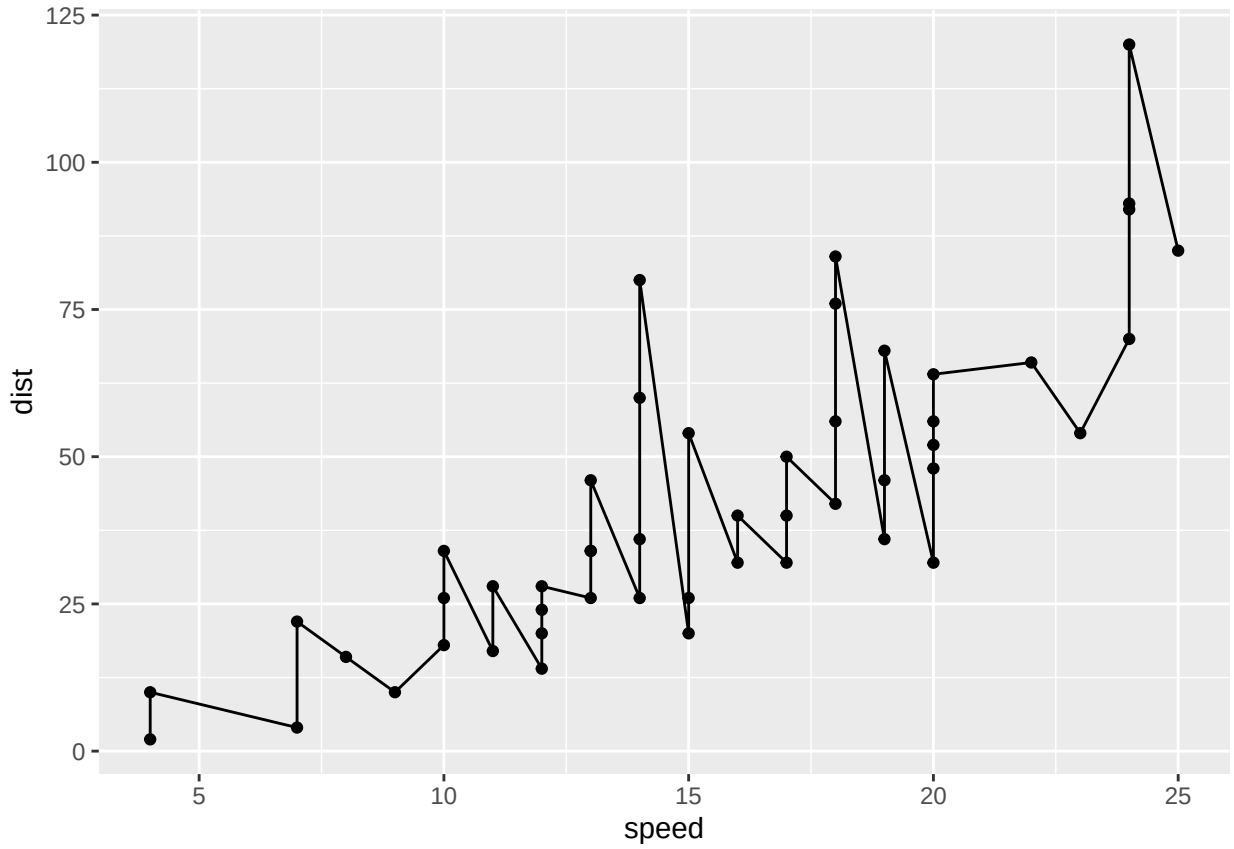


As well as choose the type of plot we want to use, for example:

- the points
- the lines
- the histograms etc

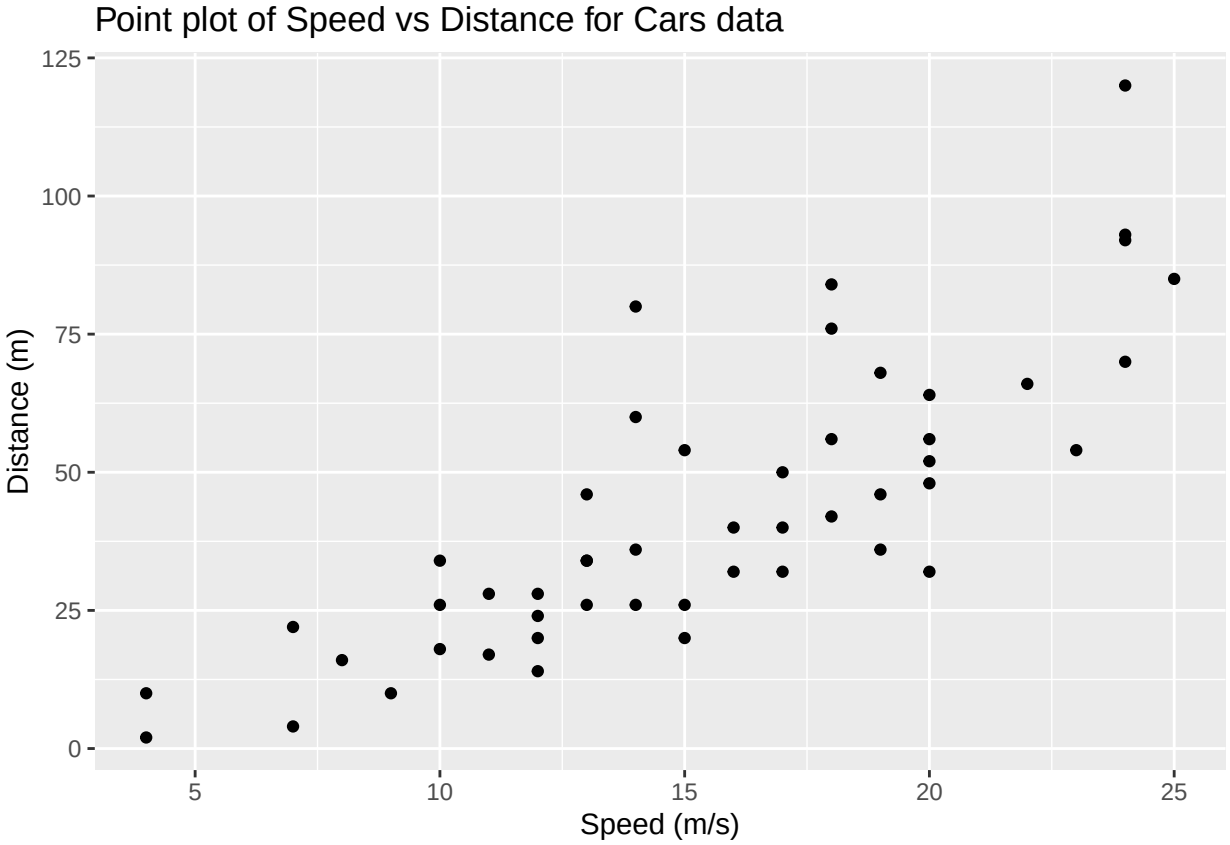
In this manner we can add multiple options to a single plot. We can also add multiple plot types, for example we can add a line plot to the above point plot as follows:

```
ggplot(
  data = cars,
  mapping = aes(
    x = speed,
    y = dist
  )
) +
  geom_point() +
  geom_line()
```



Step 4: Add Title and label axis

```
ggplot(
  data = cars,
  mapping = aes(
    x = speed,
    y = dist
  )
) +
  geom_point() +
  labs(x = "Speed (m/s)",
       y = "Distance (m)",
       title = "Point plot of Speed vs Distance for Cars data")
```



Step 5: Interpretation of the plot

In the plot above, a positive linear relationship is observed between speed and distance. As the speed increases so does the distance. There is one possible maximum outlier observed around the speed value of 24 which needs to be investigated.

A list of the geom functions that produce common graphs:

- Geom_density() - density plot for univariate continuous variable.
- Geom_density() - density plot for univariate continuous variable.
- Geom_histogram() - histogram for univariate continuous variable.
- Geom_bar() - bar graph for univariate discrete variable.
- Geom_point() - plot of point for bivariate variable.
- Geom_rug() - rug plot for univariate or bivariate variable.
- Geom_boxplot() - plots the five number summary.
- Geom_violin() - violin plots are similar to box plots, except that they also show the kernel probability density of the data at different values. (A violin plot is a mirrored density plot displayed in the same way as a boxplot).
- Geom_line() - plots a line graph.

- Geom_hline() , geom_vline(), geom_abline() - adds a horizontal/vertical/slope line to the graph.

View the ggplot2 package help for a full list of functions. Data visualization does not end at simply plotting variable(s). The plots need to help the reader to understand the meaning of the visual representation by providing context. Context can be added in two ways: making the plots descriptive and adding an interpretation to the plots. The interpretation should include a reason justifying the use of a specific plot as well as the information provided by the plot.

Adding context to plots

To ensure that a plot is descriptive the following need to be added:

- Label axes, include units.
- Add reference lines and markers for important values, these values should be discussed in the interpretation.
- Label points of unusual/interesting observations, these values should be discussed in the interpretation.
- Use color and plotting symbols to convey additional information.
- Add legends and labels, especially when plotting multiple variables on the same grid.
- Describe what you see in the caption, captions should be comprehensive. Captions should describe what has been graphed.

Every graph should be accompanied with an interpretation. Communication of findings and results is key in statistics. The interpretation should explain the patterns or lack thereof in the plot in layman terms.

3.3 Data Wrangling

In this chapter, we will focus on data wrangling. Wrangling skills provides a sound practical foundation for working with modern data.

What is data wrangling? It is the art of getting your data in a useful form via the process of cleaning and unifying messy, incomplete and complex data sets for easy access and analysis. It is a core data analysis technique; working as an iterative process that helps get to the cleanest, most usable data possible prior to analysis. Each step in the process exposes new potential ways for the data to be "re-wrangled", with the ultimate goal of generating the most robust data for analysis. The process of transforming a table into a data table that contains information explicitly is called data wrangling. Data wrangling can be assessed by considering the following six processes:

- Discovering – understanding the data and how it can be used for analytic exploration and analysis.

- Structuring – formatting the data of different shapes and sizes into a data table that can be used in analysis.
- Cleaning – correcting errors resulting from data storage or data entry and standardizing the data.
- Enriching – improvements that can be added to the representation of information to allow for efficient analysis and reporting.
- Validating – identifying consistency issues and data quality.
- Publishing – providing correctly formatted and structured data that can be used for analysis.

We will group the above process into two steps;

- exploring the data; and
- preparing the data for analysis.

We will focus on data wrangling in R, mainly using the "dplyr", "tidyr" and "lubridate" packages.

Step 1: Exploration

In the exploration step, we discover the data, understand the structure of the data and visualize the data. Given a dataset, to understand the data we need to determine certain aspects such as the size of the dataset (number of observations and variables) and the type of data it contains (categorical or quantitative). It would also be great to be able to preview the dataset without printing out the entire dataset, especially for large datasets. To help understand the data, the following functions in R can be used:

- class( ) – returns the class of a data object. Refer to Introduction to R software chapter for data types in R.
- dim( ) – returns the dimension of the data.
- names( ) – returns the column names of the data.
- str( ) – returns a preview of the data with details.
- summary( ) – returns a summary of the data.

Note: The glimpse( ) function can also be used to preview the data with details, however this function needs the "dplyr" package.

Example

Consider the dataset Iris. To explore this dataset we will apply each of the above functions to gain a better understanding of how the dataset looks.

```r
class(dat)
```

```
## [1] "data.frame"
```

```r
dim(dat)
```

```
## [1] 150   5
```

```r
names(dat)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

```r
str(dat)
```

```
## '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 ...
```

```r
glimpse(dat)
```

```
## Observations: 150
## Variables: 5
## $ Sepal.Length <dbl> 5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4.6, 5.0, 4.4, 4.9, 5.4,...
## $ Sepal.Width  <dbl> 3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 2.9, 3.1, 3.7,...
## $ Petal.Length <dbl> 1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5, 1.4, 1.5, 1.5,...
## $ Petal.Width  <dbl> 0.2, 0.2, 0.2, 0.2, 0.2, 0.4, 0.3, 0.2, 0.2, 0.1, 0.2,...
## $ Species      <fct> setosa, setosa, setosa, setosa, setosa, setosa, setosa...
```

```r
summary(dat)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa    :50
##  versicolor:50
##  virginica :50
##
##
##
```

To get a preview of how the data looks; numerically and visually; without printing the entire

dataset, we will use the following functions in R:

- head( ) – gives a view of the top of a dataset.
- tail( ) – gives a view of the bottom of a dataset.
- hist( ) – prints a histogram of a specific variable.
- plot( ) – prints a plot of two variables.

Note: The print( ) function is also available to view the entire dataset but it is not recommended especially for large datasets.

Example

Consider the dataset Iris. For a preview of the data, we apply the functions above as follows:

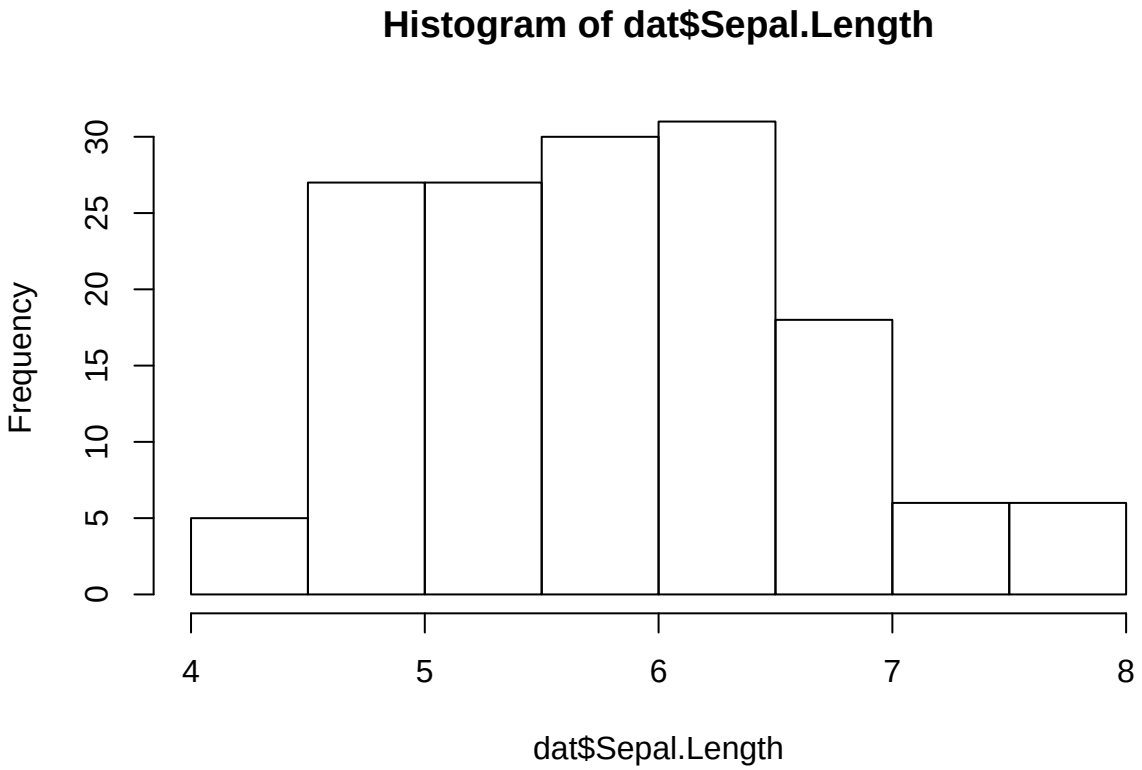
```
head(dat)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
tail(dat)
```

```
##     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
## 145          6.7         3.3          5.7         2.5 virginica
## 146          6.7         3.0          5.2         2.3 virginica
## 147          6.3         2.5          5.0         1.9 virginica
## 148          6.5         3.0          5.2         2.0 virginica
## 149          6.2         3.4          5.4         2.3 virginica
## 150          5.9         3.0          5.1         1.8 virginica
```

```
hist(dat$Sepal.Length)
```



Now that we have explored the data and have a better understanding of what the data looks like and the information it contains, we can move to the next step and prepare the data.

Step 2: Preparation

In this step, we tidy, clean and validate the data to ensure that it is ready to use for analysis.

Structuring the Data

Firstly, we tidy the structure of the data by making sure that the columns and rows in the dataset are organized in a manner that allows for efficient analysis and avoids redundancy of variables and/or observations. Given a dataset, it is important that certain principles hold. The rows of the dataset should contain observations; the columns should contain the variables (or attributes) and there should be one observational unit per dataset. Depending on the aim of a study, datasets can be designed to be either long or wide to ensure the principles hold. This can be achieved as in Part I or by using the equivalent built-in R functions. Data are called 'tidy' because they are organized as follows:

- Each observation is saved in its own row.

- Each variable is saved in its own column.
- Each cell contains a single value.

When data are in tidy form, it is relatively straightforward to transform the data into arrangements that are more useful for answering interesting questions.

Variable Types

The class( ) function is used to determine the type of data object. The type of variable can be converted using the as.new type. If the variable type is a list then the unlist( ) function in R is used to convert the list to a numeric vector.

Example

Consider the dataset Iris. The variable type for "Species" can be converted to a factor for grouping as follows:

```
new_species <- as.factor(dat$Species)
class(new_species)
```

```
## [1] "factor"
```

Date Formats

Certain datasets also have dates and times recorded in different formats in the same column. This defies the tidy data principles. To ensure that the date and time format align we use the "lubridate" package in R. This package allows us to specify the format of the date and time.

Example

The example below illustrates how the lubridate package in R can be used to format dates.

```
ymd("2020-02-20")
```

```
## [1] "2020-02-20"
```

```
ymd("2020 February 25")
```

```
## [1] "2020-02-25"
```

```
mdy("August 24, 2020")
```

```
## [1] "2020-08-24"
```

```
hms("13:33:09")
```

```
## [1] "13H 33M 9S"
```

```
ymd_hms("2020/02/20 13:33.09")
```

```
## [1] "2020-02-20 13:33:09 UTC"
```

Now that the rows and columns are in order, we need to clean the entries.

Missing and Special values

Missing values and special values need to be identified and corrected. Missing values may be random or sometimes are associated with an outcome/variable. It is not advised to make prior assumptions regarding the missing values. Missing values are represented differently in different software's, for example, in R it is a **"NA"**, in Excel it is #N/A and others include an empty string or a single dot. Special values include infinity, represented as **"Inf"** in R, and not a number entries, represented as **"NaN"** in R. How do we check for these values?

The is.na(data_frame_name) function is used to check for any missing values. This function will return a logical matrix specifying **"FALSE"** if the values are not NA and **"TRUE"** for any NA values. However, this function requires manual inspection of missing values. Instead, we can use the following functions to determine whether there are any missing values and if so how many missing values are present,

- The any(is.na(data frame)) function returns one logical value stating "TRUE" if there are missing values and "FALSE" otherwise.
- The sum(is.na(data frame)) specifies the number of missing values.
- Another method for identifying missing values as well as in which columns they appear is by using the summary(data frame) function.

A number of principled approaches have been developed to account for missing values, most notably multiple imputation (process of replacing missing values with substitute values based on the data). In this course, we will only focus on identifying missing values.

Example

Investigating missing values. Consider the dataset created below.

```
dat <- data.frame(A = c(7, NA, 5, 2),
                  B = c(1, 6, NA, 3),
                  C = c(NA, 5, 3, 1))

is.na(dat)
```

```
##          A     B     C
## [1,] FALSE FALSE  TRUE
## [2,]  TRUE FALSE FALSE
## [3,] FALSE  TRUE FALSE
## [4,] FALSE FALSE FALSE
```

```
any(is.na(dat))
```

```
## [1] TRUE
```

```
sum(is.na(dat))
```

```
## [1] 3
```

```
summary(dat)
```

```
##        A               B               C
##  Min.   :2.000   Min.   :1.000   Min.   :1
##  1st Qu.:3.500   1st Qu.:2.000   1st Qu.:2
##  Median :5.000   Median :3.000   Median :3
##  Mean   :4.667   Mean   :3.333   Mean   :3
##  3rd Qu.:6.000   3rd Qu.:4.500   3rd Qu.:4
##  Max.   :7.000   Max.   :6.000   Max.   :5
##  NA's   :1       NA's   :1       NA's   :1
```

The other types of values that need to be inspected are **outliers** and obvious **errors**. Outliers are extreme values that differ from other values. Causes include:

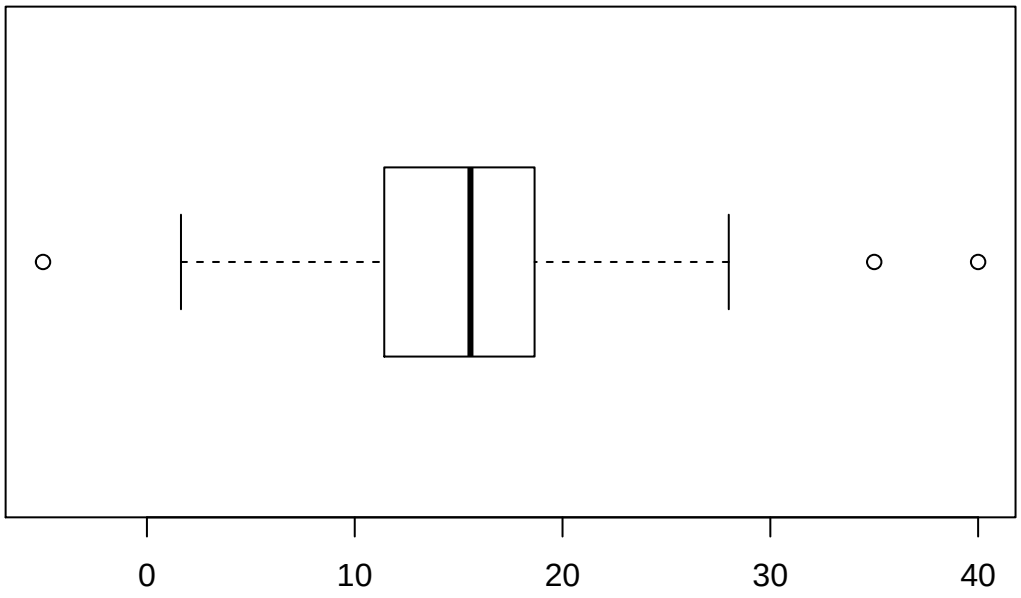
- Valid measurements.
- Variability in measurement.
- Experimental error.
- Data entry error.

These values may be discarded or retained depending on cause. Outliers may only be discarded if there is a strong justification to prove that the value is an error, otherwise, the value is retained in the data set. Outliers can generally be detected by a boxplot.

Example

Consider the dataset, X, which contains simulated values. The box plot below shows the outliers.

```
x <- c(rnorm(50, mean = 15, sd = 5), -5, 28, 35, 40)
boxplot(x, horizontal = TRUE)
```



Obvious errors may appear in many forms such as values so extreme they cannot be plausible (e.g., person aged 243) and values that do not make sense (e.g. negative age). There are several causes:

- Measurement error.
- Data entry error.
- Special code for missing data (e.g. -1 means missing).

These values should generally be removed or replaced. Obvious errors can be detected by using a boxplot, histogram or a five number summary.

Example

Consider the dataset, dat2, which contains simulated values.

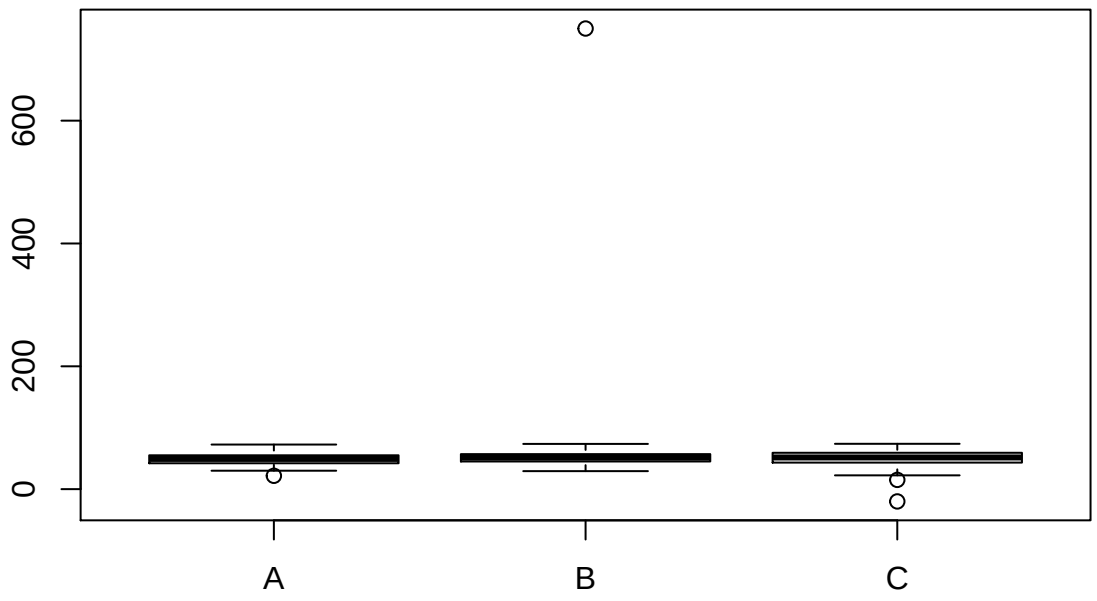
```
dat2 <- data.frame(A = rnorm(100, 50, 10),
                   B = c(rnorm(99, 50, 10), 750),
                   C = c(rnorm(99, 50, 10), -20))
summary(dat2)
```

```
##        A                B                C
##  Min.   :21.73   Min.   : 29.29   Min.   :-20.00
##  1st Qu.:42.29   1st Qu.: 45.09   1st Qu.: 43.36
```

```
##  Median :49.19   Median : 51.28   Median : 51.59
##  Mean   :49.21   Mean   : 57.32   Mean   : 50.21
##  3rd Qu.:54.99   3rd Qu.: 56.96   3rd Qu.: 59.20
##  Max.   :72.61   Max.   :750.00   Max.   : 73.81
```

```
hist(dat2$B, breaks = 20)
```


```
boxplot(dat2)
```



3.4 Supervised Learning

In this chapter, we will focus on techniques used for supervised learning, specifically on classification techniques.

3.4.1 Logistic Regression

In this section, we will focus on building a logistic regression model for predictive analysis.

Classification models are used when the response variable is qualitative and there are predetermined classes. For a logistic regression model we will consider the case when the response variable is binary. In general, the logistic model is given by,

$$ln(\frac{p(y=1)}{1-p(y=1)}) = \beta_0 + \beta_1 x_1 + ... + \beta_n x_n$$

Here, y is the outcome variable that takes either 0 or 1, where 1 is typically the outcome of interest. This outcome is linked to a set of independent variables, $x_1, ..., x_n$. $\frac{p(y=1)}{1-p(y=1)}$ refer to the odds that the outcome variable will take the value of 1. To ensure a linear relationship between the outcome

variable and the independent variables, we take the log of the odds. The probabilities from the logistic regression can be calculated as,

$$p = \frac{\exp(\beta_0 + \beta_1 x_1 + ... + \beta_n x_n)}{1 + exp(\beta_0 + \beta_1 x_1 + ... + \beta_n x_n)}$$

The glm( ) function in R makes it easy for us to fit a logistic regression model. The general syntax for this model is,

$$glm(y \sim x1 + x2 + x3, data = my\_dataset, family = "binomial")$$

OR

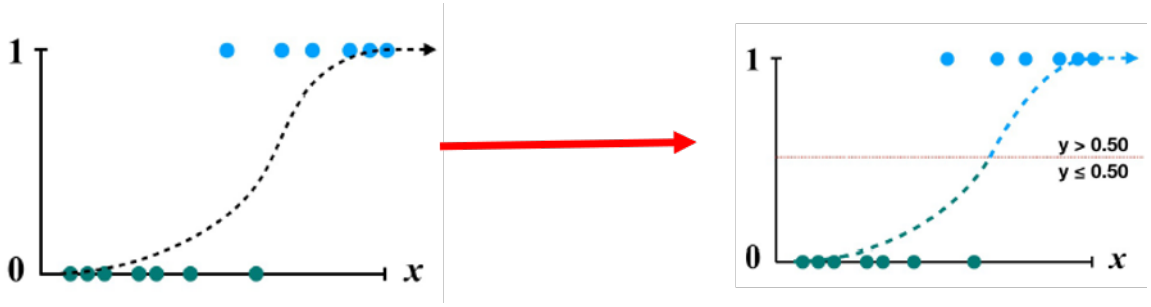
$$glm(y \sim ., data = my\_dataset, family = "binomial")$$

Where the first argument is the model function, the second argument the data set and the third argument specifies the binary response. Note that an alternative code is given where the model is specified in the with "y ~ .", this is a shorthand for specifying all the variables in the dataset except the response variable.

The glm( ) function is used to build the model, and the predict( ) function is used to make predictions based on this model.

$$prob \leftarrow predict(model, new\_dataset, type = "response")$$

The response variable in the dataset is categorical; however, the fitted model will return probabilities. To obtain a valid outcome we need to set a threshold value to classify each predicted outcome to a specific category. The norm is taken as 0.5. For example, if a probability is greater than 0.5 then it is classified into group 1 and if it is less than or equal to 0.5 then it goes to group 2.



This is made easy in R, by using the "if" statement the probabilities are classified into the groups.

$$pred \leftarrow ifelse(prob > 0.50, 1, 0)$$

Where the first argument defines that logical statement, the second argument is the assigned value if the logical test is true and the third argument is the value for if the logical test is false.

Dummy Variables

We have defined the response variable to be a binary (categorical) variable. This means that the variable may be composed of characters such as gender, or default status for loan repayment. To help represent these categories in a numeric form we introduce a dummy variable. A dummy variable takes on the value 0 or 1 to indicate the absence or presence of a factor. For example, if we consider the response variable to be the default status of a loan repayment then we can create a dummy variable that takes on the value of 1 if the loan was not repaid and 0 if the loan was repaid. When we build our model we will now use the dummy variable as our response variable.

Train/Test split

We will split the data set into a training set and test set. The logistic model will be built on the training set and the performance will be evaluated using the test set.

Real world examples of binary classification problems

In case you were wondering what kind of problems you can use logistic regression for, here are some real world binary classification problems:

- Spam detection: Predicting whether an email is spam or not.
- Credit card fraud: Predicting if a given transaction is fraud or not.
- Health: Predicting if a given mass of tissue is benign or malignant.
- Marketing: Predicting if a given user will buy a product or not.
- Banking: Predicting if a customer will default on a loan.

Example

Consider the Breast Cancer dataset. You would like to model and predict if a given specimen (row in dataset) is benign or malignant, based on 9 other cell features. A preview of the dataset:

```
head(bc)
```

```
##   X Cl.thickness Cell.size Cell.shape Marg.adhesion Epith.c.size Bare.nuclei
## 1 1            5         1          1             1            2           1
## 2 2            5         4          4             5            7          10
## 3 3            3         1          1             1            2           2
## 4 4            6         8          8             1            3           4
## 5 5            4         1          1             3            2           1
## 6 6            8        10         10             8            7          10
##   Bl.cromatin Normal.nucleoli Mitoses     Class
## 1           3               1       1    benign
```

```
## 2           3               2       1    benign
## 3           3               1       1    benign
## 4           3               7       1    benign
## 5           3               1       1    benign
## 6           9               7       1 malignant
```

Step 1: We need to define a dummy variable for the response variable as the data type is a character. We will indicate a malignant response with a 1 and a benign response with 0. Also it is important that we define this dummy variable to be a factor variable and not numeric.

```
# Change class values to 1's (malignant) and 0's (benign)
bc$Class <- ifelse(bc$Class == "malignant", 1, 0)
bc$Class <- factor(bc$Class, levels = c(0, 1))
```

Step 2: We randomly the data into a training and test set (80/20 split).

```
# set split size
split <- round(nrow(bc)*0.80)

train_ind  <- sample(1:nrow(bc),
                     split,
                     replace=FALSE)
# Create train set
trainData     <- bc[train_ind, ]
# Create test set
testData       <- bc[-train_ind, ]
```

Step 3: Build a logistic regression model with Class being our response variable and the other 9 features the predictors.

```
# Build Logistic Model
logistic_mod <- glm(Class ~ ., family = "binomial", data = trainData)
summary(logistic_mod)
```

```
##
## Call:
## glm(formula = Class ~ ., family = "binomial", data = trainData)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -3.5639  -0.0949  -0.0468   0.0227   2.4238
##
## Coefficients:
##                   Estimate Std. Error z value Pr(>|z|)
## (Intercept)     -10.056078   1.576456  -6.379 1.78e-10 ***
## X                -0.001195   0.001727  -0.692 0.489169
## Cl.thickness      0.572343   0.164752   3.474 0.000513 ***
## Cell.size         0.025248   0.234029   0.108 0.914088
```

```
## Cell.shape        0.373705   0.263103   1.420 0.155498
## Marg.adhesion     0.457827   0.164420   2.784 0.005361 **
## Epith.c.size     -0.049018   0.208472  -0.235 0.814109
## Bare.nuclei       0.361317   0.109290   3.306 0.000946 ***
## Bl.cromatin       0.433784   0.181210   2.394 0.016674 *
## Normal.nucleoli   0.216971   0.127973   1.695 0.089992 .
## Mitoses           0.535688   0.391844   1.367 0.171596
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 715.173  on 545  degrees of freedom
## Residual deviance:  73.487  on 535  degrees of freedom
## AIC: 95.487
##
## Number of Fisher Scoring iterations: 8
```

Step 4: Predict the response variables for the test set.

```
pred <- predict(logistic_mod, newdata = testData, type = "response")
```

Step 5: Calculate the performance metrics. The confusion matrix function in R provides the performance metric values. Note that for the confusion matrix, both input arguments need to be the same type of variable, i.e. they both need to be factors. Hence, we first convert the numeric predictors response to factors.

```
# Recode factors
y_pred_num <- ifelse(pred > 0.5, 1, 0)
y_pred <- factor(y_pred_num, levels = c(0,1))
y_act <- testData$Class

# Performance metrics
confusionMatrix(data = y_pred, reference = testData$Class)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  0  1
##          0 93  6
##          1  3 35
##
##                Accuracy : 0.9343
##                  95% CI : (0.879, 0.9695)
##     No Information Rate : 0.7007
##     P-Value [Acc > NIR] : 1.396e-11
##
```

```
##                  Kappa : 0.84
##
##  Mcnemar's Test P-Value : 0.505
##
##             Sensitivity : 0.9688
##             Specificity : 0.8537
##          Pos Pred Value : 0.9394
##          Neg Pred Value : 0.9211
##              Prevalence : 0.7007
##          Detection Rate : 0.6788
##    Detection Prevalence : 0.7226
##       Balanced Accuracy : 0.9112
##
##        'Positive' Class : 0
##
```

3.5 Unsupervised Learning

In this chapter, we will explore unsupervised learning techniques. We will specifically focus on K-means clustering. Clustering is a technique that groups samples such that objects within a group are similar (homogenous) and the objects between groups are dissimilar (heterogeneous). There are different types of clustering methods for example, hierarchical clustering, k nearest neighbours and k means clustering.

For k-means clustering the objective is to divide the data into k clusters and classify each observation in to a cluster. It is an iterative algorithm that calculates the centroids (means of the groups) of k clusters and assigns a data point to that cluster having least distance between its centroid and the object. To obtain a classification model for k means the following algorithm is used:

- Firstly, the number of clusters k is decided upon. An approach is to plot a scatter plot of the data and choose k according to the number of distinct groups visible from the plot. Another approach is to plot a scree plot to choose the optimal k value.
- Once k is selected, the "kmeans(data, centers=k, nstart=20)" function in R is applied where the data is specified as the first argument, the number of clusters is specified in the second argument and the nstart=20 specifies the iterations.
- The results will provide the centroids for each cluster as well as the classification of each object in the dataset.

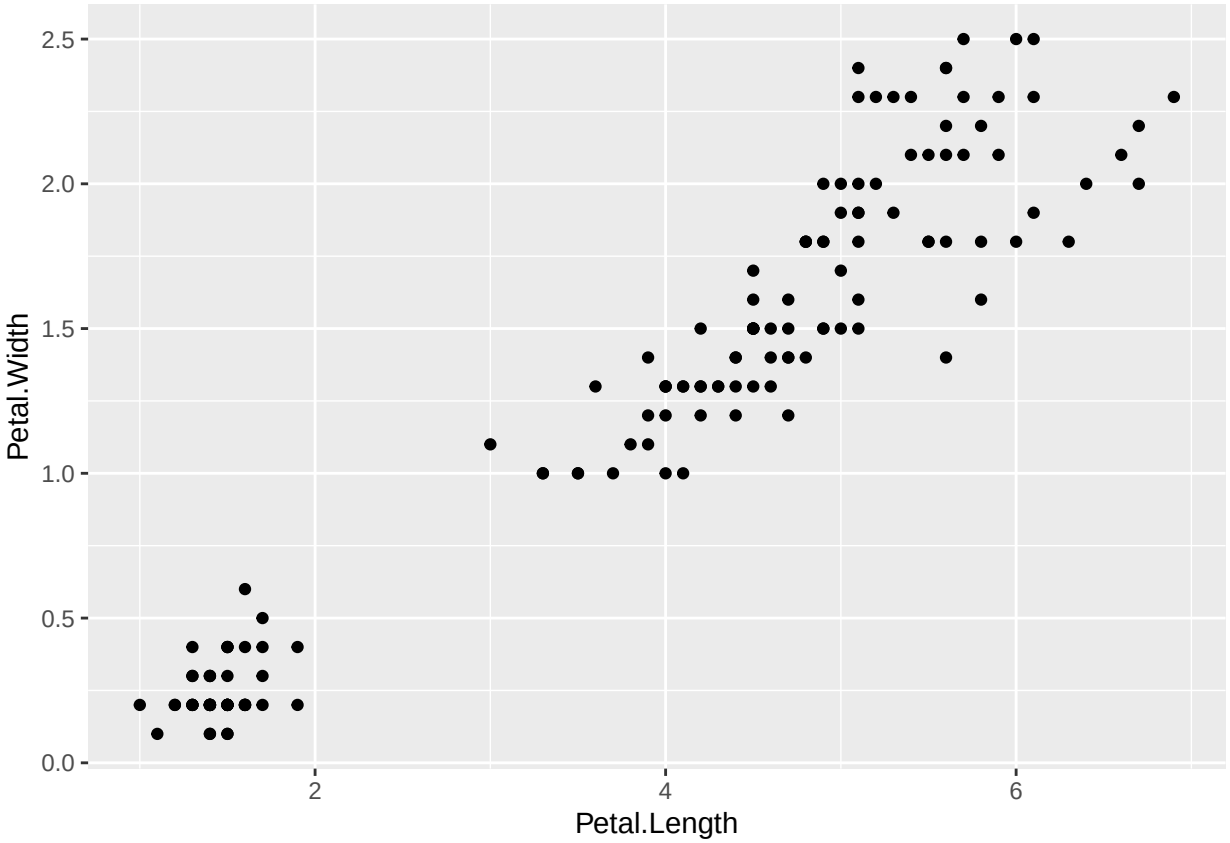
Once we obtain the model, the final step is to perform a model evaluation.

Example

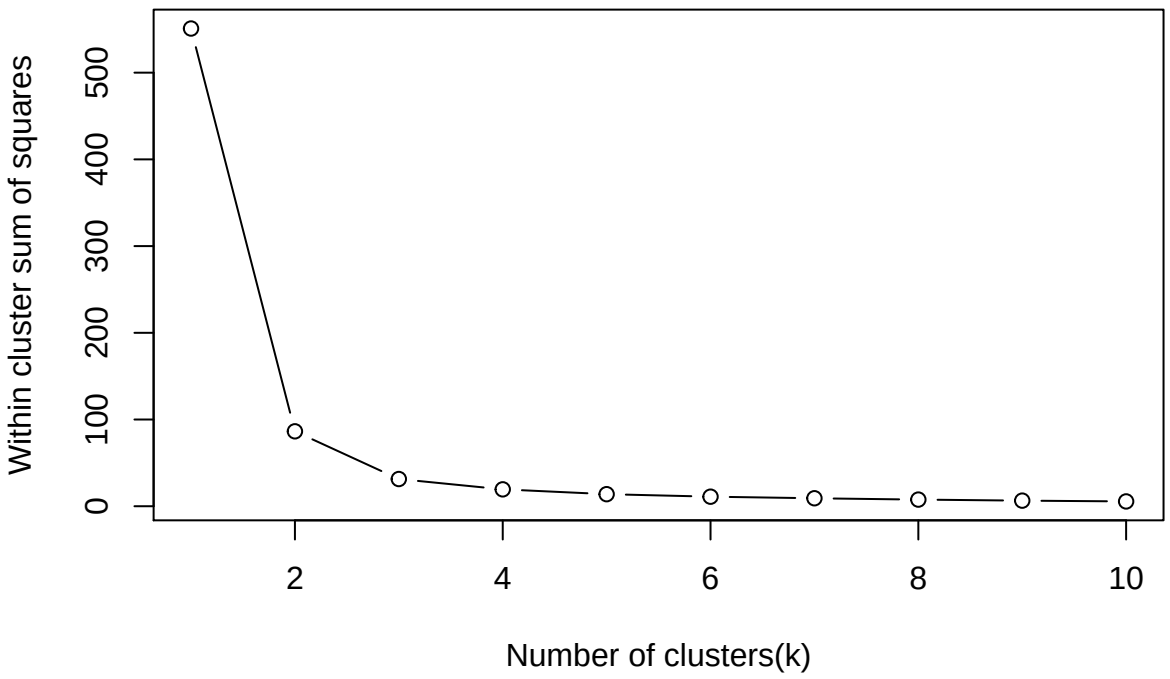
Consider the dataset Iris. The dataset consists of five variables, however we will focus on Petal.Width and Petal.Length. We would like to cluster these observations into categories that are similar within clusters and different between clusters.

Step 1: Plot the data to try to identify the number of clusters.

```
ggplot(dat,aes(x = Petal.Length, y = Petal.Width)) + geom_point()
```



From the plot, we could say that maybe 2 clusters will be a good fit. We can perform a k-means clustering with k = 2 and test if it is a good fit using an error measure. Another method of obtaining an optimal k is by using a scree plot. The scree plot, plots the within group sum of squares error for different k values. The optimal k is chosen as the point where an elbow occurs in the plot. The ideal plot will have an elbow where the measure improves more slowly as the number of clusters increases. This indicates that the quality of the model is no longer improving substantially as the number of clusters increases. In other words, the elbow indicates the number of clusters inherent in the data. A scree plot for the data set will produce the following plot:



From the scree plot, we can conclude that the optimal number of cluster is 3 as the elbow occurs at this point.

Step 2: Now that the number of clusters has been determined, we can use k-means with 3 clusters to classify the observations.

```
k_means <- kmeans(dat[,3:4], centers = 3, nstart = 20)
k_means
```

```
## K-means clustering with 3 clusters of sizes 50, 48, 52
##
## Cluster means:
##   Petal.Length Petal.Width
## 1     1.462000    0.246000
## 2     5.595833    2.037500
## 3     4.269231    1.342308
##
## Clustering vector:
##   [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [38] 1 1 1 1 1 1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
##  [75] 3 3 3 2 3 3 3 3 3 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 2 2 2 2 2 3 2 2 2 2
## [112] 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2 2 3 2 2 2 2 2 2 2 2 2
## [149] 2 2
```

```
##
## Within cluster sum of squares by cluster:
## [1]  2.02200 16.29167 13.05769
##  (between_SS / total_SS =  94.3 %)
##
## Available components:
##
## [1] "cluster"      "centers"      "totss"        "withinss"     "tot.withinss"
## [6] "betweenss"    "size"         "iter"         "ifault"
```

- The cluster means return the centroid values for each cluster. The clustering vector classifies each row of the dataset into a category.
- To evaluate the performance we calculate the within sum of squares and the between sum of squares.

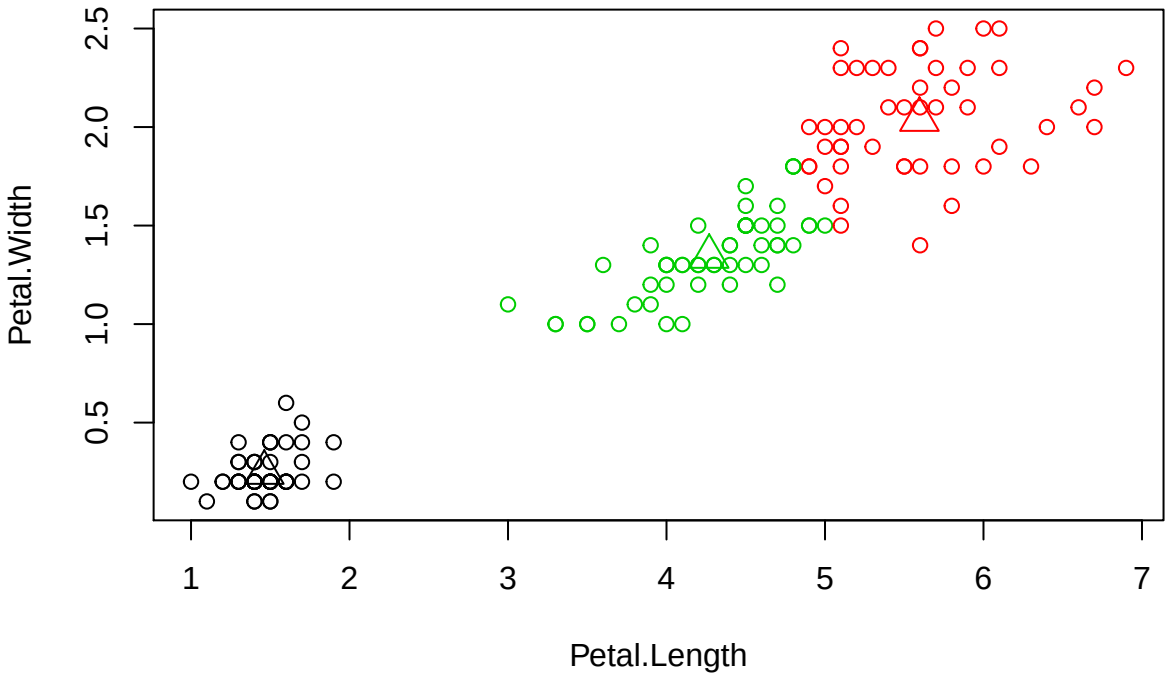
We can compare the clusters for each of the species:

```
table(k_means$cluster, dat$Species)
```

```
##
##     setosa versicolor virginica
##   1     50          0         0
##   2      0          2        46
##   3      0         48         4
```

A visual representation of the clustered observations where each cluster is represented by a different colour and the centroids of each cluster is illustrated by the triangles.

```
plot(dat[,3:4][c("Petal.Length", "Petal.Width")], col=k_means$cluster)
points(k_means$centers[,c("Petal.Length", "Petal.Width")], col=1:3, pch=2, cex=2)
```


Petal.Width
0.5
1.0
1.5
2.0
2.5
1
2
3
4
5
6
7
Petal.Length