DATA SCıENCE

DATA SCIENCE

COURSE OBJECTIVES:

The students should be able to:

1. Understand the data science process.
2. Conceive the methods in R to load, explore and manage large data.
3. Choose and evaluate the models for analysis.
4. Describe the regression analysis.
5. Select the methods for displaying the predicted results.

UNIT –I

Introduction to Data Science

Data Science Process: Roles in a data science project, Stages in a data science project, Applications of data science.

Overview of R: Basic Features of R, R installation, basic data types: Numeric, Integer, Complex, Logical, Character. Data Structures: vectors, lists, matrices, array, data frames, factors.

Control Structures: if, if-else, for loop, while loop, next, break. Functions:
named arguments, default parameters, return values.

UNIT –II

Loading, Exploring and Managing Data

Working with data from files: Reading and writing data, reading data files with read. table (), Reading in larger datasets with read. table. Working with relational databases.

Data manipulation packages:dplyr, data.table, reshape2, tidyr, lubridate.

UNIT–III

Exploratory Data Analysis and Validation Approaches

Data validation: handling missing values, null values, duplicate values, outlier detection, data cleaning, data loading and inspection, data transformation.

Cross validation: Validaton set approach, leave one out cross validation, k-fold cross validation, repeated k -fold cross validation.

UNIT – IV

Modelling Methods

Supervised: Regression Analysis in R, linear regression, logistic regression,
naive bayes classifier, decision tree, random forest, knn classifier,

Unsupervised: kmeans clustering, association rule mining, apriori algorithm.

UNIT – V

Data Visualization in R

Introduction to ggplot2: Univariate graphs: categorical, quantitative, bivariate graphs categorical vs. categorical, quantitative vs quantitative, categorical vs. quantitative, multivariate graphs : grouping, faceting.

TEXT BOOKS:

1. Practical Data Science with R, Nina Zumel & John Mount , Manning Publications
NY, 2014.
2. Beginning Data Science in R-Data Analysis, Visualization, and Modelling for the Data
Scientist -Thomas Mailund –Apress -2017.

REFERENCE BOOKS:

1. The Comprehensive R Archive Network-https://cran.r-project.org.
2. R for Data Science by Hadley Wickham and Garrett Grolemund , 2017 , Published by O Reilly Media, Inc.
3. R Programming for Data Science -Roger D. Peng, 2015 , Lean Publishing.
4. https://rkabacoff.github.io/datavis/IntroGGPLOT.html.

COURSE OUTCOMES:

The students will be able to:

- Analyze the basics in R programming in terms of constructs, control statements and Functions.
- Implement Data Preprocessing using R Libraries.
- Apply the R programming from a statistical perspective and Modelling Methods.
- Build regression models for a given problem.
- Illustrate R programming tools for Graphs.

INDEX

UNIT-I

Introduction to Data Science and Overview of R

Data Science Process: Roles in a data science project, Stages in a data science project, Setting expectations.

Overview of R: Basic Features of R, R installation, Basic Data Types: Numeric, Integer, Complex, Logical, Character.

Data Structures: Vectors, Matrix, Lists, Indexing, Named Values, Factors. Subsetting R Objects: Sub setting a Vector, Matrix, Lists, Partial Matching, Removing NAValues.

Control Structures: if-else, for Loop, while Loop, next, break. Functions: Named Arguments , Default Parameters, Return Values.

Roles in a data science project

Table Data science project roles and responsibilities

Role	Responsibilities
Project sponsor	Represents the business interests; champions the project
Client	Represents end users' interests; domain expert
Data scientist	Sets and executes analytic strategy; communicates with sponsor and client
Data architect	Manages data and data storage; sometimes manages data collection
Operations	Manages infrastructure; deploys final project results

PROJECT SPONSOR

The most important role in a data science project is the project sponsor. The sponsor is the person who wants the data science result; generally they represent the business interests. The sponsor is responsible for deciding whether the project is a success or failure. The ideal sponsor meets the following condition: if they're satisfied with the project outcome, then the project is by definition a success.

KEEP THE SPONSOR INFORMED AND INVOLVED

It's critical to keep the sponsor informed and involved. Show them plans, progress, and intermediate successes or failures in terms they can understand.

CLIENT

While the sponsor is the role that represents the business interest, the client is the role that represents the model's end users' interests. The client is more hands-on than the sponsor; they're the interface between the technical details of building a good model and the day-to-day work process into which the model will be deployed. They aren't necessarily mathematically or statistically sophisticated, but are familiar with the relevant business processes and serve as the domain expert on the team.As with the sponsor, you should keep the client informed and involved. Ideally you'd like to have regular meetings with them to keep your efforts aligned with the needs of the end users.

DATA SCIENTIST

The next role in a data science project is the data scientist, who's responsible for taking all necessary steps to make the project succeed, including setting the project strategy and keeping the client informed. They design the project steps, pick the data sources, and pick the tools to be used. Since they

pick the techniques that will be tried, they have to be well informed about statistics and machine learning. They're also responsible for project planning and tracking, though they may do this with a project management partner.

DATA ARCHITECT

The data architect is responsible for all of the data and its storage. Often this role is filled by someone outside of the data science group, such as a database administrator or architect. Data architects often manage data warehouses for many different projects, and they may only be available for quick consultation.

OPERATIONS

The operations role is critical both in acquiring data and delivering the final results. The person filling this role usually has operational responsibilities outside of the data science group. For example, if you're deploying a data science result that affects howproducts are sorted on an online shopping site, then the person responsible for running the site will have a lot to say about how such a thing can be deployed. This person will likely have constraints on response time, programming language, or data size that you need to respect in deployment. The person in the operations role may already be supporting your sponsor or your client, so they're often easy to find.

The Lifecycle of Data Science

The major steps in the life cycle of Data Science project are as follows:

1. Problem identification

This is the crucial step in any Data Science project. First thing is understanding in what way Data Science is useful in the domain under consideration and identification of appropriate tasks which are useful for the same. Domain experts and Data Scientists are the key persons in the problem identification of problem. Domain expert has in depth knowledge of the application domain and exactly what is the problem to be solved. Data Scientist understands the domain and help in identification of problem and possible solutions to the problems.

2. Business Understanding

Understanding what customer exactly wants from the business perspective is nothing but Business Understanding. Whether customer wish to do predictions or want to improve sales or minimise the loss or optimise any particular process etc forms the business goals. During business understanding two important steps are followed:

- **KPI (Key Performance Indicator)**

For any data science project, key performance indicators define the performance or success of the project. There is a need to be an agreement between the customer and data science project team on Business related indicators and related data science project goals. Depending on the business need the business indicators are devised and then accordingly the data science project team decides the goals and indicators. To better understand this let us see an example. Suppose the business need is to optimise the overall spendings of the company, then the data science goal will be to use the existing resources to manage double the clients. Defining the Key performance Indicators is very crucial for any data science projects as the cost of the solutions will be different for different goals.

- **SLA (Service Level Agreement)**

Once the performance indicators are set then finalizing the service level agreement is important. As per the business goals the service level agreement terms are decided. For example, for any airline reservation system simultaneous processing of say 1000 users is required. Then the product must satisfy this service requirement is the part of service level agreement. Once the performance indicators are agreed and service level agreement is completed then the project proceeds to the next important step.

3. Collecting Data

The basic data collection can be done using the surveys. Generally, the data collected through surveys provide important insights. Much of the data is collected from the various processes followed in the enterprise. At various steps the data is recorded in various software systems used in the enterprise which is important to understand the process followed from the product development to deployment and delivery. The historical data available through archives is also important to better understand the business. Transactional data also plays a vital role as it is collected on a daily basis. Many statistical methods are applied to the data to extract the important information related to business. In data science project the major role is played by data and so proper data collection methods are important.

4. Pre-processing data

Large data is collected from archives, daily transactions and intermediate records. The data is available in various formats and in various forms. Some data may be available in hard copy formats also. The data is scattered at various places on various servers. All these data are extracted and converted into single format and then processed. Typically, as data warehouse is constructed where the Extract, Transform and Loading (ETL) process or operations are carried out. In the data science project this ETL operation is vital and important. A data architect role is important in this stage who decides the structure of data warehouse and perform the steps of ETL operations.

5. Analyzing data

Now that the data is available and ready in the format required then next important step is to understand the data in depth. This understanding comes from analysis of data using various statistical tools available. A data engineer plays a vital role in analysis of data. This step is also called as Exploratory Data Analysis (EDA). Here the data is examined by formulating the various statistical functions and dependent and independent variables or features are identified. Careful analysis of data revels which data or features are important and what is the spread of data. Various plots are utilized to visualize the data for better understanding. The tools like Tableau, PowerBI etc are famous for performing Exploratory Data Analysis and Visualization. Knowledge of Data Science with Python and R is important for performing EDA on any type of data.

6. Data Modelling

Data modelling is the important next step once the data is analysed and visualized. The important components are retained in the dataset and thus data is further refined. Now the important is to decide how to model the data? What tasks are suitable for modelling? The tasks, like classification or regression, which is suitable is dependent upon what business value is required. In these tasks also many ways of modelling are available. The Machine Learning engineer applies various algorithms to the data and generates the output. While modelling the data many a times the models are first tested on dummy data similar to actual data.

7. Model Evaluation/ Monitoring

As there are various ways to model the data so it is important to decide which one is effective. For that model evaluation and monitoring phase is very crucial and important. The model is now tested with actual data. The data may be very few and in that case the output is monitored for improvement. There may be changes in data while model is being evaluated or tested and the output will drastically change depending on changes in data. So, while evaluating the model following two phases are important:

- **Data Drift Analysis**

Changes in input data is called as data drift. Data drift is common phenomenon in data science as depending on the situation there will be changes in data. Analysis of this change is called Data Drift Analysis. The accuracy of the model depends on how well it handles this data drift. The changes in data are majorly because of change in statistical properties of data.

- **Model Drift Analysis**

To discover the data drift machine learning techniques can be used. Also, more sophisticated methods like Adaptive Windowing, Page Hinkley etc. are available for use. Modelling Drift Analysis is important as we all know change is constant. Incremental learning also can be used effectively where the model is exposed to new data incrementally.

8. Model Training

Once the task and the model are finalised and data drift analysis modelling is finalized then the important step is to train the model. The training can be done is phases where the important parameters can be further fine tuned to get the required accurate output. The model is exposed to the actual data in production phase and output is monitored.

9. Model Deployment

Once the model is trained with the actual data and parameters are fine tuned then model is deployed. Now the model is exposed to real time data flowing into the system and output is generated. The model can be deployed as web service or as an embedded application in edge or mobile application. This is very important step as now model is exposed to real world.

10. Driving insights and generating BI reports

After model deployment in real world, next step is to find out how model is behaving in real world scenario. The model is used to get the insights which aid in strategic decisions related to business. The business goals are bound to these insights. Various reports are generated to see how business is driving. These reports help in finding out if key process indicators are achieved or not.

11. Taking a decision based on insight

For data science to make wonders, every step indicated above has to be done very carefully and accurately. When the steps are followed properly then the reports generated in above step helps in taking key decisions for the organization. The insights generated helps in taking strategic decisions like for example the organization can predict that there will be need of raw material in advance. The data science can be of great help in taking many important decisions related to business growth and better revenue generation.

Setting Expectations

Developing expectations is the process of deliberately thinking about what you expect before you do anything, such as inspect your data, perform a procedure, or enter a command. For experienced data analysts, in some circumstances, developing expectations may be an automatic, almost subconscious process, but it's an important activity to cultivate and be deliberate about.For example, you may be going out to dinner with friends at a cash-only establishment and need to stop by the ATM to withdraw money before meeting up. To make a decision about the amount of money you're going to withdraw, you have to have developed some expectation of the cost of dinner. This may be an automatic expectation because you dine at this establishment regularly so you know what the typical cost of a meal is there, which would be an example of a priori knowledge. Another example of a priori knowledge would be knowing what a typical meal costs at a restaurant in your city, or knowing what a meal at the most expensive restaurants in your city costs. Using that information, you could perhaps place an upper and lower bound on how much the meal will cost.You may have also sought out external information to develop your expectations, which could include asking your friends who will be joining you or who have eaten at the restaurant before and/or Googling the restaurant to find general cost information online or a menu with prices. This same process, in which you use any a

priori information you have and/or external sources to determine what you expect when you inspect your data or execute an analysis procedure, applies to each core activity of the data analysis process.

Features Of R

1) Open Source

An open-source language is a language on which we can work without any need for a license or a fee. R is an open-source language. We can contribute to the development of R by optimizing our packages, developing new ones, and resolving issues.

2) Platform Independent

R is a platform-independent language or cross-platform programming language which means its code can run on all operating systems. R enables programmers to develop software for several competing platforms by writing a program only once. R can run quite easily on Windows, Linux, and Mac.

3) Machine Learning Operations

R allows us to do various machine learning operations such as classification and regression. For this purpose, R provides various packages and features for developing the artificial neural network. R is used by the best data scientists in the world.

4) Exemplary support for data wrangling

R allows us to perform data wrangling. R provides packages such as dplyr, readr which are capable of transforming messy data into a structured form.

5) Quality plotting and graphing

R simplifies quality plotting and graphing. R libraries such as ggplot2 and plotly advocates for visually appealing and aesthetic graphs which set R apart from other programming languages.

6) The array of packages

R has a rich set of packages. R has over 10,000 packages in the CRAN repository which are constantly growing. R provides packages for data science and machine learning operations.

7) Statistics

R is mainly known as the language of statistics. It is the main reason why R is predominant than other programming languages for the development of statistical tools.

8) Continuously Growing

R is a constantly evolving programming language. Constantly evolving means when something evolves, it changes or develops over time, like our taste in music and clothes, which evolve as we get older. R is a state of the art which provides updates whenever any new feature is added.

Limitations of R

1) Data Handling

In R, objects are stored in physical memory. It is in contrast with other programming languages like Python. R utilizes more memory as compared to Python. It requires the entire data in one single place which is in the memory. It is not an ideal option when we deal with Big Data.

2) Basic Security

R lacks basic security. It is an essential part of most programming languages such as Python. Because of this, there are many restrictions with R as it cannot be embedded in a web-application.

3) Complicated Language

R is a very complicated language, and it has a steep learning curve. The people who don't have prior knowledge or programming experience may find it difficult to learn R.

4) Weak Origin

The main disadvantage of R is, it does not have support for dynamic or 3D graphics. The reason behind this is its origin. It shares its origin with a much older programming language "S."

5) Lesser Speed

R programming language is much slower than other programming languages such as MATLAB and Python. In comparison to other programming language, R packages are much slower.

In R, algorithms are spread across different packages. The programmers who have no prior knowledge of packages may find it difficult to implement algorithms.

Basic Data Types

The Numeric Type

The numeric type is what you get any time you write a number into R. You can test if an object is numeric using the is.numeric function or by getting the class object.

```
is.numeric(2)
## [1] TRUE
class(2)
## [1] "numeric"
```

The Integer Type

The integer type is used for, well, integers. Surprisingly, the 2 is not an integer in R. It is a numeric type which is the larger type that contains all floating-point numbers as well as integers. To get an integer you have to make the value explicitly an integer, and you can do that using the function as.integer or writing L after the literal.

```
is.integer(2)
## [1] FALSE
is.integer(2L)
## [1] TRUE
x <- as.integer(2)
is.integer(x)
## [1] TRUE
class(x)
## [1] "integer"
```

If you translate a non-integer into an integer, you just get the integer part.

```
as.integer(3.2)
## [1] 3
as.integer(9.9)
## [1] 9
```

The Complex Type

If you ever find that you need to work with complex numbers, R has those as well. You construct them by adding an imaginary number—a number followed by i—to any number or explicitly using the function as. complex. The imaginary number can be zero, 0i, which creates a complex number that only has a non-zero real part.

```
1 + 0i
## [1] 1+0i
is.complex(1 + 0i)
## [1] TRUE
class(1 + 0i)
## [1] "complex"
sqrt(as.complex(-1))
## [1] 0+1i
```

The Logical Type

Logical values are what you get if you explicitly type in TRUE or FALSE, but it is also what you get if you make, for example, a comparison.

```
x <- 5 > 4
x
```

[1] TRUE
class(x)
[1] "logical"
is.logical(x)
[1] TRUE
The Character Type
Finally, characters are what you get when you type in a string such as "hello, world".
x <- "hello, world"
class(x)
[1] "character"
is.character(x)
[1] TRUE
Unlike in some languages, character doesn't mean a single character but any text. So it is not like in C or Java where you have single character types, 'c', and multi-character strings, "string", they are both just characters. You can, similar to the other types, explicitly convert a value into a character (string) using as. character:
as.character(3.14)
[1] "3.14"
Unlike in some languages, character doesn't mean a single character but any text. So it is not like in C or Java where you have single character types, 'c', and multi-character strings, "string", they are both just characters. You can, similar to the other types, explicitly convert a value into a character (string) using as. character:
as.character(3.14)
[1] "3.14"

Data Structures
vectors
vectors, which are sequences of values all of the same type.
v <- **c**(1, 2, 3)
or through some other operator or function, e.g., the : operator or the rep function
1:3
[1] 1 2 3
rep("foo", 3)
[1] "foo""foo""foo"
We can test if something is this kind of vector using the is.atomic function:
v <- 1:3
is.atomic(v)
[1] TRUE
v <- 1:3
is.vector(v)
[1] TRUE
It is just that R only consider such a sequence a vector—in the sense that is.vector returns TRUE—if the object doesn't have any attributes (except for one, names, which it is allowed to have).
Attributes are meta-information associated with an object, and not something we will deal with much here, but you just have to know that is.vector will be FALSE if something that is a perfectly good vector gets
an attribute.
v <- 1:3
is.vector(v)
[1] TRUE
attr(v, "foo") <- "bar"
v

[1] 1 2 3
attr(,"foo")
[1] "bar"
is.vector(v)
[1] FALSE
So if you want to test if something is the kind of vector I am talking about here, use is.atomic instead. When you concatenate (atomic) vectors, you always get another vector back. So when you combine several c() calls you don't get any kind of tree structure if you do something like this:
c(1, 2, **c**(3, 4), **c**(5, 6, 7))
[1] 1 2 3 4 5 6 7
The type might change, if you try to concatenate vectors of different types, R will try to translate the type into the most general type of the vectors.
c(1, 2, 3, "foo")
[1] "1""2""3""foo"
Matrix
If you want a matrix instead of a vector, what you really want is just a two-dimensional vector. You can set the dimensions of a vector using the dim function—it sets one of those attributes we talked about previously where you specify the number of rows and the number of columns you want the matrix to have.
v <- 1:6
attributes(v)
NULL
dim(v) <- **c**(2, 3)
attributes(v)
$dim
[1] 2 3
dim(v)
[1] 2 3
v
[,1] [,2] [,3]
[1,] 1 3 5
[2,] 2 4 6
When you do this, the values in the vector will go in the matrix column-wise, i.e., the values in the vector will go down the first column first and then on to the next column and so forth. You can use the convenience function matrix to create matrices and there you can specify if you want
the values to go by column or by row using the byrow parameter.
v <- 1:6
matrix(data = v, nrow = 2, ncol = 3, byrow = FALSE)
[,1] [,2] [,3]
[1,] 1 3 5
[2,] 2 4 6
matrix(data = v, nrow = 2, ncol = 3, byrow = TRUE)
[,1] [,2] [,3]
[1,] 1 2 3
[2,] 4 5 6
the * operator will not do matrix multiplication. You use * if you want to make element-wise multiplication; for matrix multiplication you need the operator %*% instead.
(A <- **matrix**(1:4, nrow = 2))
[,1] [,2]
[1,] 1 3
[2,] 2 4
(B <- **matrix**(5:8, nrow = 2))

```
## [,1] [,2]
## [1,] 5 7
## [2,] 6 8
```
A * B
```
## [,1] [,2]
## [1,] 5 21
## [2,] 12 32
```
A %*% B
```
## [,1] [,2]
## [1,] 23 31
## [2,] 34 46
```
If you want to transpose a matrix, you use the t function and, if you want to invert it, you use the solve function.

t(A)
```
## [,1] [,2]
## [1,] 1 2
## [2,] 3 4
```
solve(A)
```
## [,1] [,2]
## [1,] -2 1.5
## [2,] 1 -0.5
```
Lists

Lists, like vectors, are sequences, but unlike vectors, the elements of a list can be any kind of objects, and they do not have to be the same type of objects. This means that you can construct more complex data structures out of lists.

For example, we can make a list of two vectors:

list(1:3, 5:8)
```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] 5 6 7 8
```
Notice how the vectors do not get concatenated like they would if we combined them with c(). The result of this command is a list of two elements that happens to be both vectors.

They didn't have to have the same type either, we could make a list like this, which also consist of two vectors but vectors of different types:

list(1:3, **c**(TRUE, FALSE))
```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1] TRUE FALSE
```
You can flatten a list into a vector using the function unlist(). This will force the elements in the list to be converted into the same type, of course, since that is required of vectors.

unlist(**list**(1:4, 5:7))
```
## [1] 1 2 3 4 5 6 7
```

Indexing

```
v <- 1:4
v[2]
## [1] 2
```

We can also know that you can get a subsequence out of the vector using a range of indices:

```
v[2:3]
## [1] 2 3
```

Here we are indexing with positive numbers, which makes sense since the elements in the vector have positive indices, but it is also possible to use negative numbers to index in R. If you do that it is interpreted as specifying the complement of the values you want. So if you want all elements except the first element, you can use:

You can also use multiple negative indices to remove some values:

```
v[-(1:2)]
## [1] 3 4
```

Another way to index is to use a Boolean vector. This vector should be the same length as the vector you index into, and it will pick out the elements where the Boolean vector is true.

```
v[v %% 2 == 0]
## [1] 2 4
```

If you want to assign to a vector you just assign to elements you index; as long as the vector to the right of the assignment operator has the same length as the elements the indexing pulls out you will be assigning to the vector.

```
v[v %% 2 == 0] <- 13
v
## [1] 1 13 3 13
```

If the vector has more than one dimension—remember that matrices and arrays are really just vectors with more dimensions—then you subset them by subsetting each dimension. If you leave out a dimension, you will get whole range of values in that dimension, which is a simple way to of getting rows and columns of a matrix:

```
m <- matrix(1:6, nrow = 2, byrow = TRUE)
m
## [,1] [,2] [,3]
## [1,] 1 2 3
## [2,] 4 5 6
m[1,]
## [1] 1 2 3
m[,1]
## [1] 1 4
```

You can also index out a submatrix this way by providing ranges in one or more dimensions:

```
m[1:2,1:2]
## [,1] [,2]
## [1,] 1 2
## [2,] 4 5
```

If you want to get to the actual element in there, you need to use the [[]] operator instead.

```
L <- list(1,2,3)
L[[1]]
## [1] 1
```

Named Values

The elements in a vector or a list can have names. These are attributes that do not affect the values of the elements but can be used to refer to them. You can set these names when you create the vector or list:

```
v <- c(a = 1, b = 2, c = 3, d = 4)
v
## a b c d
## 1 2 3 4
L <- list(a = 1:5, b = c(TRUE, FALSE))
L
## $a
## [1] 1 2 3 4 5
##
## $b
## [1] TRUE FALSE
```

Or you can set the names using the names<- function. That weird name, by the way, means that you are dealing with the names() function combined with assignment:

```
names(v) <- LETTERS[1:4]
v
## A B C D
## 1 2 3 4
```

You can use names to index vectors and lists (where the [] and [[]] returns either a list or the element of the list, as before):

```
v["A"]
## A
## 1
L["a"]
## $a
## [1] 1 2 3 4 5
L[["a"]]
## [1] 1 2 3 4 5
```

factors

In the first step, we create a vector.

1. Next step is to convert the vector into a factor,

R provides factor() function to convert the vector into factor. There is the following syntax of factor() function

1. factor_data<- factor(vector)
   data <- c("Shubham","Nishka","Arpita","Nishka","Shubham","Sumit","Nishka","Shubham","Sumit","Arpita","Sumit")
2. print(data)
3. print(is.factor(data))
4. # Applying the factor function.
5. factor_data<- factor(data)
6. print(factor_data)
7. print(is.factor(factor_data))

   output:[1] "Shubham""Nishka""Arpita""Nishka""Shubham""Sumit""Nishka"
[8] "Shubham""Sumit""Arpita""Sumit"
[1] FALSE
[1] Shubham Nishka Arpita Nishka Shubham Sumit Nishka Shubham Sumit

[10] Arpita Sumit
Levels: Arpita Nishka Shubham Sumit
[1] TRUE

Accessing components of factor

Like vectors, we can access the components of factors. The process of accessing components of factor is much more similar to the vectors. We can access the element with the help of the indexing method or using logical vectors. Let's see an example in which we understand the different-different ways of accessing the components.

```
# Creating a vector as input.
data <- c("Shubham","Nishka","Arpita","Nishka","Shubham","Sumit","Nishka","Shubham","Sumit","Arpita", "Sumit")
factor_data<- factor(data)
print(factor_data)
print(factor_data[4])
print(factor_data[c(5,7)])
print(factor_data[-4])
print(factor_data[c(TRUE,FALSE,FALSE,FALSE,TRUE,TRUE,TRUE,FALSE,FALSE,FALSE,TRUE)])
```

```
[1] Shubham Nishka  Arpita  Nishka  Shubham Sumit  Nishka  Shubham Sumit
[10] Arpita  Sumit
Levels: Arpita Nishka Shubham Sumit
[1] Nishka
Levels: Arpita Nishka Shubham Sumit
[1] Shubham Nishka
Levels: Arpita Nishka Shubham Sumit
[1] Shubham Nishka  Arpita  Shubham Sumit  Nishka  Shubham Sumit  Arpita
[10] Sumit
Levels: Arpita Nishka Shubham Sumit
[1] Shubham Shubham Sumit  Nishka  Sumit
Levels: Arpita Nishka Shubham Sumit
```

Modification of factor

Like data frames, R allows us to modify the factor. We can modify the value of a factor by simply re-assigning it. In R, we cannot choose values outside of its predefined levels means we cannot insert value if it's level is not present on it. For this purpose, we have to create a level of that value, and then we can add it to our factor.

```
data <- c("Shubham","Nishka","Arpita","Nishka","Shubham")
factor_data<- factor(data)
print(factor_data)
factor_data[4] <-"Arpita"
print(factor_data)
factor_data[4] <- "Gunjan"
print(factor_data)
levels(factor_data) <- c(levels(factor_data),"Gunjan")
factor_data[4] <- "Gunjan"
print(factor_data)
```

output:

```
[1] Shubham Nishka Arpita Nishka Shubham
Levels: Arpita Nishka Shubham
[1] Shubham Nishka  Arpita  Arpita  Shubham
```

Levels: Arpita Nishka Shubham
Warning message:
In `[<-.factor`(`*tmp*`, 4, value = "Gunjan") :
invalid factor level, NA generated
[1] Shubham Nishka Arpita <NA> Shubham
Levels: Arpita Nishka Shubham
[1] Shubham Nishka Arpita Gunjan Shubham
Levels: Arpita Nishka Shubham Gunjan

Generating Factor Levels

R provides gl() function to generate factor levels. This function takes three arguments i.e., n, k, and labels. Here, n and k are the integers which indicate how many levels we want and how many times each level is required.

There is the following syntax of gl() function which is as follows

1. gl(n, k, labels)
1. n indicates the number of levels.
2. k indicates the number of replications.
3. labels is a vector of labels for the resulting factor levels.

Example

1. gen_factor<- gl(3,5,labels=c("BCA","MCA","B.Tech"))
2. gen_factor

Output

[1] BCA BCA BCA BCA BCA MCA MCA MCA MCA MCA
[11] B.Tech B.Tech B.Tech B.Tech B.Tech
Levels: BCA MCA B.Tech

```
height <- c(132,151,162,139,166,147,122)
weight <- c(48,49,66,53,67,52,40)
gender <- c("male","male","female","female","male","female","male")
input_data <- data.frame(height,weight,gender)
print(input_data)
print(is.factor(input_data$gender))
print(input_data$gender)
```

When we execute the above code, it produces the following result −

```
height weight gender
1    132     48  male
2    151     49  male
3    162     66 female
4    139     53 female
5    166     67  male
6    147     52 female
7    122     40  male
[1] TRUE
[1] male  male  female female male  female male
Levels: female male
```

Changing the Order of Levels

The order of the levels in a factor can be changed by applying the factor function again with new order of the levels.

```
data <- c("East","West","East","North","North","East","West",
"West","West","East","North")
factor_data <- factor(data)
print(factor_data)
```

```
new_order_data <- factor(factor_data,levels = c("East","West","North"))
print(new_order_data)
```
When we execute the above code, it produces the following result −
```
[1] East  West  East  North North East  West  West  West  East  North
Levels: East North West
[1] East  West  East  North North East  West  West  West  East  North
Levels: East West North
```

Subsetting R Objects

There are three operators that can be used to extract subsets of R objects.

- The [operator always returns an object of the same class as the original. It can be used to select multiple elements of an object
- The [[operator is used to extract elements of a list or a data frame. It can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame.
- The $ operator is used to extract elements of a list or data frame by literal name. Its semantics are similar to that of [[.

Subsetting a Vector

Vectors are basic objects in R and they can be subsetted using the [operator.

```
> x <- c("a", "b", "c", "c", "d", "a")
> x[1]    ## Extract the first element
[1] "a"
> x[2]    ## Extract the second element
[1] "b"
```
The [operator can be used to extract multiple elements of a vector by passing the operator an integer sequence. Here we extract the first four elements of the vector.

```
> x[1:4]
[1] "a""b""c""c"
```
The sequence does not have to be in order; you can specify any arbitrary integer vector.

```
> x[c(1, 3, 4)]
[1] "a""c""c"
```
We can also pass a logical sequence to the [operator to extract elements of a vector that satisfy a given condition. For example, here we want the elements of x that come lexicographically after the letter “a”.

```
> u <- x >"a"
> u
[1] FALSE  TRUE  TRUE  TRUE  TRUE FALSE
> x[u]
[1] "b""c""c""d"
```
Another, more compact, way to do this would be to skip the creation of a logical vector and just subset the vector directly with the logical expression

```
> x[x >"a"]
[1] "b""c""c""d"
```

Subsetting a Matrix

Matrices can be subsetted in the usual way with (i,j) type indices. Here, we create simple 2×3 matrix with the matrix function.

```
> x <- matrix(1:6, 2, 3)
> x
[,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

We can access the (1,2)or the (2,1)element of this matrix using the appropriate indices.

```
> x[1, 2]
[1] 3
> x[2, 1]
[1] 2
```

Indices can also be missing. This behavior is used to access entire rows or columns of a matrix.

```
> x[1, ] ## Extract the first row
[1] 1 3 5
> x[, 2]  ## Extract the second column
[1] 3 4
```

Subsetting Lists

Lists in R can be subsetted using all three of the operators mentioned above, and all three are used for different purposes.

```
> x <- list(foo = 1:4, bar = 0.6)
> x
$foo
[1] 1 2 3 4

$bar
[1] 0.6
```

The [[operator can be used to extract single elements from a list. Here we extract the first element of the list.

```
> x[[1]]
[1] 1 2 3 4
```

The [[operator can also use named indices so that you don't have to remember the exact ordering of every element of the list. You can also use the $ operator to extract elements by name.

```
> x[["bar"]]
[1] 0.6
> x$bar
[1] 0.6
```

Notice you don't need the quotes when you use the $ operator.

One thing that differentiates the [[operator from the $ is that the [[operator can be used with computed indices. The $ operator can only be used with literal names.

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hello")
> name <- "foo"
>
> ## computed index for "foo"
> x[[name]]
[1] 1 2 3 4
>
> ## element "name" doesn’t exist! (but no error here)
> x$name
NULL
>
> ## element "foo" does exist
> x$foo
[1] 1 2 3 4
```

Partial Matching

Partial matching of names is allowed with [[and $. This is often very useful during interactive work if the object you’re working with has very long element names. You can just abbreviate those names and R will figure out what element you’re referring to.

```
> x <- list(aardvark = 1:5)
> x$a
[1] 1 2 3 4 5
> x[["a"]]
NULL
> x[["a", exact = FALSE]]
[1] 1 2 3 4 5
```

Removing NA Values

A common task in data analysis is removing missing values (NAs).

```
> x <- c(1, 2, NA, 4, NA, 5)
> bad <- is.na(x)
> print(bad)
[1] FALSE FALSE  TRUE FALSE  TRUE FALSE
> x[!bad]
[1] 1 2 4 5
```

What if there are multiple R objects and you want to take the subset with no missing values in any of those objects?

```
> x <- c(1, 2, NA, 4, NA, 5)
> y <- c("a", "b", NA, "d", NA, "f")
> good <- complete.cases(x, y)
> good
[1]  TRUE  TRUE FALSE  TRUE FALSE  TRUE
> x[good]
[1] 1 2 4 5
> y[good]
[1] "a""b""d""f"
```

Control Structures

if condition

This control structure checks the expression provided in parenthesis is true or not. If true, the execution of the statements in braces {} continues.

Syntax:

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

Example:

```
x <-100
if(x > 10){
print(paste(x, "is greater than 10"))
}
```

Output:

```
[1] "100 is greater than 10"
```

if-else condition

It is similar to **if** condition but when the test expression in if condition fails, then statements in **else** condition are executed.

Syntax:

```
if(expression)
{
statements
....
....
}
else
{
statements
....
....
}
```

Example:

```
x <-5

# Check value is less than or greater than 10
if(x > 10){
  print(paste(x, "is greater than 10"))
}else{
  print(paste(x, "is less than 10"))
}
```

Output:

```
[1] "5 is less than 10"
```

for loop

It is a type of loop or sequence of statements executed repeatedly until exit condition is reached.

Syntax:

```
for(value in vector)
{
statements
....
....
}
```

Example:

```
x <-letters[4:10]

for(i inx){
  print(i)
}
```

Output:

```
[1] "d"
[1] "e"
[1] "f"
[1] "g"
[1] "h"
[1] "i"
[1] "j"
```

Nested loops

Nested loops are similar to simple loops. Nested means loops inside loop. Moreover, nested loops are used to manipulate the matrix.

```
for(i in1:3)
{

for(j in1:5)
{

        print(paste("This is iteration i =", i, "and j =", j))# Some output
}
}
# [1] "This is iteration i = 1 and j = 1"
# [1] "This is iteration i = 1 and j = 2"
# [1] "This is iteration i = 1 and j = 3"
# [1] "This is iteration i = 1 and j = 4"
# [1] "This is iteration i = 1 and j = 5"
# [1] "This is iteration i = 2 and j = 1"
# [1] "This is iteration i = 2 and j = 2"
# [1] "This is iteration i = 2 and j = 3"
# [1] "This is iteration i = 2 and j = 4"
# [1] "This is iteration i = 2 and j = 5"
# [1] "This is iteration i = 3 and j = 1"
# [1] "This is iteration i = 3 and j = 2"
# [1] "This is iteration i = 3 and j = 3"
# [1] "This is iteration i = 3 and j = 4"
# [1] "This is iteration i = 3 and j = 5"
```

while loop

while loop is another kind of loop iterated until a condition is satisfied. The testing expression is checked first before executing the body of loop.

Syntax:

```
while(expression)
{
statement
....
....
}
```

Example:

```
x =1

# Print 1 to 5
while(x <=5){
  print(x)
  x =x +1
}
```

Output:

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

repeat loop and break statement

repeat is a loop which can be iterated many number of times but there is no exit condition to come out from the loop. So, break statement is used to exit from the loop. **break** statement can be used in any type of loop to exit from the loop.

Syntax:

```
repeat {
statements
....
....
if(expression) {
break
}
}
```

Example:

```
x =1

# Print 1 to 5
repeat{
print(x)
  x =x +1
  if(x > 5){
    break
   }
}
```

Output:

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

next statement

next statement is used to skip the current iteration without executing the further statements and continues the next iteration cycle without terminating the loop.

Example:

```
# Defining vector
x <-1:10

# Print even numbers
for(i inx){
if(i%%2!=0){
    next#Jumps to next loop
  }
  print(i)
}
```

Output:

```
[1] 2
[1] 4
[1] 6
[1] 8
[1] 10
```

Functions

name <- function(arguments) expression

Where name can be any variable name, arguments is a list of formal arguments to the function, and expression is what the function will do when you call it. It says expression because you might as well thinkabout the body of a function as an expression, but typically it is a sequence of statements enclosed by curlybrackets:

name <- function(arguments) { statements }

It is just that such a sequence of statements is also an expression; the result of executing a series of statements is the value of the last statement.

The following function will print a statement and return 5 because the statements in the function bodyare first a print statement and then just the value 5 that will be the return value of the function:

```
f <- function()
{
print("hello, world")
5
}
f()
## [1] "hello, world"
## [1] 5
```

```
plus <- function(x, y) {
print(paste(x, "+", y, "is", x + y))
x + y
}
div <- function(x, y) {
print(paste(x, "/", y, "is", x / y))
x / y
}
plus(2, 2)
## [1] "2 + 2 is 4"
## [1] 4
div(6, 2)
## [1] "6 / 2 is 3"
## [1] 3
```

Named Arguments

The order of arguments matters when you call a function because it determines which argument gets set to which value:

```
div(6,2)
## [1] "6 / 2 is 3"
## [1] 3
div(2,6)
## [1] "2 / 6 is 0.333333333333333"
## [1] 0.3333333
```

If a function has many arguments, though, it can be hard always to remember the order, so there is an alternative way to specify which variable is given which values: named arguments. It means that when you call a function, you can make explicit which parameter each argument should be set to.

```
div(x = 6, y = 2)
## [1] "6 / 2 is 3"
## [1] 3
div(y = 2, x = 6)
## [1] "6 / 2 is 3"
## [1] 3
```

This makes explicit which parameter gets assigned which value, and you can think of it as an assignment operator. You shouldn't, though, because although you *can* use = as an assignment operator you *cannot* use <- for specifying named variables. It looks like you can, but it doesn't do what you want it to do (unless you want something really weird):

```
div(x <- 6, y <- 2)
## [1] "6 / 2 is 3"
## [1] 3
div(y <- 2, x <- 6)
## [1] "2 / 6 is 0.333333333333333"
## [1] 0.3333333
```

The assignment operator <- returns a value and that is passed along to the function as positional arguments. So in the second function call above you are assigning 2 to y and 6 to x in the scope outside the function, but the values you pass to the function are positional so inside the function you have given 2 to x and 6 to y.

Default Parameters

When you define a function, you can provide default values to parameters like this:

```
pow <- function(x, y = 2) x^y
pow(2)
## [1] 4
pow(3)
## [1] 9
pow(2, 3)
## [1] 8
pow(3, 3)
## [1] 27
```

Default parameter values will be used whenever you do not provide the parameter at the function call.

Return Value from R Function

Method 1: R function with return value

In this scenario, we will use the return statement to return some value

Syntax:

```
function_name <- function(parameters)
{
statements
 return(value)
}
function_name(values)
```

Where,

- function_name is the name of the function
- parameters are the values that are passed as arguments
- return() is used to return a value
- function_name(values) is used to pass values to the parameters

```
addition= function(val1,val2)
 {
add=val1+val2
return(add)
}
addition(10,20)
```

Output:

[1] 30

Method 2: R function to return multiple values as a list

In this scenario, we will use the list() function in the return statement to return multiple values.

Syntax:

```
function_name <- function(parameters) {
statements
return(list(value1,value2,.,value n)
}
function_name(values)
```

where,

- function_name is the name of the function
- parameters are the values that are passed as arguments
- return() function takes list of values as input
- function_name(values) is used to pass values to the parameters

Example: R program to perform arithmetic operations and return those values

```
arithmetic = function(val1,val2)
 {
add=val1+val2
sub=val1-val2
mul=val1*val2
div=val2/val1
return(list(add,sub,mul,div))
}
arithmetic(10,20)
```

Output:

```
[[1]]
[1] 30

[[2]]
[1] -10

[[3]]
[1] 200

[[4]]
[1] 2
```

Write R Programming: Create a 5 × 4 matrix, 3 × 3 matrix with labels and fill the matrix by rows and 2 × 2 matrix with labels and fill the matrix by columns

Program:

```
m1 = matrix(1:20, nrow=5, ncol=4)
print("5 × 4 matrix:")
print(m1)
cells = c(1,3,5,7,8,9,11,12,14)
rnames = c("Row1", "Row2", "Row3")
cnames = c("Col1", "Col2", "Col3")
m2 = matrix(cells, nrow=3, ncol=3, byrow=TRUE, dimnames=list(rnames, cnames))
print("3 × 3 matrix with labels, filled by rows: ")
print(m2)
print("3 × 3 matrix with labels, filled by columns: ")
m3 = matrix(cells, nrow=3, ncol=3, byrow=FALSE, dimnames=list(rnames, cnames))
print(m3)
```

Write a R program to create a Dataframes which contain details of 5 employees and display the details.

```
Employees = data.frame(Name=c("Anastasia S","Dima R","Katherine S", "JAMES A","LAURA MARTIN"),
                       Gender=c("M","M","F","F","M"),
                       Age=c(23,22,25,26,32),
                       Designation=c("Clerk","Manager","Exective","CEO","ASSISTANT"),
                       SSN=c("123-34-2346","123-44-779","556-24-433","123-98-987","679-77-576")
                       )
print("Details of the employees:")
print(Employees)
```

Write a R program to create the system's idea of the current date with and without time.

```
print("System's idea of the current date with and without time:")
print(Sys.Date())
print(Sys.time())
```

UNIT–II

Loading, Exploring and Managing Data

Working with data from files: Reading and Writing Data, Reading Data Files with read.table (), Reading in Larger Datasets with read.table.Working with relational databases. Data manipulation packages: dplyr, data.table, reshape2, tidyr, lubridate.

Reading and Writing Data

One of the important formats to store a file is in a text file. R provides various methods that one can read data from a text file.

- **read.delim()**: This method is used for reading "tab-separated value" files (".txt"). By default, point (".") is used as decimal points.
- **syntax:** read.delim(file, header = TRUE, sep = "\t", dec = ".", …)
- myData = read.delim("1.txt", header = FALSE)
- print(myData)

Output:

1 A computer science portal.

- **read.delim2()**: This method is used for reading "tab-separated value" files (".txt"). By default, point (",") is used as decimal points.
  - **Syntax:** read.delim2(file, header = TRUE, sep = "\t", dec = ",", …)

```
myData = read.delim2("1.txt", header = FALSE)
print(myData)
```

file.choose(): In R it's also possible to choose a file interactively using the function **file.choose.**

```
myFile = read.delim(file.choose(), header = FALSE)
print(myFile)
```

Output:

1 A computer science portal.

- **read_tsv()**: This method is also used for to read a tab separated ("\t") values by using the help of **readr** package.

Syntax: read_tsv(file, col_names = TRUE)

```
library(readr)
 myData = read_tsv("1.txt", col_names = FALSE)
print(myData)
```

Output:

```
# A tibble: 1 x 1
  X1
1 A computer science portal .
```

Reading one line at a time

read_lines(): This method is used for the reading line of your own choice whether it's one or two or ten lines at a time. To use this method we have to import **reader** package.

Syntax: read_lines(file, skip = 0, n_max = -1L)

```
library(readr)
myData = read_lines("1.txt", n_max = 1)
print(myData)
 myData = read_lines("1.txt", n_max = 2)
print(myData)
```

Output:

[1] "c."
[1] "c++"
[2] "java"

Reading the whole file

read_file(): This method is used for reading the whole file. To use this method we have to import reader package.

Syntax: read_lines(file)
file: the file path

program:

```
library(readr)
myData = read_file("1.txt")
print(myData)
```

Output:

[1] “cc++java”

Reading a file in a table format

Another popular format to store a file is in a tabular format. R provides various methods that one can read data from a tabular formatted data file.

read.table(): read.table() is a general function that can be used to read a file in table format. The data will be imported as a data frame.

Syntax: read.table(file, header = FALSE, sep = “”, dec = “.”)

```
myData = read.table("basic.csv")
print(myData)
```

Output:

1 Name,Age,Qualification,Address
2 Amiya,18,MCA,BBS
3 Niru,23,Msc,BLS
4 Debi,23,BCA,SBP
5 Biku,56,ISC,JJP

read.csv(): read.csv() is used for reading “comma separated value” files (“.csv”). In this also the data will be imported as a data frame.

Syntax: read.csv(file, header = TRUE, sep = “,”, dec = “.”, …)

```
myData = read.csv("basic.csv")
print(myData)
```

Output:

Name Age Qualification Address
1 Amiya 18 MCA BBS
2 Niru 23 Msc BLS
3 Debi 23 BCA SBP
4 Biku 56 ISC JJP

read.csv2(): read.csv() is used for variant used in countries that use a comma “,” as decimal point and a semicolon “;” as field separators.

Syntax: read.csv2(file, header = TRUE, sep = “;”, dec = “,”, …)

```
myData = read.csv2("basic.csv")
print(myData)
```

Output:

Name.Age.Qualification.Address
1 Amiya,18,MCA,BBS
2 Niru,23,Msc,BLS

3 Debi,23,BCA,SBP
4 Biku,56,ISC,JJP

file.choose(): You can also use **file.choose()** with **read.csv()** just like before.

```
myData = read.csv(file.choose())
print(myData)
```

Output:

```
Name Age Qualification Address
1 Amiya  18         MCA     BBS
2  Niru  23         Msc     BLS
3  Debi  23         BCA     SBP
4  Biku  56         ISC     JJP
```

read_csv(): This method is also used for to read a comma (“,”) separated values by using the help of readr package.

Syntax: read_csv(file, col_names = TRUE)

```
library(readr)
 myData = read_csv("basic.csv", col_names = TRUE)
print(myData)
```

Output:

```
Parsed with column specification:
cols(
  Name = col_character(),
  Age = col_double(),
  Qualification = col_character(),
  Address = col_character()
)
# A tibble: 4 x 4
  Name    Age Qualification Address
1 Amiya    18 MCA           BBS
2 Niru     23 Msc           BLS
3 Debi     23 BCA           SBP
4 Biku     56 ISC           JJP
```

Reading a file from the internet

It’s possible to use the functions **read.delim()**, **read.csv()** and **read.table()** to import files from the web.

```
myData = read.delim("http://www.sthda.com/upload/boxplot_format.txt")
print(head(myData))
```

Output:

```
Nom variable Group
1 IND1       10     A
2 IND2        7     A
3 IND3       20     A
4 IND4       14     A
5 IND5       14     A
6 IND6       12     A
```

Reading a CSV File

Following is a simple example of **read.csv()** function to read a CSV file available in your current working directory −

```
data <- read.csv("input.csv")
print(data)
```

When we execute the above code, it produces the following result −

```
    id,  name,   salary,  start_date,    dept
1    1   Rick    623.30   2012-01-01      IT
2    2   Dan     515.20   2013-09-23      Operations
3    3   Michelle 611.00  2014-11-15      IT
4    4   Ryan    729.00   2014-05-11      HR
5   NA   Gary    843.25   2015-03-27      Finance
6    6   Nina    578.00   2013-05-21      IT
7    7   Simon   632.80   2013-07-30      Operations
8    8   Guru    722.50   2014-06-17      Finance
```

Analyzing the CSV File

By default the **read.csv()** function gives the output as a data frame. This can be easily checked as follows. Also we can check the number of columns and rows.

```
data <- read.csv("input.csv")
print(is.data.frame(data))
print(ncol(data))
print(nrow(data))
```

When we execute the above code, it produces the following result −

```
[1] TRUE
[1] 5
[1] 8
```

Once we read data in a data frame, we can apply all the functions applicable to data frames as explained in subsequent section.

Get the maximum salary

```
# Create a data frame.
data <- read.csv("input.csv")
# Get the max salary from data frame.
sal <- max(data$salary)
print(sal)
```

When we execute the above code, it produces the following result −

```
[1] 843.25
```

Get the details of the person with max salary

We can fetch rows meeting specific filter criteria similar to a SQL where clause.

```
# Create a data frame.
data <- read.csv("input.csv")
# Get the max salary from data frame.
sal <- max(data$salary)
# Get the person detail having max salary.
retval <- subset(data, salary == max(salary))
print(retval)
```

When we execute the above code, it produces the following result −

```
   id   name  salary  start_date    dept
5   NA   Gary  843.25  2015-03-27    Finance
```

Get all the people working in IT department

```
# Create a data frame.
data <- read.csv("input.csv")
retval <- subset( data, dept == "IT")
print(retval)
```

When we execute the above code, it produces the following result −

```
   id  name   salary  start_date  dept
1   1  Rick   623.3   2012-01-01  IT
3   3  Michelle 611.0  2014-11-15 IT
6   6  Nina   578.0   2013-05-21  IT
```

Get the persons in IT department whose salary is greater than 600

```
data <- read.csv("input.csv")
info <- subset(data, salary > 600 & dept == "IT")
print(info)
```

When we execute the above code, it produces the following result −

```
   id  name   salary  start_date  dept
1   1  Rick   623.3   2012-01-01  IT
3   3  Michelle 611.0  2014-11-15 IT
```

Get the people who joined on or after 2014

```
data <- read.csv("input.csv")
retval <- subset(data, as.Date(start_date) > as.Date("2014-01-01"))
print(retval)
```

When we execute the above code, it produces the following result −

```
   id  name   salary  start_date  dept
3   3  Michelle 611.00  2014-11-15  IT
4   4  Ryan   729.00  2014-05-11  HR
5  NA  Gary   843.25  2015-03-27  Finance
8   8  Guru   722.50  2014-06-17  Finance
```

Writing into a CSV File

R can create csv file form existing data frame. The **write.csv()** function is used to create the csv file. This file gets created in the working directory.

```
data <- read.csv("input.csv")
retval <- subset(data, as.Date(start_date) > as.Date("2014-01-01"))
write.csv(retval,"output.csv")
newdata <- read.csv("output.csv")
print(newdata)
```

When we execute the above code, it produces the following result −

```
  X   id  name   salary  start_date  dept
1 3   3  Michelle 611.00  2014-11-15  IT
2 4   4  Ryan   729.00  2014-05-11  HR
3 5  NA  Gary   843.25  2015-03-27  Finance
4 8   8  Guru   722.50  2014-06-17  Finance
```

Install xlsx Package

You can use the following command in the R console to install the "xlsx" package. It may ask to install some additional packages on which this package is dependent. Follow the same command with required package name to install the additional packages.

```
install.packages("xlsx")
```

Verify and Load the "xlsx" Package

Use the following command to verify and load the "xlsx" package.

```
any(grepl("xlsx",installed.packages()))
library("xlsx")
```

When the script is run we get the following output.

```
[1] TRUE
Loading required package: rJava
Loading required package: methods
Loading required package: xlsxjars
```

Input as xlsx File

Open Microsoft excel. Copy and paste the following data in the work sheet named as sheet1.

id	name	salary	start_date	dept
1	Rick	623.3	1/1/2012	IT
2	Dan	515.2	9/23/2013	Operations
3	Michelle	611	11/15/2014	IT
4	Ryan	729	5/11/2014	HR
5	Gary	43.25	3/27/2015	Finance
6	Nina	578	5/21/2013	IT
7	Simon	632.8	7/30/2013	Operations
8	Guru	722.5	6/17/2014	Finance

Also copy and paste the following data to another worksheet and rename this worksheet to "city".

name	city
Rick	Seattle
Dan	Tampa
Michelle	Chicago
Ryan	Seattle
Gary	Houston
Nina	Boston
Simon	Mumbai
Guru	Dallas

Save the Excel file as "input.xlsx". You should save it in the current working directory of the R workspace.

Reading the Excel File

The input.xlsx is read by using the **read.xlsx()** function as shown below. The result is stored as a data frame in the R environment.

```
data <- read.xlsx("input.xlsx", sheetIndex = 1)
print(data)
```

When we execute the above code, it produces the following result −

	id,	name,	salary,	start_date,	dept
1	1	Rick	623.30	2012-01-01	IT
2	2	Dan	515.20	2013-09-23	Operations
3	3	Michelle	611.00	2014-11-15	IT
4	4	Ryan	729.00	2014-05-11	HR
5	NA	Gary	843.25	2015-03-27	Finance
6	6	Nina	578.00	2013-05-21	IT
7	7	Simon	632.80	2013-07-30	Operations
8	8	Guru	722.50	2014-06-17	Finance

XML is a file format which shares both the file format and the data on the World Wide Web, intranets, and elsewhere using standard ASCII text. It stands for Extensible Markup Language (XML). Similar to HTML it contains markup tags. But unlike HTML where the markup tag describes structure of the page, in xml the markup tags describe the meaning of the data contained into the file.

install.packages("XML")

Input Data

Create a XMl file by copying the below data into a text editor like notepad. Save the file with a **.xml** extension and choosing the file type as **all files(*.*)**.

```
<RECORDS>
<EMPLOYEE>
<ID>1</ID>
<NAME>Rick</NAME>
<SALARY>623.3</SALARY>
<STARTDATE>1/1/2012</STARTDATE>
<DEPT>IT</DEPT>
</EMPLOYEE>

<EMPLOYEE>
<ID>2</ID>
<NAME>Dan</NAME>
<SALARY>515.2</SALARY>
<STARTDATE>9/23/2013</STARTDATE>
<DEPT>Operations</DEPT>
</EMPLOYEE>

<EMPLOYEE>
<ID>3</ID>
<NAME>Michelle</NAME>
<SALARY>611</SALARY>
<STARTDATE>11/15/2014</STARTDATE>
<DEPT>IT</DEPT>
</EMPLOYEE>

<EMPLOYEE>
<ID>4</ID>
<NAME>Ryan</NAME>
<SALARY>729</SALARY>
<STARTDATE>5/11/2014</STARTDATE>
<DEPT>HR</DEPT>
</EMPLOYEE>

<EMPLOYEE>
<ID>5</ID>
<NAME>Gary</NAME>
<SALARY>843.25</SALARY>
<STARTDATE>3/27/2015</STARTDATE>
<DEPT>Finance</DEPT>
</EMPLOYEE>

<EMPLOYEE>
```

<ID>6</ID>
<NAME>Nina</NAME>
<SALARY>578</SALARY>
<STARTDATE>5/21/2013</STARTDATE>
<DEPT>IT</DEPT>
</EMPLOYEE>

<EMPLOYEE>
<ID>7</ID>
<NAME>Simon</NAME>
<SALARY>632.8</SALARY>
<STARTDATE>7/30/2013</STARTDATE>
<DEPT>Operations</DEPT>
</EMPLOYEE>

<EMPLOYEE>
<ID>8</ID>
<NAME>Guru</NAME>
<SALARY>722.5</SALARY>
<STARTDATE>6/17/2014</STARTDATE>
<DEPT>Finance</DEPT>
</EMPLOYEE>

</RECORDS>

Reading XML File

The xml file is read by R using the function **xmlParse()**. It is stored as a list in R.

```
library("XML")
library("methods")
result <- xmlParse(file = "input.xml")
print(result)
```

When we execute the above code, it produces the following result −

```
1
Rick
623.3
1/1/2012
IT

2
Dan
515.2
9/23/2013
Operations

3
Michelle
611
11/15/2014
IT

4
Ryan
729
```

5/11/2014
HR

5
Gary
843.25
3/27/2015
Finance

6
Nina
578
5/21/2013
IT

7
Simon
632.8
7/30/2013
Operations

8
Guru
722.5
6/17/2014
Finance

Get Number of Nodes Present in XML File

```
# Load the packages required to read XML files.
library("XML")
library("methods")
# Give the input file name to the function.
result <- xmlParse(file = "input.xml")
# Exract the root node form the xml file.
rootnode <- xmlRoot(result)
# Find number of nodes in the root.
rootsize <- xmlSize(rootnode)
# Print the result.
print(rootsize)
```

When we execute the above code, it produces the following result −

output

```
[1] 8
```

Details of the First Node

Let's look at the first record of the parsed file. It will give us an idea of the various elements present in the top level node.

```
# Load the packages required to read XML files.
library("XML")
library("methods")
# Give the input file name to the function.
result <- xmlParse(file = "input.xml")
# Exract the root node form the xml file.
rootnode <- xmlRoot(result)
# Print the result.
```

```
print(rootnode[1])
```
When we execute the above code, it produces the following result −
```
$EMPLOYEE
  1
  Rick
  623.3
  1/1/2012
  IT
attr(,"class")
[1] "XMLInternalNodeList" "XMLNodeList"
```
Get Different Elements of a Node
```
# Load the packages required to read XML files.
library("XML")
library("methods")
# Give the input file name to the function.
result <- xmlParse(file = "input.xml")
# Exract the root node form the xml file.
rootnode <- xmlRoot(result)
# Get the first element of the first node.
print(rootnode[[1]][[1]])
# Get the fifth element of the first node.
print(rootnode[[1]][[5]])
# Get the second element of the third node.
print(rootnode[[3]][[2]])
```
When we execute the above code, it produces the following result −
```
1
IT
Michelle
```
JSON file stores data as text in human-readable format. Json stands for JavaScript Object Notation. R can read JSON files using the rjson package.

Install rjson Package

In the R console, you can issue the following command to install the rjson package.
```
install.packages("rjson")
```
Input Data

Create a JSON file by copying the below data into a text editor like notepad. Save the file with a **.json** extension and choosing the file type as **all files(*.*)**.
```
{
  "ID":["1","2","3","4","5","6","7","8" ],
  "Name":["Rick","Dan","Michelle","Ryan","Gary","Nina","Simon","Guru" ],
  "Salary":["623.3","515.2","611","729","843.25","578","632.8","722.5" ],
  "StartDate":[ "1/1/2012","9/23/2013","11/15/2014","5/11/2014","3/27/2015","5/21/2013",
    "7/30/2013","6/17/2014"],
  "Dept":[ "IT","Operations","IT","HR","Finance","IT","Operations","Finance"]
}
```
Read the JSON File

The JSON file is read by R using the function from **JSON()**. It is stored as a list in R.
```
# Load the package required to read JSON files.
library("rjson")
# Give the input file name to the function.
result <- fromJSON(file = "input.json")
# Print the result.
```

```
print(result)
```
When we execute the above code, it produces the following result −
```
$ID
[1] "1"   "2"   "3"  "4"   "5"   "6"   "7"   "8"
$Name
[1] "Rick"    "Dan"     "Michelle" "Ryan"    "Gary"    "Nina"    "Simon"   "Guru"
$Salary
[1] "623.3"  "515.2"  "611"    "729"    "843.25" "578"    "632.8" "722.5"
$StartDate
[1] "1/1/2012"  "9/23/2013"  "11/15/2014" "5/11/2014"  "3/27/2015"  "5/21/2013"
  "7/30/2013" "6/17/2014"
$Dept
[1] "IT"        "Operations" "IT"        "HR"        "Finance"   "IT"
  "Operations" "Finance"
```
Convert JSON to a Data Frame

We can convert the extracted data above to a R data frame for further analysis using the **as.data.frame()** function.
```
# Load the package required to read JSON files.
library("rjson")
# Give the input file name to the function.
result <- fromJSON(file = "input.json")
# Convert JSON file to a data frame.
json_data_frame <- as.data.frame(result)
print(json_data_frame)
```
When we execute the above code, it produces the following result −
```
    id,  name,   salary,  start_date,    dept
1    1   Rick    623.30   2012-01-01     IT
2    2   Dan     515.20   2013-09-23     Operations
3    3   Michelle 611.00  2014-11-15     IT
4    4   Ryan    729.00   2014-05-11     HR
5   NA   Gary    843.25   2015-03-27     Finance
6    6   Nina    578.00   2013-05-21     IT
7    7   Simon   632.80   2013-07-30     Operations
8    8   Guru    722.50   2014-06-17     Finance
```
Reading in Larger Datasets with read.table

R is known to have difficulties handling large data files. Here we will explore some tips that make working with such files in R less painfull.

- If you can comfortably work with the entire file in memory, but reading the file is rather slow, consider using the data.table package and read the file with its fread function.
- If your file does not comfortably fit in memory:
    - Use sqldf if you have to stick to csv files.
    - Use a SQLite database and query it using either SQL queries or dplyr.
    - Convert your csv file to a sqlite database in order to query

Loading a large dataset: use fread() or functions from readr instead of read.xxx().
```
library("data.table")
library("readr")
```
Tto read an entire csv in memory, by default, R users use the read.table method or variations thereof (such as read.csv). However, fread from the data.table package is a lot faster. Furthermore, the readr package also provides more optimized reading functions (read_csv, read_delim,…). Let's measure the time to read in the data using these three different methods.

```
read.table.timing <- system.time(read.table(csv.name, header = TRUE, sep = ","))
readr.timing <- system.time(read_delim(csv.name, ",", col_names = TRUE))
data.table.timing <- system.time(allData <- fread(csv.name, showProgress = FALSE))
data <- data.frame(method = c('read.table', 'readr', 'fread'),
                   timing = c(read.table.timing[3], readr.timing[3], data.table.timing[3]))
## 1 read.table 183.732
## 2      readr  3.625
## 3      fread  12.564
```

Data files that don't fit in memory

If you are not able to read in the data file, because it does not fit in memory (or because R becomes too slow when you load the entire dataset), you will need to limit the amount of data that will actually be stored in memory. There are a couple of options which we will investigate:

1. limit the number of lines you are trying to read for some exploratory analysis. Once you are happy with the analysis you want to run on the entire dataset, move to another machine.
2. limit the number of columns you are reading to reduce the memory required to store the data.
3. limit both the number of rows and the number of columns using sqldf.
4. stream the data.

1. Limit the number of lines you read (fread)

Limiting the number of lines you read is easy. Just use the nrows and/or skip option (available to both read.table and fread). skip can be used to skip a number of rows, but you can also pass a string to this parameter causing fread to only start reading lines from the first line matching that string. Let's say we only want to start reading lines after we find a line matching the pattern 2015-06-12 15:14:39. We can do that like this:

```
sprintf("Number of lines in full data set: %s", nrow(allData))
## [1] "Number of lines in full data set: 3761058"
subSet <- fread(csv.name, skip = "2015-06-12 15:14:39", showProgress = FALSE)
sprintf("Number of lines in data set with skipped lines: %s", nrow(subSet))
## [1] "Number of lines in data set with skipped lines: 9998"
```

Skipping rows this way is obviously not giving you the entire dataset, so this strategy is only useful for doing exploratory analysis on a subset of your data. Note that also read_delim provides a n_max argument to limit the number of lines to read. If you want to explore the whole dataset, limiting the number of columns you read can be a more useful strategy.

2. Limit the number of columns you read (fread)

If you only need 4 columns of the 21 columns present in the file, you can tell fread to only select those 4. This can have a major impact on the memory footprint of your data. The option you need for this is: select. With this, you can specify a number of columns to keep. The opposite - specifying the columns you want to drop - can be accomplished with the drop option.

```
fourColumns = fread(csv.name, select = c("device_info_serial", "date_time", "latitude", "longitude"), showProgress = FALSE)
sprintf("Size of total data in memory: %s MB", utils::object.size(allData)/1000000)
## [1] "Size of total data in memory: 1173.480728 MB"
sprintf("Size of only four columns in memory: %s MB", utils::object.size(fourColumns)/1000000)
## [1] "Size of only four columns in memory: 105.311936 MB"
```

The difference might not be as large as you would expect. R objects claim more memory than needed to store the data alone, as they keep pointers, and other object attributes. But still, the difference could save you.

3. Limiting both the number of rows and the number of columns using sqldf

The sqldf package allows you to run SQL-like queries on a file, resulting in only a selection of the file being read. It allows you to limit both the number of lines and the number of rows at the same

time. In the background, this actually creates a sqlite database on the fly to execute the query. Consider using the package when starting from a csv file, but the actual strategy boils down to making a sqlite database file of your data.

4. Streaming data

Streaming a file means reading it line by line and only keeping the lines you need or do stuff with the lines while you read through the file. It turns out that R is really not very efficient in streaming files. The main reason is the memory allocation process that has difficulties with a constantly growing object (which can be a dataframe containing only the selected lines).

Working with relational databases

In many production environments, the data you want lives in a relational or SQL database, not in files. Public data is often in files (as they are easier to share), but your most important client data is often in databases. Relational databases scale easily to the millions of records and supply important production features such as parallelism, consistency, transactions, logging, and audits. When you're working with transaction data, you're likely to find it already stored in a relational database, as relational data- bases excel at online transaction processing (OLTP). Often you can export the data into a structured file and use the methods of our previous sections to then transfer the data into R. But this is generally not the right way to do things. Exporting from databases to files is often unreliable and idiosyn- cratic due to variations in database tools and the typically poor job these tools do when quoting and escaping characters that are confused with field separators. Data in a database is often stored in what is called a normalized form, which requires relational preparations called joins before the data is ready for analysis. Also, you often don't want a dump of the entire database, but instead wish to freely specify which columns and aggregations you need during analysis.

Loading data with SQL Screwdriver

java -classpath SQLScrewdriver.jar:h2-1.3.170.jar \ com.winvector.db.LoadFiles \ file:dbDef.xml \ , \ hus \ file:csv_hus/ss11husa.csv file:csv_hus/ss11husb.csv java -classpath SQLScrewdriver.jar:h2-1.3.170.jar \ com.winvector.db.LoadFiles \ file:dbDef.xml , pus \ file:csv_pus/ss11pusa.csv file:csv_pus/ss11pusb.csv

Loading data from a database into R

To load data from a database, we use a database connector. Then we can directly issueSQL queries from R. SQL is the most common database query language and allows usto specify arbitrary joins and aggregations. SQL is called a declarative language (asopposed to a procedural language) because in SQL we specify what relations we wouldlike our data sample to have, not how to compute them. For our example, we load asample of the household data from the hus table and the rows from the person table(pus) that are associated with those households.

```
options( java.parameters = "-Xmx2g" )
drv <- JDBC("org.h2.Driver","h2-1.3.170.jar",identifier.quote="'")
options<-";LOG=0;CACHE_SIZE=65536;LOCK_MODE=0;UNDO_LOG=0"
conn <- dbConnect(drv,paste("jdbc:h2:H2DB",options,sep=''),"u","u")
dhus <- dbGetQuery(conn,"SELECT * FROM hus WHERE ORIGRANDGROUP<=1")
dpus <- dbGetQuery(conn,"SELECT pus.* FROM pus WHERE pus.SERIALNO IN \
(SELECT DISTINCT hus.SERIALNO FROM hus \
WHERE hus.ORIGRANDGROUP<=1)")
dbDisconnect(conn)
save(dhus,dpus,file='phsample.RData')
```

And we're in business; the data has been unpacked from the Census-supplied .csv filesinto our database and a useful sample has been loaded into R for analysis. We haveactually accomplished a lot. Generating, as we have, a uniform sample of householdsand matching people would be tedious using shell tools. It's exactly what SQL data-bases are designed to do well.

Data manipulation packages

Data Manipulation is a loosely used term with 'Data Exploration'. It involves 'manipulating' data using available set of variables. This is done to enhance accuracy and precision associated with data.

1.dplyr Package

This packages is created and maintained by Hadley Wickham. This package has everything (almost) to accelerate your data manipulation efforts. It is known best for data exploration and transformation. It's chaining syntax makes it highly adaptive to use. It includes 5 major data manipulation commands:

1. filter – It filters the data based on a condition
2. select – It is used to select columns of interest from a data set
3. arrange – It is used to arrange data set values on ascending or descending order
4. mutate – It is used to create new variables from existing variables
5. summarise (with group_by) – It is used to perform analysis by commonly used operations such as min, max, mean count etc

Simple focus on these commands and do great in data exploration. Let's understand these commands one by one. I have used 2 pre-installed R data sets namely mtcars and iris.

```
> library(dplyr)
> data("mtcars")
> data('iris')
> mydata <- mtcars
#read data
> head(mydata)
```

	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
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

#creating a local dataframe. Local data frame are easier to read

```
> mynewdata <- tbl_df(mydata)
> myirisdata <- tbl_df(iris)

#now data will be in tabular structure
> mynewdata
```

```
Source: local data frame [32 x 11]

     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
   (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl)
1   21.0     6 160.0   110  3.90 2.620 16.46     0     1     4     4
2   21.0     6 160.0   110  3.90 2.875 17.02     0     1     4     4
3   22.8     4 108.0    93  3.85 2.320 18.61     1     1     4     1
4   21.4     6 258.0   110  3.08 3.215 19.44     1     0     3     1
5   18.7     8 360.0   175  3.15 3.440 17.02     0     0     3     2
6   18.1     6 225.0   105  2.76 3.460 20.22     1     0     3     1
7   14.3     8 360.0   245  3.21 3.570 15.84     0     0     3     4
8   24.4     4 146.7    62  3.69 3.190 20.00     1     0     4     2
9   22.8     4 140.8    95  3.92 3.150 22.90     1     0     4     2
10  19.2     6 167.6   123  3.92 3.440 18.30     1     0     4     4
..   ...   ...   ...   ...   ...   ...   ...   ...   ...   ...   ...
```

> myirisdata

```
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
          (dbl)       (dbl)        (dbl)       (dbl)  (fctr)
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
7           4.6         3.4          1.4         0.3  setosa
8           5.0         3.4          1.5         0.2  setosa
9           4.4         2.9          1.4         0.2  setosa
10          4.9         3.1          1.5         0.1  setosa
..          ...         ...          ...         ...     ...
```

#use filter to filter data with required condition

> filter(mynewdata, cyl > 4 & gear > 4)

```
Source: local data frame [3 x 11]

    mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
  (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl)
1  15.8     8   351   264  4.22  3.17  14.5     0     1     5     4
2  19.7     6   145   175  3.62  2.77  15.5     0     1     5     6
3  15.0     8   301   335  3.54  3.57  14.6     0     1     5     8
```

> filter(mynewdata, cyl > 4)

```
Source: local data frame [21 x 11]

     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
   (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl)
1   21.0     6 160.0   110  3.90 2.620 16.46     0     1     4     4
2   21.0     6 160.0   110  3.90 2.875 17.02     0     1     4     4
3   21.4     6 258.0   110  3.08 3.215 19.44     1     0     3     1
4   18.7     8 360.0   175  3.15 3.440 17.02     0     0     3     2
5   18.1     6 225.0   105  2.76 3.460 20.22     1     0     3     1
6   14.3     8 360.0   245  3.21 3.570 15.84     0     0     3     4
7   19.2     6 167.6   123  3.92 3.440 18.30     1     0     4     4
8   17.8     6 167.6   123  3.92 3.440 18.90     1     0     4     4
9   16.4     8 275.8   180  3.07 4.070 17.40     0     0     3     3
10  17.3     8 275.8   180  3.07 3.730 17.60     0     0     3     3
..   ...   ...   ...   ...   ...   ...   ...   ...   ...   ...   ...
```

> filter(myirisdata, Species %in% c('setosa', 'virginica'))

```
Source: local data frame [100 x 5]

   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
          (dbl)       (dbl)        (dbl)       (dbl)  (fctr)
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
7           4.6         3.4          1.4         0.3  setosa
8           5.0         3.4          1.5         0.2  setosa
9           4.4         2.9          1.4         0.2  setosa
10          4.9         3.1          1.5         0.1  setosa
..          ...         ...          ...         ...     ...
```

#use select to pick columns by name
> select(mynewdata, cyl,mpg,hp)

```
    cyl   mpg    hp
   (dbl) (dbl) (dbl)
1      6  21.0   110
2      6  21.0   110
3      4  22.8    93
4      6  21.4   110
5      8  18.7   175
6      6  18.1   105
7      8  14.3   245
8      4  24.4    62
9      4  22.8    95
10     6  19.2   123
..   ...   ...   ...
```

#here you can use (-) to hide columns
> select(mynewdata, -cyl, -mpg)

```
    disp    hp  drat    wt  qsec    vs    am  gear  carb
   (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl) (dbl)
1  160.0   110  3.90 2.620 16.46     0     1     4     4
2  160.0   110  3.90 2.875 17.02     0     1     4     4
3  108.0    93  3.85 2.320 18.61     1     1     4     1
4  258.0   110  3.08 3.215 19.44     1     0     3     1
5  360.0   175  3.15 3.440 17.02     0     0     3     2
6  225.0   105  2.76 3.460 20.22     1     0     3     1
7  360.0   245  3.21 3.570 15.84     0     0     3     4
8  146.7    62  3.69 3.190 20.00     1     0     4     2
9  140.8    95  3.92 3.150 22.90     1     0     4     2
10 167.6   123  3.92 3.440 18.30     1     0     4     4
..   ...   ...   ...   ...   ...   ...   ...   ...   ...
```

#hide a range of columns
> select(mynewdata, -c(cyl,mpg))

	disp	hp	drat	wt	qsec	vs	am	gear	carb
	(dbl)	(dbl)	(dbl)	(dbl)	(dbl)	(dbl)	(dbl)	(dbl)	(dbl)
1	160.0	110	3.90	2.620	16.46	0	1	4	4
2	160.0	110	3.90	2.875	17.02	0	1	4	4
3	108.0	93	3.85	2.320	18.61	1	1	4	1
4	258.0	110	3.08	3.215	19.44	1	0	3	1
5	360.0	175	3.15	3.440	17.02	0	0	3	2
6	225.0	105	2.76	3.460	20.22	1	0	3	1
7	360.0	245	3.21	3.570	15.84	0	0	3	4
8	146.7	62	3.69	3.190	20.00	1	0	4	2
9	140.8	95	3.92	3.150	22.90	1	0	4	2
10	167.6	123	3.92	3.440	18.30	1	0	4	4
..	...	...	...	...	...	...	...	...	...

#select series of columns

> select(mynewdata, cyl:gear)

	cyl	disp	hp	drat	wt	qsec	vs	am	gear
	(dbl)	(dbl)	(dbl)	(dbl)	(dbl)	(dbl)	(dbl)	(dbl)	(dbl)
1	6	160.0	110	3.90	2.620	16.46	0	1	4
2	6	160.0	110	3.90	2.875	17.02	0	1	4
3	4	108.0	93	3.85	2.320	18.61	1	1	4
4	6	258.0	110	3.08	3.215	19.44	1	0	3
5	8	360.0	175	3.15	3.440	17.02	0	0	3
6	6	225.0	105	2.76	3.460	20.22	1	0	3
7	8	360.0	245	3.21	3.570	15.84	0	0	3
8	4	146.7	62	3.69	3.190	20.00	1	0	4
9	4	140.8	95	3.92	3.150	22.90	1	0	4
10	6	167.6	123	3.92	3.440	18.30	1	0	4
..	...	...	...	...	...	...	...	...	...

#chaining or pipelining - a way to perform multiple operations
#in one line

```
> mynewdata %>%
    select(cyl, wt, gear)%>%
    filter(wt > 2)
```

	cyl	wt	gear
	(dbl)	(dbl)	(dbl)
1	6	2.620	4
2	6	2.875	4
3	4	2.320	4
4	6	3.215	3
5	8	3.440	3
6	6	3.460	3
7	8	3.570	3
8	4	3.190	4
9	4	3.150	4
10	6	3.440	4
..	...	...	...

#arrange can be used to reorder rows

```
> mynewdata%>%
    select(cyl, wt, gear)%>%
    arrange(wt)
```

	cyl	wt	gear
	(dbl)	(dbl)	(dbl)
1	4	1.513	5
2	4	1.615	4
3	4	1.835	4
4	4	1.935	4
5	4	2.140	5
6	4	2.200	4
7	4	2.320	4
8	4	2.465	3
9	6	2.620	4
10	6	2.770	5
..	...	...	...

```
> mynewdata %>%
    select(mpg, cyl)%>%
    mutate(newvariable = mpg*cyl)
```

	mpg	cyl	newvariable
	(dbl)	(dbl)	(dbl)
1	21.0	6	126.0
2	21.0	6	126.0
3	22.8	4	91.2
4	21.4	6	128.4
5	18.7	8	149.6
6	18.1	6	108.6
7	14.3	8	114.4
8	24.4	4	97.6
9	22.8	4	91.2
10	19.2	6	115.2
..	...	...	...

```
#or
> newvariable <- mynewdata %>% mutate(newvariable = mpg*cyl)
#summarise - this is used to find insights from data
> myirisdata%>%
     group_by(Species)%>%
     summarise(Average = mean(Sepal.Length, na.rm = TRUE))
```

	Species	Average
	(fctr)	(dbl)
1	setosa	5.006
2	versicolor	5.936
3	virginica	6.588

```
#or use summarise each
> myirisdata%>%
    group_by(Species)%>%
    summarise_each(funs(mean, n()), Sepal.Length, Sepal.Width)
```

	Species	Sepal.Length_mean	Sepal.Width_mean	Sepal.Length_n	Sepal.Width_n
	(fctr)	(dbl)	(dbl)	(int)	(int)
1	setosa	5.006	3.428	50	50
2	versicolor	5.936	2.770	50	50
3	virginica	6.588	2.974	50	50

```
#You can create complex chain commands using these 5 verbs.
#you can rename the variables using rename command
> mynewdata %>% rename(miles = mpg)
```

	miles (dbl)	cyl (dbl)	disp (dbl)	hp (dbl)	drat (dbl)	wt (dbl)	qsec (dbl)	vs (dbl)	am (dbl)	gear (dbl)	carb (dbl)
1	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
2	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
3	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
4	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
5	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
6	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
7	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
8	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
9	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
10	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
..	...	...	...	...	...	...	...	...	...	...	...

2.data.table Package

This package allows you to perform faster manipulation in a data set. Leave your traditional ways of sub setting rows and columns and use this package. With minimum coding, you can do much more. Using data.table helps in reducing computing time as compared to data.frame. You'll be astonished by the simplicity of this package.

A data table has 3 parts namely DT[i,j,by]. You can understand this as, we can tell R to subset the rows using 'i', to calculate 'j' which is grouped by 'by'. Most of the times, 'by' relates to categorical variable. In the code below, I've used 2 data sets (airquality and iris).

#load data

```
> data("airquality")
> mydata <- airquality
> head(airquality,6)
```

	Ozone	Solar.R	Wind	Temp	Month	Day
1	41	190	7.4	67	5	1
2	36	118	8.0	72	5	2
3	12	149	12.6	74	5	3
4	18	313	11.5	62	5	4
5	NA	NA	14.3	56	5	5
6	28	NA	14.9	66	5	6

```
> data(iris)
> myiris <- iris
#load package
> library(data.table)
> mydata <- data.table(mydata)
> mydata
```

	Ozone	Solar.R	Wind	Temp	Month	Day
1:	41	190	7.4	67	5	1
2:	36	118	8.0	72	5	2
3:	12	149	12.6	74	5	3
4:	18	313	11.5	62	5	4
5:	NA	NA	14.3	56	5	5

149:	30	193	6.9	70	9	26
150:	NA	145	13.2	77	9	27
151:	14	191	14.3	75	9	28
152:	18	131	8.0	76	9	29
153:	20	223	11.5	68	9	30

```
> mydata[2:4,]
```

```
   Ozone Solar.R Wind Temp Month Day
1:    36     118  8.0   72     5   2
2:    12     149 12.6   74     5   3
3:    18     313 11.5   62     5   4
```

#select columns with particular values
> myiris[Species == 'setosa']

```
    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
 7:          4.6         3.4          1.4         0.3  setosa
 8:          5.0         3.4          1.5         0.2  setosa
 9:          4.4         2.9          1.4         0.2  setosa
10:          4.9         3.1          1.5         0.1  setosa
11:          5.4         3.7          1.5         0.2  setosa
```

#select columns with multiple values. This will give you columns with Setosa
#and virginica species
> myiris[Species %in% c('setosa', 'virginica')]

#select columns. Returns a vector
> mydata[,Temp]

```
  [1] 67 72 74 62 56 66 65 59 61 69 74 69 66 68 58 64 66 57 68 62 59 73 61 61 57 58 57
 [28] 67 81 79 76 78 74 67 84 85 79 82 87 90 87 93 92 82 80 79 77 72 65 73 76 77 76 76
 [55] 76 75 78 73 80 77 83 84 85 81 84 83 83 88 92 92 89 82 73 81 91 80 81 82 84 87 85
 [82] 74 81 82 86 85 82 86 88 86 83 81 81 81 82 86 85 87 89 90 90 92 86 86 82 80 79 77
[109] 79 76 78 78 77 72 75 79 81 86 88 97 94 96 94 91 92 93 93 87 84 80 78 75 73 81 76
[136] 77 71 71 78 67 76 68 82 64 71 81 69 63 70 77 75 76 68
```

> mydata[,.(Temp,Month)]

```
     Temp Month
  1:   67     5
  2:   72     5
  3:   74     5
  4:   62     5
  5:   56     5
 ---
149:   70     9
150:   77     9
151:   75     9
152:   76     9
153:   68     9
```

#returns sum of selected column
> mydata[,sum(Ozone, na.rm = TRUE)]

[1]4887
#returns sum and standard deviation
> mydata[,.(sum(Ozone, na.rm = TRUE), sd(Ozone, na.rm = TRUE))]

```
     V1       V2
1: 4887 32.98788
```

#print and plot
> myiris[,{print(Sepal.Length)
> plot(Sepal.Width)
NULL}]

```
  [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4 5.1 5.7 5.1
 [21] 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5 4.9 5.0 5.5 4.9 4.4 5.1
 [41] 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2
 [61] 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7
 [81] 5.5 5.5 5.8 6.0 5.4 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7
[101] 6.3 5.8 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7 6.0
[121] 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7 6.3 6.4 6.0 6.9
[141] 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

#grouping by a variable
> myiris[,.(sepalsum = sum(Sepal.Length)), by=Species]

```
      Species sepalsum
1:     setosa    250.3
2: versicolor    296.8
3:  virginica    329.4
```

#select a column for computation, hence need to set the key on column
> setkey(myiris, Species)

#selects all the rows associated with this data point
> myiris['setosa']
> myiris[c('setosa', 'virginica')]

3.reshape2 Package

As the name suggests, this package is useful in reshaping data. We all know the data come in many forms. Hence, we are required to tame it according to our need. Usually, the process of reshaping data in R is tedious and worrisome. R base functions consist of 'Aggregation' option using which data can be reduced and rearranged into smaller forms, but with reduction in amount of information. Aggregation includes tapply, by and aggregate base functions. The reshape package overcome these problems. Here we try to combine features which have unique values. It has 2 functions namely *melt* and *cast*.

melt : This function converts data from wide format to long format. It's a form of restructuring where multiple categorical columns are 'melted' into unique rows. Let's understand it using the code below.

#create a data
> ID <- c(1,2,3,4,5)
> Names <- c('Joseph','Matrin','Joseph','James','Matrin')
> DateofBirth <- c(1993,1992,1993,1994,1992)
> Subject<- c('Maths','Biology','Science','Psycology','Physics')
> thisdata <- data.frame(ID, Names, DateofBirth, Subject)
> data.table(thisdata)

```
   ID  Names DateofBirth   Subject
1:  1 Joseph        1993     Maths
2:  2 Matrin        1992   Biology
3:  3 Joseph        1993   Science
4:  4  James        1994 Psycology
5:  5 Matrin        1992   Physics
```

#load package
> install.packages('reshape2')
> library(reshape2)
#melt
> mt <- melt(thisdata, id=(c('ID','Names')))
> mt

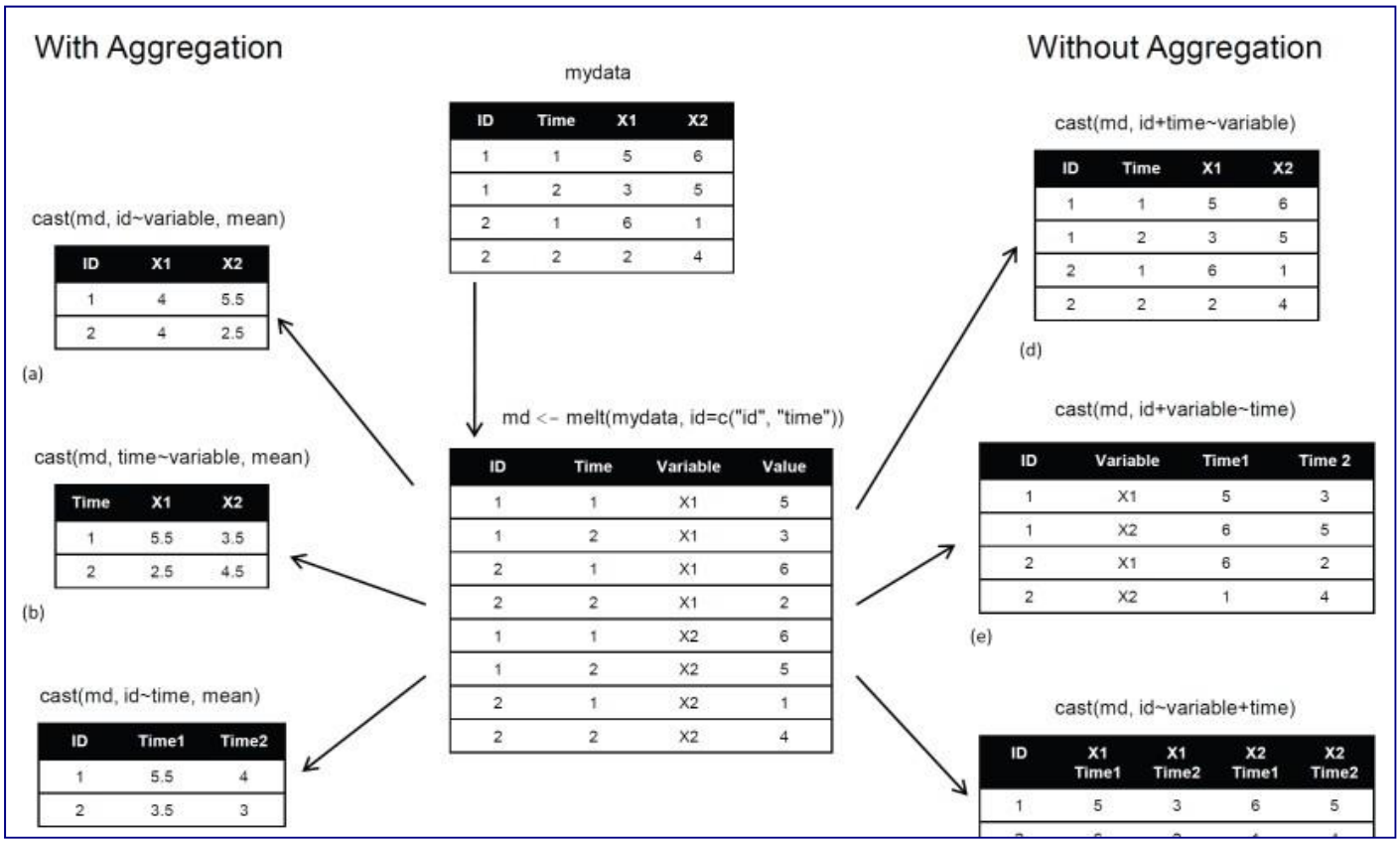
	ID	Names	variable	value
1	1	Joseph	DateofBirth	1993
2	2	Matrin	DateofBirth	1992
3	3	Joseph	DateofBirth	1993
4	4	James	DateofBirth	1994
5	5	Matrin	DateofBirth	1992
6	1	Joseph	Subject	Maths
7	2	Matrin	Subject	Biology
8	3	Joseph	Subject	Science
9	4	James	Subject	Psycology
10	5	Matrin	Subject	Physics

cast : This function converts data from long format to wide format. It starts with melted data and reshapes into long format. It's just the reverse of *melt* function. It has two functions namely, *dcast* and *acast*. dcast returns a data frame as output. acast returns a vector/matrix/array as the output. Let's understand it using the code below.

#cast
> mcast <- dcast(mt, DateofBirth + Subject ~ variable)
> mcast

	DateofBirth	Subject	DateofBirth	Subject
1	1992	Biology	1992	Biology
2	1992	Physics	1992	Physics
3	1993	Maths	1993	Maths
4	1993	Science	1993	Science
5	1994	Psycology	1994	Psycology

Note: While doing research work, I found this image which aptly describes reshape package.



4.tidyr Package

This package can make your data look 'tidy'. It has 4 major functions to accomplish this task. Needless to say, if you find yourself stuck in data exploration phase, you can use them anytime (along with dplyr). This duo makes a formidable team. They are easy to learn, code and implement. These 4 functions are:

- gather() – it 'gathers' multiple columns. Then, it converts them into key:value pairs. This function will transform wide from of data to long form. You can use it as in alternative to 'melt' in reshape package.
- spread() – It does reverse of gather. It takes a key:value pair and converts it into separate columns.
- separate() – It splits a column into multiple columns.
- unite() – It does reverse of separate. It unites multiple columns into single

columnLet's understand it closely using the code below:

```
#load package
> library(tidyr)
#create a dummy data set
> names <- c('A','B','C','D','E','A','B')
> weight <- c(55,49,76,71,65,44,34)
> age <- c(21,20,25,29,33,32,38)
> Class <- c('Maths','Science','Social','Physics','Biology','Economics','Accounts')
#create data frame
> tdata <- data.frame(names, age, weight, Class)
> tdata
```

	names	age	weight	Class
1	A	21	55	Maths
2	B	20	49	Science
3	C	25	76	Social
4	D	29	71	Physics
5	E	33	65	Biology
6	A	32	44	Economics
7	B	38	34	Accounts

#using gather function
> long_t <- tdata %>% gather(Key, Value, weight:Class)
> long_t

	names	age	Key	Value
1	A	21	weight	55
2	B	20	weight	49
3	C	25	weight	76
4	D	29	weight	71
5	E	33	weight	65
6	A	32	weight	44
7	B	38	weight	34
8	A	21	Class	Maths
9	B	20	Class	Science
10	C	25	Class	Social
11	D	29	Class	Physics
12	E	33	Class	Biology
13	A	32	Class	Economics
14	B	38	Class	Accounts

Separate function comes best in use when we are provided a date time variable in the data set. Since, the column contains multiple information, hence it makes sense to split it and use those values individually. Using the code below, I have separated a column into date, month and year.
#create a data set
> Humidity <- c(37.79, 42.34, 52.16, 44.57, 43.83, 44.59)
> Rain <- c(0.971360441, 1.10969716, 1.064475853, 0.953183435, 0.98878849, 0.939676146)
> Time <- c("27/01/2015 15:44","23/02/2015 23:24", "31/03/2015 19:15", "20/01/2015 20:52", "23/02/2015 07:46", "31/01/2015 01:55")

#build a data frame
> d_set <- data.frame(Humidity, Rain, Time)

#using separate function we can separate date, month, year
> separate_d <- d_set %>% separate(Time, c('Date', 'Month','Year'))
> separate_d

	Humidity	Rain	Date	Month	Year
1	37.79	0.9713604	27	01	2015
2	42.34	1.1096972	23	02	2015
3	52.16	1.0644759	31	03	2015
4	44.57	0.9531834	20	01	2015
5	43.83	0.9887885	23	02	2015
6	44.59	0.9396761	31	01	2015

#using unite function - reverse of separate
> unite_d <- separate_d%>% unite(Time, c(Date, Month, Year), sep = "/")
> unite_d

	Humidity	Rain	Time
1	37.79	0.9713604	27/01/2015
2	42.34	1.1096972	23/02/2015
3	52.16	1.0644759	31/03/2015
4	44.57	0.9531834	20/01/2015
5	43.83	0.9887885	23/02/2015
6	44.59	0.9396761	31/01/2015

#using spread function - reverse of gather
> wide_t <- long_t %>% spread(Key, Value)
> wide_t

	names	age	weight	Class
1	A	21	55	Maths
2	A	32	44	Economics
3	B	20	49	Science
4	B	38	34	Accounts
5	C	25	76	Social
6	D	29	71	Physics
7	E	33	65	Biology

5.Lubridate Package

Lubridate package reduces the pain of working of data time variable in R. This includes update function, duration function and date extraction.

> install.packages('lubridate')
> library(lubridate)
#current date and time
> now()
[1] "2015-12-11 13:23:48 IST"
#assigning current date and time to variable n_time
> n_time <- now()
#using update function
> n_update <- update(n_time, year = 2013, month = 10)
> n_update
[1] "2013-10-11 13:24:28 IST"
#add days, months, year, seconds
> d_time <- now()
> d_time + ddays(1)
[1] "2015-12-12 13:24:54 IST"
> d_time + dweeks(2)
[1] "2015-12-12 13:24:54 IST"
> d_time + dyears(3)
[1] "2018-12-10 13:24:54 IST"
> d_time + dhours(2)
[1] "2015-12-11 15:24:54 IST"
> d_time + dminutes(50)
[1] "2015-12-11 14:14:54 IST"
> d_time + dseconds(60)
[1] "2015-12-11 13:25:54 IST"
#extract date,time
> n_time$hour <- hour(now())
> n_time$minute <- minute(now())
> n_time$second <- second(now())
> n_time$month <- month(now())
> n_time$year <- year(now())
#check the extracted dates in separate columns

```
> new_data <- data.frame(n_time$hour, n_time$minute, n_time$second, n_time$month,
n_time$year)
> new_data
  n_time.hour n_time.minute n_time.second n_time.month n_time.year
           13            27       41.65723           12        2015
```

Write R program to illustrate working with binary file

```
# Creating a data frame
df =data.frame(
"ID"=c(1, 2, 3, 4),
  "Name"=c("Tony", "Thor", "Loki", "Hulk"),
  "Age"=c(20, 34, 24, 40),
  "Pin"=c(756083, 756001, 751003, 110011)
)

# Creating a connection object
# to write the binary file using mode "wb"
con =file("myfile.dat", "wb")

# Write the column names of the data frame
# to the connection object
writeBin(colnames(df), con)

# Write the records in each of the columns to the file
writeBin(c(df$ID, df$Name, df$Age, df$Pin), con)

# Close the connection object
close(con)
```

output:

UNIT-III

Exploratory Data Analysis and Validation Approaches
Data validation: handling missing values, null values, duplicate values, outlier detection, data cleaning, data loading and inspection, data transformation.
Cross validation: Validaton set approach, leave one out cross validation, k-fold cross validation, repeated k -fold cross validation.

DATA VALIDATION

Data validation is a critical process in data science, as it ensures the quality and reliability of your data before using it for analysis, modeling, or decision-making. In a data science context, data validation goes beyond simple checks like those used in data entry forms. It involves a deeper understanding of the data, the context of the problem, and the intended use of the data.

Why Data Validation is Essential in Data Science:

- **Accurate Results:** Invalid data can lead to incorrect conclusions and faulty models. Data validation helps ensure the accuracy of your analysis and predictions.
- **Data Integrity:** By identifying and correcting errors, inconsistencies, and outliers, you maintain the integrity of your datasets, making them more reliable for future use.
- **Time and Resource Savings:** Early detection and correction of data issues prevent wasted time and resources spent on analysis based on faulty data.
- **Reproducibility:** Validated datasets are easier to reproduce, which is crucial for scientific rigor and collaborative projects.

Types of Data Validation in Data Science:

1. **Data Type Validation:**
   - Ensures that variables are of the correct data type (e.g., numeric, categorical, date/time).
   - Checks for type mismatches, such as numeric values stored as strings.
2. **Range and Constraint Validation:**
   - Verifies that numeric values fall within expected ranges (e.g., ages, temperatures).
   - Checks that categorical values belong to a predefined set of options.
3. **Consistency Checks:**
   - Identifies logical inconsistencies within or between datasets.
   - Example: Ensuring a birth date is not later than a hire date.
4. **Outlier Detection and Handling:**
   - Identifies extreme or unusual values that could distort analysis.
   - Decides whether to remove, correct, or impute outliers based on the context.
5. **Missing Value Identification and Imputation:**
   - Detects and handles missing data appropriately.
   - Options include removing rows/columns with missing values, imputing values, or using algorithms that can handle missing data.
6. **Uniqueness Validation:**
   - Ensures that unique identifiers (e.g., ID numbers) are indeed unique.
   - Identifies and addresses duplicate records.

Tools and Techniques for Data Validation:

- **Python Libraries:**
  - Pandas: Provides powerful data manipulation and cleaning functions.
  - NumPy: Offers mathematical operations for numerical data validation.
  - Scikit-learn: Includes tools for outlier detection and missing value imputation.
- **R Libraries:**
  - dplyr, tidyr: For data manipulation and cleaning.
  - validate: A package specifically designed for data validation.
  - ggplot2: For visualizing data distributions and potential issues.
- **SQL Queries:**
  - Can be used to perform validation checks directly in a database.
- **Domain Knowledge:**
  - Understanding the context and expected values of your data is crucial for effective validation.

Handling Missing Values

Handling missing values is a crucial part of data validation in data science. Missing values can arise due to various reasons, such as data entry errors, sensor malfunctions, or incomplete surveys. Leaving them unaddressed can lead to biased analysis and incorrect conclusions.

Here are several approaches to handling missing values during data validation:

1. Deletion:

- **Listwise Deletion:** Remove entire rows or observations with missing values.
- **Pairwise Deletion:** Exclude only the specific missing values from calculations involving that variable.

Pros: Simple to implement.

Cons: Potential loss of valuable information if a significant portion of the data is missing. Can introduce bias if missing values are not random.

2. Imputation:

- **Mean/Median/Mode Imputation:** Replace missing values with the mean, median, or mode (most frequent value) of the non-missing values in that variable.
- **Regression Imputation:** Predict missing values based on other variables in the dataset.
- **Multiple Imputation:** Create multiple imputed datasets, each with different plausible values for missing data, and combine results from the analyses of these datasets.

Pros: Preserves data and can be effective if the imputation method is appropriate.

Cons: May introduce bias or reduce the variability of the data if the imputation method is not well-suited.

3. Advanced Techniques:

- **k-Nearest Neighbors (kNN) Imputation:** Impute missing values based on the values of the k most similar neighbors.
- **Maximum Likelihood Estimation (MLE):** Estimate missing values based on a statistical model that maximizes the likelihood of the observed data.

- **Expectation-Maximization (EM):** An iterative algorithm that alternates between estimating missing values and model parameters.

Pros: Can be more accurate than simpler methods.

Cons: More complex to implement and may require more computational resources.

Choosing the Right Approach

The choice of handling missing values depends on various factors:

- **Amount of missing data:** If a small percentage of data is missing, deletion might be acceptable. For larger proportions, imputation is often preferred.
- **Pattern of missingness:** If missing values are not random, simply deleting or imputing them can introduce bias. You might need more sophisticated methods like multiple imputation.
- **Type of variable:** The type of variable with missing values (categorical, numerical) influences the choice of imputation method.
- **Purpose of analysis:** The specific goals of your analysis will also guide your decision. For example, if the analysis is sensitive to outliers, you might avoid imputation methods that can introduce artificial values.

Example: Handling Missing Age in a Survey Dataset

Let's say you have a survey dataset where some respondents didn't provide their age.

- **Deletion:** If only a few respondents have missing age values, you might consider deleting those rows.
- **Mean/Median Imputation:** If the missingness is random, you could replace missing age values with the mean or median age of the respondents who provided their age.
- **Regression Imputation:** If you have other variables (e.g., income, education level) that are correlated with age, you could build a regression model to predict missing age values based on those variables.

Handling Null Values

In data science, null values represent missing or unknown data points within a dataset. They are a common occurrence in real-world data and can pose significant challenges during analysis and modeling.

Representations of Null Values:

- **NaN (Not a Number):** Commonly used in numerical datasets to represent missing or undefined values.
- **NULL or None:** Often used in databases and programming languages like SQL or Python to denote missing values.
- **Blank or Empty Cells:** In spreadsheets or flat files, null values may be represented as simply blank or empty cells.

Impact on Data Analysis:

- **Reduced Sample Size:** Null values can reduce the effective sample size of your dataset, leading to less reliable statistical analysis.
- **Biased Results:** If the missingness of data is not random, ignoring null values can introduce bias and lead to incorrect conclusions. For instance, if respondents with lower income are less likely to report their salary, your analysis might overestimate the average income.
- **Algorithm Issues:** Many machine learning algorithms cannot handle null values directly and may require preprocessing steps like imputation or removal.

Strategies for Handling Null Values:

The appropriate strategy depends on the amount of missing data, the pattern of missingness, the type of variable, and the purpose of your analysis. Here are some common approaches:

1. **Deletion:**
    - **Listwise Deletion:** Remove entire rows with any null values. Suitable when the proportion of missing data is small and the missingness is random.
    - **Pairwise Deletion:** Exclude only the missing values from specific calculations. This preserves more data but can lead to inconsistencies in results.
2. **Imputation:**
    - **Mean/Median/Mode Imputation:** Replace missing values with the mean, median, or mode of the non-missing values for that variable. Simple but may introduce bias if missingness is not random.
    - **Regression Imputation:** Predict missing values based on other variables using regression models. Can be effective if there are strong relationships between variables.
    - **Multiple Imputation:** Create multiple imputed datasets, each with different plausible values for missing data, and combine results from the analyses. This is a more robust approach that accounts for the uncertainty in imputation.
    - **K-Nearest Neighbors (kNN) Imputation:** Impute missing values based on the values of the k most similar neighbors.
3. **Advanced Techniques:**
    - **Maximum Likelihood Estimation (MLE):** Estimate missing values based on a statistical model that maximizes the likelihood of the observed data.
    - **Expectation-Maximization (EM):** An iterative algorithm that alternates between estimating missing values and model parameters.

Example: Handling Null Values in House Price Prediction

If you are building a model to predict house prices, and you have some null values in the "square footage" variable, you have a few options:

- **Deletion:** Remove rows with missing square footage. This may be acceptable if only a few rows are affected.
- **Imputation:**
    - **Mean/Median Imputation:** Replace missing values with the average or median square footage of similar houses (e.g., those with the same number of bedrooms and bathrooms).
    - **Regression Imputation:** Build a model to predict square footage based on other features like the number of bedrooms, bathrooms, and location.

Duplicate Values

In data science, duplicate values refer to the occurrence of identical or nearly identical records within a dataset. These duplicates can arise due to various reasons, such as data entry errors, multiple data sources, or intentional data collection methods.

Impact of Duplicate Values on Data Analysis:

- **Overrepresentation:** Duplicate values can lead to overrepresentation of certain observations, skewing statistical analysis and potentially leading to biased conclusions.
- **Model Overfitting:** In machine learning, duplicate values can cause models to overfit to specific patterns in the data, resulting in poor generalization to new data.
- **Misleading Metrics:** Duplicates can artificially inflate metrics like counts or sums, making it difficult to assess the true distribution of the data.

Identifying Duplicate Values:

1. **Exact Duplicates:**
    - **Python:** Use the `duplicated()` method in pandas to identify exact duplicate rows.
    - **SQL:** Use the `DISTINCT` keyword or `GROUP BY` clause to find duplicates in a database table.
2. **Near Duplicates:**
    - **Fuzzy Matching:** Use techniques like fuzzy string matching or similarity measures to identify near-duplicates, where values might differ slightly due to typos or variations.

Handling Duplicate Values:

The approach to handling duplicates depends on the context and the reasons for their occurrence.

1. **Removal:**
    - **Drop Duplicates:** If duplicates are true errors, remove them using methods like `drop_duplicates()` in pandas or `DELETE` statements in SQL.
    - **Keep First/Last:** If duplicates represent valid but redundant information, you can keep the first or last occurrence of a duplicate.
2. **Aggregation:**
    - **Group and Aggregate:** If duplicates represent multiple measurements of the same entity, you can group the data by unique identifiers and aggregate relevant columns (e.g., sum, average).
3. **Investigation:**
    - **Analyze Causes:** If duplicates are unexpected, investigate the data collection process to understand the reasons for their occurrence.
    - **Correct Errors:** If duplicates are due to errors, correct the data at the source.

Example: Handling Duplicate Customer Records

Imagine you have a customer dataset where some customers appear multiple times due to data entry errors. You could:

1. **Identify Duplicates:** Use pandas to identify rows with identical values in key fields like customer ID or email.
2. **Investigate:** Look for patterns in the duplicates to understand why they occurred.

3. **Remove Duplicates:** Drop the duplicate rows, keeping only one unique record for each customer.
4. **Correct Errors:** If the duplicates were caused by typos or inconsistencies in data entry, correct the original data source.

Tools for Handling Duplicates:

- **Python:** pandas library (e.g., `duplicated()`, `drop_duplicates()`)
- **SQL:** `DISTINCT`, `GROUP BY`, `DELETE` statements
- **R:** dplyr package (e.g., `distinct()`, `group_by()`, `summarise()`)

Outlier Detection, Data Cleaning

1. Outlier Detection

Outliers are data points that significantly deviate from the rest of your dataset. They can be legitimate extreme values or errors that crept into your data during collection or processing. Detecting and handling outliers is crucial because they can distort analysis results and undermine model performance.

Types of Outliers:

- **Global Outliers:** Data points that are significantly different from the overall distribution of the data.
- **Contextual Outliers:** Data points that are unusual within a specific context or subgroup of the data.
- **Collective Outliers:** A subset of data points that, when considered together, deviate significantly from the rest of the data.

Outlier Detection Techniques:

- **Statistical Methods:**
  - **Z-score:** Calculates how many standard deviations a data point is from the mean. Points beyond a certain threshold (e.g., +/- 3 standard deviations) are considered outliers.
  - **Modified Z-score:** Similar to Z-score but more robust to outliers in the data itself.
  - **Interquartile Range (IQR):** Outliers are points that fall below Q1 - 1.5 * IQR or above Q3 + 1.5 * IQR.
  - **Statistical tests:** Grubbs' test, Dixon's Q test, etc., for formally testing if a data point is an outlier.
- **Visual Methods:**
  - **Boxplots:** Visually identify outliers as points outside the whiskers.
  - **Scatterplots:** Outliers can be visually spotted as isolated points.
- **Machine Learning:**
  - **Isolation Forest:** A tree-based algorithm that isolates outliers by randomly partitioning the data.
  - **Local Outlier Factor (LOF):** Measures the local density deviation of a data point compared to its neighbors.

2. Data Cleaning

Data cleaning is the process of identifying and rectifying errors, inconsistencies, and inaccuracies in your data. It's a crucial preprocessing step in data science to ensure the reliability and quality of your analysis.

Common Data Cleaning Tasks:

- **Handling Missing Values:**
  - **Deletion:** Remove rows or columns with missing values.
  - **Imputation:** Replace missing values with estimated values (mean, median, mode, regression, k-NN, etc.).
- **Handling Outliers:** (As discussed above)
- **Removing Duplicates:** Identify and remove identical or near-identical records.
- **Correcting Errors:** Fix typos, incorrect data types, inconsistencies in units, and violations of business rules.
- **Standardization/Normalization:** Scale numerical data to have a mean of 0 and standard deviation of 1 or to a specific range (e.g., 0-1).
- **Data Transformation:** Apply transformations (e.g., log, square root) to make the data more suitable for analysis.
- **Feature Engineering:** Create new features from existing ones to improve model performance.

Tools and Libraries for Outlier Detection and Data Cleaning:

- **Python:** Pandas, NumPy, SciPy, Scikit-learn
- **R:** dplyr, tidyr, outliers package
- **SQL:** For cleaning data directly in databases

Example: Analyzing Customer Data

1. **Outlier Detection:** Identify customers with unusual spending patterns or extreme demographic values.
2. **Data Cleaning:**
   - Impute missing income values using the median income for similar demographics.
   - Correct invalid zip codes or email addresses.
   - Remove duplicate customer records.
   - Standardize spending amounts for comparison across customers.

Data Loading And Inspection

In data science, data loading and inspection are fundamental steps in the data analysis pipeline. They involve importing data from various sources and examining its structure, content, and quality to prepare it for further analysis and modeling.

Data Loading

Data loading is the process of bringing data from external sources into your data analysis environment. Common sources include:

- **Files:** CSV, Excel, JSON, XML, etc.
- **Databases:** SQL databases (MySQL, PostgreSQL), NoSQL databases (MongoDB, Cassandra).

- **APIs:** Web APIs that provide structured data (e.g., Twitter API, Google Maps API).
- **Cloud Storage:** Amazon S3, Google Cloud Storage, Azure Blob Storage.

Tools for Data Loading

- **Python:**
  - `pandas`: The primary library for data loading and manipulation in Python. It provides functions like `read_csv`, `read_excel`, `read_json`, `read_sql`, and more.
  - `sqlalchemy`: For connecting to and querying SQL databases.
  - `requests`: For interacting with web APIs.
- **R:**
  - `readr`, `readxl`: For reading files.
  - `DBI`: For connecting to databases.
  - `httr`: For working with APIs.
- **Other Tools:**
  - **Database Clients:** Tools like SQL Server Management Studio or MySQL Workbench can be used to directly load data from databases.
  - **Cloud Storage Clients:** Cloud providers offer tools to access and download data from their storage services.

Data Inspection

Data inspection involves examining the loaded data to understand its characteristics, identify potential issues, and prepare it for cleaning and analysis. Common inspection tasks include:

- **Displaying Data:**
  - Viewing the first few rows (e.g., using `df.head()` in pandas) to get a quick overview of the data.
  - Checking the dimensions (number of rows and columns) of the dataset.
  - Displaying column names to understand the variables present.
- **Checking Data Types:**
  - Verifying that each column is of the expected data type (numeric, categorical, date/time).
  - Identifying any columns that might need type conversion (e.g., converting string dates to datetime objects).
- **Summarizing Data:**
  - Calculating descriptive statistics (mean, median, standard deviation, etc.) for numerical variables.
  - Getting frequency counts for categorical variables.
- **Detecting Missing Values:**
  - Identifying columns or rows with missing data.
  - Deciding on a strategy to handle missing values (deletion, imputation, etc.).
- **Identifying Outliers:**
  - Checking for values that are far from the rest of the data distribution.
  - Investigating the cause of outliers (errors, unusual events, natural variation).
- **Checking for Duplicates:**
  - Identifying and potentially removing duplicate rows.
- **Data Visualization:**
  - Creating histograms, scatterplots, boxplots, or other visualizations to explore the distribution and relationships between variables.

Tools for Data Inspection:

- **Python:**
  - `pandas`: Provides functions like `describe()`, `info()`, `isnull()`, `value_counts()`, and many others for data inspection.
  - `matplotlib`, `seaborn`: For creating visualizations.
- **R:**
  - `summary()`, `str()`, `table()`: For summary statistics and data type information.
  - `ggplot2`: For creating visualizations.

Data Transformation

Data transformation is a crucial step in data science where you modify or convert raw data into a format that is more suitable for analysis, modeling, or visualization. It's a key component of the data preprocessing pipeline, helping to improve the quality and usability of your data.

Why Data Transformation is Important:

- **Feature Engineering:** Creating new features or modifying existing ones to improve the performance of machine learning models.
- **Data Cleaning:** Addressing issues like missing values, outliers, or inconsistent formats.
- **Normalization and Scaling:** Transforming data to a common scale to ensure fair comparison and prevent certain features from dominating others in models.
- **Dimensionality Reduction:** Reducing the number of features while retaining essential information. This can simplify models, improve performance, and reduce computational complexity.
- **Data Integration:** Combining data from multiple sources and converting them into a unified format for analysis.

Common Data Transformation Techniques:

1. **Scaling:**
   - **Standardization:** Transforms data to have a mean of 0 and a standard deviation of 1. Useful for algorithms that are sensitive to the scale of features, like linear regression and k-means clustering.
   - **Normalization:** Rescales data to a specific range (e.g., 0 to 1). Useful when the distribution of the data is not Gaussian.
2. **Encoding:**
   - **One-Hot Encoding:** Converts categorical variables into binary (0/1) dummy variables. Useful for algorithms that require numerical input.
   - **Label Encoding:** Assigns a unique numerical label to each category. Suitable for ordinal variables where the order of categories matters.
   - **Ordinal Encoding:** Converts ordinal categorical variables into numerical representations while preserving their order.
3. **Log Transformation:**
   - **Log Transformation:** Compresses the range of skewed data, making it more normally distributed. Useful for features with a long tail, such as income or population.
4. **Power Transformation:**
   - **Box-Cox Transformation:** Finds an optimal power transformation to make data more normally distributed.
   - **Yeo-Johnson Transformation:** Similar to Box-Cox but can handle negative values.
5. **Aggregation:**
   - **Grouping:** Combining data points based on a categorical variable and calculating summary statistics (mean, sum, count, etc.).

    - **Rolling:** Calculating statistics over a moving window of time or observations.
6. **Discretization:**
    - **Binning:** Dividing continuous data into discrete intervals or bins. Can help capture non-linear relationships.
7. **Feature Creation:**
    - **Creating Interaction Terms:** Combining two or more features to capture their combined effect on the target variable.
    - **Polynomial Features:** Adding polynomial terms (e.g., x^2, x^3) to capture non-linear relationships.

Tools and Libraries for Data Transformation:

- **Python:** Pandas, NumPy, Scikit-learn
- **R:** dplyr, tidyr, caret

Example: Transforming Housing Data

- **Scaling:** Standardize the "square footage" and "number of bedrooms" features to have a mean of 0 and a standard deviation of 1.
- **Encoding:** One-hot encode categorical features like "neighborhood" and "house style."
- **Log Transformation:** Apply a log transformation to the "price" feature, which is typically right-skewed.
- **Feature Creation:** Create a new feature "age of house" by subtracting the year built from the current year.

Cross-validation

Cross-validation (CV) is a resampling technique widely used in data science to assess the performance of machine learning models, especially in cases where you have limited data. It helps to avoid overfitting and provides a more reliable estimate of how well your model will generalize to new, unseen data.

Key Idea:

The core idea behind cross-validation is to divide your dataset into multiple subsets (folds). You then train your model on a combination of these folds and evaluate its performance on the remaining fold. This process is repeated multiple times, using a different fold as the test set each time. Finally, you average the performance metrics across all folds to get a more robust estimate of your model's performance.

Types of Cross-Validation:

1. **K-Fold Cross-Validation:**
    - The most common type.
    - Divides the dataset into K equally sized folds.
    - Trains the model on K-1 folds and tests it on the remaining fold.
    - Repeats this process K times, using each fold as the test set once.
    - Averages the performance metrics across all K folds.
2. **Stratified K-Fold Cross-Validation:**
    - Similar to K-fold but ensures that the distribution of classes (in classification problems) is preserved in each fold. This is particularly important when dealing with imbalanced datasets.

3. **Leave-One-Out Cross-Validation (LOOCV):**
    - A special case of K-fold where K is equal to the number of data points.
    - Each fold consists of a single data point.
    - Computationally expensive for large datasets.
4. **Leave-P-Out Cross-Validation:**
    - A generalization of LOOCV where you leave out P data points for testing.
    - Computationally even more expensive than LOOCV.
5. **Time Series Cross-Validation:**
    - Specifically designed for time series data.
    - Splits the data into training and test sets based on time, preserving the temporal order.

Benefits of Cross-Validation:

- **Reduced Overfitting:** By evaluating the model on multiple test sets, cross-validation reduces the risk of overfitting.
- **Better Generalization Estimate:** Provides a more reliable estimate of how the model will perform on unseen data.
- **Efficient Use of Data:** Makes better use of limited data compared to a simple train-test split.
- **Hyperparameter Tuning:** Can be used to select the best hyperparameters for your model by comparing performance across different folds.

Applications in Data Science:

- **Model Selection:** Choose the best model among different algorithms or configurations.
- **Model Evaluation:** Assess the performance of your final model on unseen data.
- **Feature Selection:** Determine which features are most important for your model.

UNIT–IV

Modelling Methods
Supervised: Regression Analysis in R, linear regression, logistic regression,naive bayes classifier, decision tree, random forest, knn classifier,
Unsupervised: kmeans clustering, association rule mining, apriori algorithm.

Supervised and unsupervised learning are two fundamental paradigms in data science and machine learning. They differ primarily in how they approach the learning process and the types of tasks they are suited for.

Supervised Learning

- **Data:** Uses labeled data, where each input data point has a corresponding output label or value.
- **Goal:** Learn a function that maps inputs to outputs, enabling predictions on new, unseen data.
- **Examples:**
    - **Classification:** Predicting if an email is spam or not (label is "spam" or "not spam").
    - **Regression:** Predicting the price of a house based on its features (label is the house price).

Unsupervised Learning

- **Data:** Uses unlabeled data, where data points do not have associated labels.
- **Goal:** Discover patterns, relationships, or structures in the data.
- **Examples:**
    - **Clustering:** Grouping similar customers based on their purchasing behavior.
    - **Dimensionality Reduction:** Finding a lower-dimensional representation of high-dimensional data.
    - **Association Rule Mining:** Discovering relationships between items in a shopping cart (e.g., people who buy bread also buy butter).

Regression Analysis

What is Regression Analysis?

Regression analysis is a statistical technique used to model the relationship between a dependent variable (the outcome you want to predict) and one or more independent variables (predictors). It's a cornerstone of supervised machine learning, where you have labeled data to train your model.

Types of Regression in R

R provides extensive support for various regression techniques, including:

- **Linear Regression (lm):** Models a linear relationship between the dependent and independent variables.
- **Generalized Linear Models (glm):** Extends linear regression to accommodate non-normal response distributions (e.g., logistic regression for binary outcomes, Poisson regression for count data).

- **Polynomial Regression:** Models non-linear relationships using polynomial functions of the predictors.
- **Robust Regression:** Deals with outliers by using robust estimation techniques.
- **Ridge Regression and Lasso Regression:** Linear regression methods that add regularization terms to prevent overfitting.
- **Elastic Net Regression:** Combines the features of Ridge and Lasso regression.

Key Steps in Regression Analysis in R

1. **Data Preparation:**
    - Load your data into an R data frame.
    - Clean the data: Handle missing values, outliers, and inconsistencies.
    - Explore the data: Visualize relationships between variables using scatterplots, histograms, etc.
2. **Model Building:**
    - Choose the appropriate regression type based on your data and research question.
    - Fit the model using functions like `lm()` or `glm()`.
    - Specify the formula, which defines the relationship between the dependent and independent variables.
3. **Model Evaluation:**
    - Assess the model's goodness of fit using metrics like R-squared, adjusted R-squared, AIC, or BIC.
    - Check model assumptions (e.g., linearity, normality of residuals, homoscedasticity) using diagnostic plots.
4. **Model Interpretation:**
    - Examine the estimated coefficients to understand the direction and magnitude of the relationship between predictors and the outcome.
    - Use confidence intervals and p-values to assess the statistical significance of the coefficients.
5. **Prediction (Optional):**
    - If your goal is prediction, use the fitted model to predict the outcome for new values of the predictors.

Example: Linear Regression in R

Code snippet

```
# Load required library
library(datasets)

# Load and explore the data (using built-in 'mtcars' dataset)
data(mtcars)
head(mtcars)

# Fit a linear regression model
model <- lm(mpg ~ wt + hp, data = mtcars)
# mpg (miles per gallon) is predicted based on weight (wt) and horsepower (hp)

# View the model summary
summary(model)

# Make predictions on new data (e.g., a car with wt = 2.5 and hp = 120)
new_data <- data.frame(wt = 2.5, hp = 120)
predicted_mpg <- predict(model, new_data)
```

Use code with caution.

content_copy

Additional Libraries in R

- `caret`: For streamlined model training, evaluation, and tuning.
- `glmnet`: For Ridge, Lasso, and Elastic Net regression.
- `MASS`: For robust regression.

Naive Bayes

What is Naive Bayes?

Naive Bayes is a probabilistic machine learning algorithm based on Bayes' Theorem. It's called "naive" because it makes a strong assumption: it assumes that all features (attributes) in your data are conditionally independent of each other given the class label. This means that the presence of one feature doesn't affect the presence of another when you know the class. While this assumption may not always hold true in real-world data, it often works surprisingly well in practice.

How Naive Bayes Works:

1. **Training:**
    - The algorithm learns the probabilities of each feature value occurring within each class.
    - It also learns the prior probabilities of each class (how likely each class is to occur overall).
2. **Prediction:**
    - Given a new data point, Naive Bayes calculates the probability of it belonging to each class using Bayes' Theorem:

```
P(class | features) = (P(features | class) * P(class)) / P(features)
```

- Where:
    - P(class | features) is the posterior probability (the probability of the class given the features).
    - P(features | class) is the likelihood (the probability of the features given the class).
    - P(class) is the prior probability of the class.
    - P(features) is the evidence (the probability of the features occurring at all).
- It then predicts the class with the highest posterior probability.

Types of Naive Bayes:

- **Gaussian Naive Bayes:** Assumes that continuous features follow a normal (Gaussian) distribution.
- **Multinomial Naive Bayes:** Suitable for discrete data (e.g., text classification, where features represent word counts).
- **Bernoulli Naive Bayes:** Used for binary data (features are either present or absent).

Advantages:

- **Simple and Easy to Implement:** The algorithm is easy to understand and code.
- **Fast:** It's computationally efficient, making it suitable for large datasets.
- **Works well with high-dimensional data:** It can handle datasets with many features.
- **Good for text classification:** Commonly used for tasks like spam filtering and sentiment analysis.

Disadvantages:

- **Naive Assumption of Independence:** The assumption that features are independent may not hold true in real-world data.
- **Zero Frequency Problem:** If a feature value is not seen in the training data, it will have a zero probability, which can affect the model's predictions.

Example: Text Classification in R

Code snippet

```
library(e1071)

# Assuming you have your data in a data frame called 'data' with a text column
'text' and a category column 'label'

# Train the model
model <- naiveBayes(label ~ text, data = data)

# Make predictions on new data (e.g., a data frame called 'new_data')
predictions <- predict(model, new_data)
```

Use code with caution.
content_copy

Applications in Data Science:

- **Spam filtering**
- **Sentiment analysis**
- **Document classification**
- **Medical diagnosis**
- **Fraud detection**

Decision Trees

What are Decision Trees?

A decision tree is a versatile supervised machine learning algorithm used for both classification and regression tasks. It's a flowchart-like model where each internal node represents a feature (or attribute), each branch represents a decision rule based on that feature, and each leaf node represents the outcome or prediction.

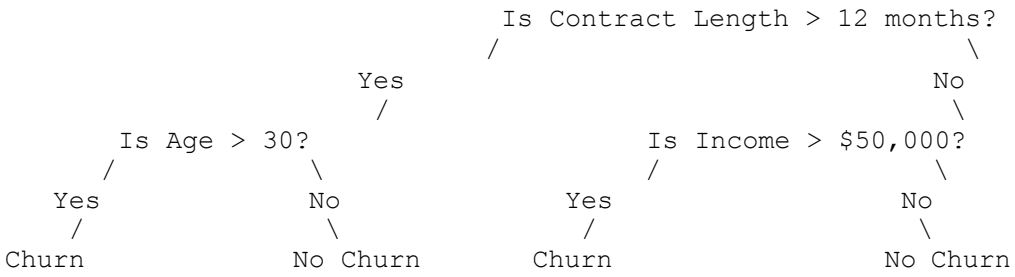
How Decision Trees Work:

1. **Recursive Partitioning:**
    - The algorithm starts at the root node, which represents the entire dataset.
    - It selects the feature that best splits the data based on a purity criterion (like Gini impurity or information gain).
    - The data is then split into subsets based on the values of the selected feature.
    - This process is repeated recursively on each subset until a stopping criterion is met (e.g., all data points in a node belong to the same class, or a maximum tree depth is reached).
2. **Prediction:**
    - To predict the outcome for a new data point, start at the root node and follow the branches based on the feature values of the data point until you reach a leaf node.
    - The value in the leaf node represents the predicted outcome.

Example: Predicting Customer Churn

Imagine you have a dataset of customers with features like age, income, and contract length. You want to build a decision tree to predict whether a customer will churn (stop using your service).

The decision tree might look like this:



Advantages of Decision Trees:

- **Interpretability:** Decision trees are easy to visualize and understand, making them valuable for explaining model predictions to stakeholders.
- **Handles Both Categorical and Numerical Data:** You don't need to preprocess categorical features.
- **Non-Linear Relationships:** Can capture complex non-linear relationships between features and the target variable.
- **Feature Importance:** The algorithm implicitly performs feature selection by prioritizing the most informative features at the top of the tree.

Disadvantages of Decision Trees:

- **Overfitting:** Prone to overfitting, especially with deep trees. Pruning techniques can help mitigate this.
- **Instability:** Small changes in the data can lead to significantly different tree structures.
- **Greedy Algorithm:** The algorithm makes locally optimal decisions at each node, which may not lead to the globally optimal tree.

Libraries for Decision Trees in R:

- `rpart`: For recursive partitioning and regression trees.
- `party`: For conditional inference trees.
- `C50`: For generating C5.0 decision trees and rule-based models.
- `tree`: For classification and regression trees.

Example: Decision Tree in R

Code snippet

```
library(rpart)
# Assuming you have a data frame 'data' with a target variable 'churn'

model <- rpart(churn ~ age + income + contract_length, data = data)
summary(model)

# Visualize the decision tree
plot(model)
text(model)
```

Random Forest

What is Random Forest?

A Random Forest is a supervised machine learning algorithm that combines the output of multiple decision trees to reach a single, more accurate prediction. It's a type of ensemble learning, where the predictions of several individual models are combined to produce a final result. The "randomness" in Random Forest comes from two main sources:

1. **Bootstrapping:** Each decision tree in the forest is trained on a random subset (bootstrap sample) of the original training data. This introduces diversity among the trees, as each one sees slightly different data.
2. **Feature Randomness:** At each node of the decision tree, the algorithm randomly selects a subset of the available features for splitting. This further decorrelates the trees and helps to prevent overfitting.

How Random Forest Works:

1. **Building the Forest:** The algorithm creates multiple decision trees (the number is a hyperparameter). Each tree is trained on a different bootstrap sample of the data, and at each node, a random subset of features is considered for splitting.
2. **Making Predictions (Classification):**
    - Each tree makes a prediction (e.g., the class label for a new data point).
    - The final prediction is determined by majority vote (the class that receives the most votes among the trees).
3. **Making Predictions (Regression):**
    - Each tree makes a prediction (e.g., a numerical value).
    - The final prediction is the average of the predictions from all the trees.

Advantages of Random Forest:

- **High Accuracy:** Often achieves state-of-the-art performance in both classification and regression tasks.
- **Handles Large Datasets and High-Dimensional Data:** Can handle datasets with many features and observations.
- **Reduces Overfitting:** The randomness in the algorithm helps to reduce overfitting and improve generalization to new data.
- **Robust to Noise and Outliers:** Less sensitive to outliers and noisy data compared to individual decision trees.
- **Provides Feature Importance:** Gives an estimate of the importance of each feature in making predictions.

Disadvantages of Random Forest:

- **Complexity:** Can be more complex to interpret compared to individual decision trees.
- **Computational Cost:** Training multiple trees can be computationally expensive, especially for large datasets.

Applications in Data Science:

- **Classification:** Identifying customer churn, detecting fraud, image classification.
- **Regression:** Predicting house prices, stock prices, or customer lifetime value.

- **Recommendation Systems:** Recommending products or movies to users.
- **Anomaly Detection:** Identifying unusual patterns or outliers in data.

Example: Random Forest in R

Code snippet

```
library(randomForest)

# Assuming you have your data in a data frame called 'data' with a target variable 'label'

# Train the model
model <- randomForest(label ~ ., data = data)

# Make predictions on new data (e.g., a data frame called 'new_data')
predictions <- predict(model, new_data)
```

Knn Classifiers

What is KNN?

KNN is a supervised machine learning algorithm commonly used for both classification and regression tasks. It's a non-parametric algorithm, meaning it doesn't make any assumptions about the underlying data distribution. Instead, it relies on the concept of similarity (or distance) between data points.

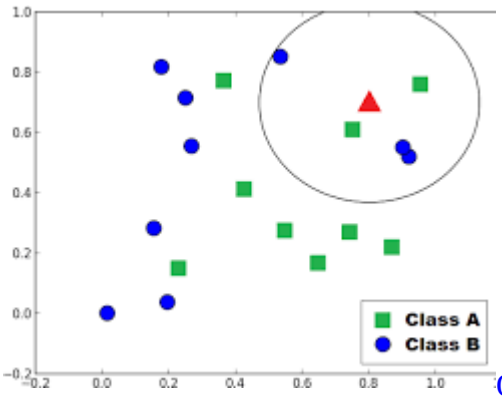
How KNN Works (Classification):

1. **Choose K:** The first step is to choose the value of K, which represents the number of nearest neighbors to consider.
2. **Calculate Distances:** For a new data point you want to classify, calculate the distance between this point and all other points in the training dataset. Common distance measures include Euclidean distance, Manhattan distance, and others.
3. **Identify K-Nearest Neighbors:** Select the K data points closest to the new data point based on the calculated distances.
4. **Majority Voting:** In classification, the new data point is assigned the class label that is most common among its K nearest neighbors.

How KNN Works (Regression):

1. **Choose K:** Similar to classification, you choose the value of K.
2. **Calculate Distances:** Calculate the distance between the new data point and all other points in the training dataset.
3. **Identify K-Nearest Neighbors:** Select the K closest data points.
4. **Average or Weighted Average:** The prediction for the new data point is either the average or weighted average of the target values of its K nearest neighbors.

Illustrative Diagram:



Opens in a new window www.researchgate.net
scatter plot of data points with different classes, highlighting the knearest neighbors of a new data point.

Advantages of KNN:

- **Simple and Easy to Understand:** The concept of KNN is intuitive and easy to grasp.
- **No Training Time:** The model doesn't need to be explicitly trained; it simply stores the training data.
- **Versatile:** Can be used for both classification and regression.
- **Non-Parametric:** Doesn't make assumptions about the data distribution.
- **Works Well with Small Datasets:** Can be effective even with limited data.

Disadvantages of KNN:

- **Computationally Expensive:** Calculating distances to all data points for each prediction can be computationally expensive, especially for large datasets.
- **Sensitive to Irrelevant Features:** If your dataset has many irrelevant features, KNN performance can suffer. Feature selection or dimensionality reduction might be necessary.
- **Sensitive to the Scale of Features:** Features with larger ranges can dominate the distance calculations. Scaling or normalization of features is often needed.
- **Choosing the Right K:** The value of K can significantly impact model performance. It's often determined through cross-validation.

Common Applications of KNN:

- **Recommendation Systems:** Recommending items based on user similarity.
- **Image Classification:** Classifying images based on similar features.
- **Anomaly Detection:** Identifying outliers or unusual patterns in data.

Example: KNN in R (Classification)

Code snippet

```
library(class)

# Assuming you have your data in a data frame called 'data' with a target variable 'label'

# Split the data into training and testing sets
train_indices <- sample(1:nrow(data), 0.7 * nrow(data))
train_data <- data[train_indices, ]
test_data <- data[-train_indices, ]

# Train the KNN model (no explicit training step for KNN)
```

```
# Make predictions on the test set (using k = 5)
predictions <- knn(train_data[, -ncol(train_data)], test_data[, -
ncol(test_data)], train_data$label, k = 5)

# Evaluate accuracy
accuracy <- mean(predictions == test_data$label)
```

K-means Clustering

What is K-means Clustering?

K-means clustering is an algorithm that aims to partition a dataset into a pre-defined number of clusters (denoted by 'K') in such a way that:

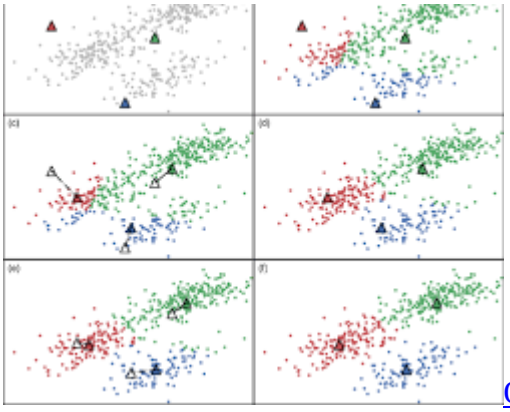
- Data points within a cluster are similar to each other.
- Data points in different clusters are dissimilar.

The similarity between data points is usually measured using distance metrics like Euclidean distance or Manhattan distance.

How K-means Clustering Works:

1. **Initialization:** Randomly choose K points as initial cluster centers (centroids).
2. **Assignment:** Assign each data point to the nearest centroid based on the chosen distance metric.
3. **Update:** Recalculate the centroids by taking the mean (average) of all the data points assigned to each cluster.
4. **Iteration:** Repeat steps 2 and 3 until the centroids no longer change significantly or a maximum number of iterations is reached.

Illustrative Diagram:


Opens in a new window www.researchgate.net
Kmeans clustering diagram showing data points assigned to different clusters and centroids being updated iteratively.

Choosing the Value of K:

The optimal value of K is not always obvious and often requires experimentation. Common methods include:

- **Elbow Method:** Plot the sum of squared distances (within-cluster sum of squares - WCSS) against different values of K. The "elbow" point where the WCSS starts to decrease more slowly is often a good choice for K.
- **Silhouette Analysis:** Measures how well each data point fits within its assigned cluster. Higher silhouette scores indicate better clustering quality.

Advantages of K-means Clustering:

- **Simple and Easy to Understand:** The algorithm is relatively straightforward and easy to implement.
- **Efficient:** It scales well to large datasets.
- **Versatile:** Can be applied to various types of data.

Disadvantages of K-means Clustering:

- **Requires Specifying K in Advance:** You need to pre-determine the number of clusters, which may not always be known.
- **Sensitive to Initialization:** The initial choice of centroids can affect the final clustering results. Running the algorithm multiple times with different initializations can help mitigate this.
- **Assumes Spherical Clusters:** K-means works best when clusters are roughly spherical and of similar sizes.
- **Outlier Sensitivity:** Outliers can significantly influence the placement of centroids.

Applications in Data Science:

- **Customer Segmentation:** Grouping customers based on their purchasing behavior or demographics.
- **Image Segmentation:** Dividing images into meaningful regions (e.g., foreground and background).
- **Anomaly Detection:** Identifying unusual patterns or outliers in data.
- **Document Clustering:** Grouping documents based on their content or topics.

Association Rule Mining

What is Association Rule Mining?

Association rule mining (ARM), also known as market basket analysis, is an unsupervised machine learning method used to discover interesting relationships (affinities) between variables in large databases. It's particularly useful for analyzing transactional data, where you have records of items that are often purchased or used together.

The key concept in ARM is to find *association rules*, which are if-then statements that express the likelihood of certain items occurring together in a transaction. For example, a classic association rule might be:

"If a customer buys bread, then they are also likely to buy butter (with 80% confidence)."

Key Concepts in Association Rule Mining:

1. **Support:** The proportion of transactions that contain both the antecedent (the "if" part) and the consequent (the "then" part) of the rule. It indicates how frequently the itemset appears in the dataset.
2. **Confidence:** The proportion of transactions containing the antecedent that also contain the consequent. It measures how often the rule is found to be true.
3. **Lift:** The ratio of the observed support of the rule to the expected support if the antecedent and consequent were independent. A lift greater than 1 indicates that the items are associated more often than would be expected by chance.

Apriori Algorithm

One of the most well-known algorithms for association rule mining is the Apriori algorithm. It uses a "bottom-up" approach, starting by identifying frequent individual items and then iteratively generating larger and larger itemsets until no more frequent itemsets can be found.

Applications of Association Rule Mining:

- **Market Basket Analysis:** Understanding customer purchasing patterns to improve product placement, promotions, and recommendations.
- **Web Usage Mining:** Analyzing user navigation patterns on websites to personalize recommendations or optimize website layout.
- **Medical Diagnosis:** Identifying patterns in patient symptoms or medical history to assist in diagnosis.
- **Fraud Detection:** Discovering unusual patterns in financial transactions to detect potential fraud.

Example: Market Basket Analysis

Imagine a grocery store wants to analyze its transaction data to identify which products are frequently bought together. Using association rule mining, they might find rules like:

- If a customer buys diapers, they are also likely to buy baby wipes (with 70% confidence).
- If a customer buys beer, they are also likely to buy chips (with 60% confidence).

This information could be used to create targeted promotions or adjust store layouts to increase sales.

Challenges and Considerations:

- **Scalability:** ARM can become computationally expensive for very large datasets.
- **Interpreting Results:** It's important to consider the context and business knowledge when interpreting association rules. Not all rules may be actionable or meaningful.
- **Rare Itemsets:** It can be challenging to find rules for items that occur infrequently.

Libraries and Tools:

- **Python:** `mlxtend`, `apyori` libraries
- **R:** `arules`, `arulesViz` packages

Linear and Logistic Regression :

Linear models are especially useful when you don't want only to predict an outcome, but also to know the relationship between the input variables and the outcome. This knowledge can prove useful because this relationship can often be used as advice on how to get the outcome that you want. We'll first define linear regression and then use it to predict customer income. Later, we will use logistic regression to predict the probability that a newborn baby will need extra medical attention. We'll also walk through the diagnostics that R produces when you fit a linear or logistic model.Linear methods can work well in a surprisingly wide range of situations. However, there can be issues when the inputs to the model are correlated or collinear. In the case of logistic regression, there can also be issues (ironically) when a subset of the variables predicts a classification output perfectly in a subset of the training data.

USING LINEAR REGRESSION :

Linear regression is the bread and butter prediction method for statisticians and data scientists. If you're trying to predict a numerical quantity like profit, cost, or sales volume, you should always try linear regression first. If it works well, you're done; if it fails, the detailed diagnostics produced can give you a good clue as to what methods you should try next.

UNDERSTANDING LINEAR REGRESSION :

Example Suppose you want to predict how many pounds a person on a diet and exercise plan will lose in a month. You will base that prediction on other facts about that person, like how much they reduce their average daily caloric intake over that month and how many hours a day they exercised. In other words, for every person i, you want to predict pounds lost[i] based on daily_cals_down[i] and daily_exercise[i].

Linear regression assumes that the outcome pounds_lost is linearly related to each of the inputs daily_cals_down[i] and daily_exercise[i]. This means that the relationship between (for instance) daily_cals_down[i] and pounds_lost looks like a (noisy) straight line, as shown in figure 7.2.1

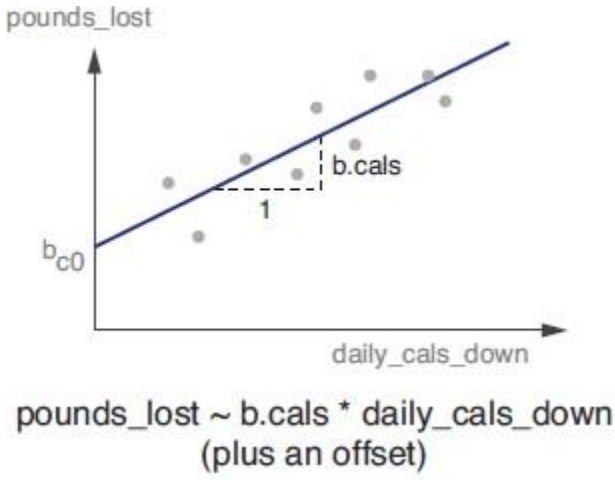


Figure 7.2 The linear relationship between `daily_cals_down` and `pounds_lost`

The relationship between daily_exercise and pounds_lost would similarly be a straight line. Suppose that the equation of the line shown in figure 7.2 is

pounds_lost = bc0 + b.cals * daily_cals_down

This means that for every unit change in daily_cals_down (every calorie reduced), the value of pounds_lost changes by b.cals, no matter what the starting value of daily_cals_down was. To make it concrete, suppose pounds_lost = 3 + 2 * daily_ cals_down. Then increasing daily_cals_down by one increases pounds_lost by 2, no matter what value of daily_cals_down you start with. This would not be true for, say, pounds_lost = 3 + 2 * (daily_cals_down^2).

Linear regression further assumes that the total pounds lost is a linear combination of our variables daily_cals_down[i] and daily_exercise[i], or the sum of the pounds lost due to reduced caloric intake, and the pounds lost due to exercise. This gives us the following form for the linear regression model of pounds_lost:

pounds_lost[i] = b0 + b.cals * daily_cals_down[i] +
b.exercise * daily_exercise[i]

The goal of linear regression is to find the values of b0, b.cals, and b.exercise so that the linear combination of daily_cals_lost[i] and daily_exercise[i] (plus some offset b0) comes very close to pounds_lost[i] for all persons i in the training data. Let's put this in more general terms. Suppose that y[i] is the numeric quantity you want to predict (called the *dependent* or *response* variable), and x[i,] is a row of inputs that corresponds to output y[i] (the x[i,] are the *independent* or *explanatory* variables). Linear regression attempts to find a function *f*(*x*) such that

$$y[i] \sim f(x[i,]) + e[i] = b[0] + b[1] * x[i,1] + ... + b[n] * x[i,n] + e[i]$$

The expression for a linear regression model

You want numbers b[0],...,b[n] (called the coefficients or betas) such that f(x[i,]) is as near as possible to y[i] for all (x[i,],y[i]) pairs in the training data. R supplies a one-line command to find these coefficients: lm(). The last term in equation 7.1, e[i], represents what are called unsystematic errors, or noise. Unsystematic errors are defined to all have a mean value of 0 (so they don't represent a net upward or net downward bias) and are defined as uncorrelated with x[i,]. In other words, x[i,] should not encode information about e[i] (or vice versa).

By assuming that the noise is unsystematic, linear regression tries to fit what is called an "unbiased" predictor. This is another way of saying that the predictor gets the right answer "on average" over the entire training set, or that it underpredicts about as much as it overpredicts. In particular, unbiased estimates tend to get totals correct.

Example Suppose you have fit a linear regression model to predict weight loss based on reduction of caloric intake and exercise. Now consider the set of subjects in the training data, LowExercise, who exercised between zero and one hour a day. Together, these subjects lost a total of 150 pounds over the course of the study. How much did the model predict they would lose?

With a linear regression model, if you take the predicted weight loss for all the subjects in Low Exercise and sum them up, that total will sum to 150 pounds, which means that the model predicts the average weight loss of a person in the Low Exercise group correctly, even though some of the individuals will have lost more than the model predicted, and some of them will have lost less. In a business setting, getting sums like this correct is critical, particularly when summing up monetary amounts. Under these assumptions (linear relationships and unsystematic noise), linear regression is absolutely relentless in finding the best coefficients b[i]. If there's some advantageous combination or cancellation of features, it'll find it. One thing that linear regression doesn't do is reshape variables to be linear. Oddly enough, linear regression often does an excellent job, even when the actual relation is not in fact linear.

INTRODUCING THE PUMS DATASET

Example Suppose you want to predict personal income of any individual in the general

public, within some relative percent, given their age, education, and other demographic variables. In addition to predicting income, you also have a secondary goal: to determine the effect of a bachelor's degree on income, relative to having no degree at all.

We can continue the example by loading psub.RDS (which you can download from https://github.com/WinVector/PDSwR2/raw/master/PUMS/psub.RDS) into your working directory, and performing the steps in the following listing.[1]

Listing 7.1 Loading the PUMS data and fitting a model

```
psub <- readRDS("psub.RDS")

set.seed(3454351)
gp <- runif(nrow(psub))

dtrain <- subset(psub, gp >= 0.5)
 dtest <- subset(psub, gp < 0.5)

model <- lm(log10(PINCP) ~ AGEP + SEX + COW + SCHL, data = dtrain)
 dtest$predLogPINCP <- predict(model, newdata = dtest)
 dtrain$predLogPINCP <- predict(model, newdata = dtrain)
```

Makes a random variable to group and partition the data

Splits 50–50 into training and test sets

Fits a linear model to log(income)

Gets the predicted log(income) on the test and training sets

For this task, you will use the 2016 US Census PUMS dataset. For simplicity, we have prepared a small sample of PUMS data to use for this example. The data preparation steps include these:

- Restricting the data to full-time employees between 20 and 50 years of age, with an income between $1,000 and $250,000.
- Dividing the data into a training set, dtrain, and a test set, dtest.

Each row of PUMS data represents a single anonymized person or household. Personal data recorded includes occupation, level of education, personal income, and many other demographic variables. For this example we have decided to predict log10(PINCP), or the logarithm of income. Fitting logarithm-transformed data typically gives results with smaller relative error, emphasizing smaller errors on smaller incomes. But this improved relative error comes at a cost of introducing a bias: on average, predicted incomes are going to be below actual training incomes. An unbiased alternative to predicting log(income) would be to use a type of generalized linear model called Poisson regression. The Poisson regression is unbiased, but typically at the cost of larger relative errors.1 For the analysis in this section, we'll consider the input variables age (AGEP), sex (SEX), class of worker (COW), and level of education (SCHL). The output variable is personal income (PINCP). We'll also set the reference level, or "default" sex to M (male); the reference level of class of worker to Employee of a private for-profit; and the reference level of education level to no high school diploma.

BUILDING A LINEAR REGRESSION MODEL

The first step in either prediction or finding relations (advice) is to build the linear regression model. The function to build the linear regression model in R is lm(), supplied by the stats package. The most important argument to lm() is a formula with ~ used in place of an equals sign. The formula specifies what column of the data frame is the quantity to be predicted, and what columns are to be used to make the predictions. Statisticians call the quantity to be predicted the dependent variable and the variables/ columns used to make the prediction the independent variables. We find it is easier to call the quantity to be predicted the y and the variables used to make the predictions the xs. Our formula is this: log10(PINCP) ~ AGEP + SEX + COW + SCHL, which is read "Predict the log base 10 of income as a function of age, sex, employment class, and education."

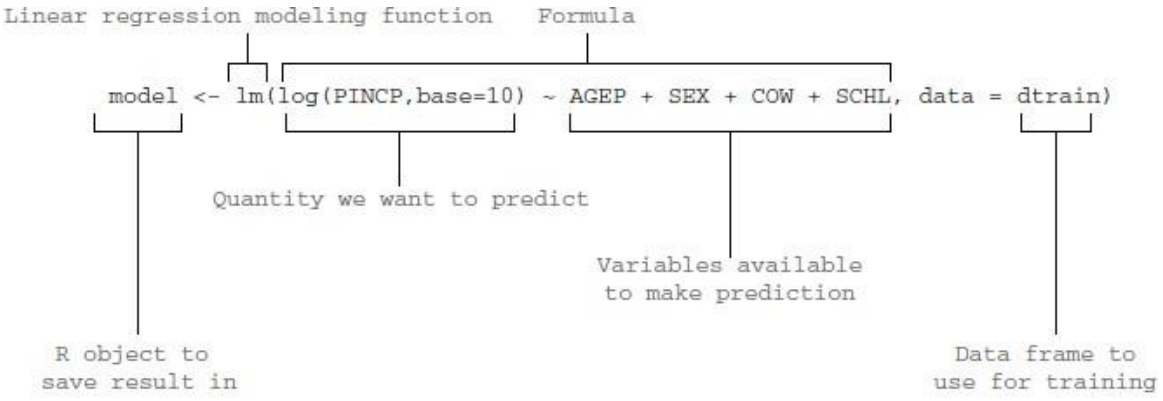


Figure 7.4 Building a linear model using `lm()`

R STORES TRAINING DATA IN THE MODEL R holds a copy of the training data in its model to supply the residual information seen in summary(model). Holding a copy of the data this way is not strictly necessary, and can needlessly run you out of memory. If you're running low on memory (or swapping), you can dispose of R objects like model using the rm() command. In this case, you'd dispose of the model by running rm("model").

MAKING PREDICTIONS:

Once you've called lm() to build the model, your first goal is to predict income. This is easy to do in R. To predict, you pass data into the predict() method. Figure demonstrates this using both the test and training data frames dtest and dtrain.

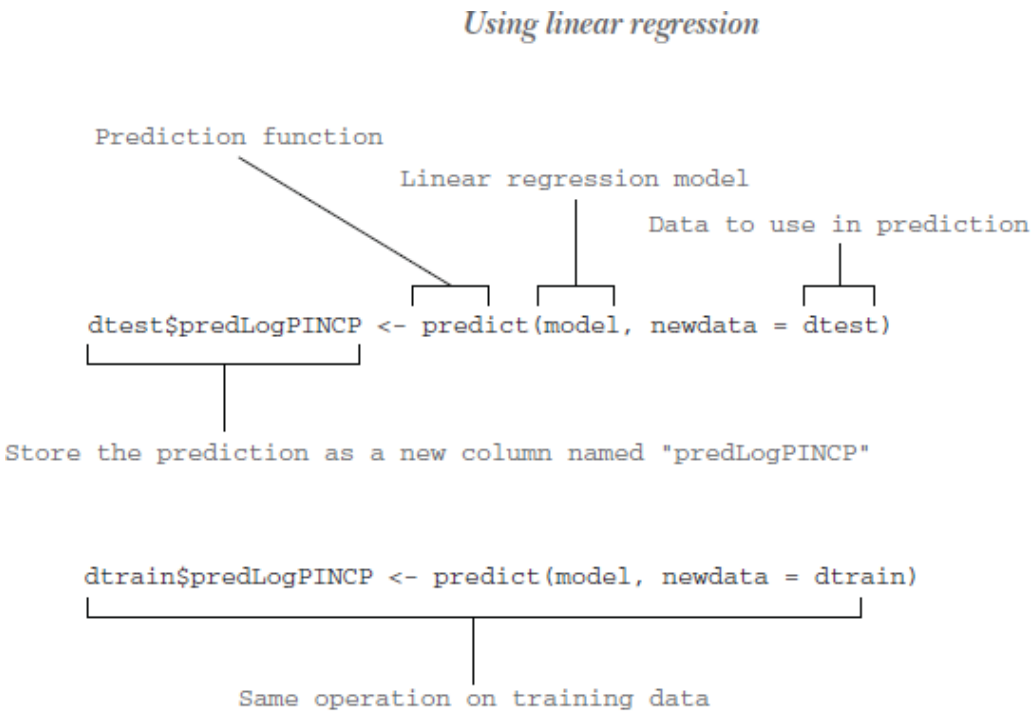


Figure 7.5 Making predictions with a linear regression model

The data frame columns dtest$predLogPINCP and dtrain$predLogPINCP now store the predictions for the test and training sets, respectively. We have now both produced and applied a linear regression model.

USING LOGISTIC REGRESSION:

Logistic regression is the most important (and probably most used) member of a class of models called generalized linear models. Unlike linear regression, logistic regression can directly predict values that are restricted to the (0, 1) interval, such as probabilities. It's the go-to method for predicting probabilities or rates, and like linear regression, the coefficients of a logistic regression model can be treated as advice. It's also a good first choice for binary classification problems. In this section, we'll use a medical classification example (predicting whether a
newborn will need extra medical attention) to work through all the steps of producing and using a logistic regression model.1 As we did with linear regression, we'll take a quick overview of logistic regression before tackling the main example.

UNDERSTANDING LOGISTIC REGRESSION

Example Suppose you want to predict whether or not a flight will be delayed, based on

facts like the flight's origin and destination, weather, and air carrier. For every flight i, you want to predict flight_delayed[i] based on origin[i], destination[i], weather[i], and air_carrier[i].
We'd like to use linear regression to predict the probability that a flight i will be delayed, but probabilities are strictly in the range 0:1, and linear regression doesn't restrict its prediction to that range.

One idea is to find a function of probability that is in the range -Infinity:Infinity, fit a linear model to predict that quantity, and then solve for the appropriate probabilities from the model predictions. So let's look at a slightly different problem: instead of predicting the probability that a flight is delayed, consider the odds that the flight is delayed, or the ratio of the probability that the flight is delayed over the probability that it is not.
odds[flight_delayed] = P[flight_delayed == TRUE] / P[flight_delayed == FALSE]
The range of the odds function isn't -Infinity:Infinity; it's restricted to be a nonnegative number. But we can take the log of the odds---the log-odds---to get a function of the probabilities that is in the range -Infinity:Infinity.
log_odds[flight_delayed] = log(P[flight_delayed == TRUE] / P[flight_delayed =
= FALSE])
Let: p = P[flight_delayed == TRUE]; then
log_odds[flight_delayed] = log(p / (1 - p))

Note that if it's more likely that a flight will be delayed than on time, the odds ratio will be greater than one; if it's less likely that a flight will be delayed than on time, the odds ratio will be less than one. So the log-odds is positive if it's more likely that the flight will be delayed, negative if it's more likely that the flight will be on time, and zero if the chances of delay are 50-50.
The log-odds of a probability p is also known as logit(p). The inverse of logit(p) is the sigmoid function, shown in figure 7.13. The sigmoid function maps values in the range from -Infinity:Infinity to the range 0:1—in this case, the sigmoid maps unbounded log-odds ratios to a probability value that is between 0 and 1.

```
logit <- function(p) { log(p/(1-p)) }
s <- function(x) { 1/(1 + exp(-x))}
s(logit(0.7))
# [1] 0.7
logit(s(-2))
# -2
```

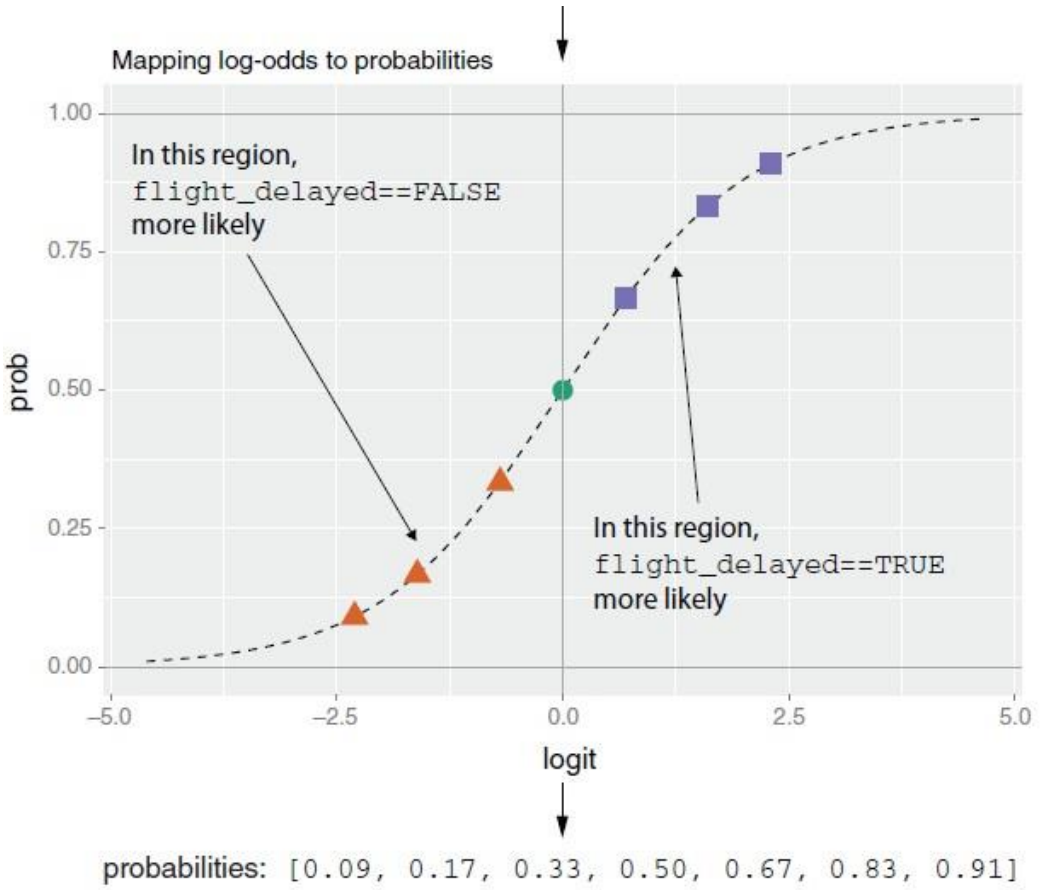


Figure 7.13 Mapping log-odds to the probability of a flight delay via the sigmoid function

BUILDING A LOGISTIC REGRESSION MODEL

The function to build a logistic regression model in R is glm(), supplied by the stats package. In our case, the dependent variable y is the logical (or Boolean) atRisk; all the other variables in table 7.1 are the independent variables x. The formula for building a model to predict atRisk using these variables is rather long to type in by hand; you can generate the formula using the mk_formula() function from the wrapr package, as shown next.

Listing 7.8 Building the model formula

```
complications <- c("ULD_MECO","ULD_PRECIP","ULD_BREECH")
riskfactors <- c("URF_DIAB", "URF_CHYPER", "URF_PHYPER",
                 "URF_ECLAM")
y <- "atRisk"
x <- c("PWGT",
       "UPREVIS",
       "CIG_REC",
       "GESTREC3",
       "DPLURAL",
       complications,
       riskfactors)
library(wrapr)
fmla <- mk_formula(y, x)
```

Now we'll build the logistic regression model, using the training dataset.

Listing 7.9 Fitting the logistic regression model

```
print(fmla)

## atRisk ~ PWGT + UPREVIS + CIG_REC + GESTREC3 + DPLURAL + ULD_MECO +
##     ULD_PRECIP + ULD_BREECH + URF_DIAB + URF_CHYPER + URF_PHYPER +
##     URF_ECLAM
## <environment: base>

model <- glm(fmla, data = train, family = binomial(link = "logit"))
```

This is similar to the linear regression call to lm(), with one additional argument:
family = binomial(link = "logit"). The family function specifies the assumed distribution of the dependent variable y. In our case, we're modeling y as a binomial distribution, or as a coin whose probability of heads depends on x. The link function "links" the output to a linear model—it's as if you pass y through the link function, and then model the resulting value as a linear function of the x values. Different combinations of family functions and link functions lead to different kinds of generalized linear models (for example, Poisson, or probit). In this book, we'll only discuss logistic models, so we'll only need to use the binomial family with the logit link

MAKING PREDICTIONS

Making predictions with a logistic model is similar to making predictions with a linear model—use the predict() function. The following code stores the predictions for the training and test sets as the column pred in the respective data frames.

Applying the logistic regression model.

```
train$pred <- predict(model, newdata=train, type = "response")
test$pred <- predict(model, newdata=test, type="response")
```

Note the additional parameter type = "response". This tells the predict() function to return the predicted probabilities y. If you don't specify type = "response", then by default predict() will return the output of the link function, logit(y). One strength of logistic regression is that it preserves the marginal probabilities of the training data. That means that if you sum the predicted probability scores for the entire training set, that quantity will be equal to the number of positive outcomes (atRisk == TRUE) in the training set. This is also true for subsets of the data determined by variables included in the model. For example, in the subset of the training data that has train$GESTREC == "<37 weeks" (the baby was premature), the sum of the predicted probabilities equals the number of positive training examples.

Create a subset of these variables from the "mtcars" dataset

1. data<-mtcars[,c("mpg","wt","disp","hp")]
2. print(head(input))

Creating Relationship Model and finding Coefficient

will use the data which we have created before to create the Relationship Model. We will use the lm() function, which takes two parameters i.e., formula and data. Let's start understanding how the lm() function is used to create the Relationship Model.

Example

```
#Creating input data.
input <- mtcars[,c("mpg","wt","disp","hp")]
# Creating the relationship model.
Model <- lm(mpg~wt+disp+hp, data = input)
# Showing the Model.
print(Model)

b0<- coef(Model)[1]
print(b0)
x_wt<- coef(Model)[2]
x_disp<- coef(Model)[3]
x_hp<- coef(Model)[4]
print(x_wt)
print(x_disp)
print(x_hp)
```

UNIT-V

Data visualization with R

Introduction to ggplot2: A worked example, Placing the data and mapping options, Graphs as objects, Univariate Graphs: Categorical, Quantitative.

Bivariate Graphs- Categorical vs. Categorical, Quantitative vs Quantitative, Categorical vs. Quantitative, Multivariate Graphs : Grouping, Faceting.

Introduction to ggplot2:

5.1 A worked example

The functions in the ggplot2 package build up a graph in layers. We'll build a complex graph by starting with a simple graph and adding additional elements, one at a time.
The example uses data from the 1985 Current Population Survey to explore the relationship between wages (*wage*) and experience (*expr*).

```
# load data
data(CPS85 , package ="mosaicData")
```

In building a ggplot2 graph, only the first two functions described below are required. The other functions are optional and can appear in any order.

5.1.1 ggplot

The first function in building a graph is the ggplot function. It specifies the

- data frame containing the data to be plotted
- the mapping of the variables to visual properties of the graph. The mappings are placed within the aes function (where *aes* stands for aesthetics).

```
# specify dataset and mapping
library(ggplot2)
ggplot(data = CPS85,
mapping =aes(x =exper, y = wage))
```

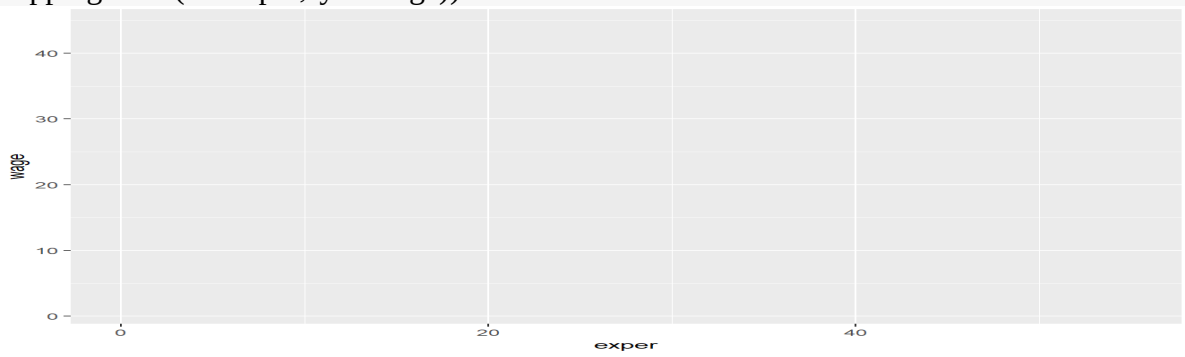


Figure: Map variables

Why is the graph empty? We specified that the *exper* variable should be mapped to the *x*-axis and that the *wage* should be mapped to the *y*-axis, but we haven't yet specified what we wanted placed on the graph.

5.1.2 geoms

Geoms are the geometric objects (points, lines, bars, etc.) that can be placed on a graph. They are added using functions that start with geom_. In this example, we'll add points using the geom_point function, creating a scatterplot.
In ggplot2 graphs, functions are chained together using the + sign to build a final plot.

```
# add points
ggplot(data = CPS85,
mapping =aes(x =exper, y = wage)) +
geom_point()
```

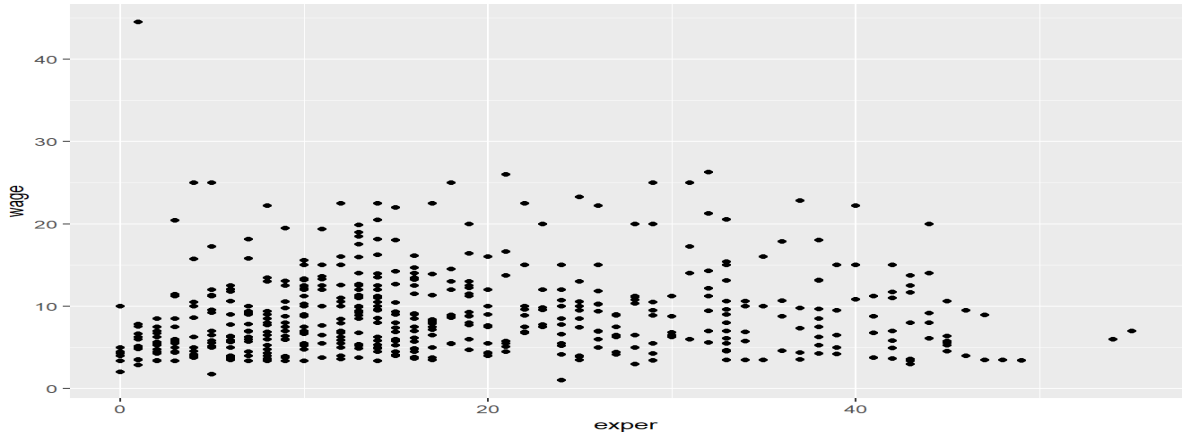


Figure: Add points

The graph indicates that there is an outlier. One individual has a wage much higher than the rest. We'll delete this case before continuing.

```
# delete outlier
library(dplyr)
plotdata<-filter(CPS85, wage <40)

# redraw scatterplot
ggplot(data =plotdata,
mapping =aes(x =exper, y = wage)) +
geom_point()
```

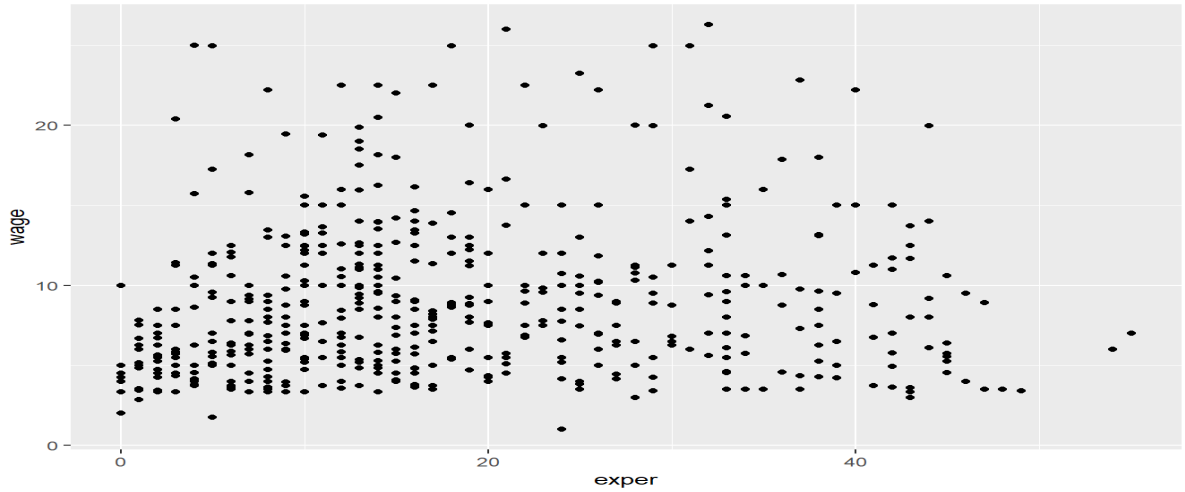


Figure: Remove outlier

A number of parameters (options) can be specified in a geom_ function. Options for the geom_point function include color, size, and alpha. These control the point color, size, and

transparency, respectively. Transparency ranges from 0 (completely transparent) to 1 (completely opaque). Adding a degree of transparency can help visualize overlapping points.

```
# make points blue, larger, and semi-transparent
ggplot(data =plotdata,
mapping =aes(x =exper, y = wage)) +
geom_point(color ="cornflowerblue",
alpha = .7,
size =3)
```

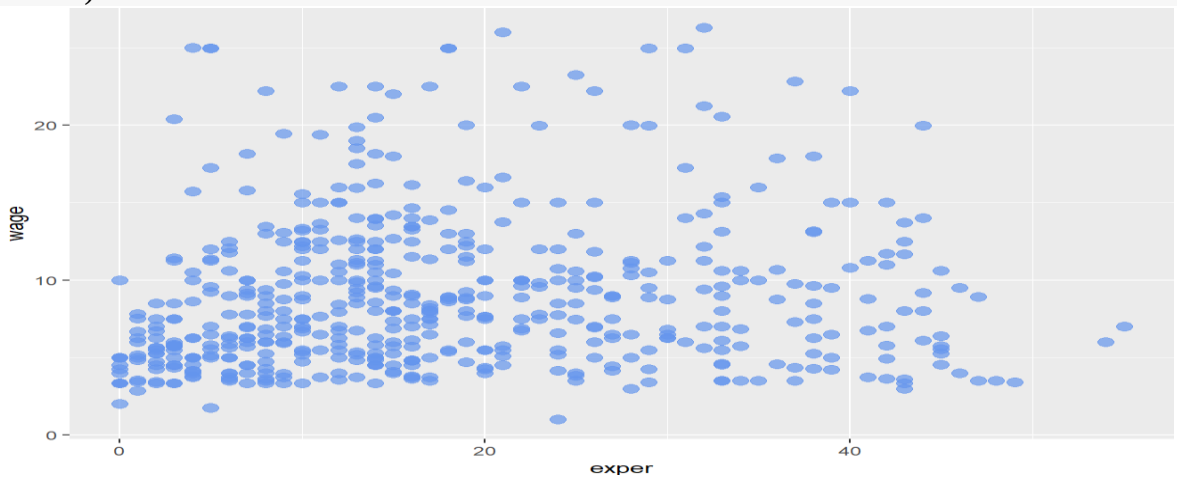


Figure: Modify point color, transparency, and size

Next, let's add a line of best fit. We can do this with the geom_smooth function. Options control the type of line (linear, quadratic, nonparametric), the thickness of the line, the line's color, and the presence or absence of a confidence interval. Here we request a linear regression (method = lm) line (where *lm* stands for linear model).

```
# add a line of best fit.
ggplot(data =plotdata,
mapping =aes(x =exper, y = wage)) +
geom_point(color ="cornflowerblue",
alpha = .7,
size =3) +
geom_smooth(method ="lm")
```

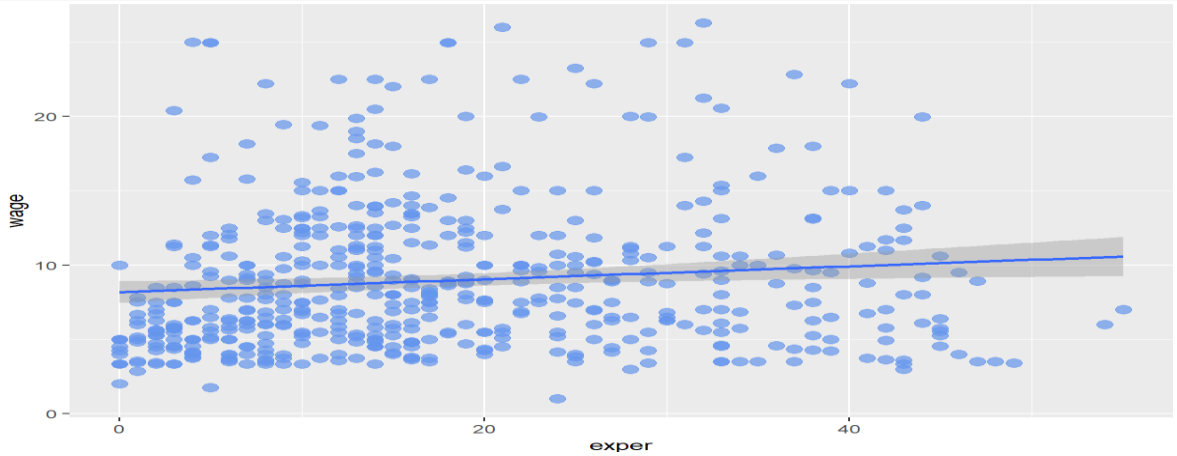


Figure: Add line of best fit

Wages appears to increase with experience.

5.1.3 grouping

In addition to mapping variables to the *x* and *y* axes, variables can be mapped to the color, shape, size, transparency, and other visual characteristics of geometric objects. This allows groups of observations to be superimposed in a single graph.

Let's add sex to the plot and represent it by color.

```
# indicate sex using color
ggplot(data =plotdata,
mapping =aes(x =exper,
y = wage,
color = sex)) +
geom_point(alpha = .7,
size =3) +
geom_smooth(method ="lm",
se =FALSE,
size =1.5)
```

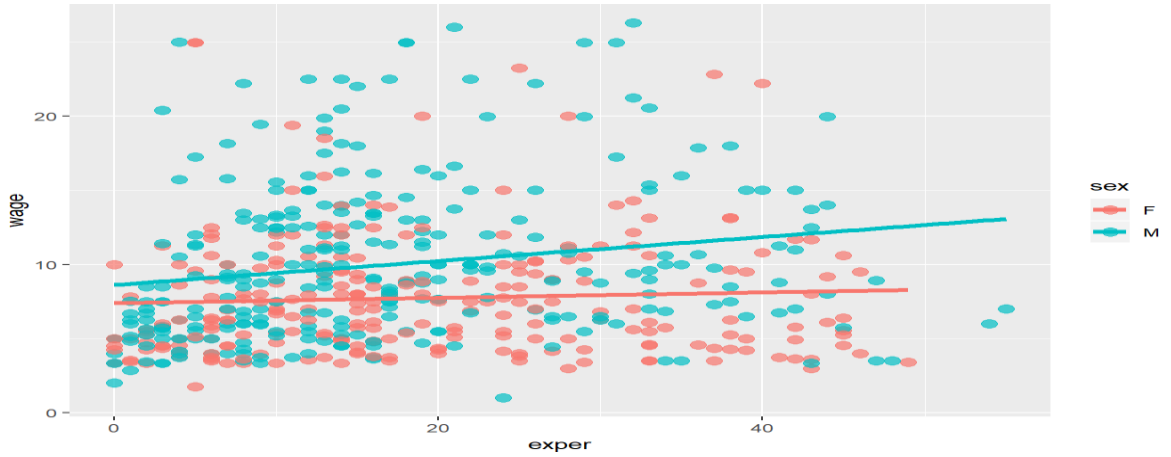


Figure: Include sex, using color

The color = sex option is placed in the aes function, because we are mapping a variable to an aesthetic. The geom_smooth option (se = FALSE) was added to suppresses the confidence intervals.
It appears that men tend to make more money than women. Additionally, there may be a stronger relationship between experience and wages for men than than for women.

5.1.4 scales

Scales control how variables are mapped to the visual characteristics of the plot. Scale functions (which start with scale_) allow you to modify this mapping. In the next plot, we'll change the *x* and *y* axis scaling, and the colors employed.

```
# modify the x and y axes and specify the colors to be used
ggplot(data =plotdata,
mapping =aes(x =exper,
y = wage,
color = sex)) +
geom_point(alpha = .7,
size =3) +
geom_smooth(method ="lm",
se =FALSE,
size =1.5) +
```

```
scale_x_continuous(breaks =seq(0, 60, 10)) +
scale_y_continuous(breaks =seq(0, 30, 5),
label =scales::dollar) +
scale_color_manual(values =c("indianred3",
"cornflowerblue"))
```

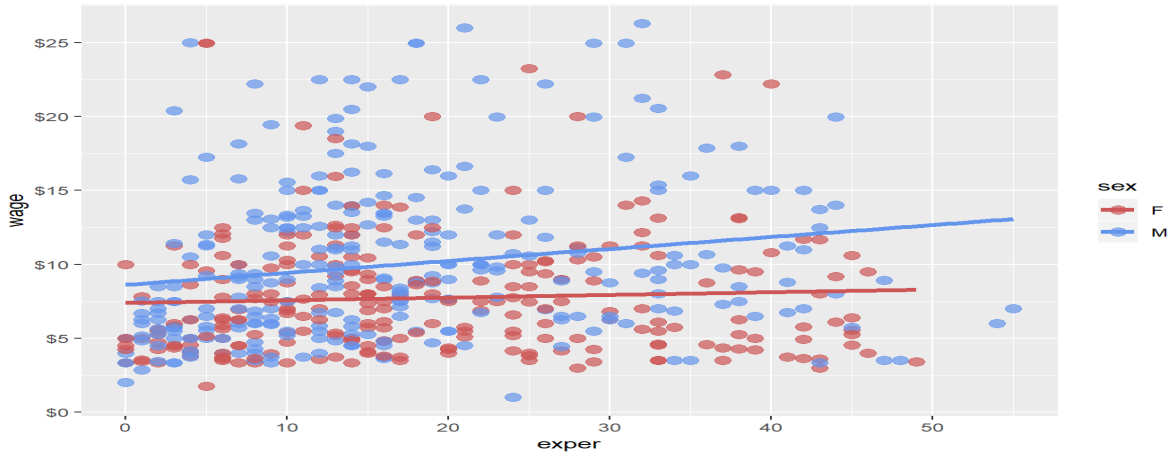


Figure: Change colors and axis labels

We're getting there. The numbers on the *x* and *y* axes are better, the *y* axis uses dollar notation, and the colors are more attractive (IMHO).

Here is a question. Is the relationship between experience, wages and sex the same for each job sector? Let's repeat this graph once for each job sector in order to explore this.

5.1.5 facets

Facets reproduce a graph for each level a given variable (or combination of variables). Facets are created using functions that start with facet_. Here, facets will be defined by the eight levels of the *sector* variable.

```
# reproduce plot for each level of job sector
ggplot(data =plotdata,
mapping =aes(x =exper,
y = wage,
color = sex)) +
geom_point(alpha = .7) +
geom_smooth(method ="lm",
se =FALSE) +
scale_x_continuous(breaks =seq(0, 60, 10)) +
scale_y_continuous(breaks =seq(0, 30, 5),
label =scales::dollar) +
scale_color_manual(values =c("indianred3",
"cornflowerblue")) +
facet_wrap(~sector)
```

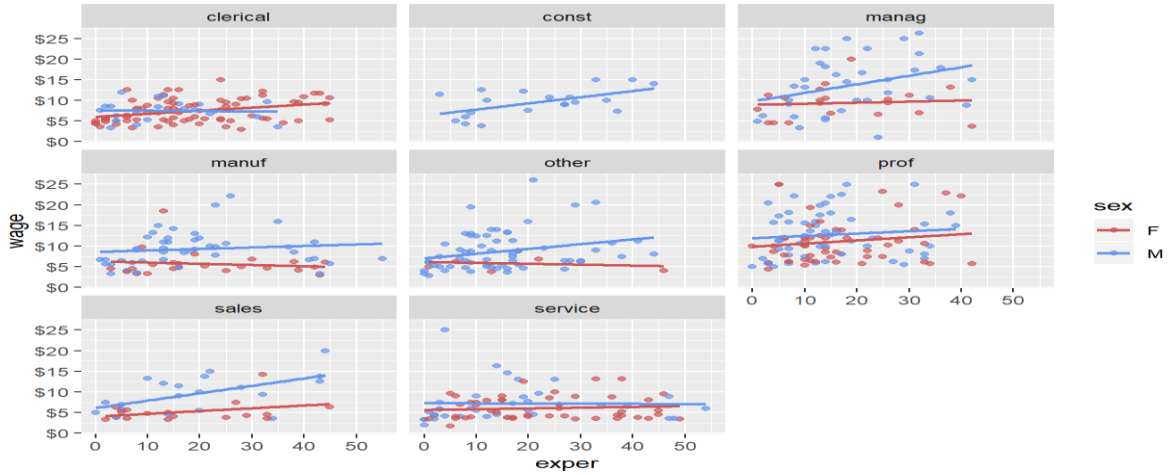


Figure: Add job sector, using faceting

It appears that the differences between mean and women depend on the job sector under consideration.

5.1.6 labels

Graphs should be easy to interpret and informative labels are a key element in achieving this goal. The labs function provides customized labels for the axes and legends. Additionally, a custom title, subtitle, and caption can be added.

```
# add informative labels
ggplot(data =plotdata,
mapping =aes(x =exper,
y = wage,
color = sex)) +
geom_point(alpha = .7) +
geom_smooth(method ="lm",
se =FALSE) +
scale_x_continuous(breaks =seq(0, 60, 10)) +
scale_y_continuous(breaks =seq(0, 30, 5),
label =scales::dollar) +
scale_color_manual(values =c("indianred3",
"cornflowerblue")) +
facet_wrap(~sector) +
labs(title ="Relationship between wages and experience",
subtitle ="Current Population Survey",
caption ="source: http://mosaic-web.org/",
x =" Years of Experience",
y ="Hourly Wage",
color ="Gender")
```

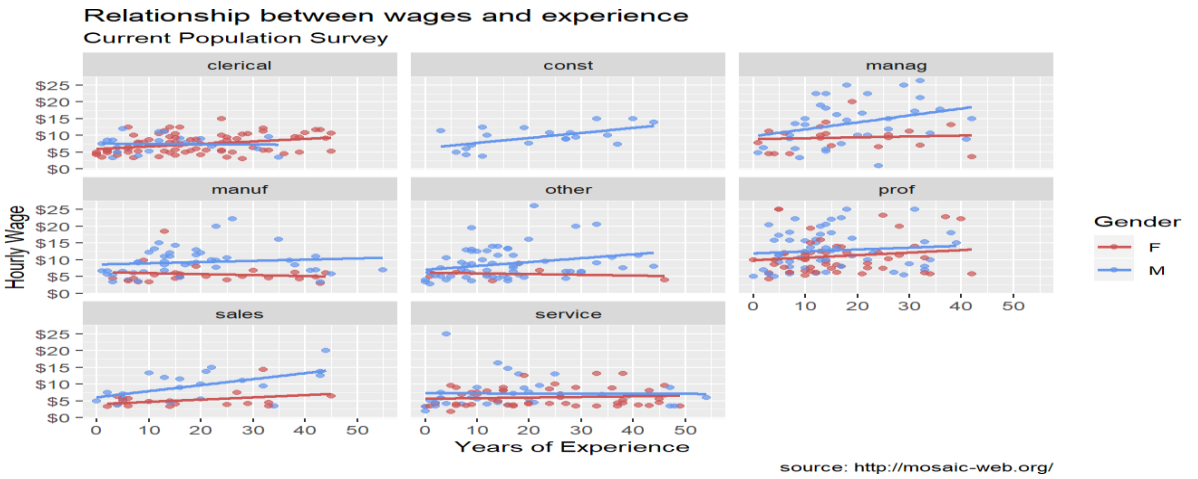


Figure: Add informative titles and labels

Now a viewer doesn't need to guess what the labels *expr* and *wage* mean, or where the data come from.

5.1.7 themes

Finally, we can fine tune the appearance of the graph using themes. Theme functions (which start with theme_) control background colors, fonts, grid-lines, legend placement, and other non-data related features of the graph. Let's use a cleaner theme.

```
# use a minimalist theme
ggplot(data =plotdata,
mapping =aes(x =exper,
y = wage,
color = sex)) +
geom_point(alpha = .6) +
geom_smooth(method ="lm",
se =FALSE) +
scale_x_continuous(breaks =seq(0, 60, 10)) +
scale_y_continuous(breaks =seq(0, 30, 5),
label =scales::dollar) +
scale_color_manual(values =c("indianred3",
"cornflowerblue")) +
facet_wrap(~sector) +
labs(title ="Relationship between wages and experience",
subtitle ="Current Population Survey",
caption ="source: http://mosaic-web.org/",
x =" Years of Experience",
y ="Hourly Wage",
color ="Gender") +
theme_minimal()
```

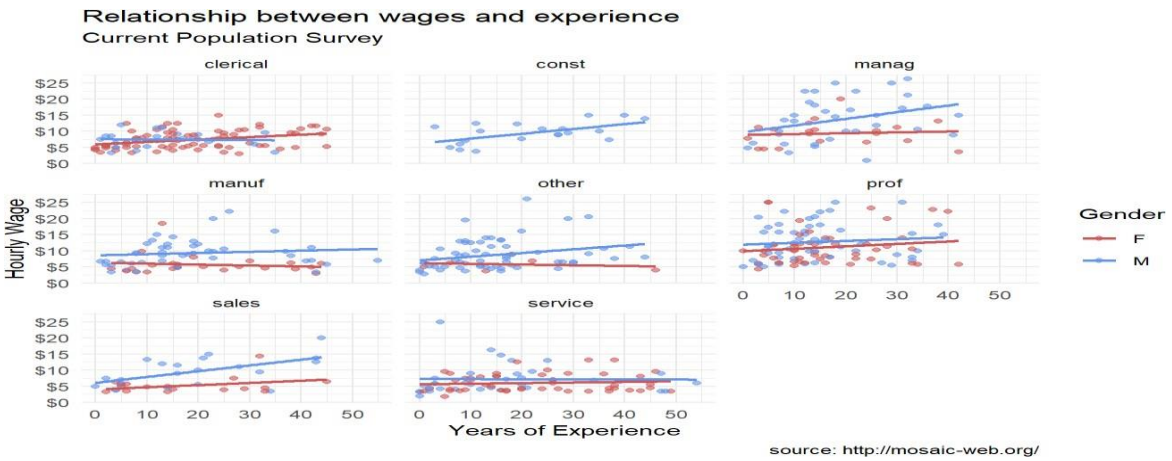


Figure: Use a simpler theme

Now we have something. It appears that men earn more than women in management, manufacturing, sales, and the "other" category. They are most similar in clerical, professional, and service positions. The data contain no women in the construction sector. For management positions, wages appear to be related to experience for men, but not for women (this may be the most interesting finding). This also appears to be true for sales.

Of course, these findings are tentative. They are based on a limited sample size and do not involve statistical testing to assess whether differences may be due to chance variation.

5.2 Placing the data and mapping options

Plots created with ggplot2 always start with the ggplot function. In the examples above, the data and mapping options were placed in this function. In this case they apply to each geom_ function that follows.
You can also place these options directly within a geom. In that case, they only apply only to that specific geom.
Consider the following graph.

```
# placingcolor mapping in the ggplot function
ggplot(plotdata,
aes(x =exper,
y =  wage,
color = sex)) +
geom_point(alpha = .7,
size =3) +
geom_smooth(method ="lm",
formula = y ~poly(x,2),
se =FALSE,
size =1.5)
```

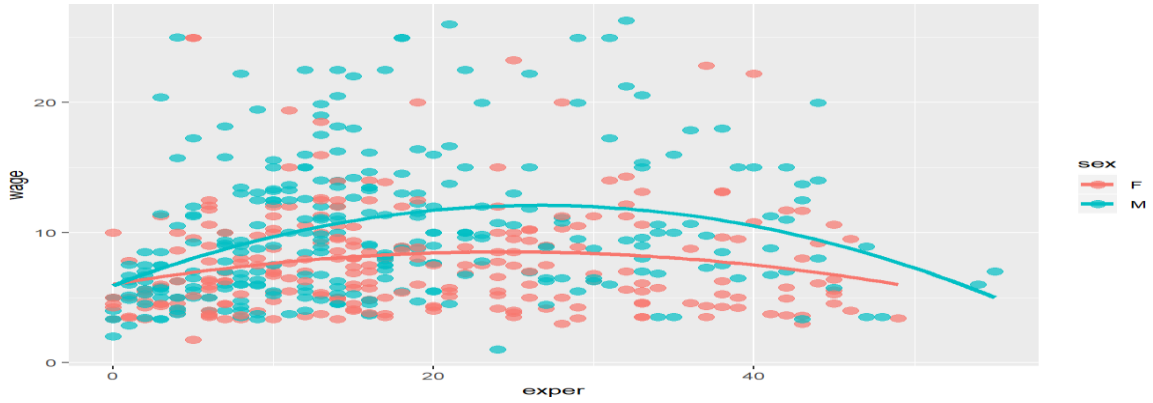


Figure: Color mapping in ggplot function

Since the mapping of sex to color appears in the ggplot function, it applies to *both* geom_point and geom_smooth. The color of the point indicates the sex, and a separate colored trend line is produced for men and women. Compare this to

```
# placingcolor mapping in the geom_point function
ggplot(plotdata,
aes(x =exper,
y = wage)) +
geom_point(aes(color = sex),
alpha = .7,
size =3) +
geom_smooth(method ="lm",
formula = y ~poly(x,2),
se =FALSE,
size =1.5)
```

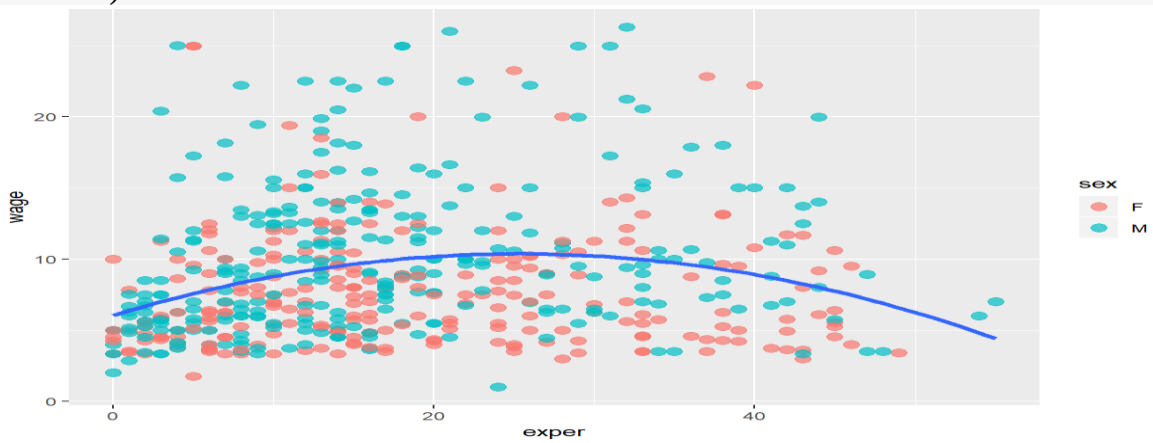


Figure12: Color mapping in ggplot function

Since the sex to color mapping only appears in the geom_point function, it is only used there. A single trend line is created for all observations.
Most of the examples in this book place the data and mapping options in the ggplot function. Additionally, the phrases *data=* and *mapping=* are omitted since the first option always refers to data and the second option always refers to mapping.

5.3 Graphs as objects

A ggplot2 graph can be saved as a named R object (like a data frame), manipulated further, and then printed or saved to disk.

```
# prepare data
data(CPS85 , package ="mosaicData")
```

```
plotdata<-CPS85[CPS85$wage <40,]

# create scatterplot and save it
myplot<-ggplot(data =plotdata,
aes(x =exper, y = wage)) +
geom_point()

# print the graph
myplot

# make the points larger and blue
# then print the graph
myplot<-myplot+geom_point(size =3, color ="blue")
myplot

# print the graph with a title and line of best fit
# but don't save those changes
myplot+geom_smooth(method ="lm") +
labs(title ="Mildly interesting graph")

# print the graph with a black and white theme
# but don't save those changes
myplot+theme_bw()
```

5.4 Univariate graphs

Univariate graphs plot the distribution of data from a single variable. The variable can be categorical (e.g., race, sex) or quantitative (e.g., age, weight).

5.4.1 Categorical

The distribution of a single categorical variable is typically plotted with a bar chart, a pie chart, or (less commonly) a tree map.

5.4.1.1 Bar chart

The Marriage dataset contains the marriage records of 98 individuals in Mobile County, Alabama. Below, a bar chart is used to display the distribution of wedding participants by race.

```
library(ggplot2)
data(Marriage, package ="mosaicData")

# plot the distribution of race
ggplot(Marriage, aes(x = race)) +
geom_bar()
```

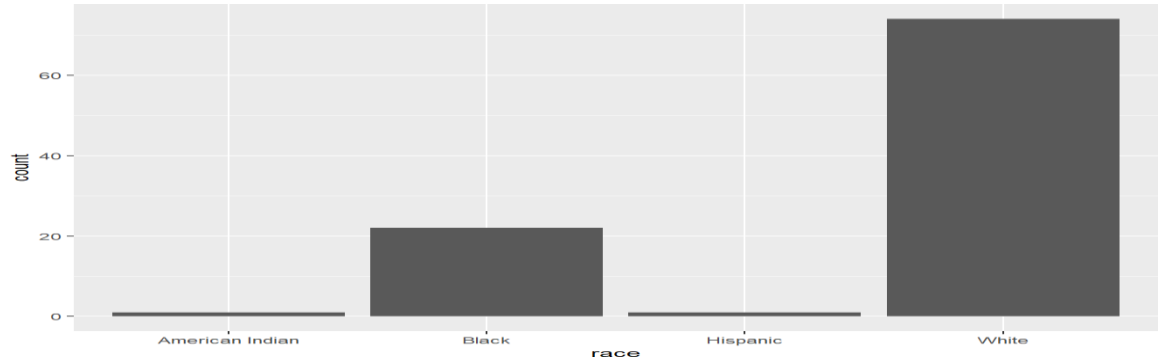


Figure: Simple barchart

5.4.1.1.1 Percents

Bars can represent percents rather than counts. For bar charts, the code aes(x=race) is actually a shortcut for aes(x = race, y = ..count..), where ..count.. is a special variable representing the frequency within each category. You can use this to calculate percentages, by specifying the y variable explicitly.

```
# plot the distribution as percentages
ggplot(Marriage,
aes(x = race,
y = ..count.. /sum(..count..))) +
geom_bar() +
labs(x ="Race",
y ="Percent",
title ="Participants by race") +
scale_y_continuous(labels = scales::percent)
```

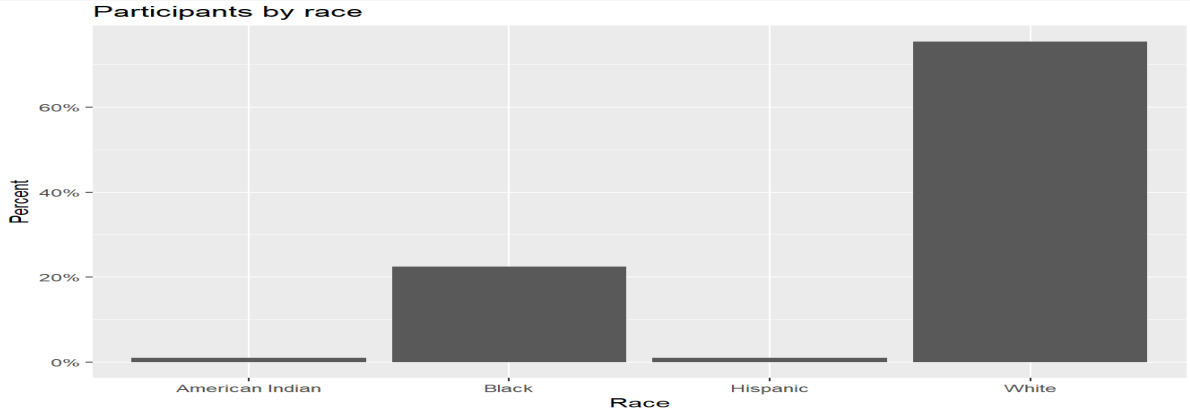


Figure: Barchart with percentages

In the code above, the scales package is used to add % symbols to the *y*-axis labels.

5.4.1.1.2 Sorting categories

It is often helpful to sort the bars by frequency. In the code below, the frequencies are calculated explicitly. Then the reorder function is used to sort the categories by the frequency. The option stat="identity" tells the plotting function not to calculate counts, because they are supplied directly.

```
# calculate number of participants in
# each race category
library(dplyr)
plotdata<-Marriage %>%
```

```
count(race)
```
The resulting dataset is give below.

Table 5.1: plotdata

race	**n**
American Indian	1
Black	22
Hispanic	1
White	74

This new dataset is then used to create the graph.

```
# plot the bars in ascending order
ggplot(plotdata,
aes(x =reorder(race, n),
y = n)) +
geom_bar(stat ="identity") +
labs(x ="Race",
y ="Frequency",
title ="Participants by race")
```

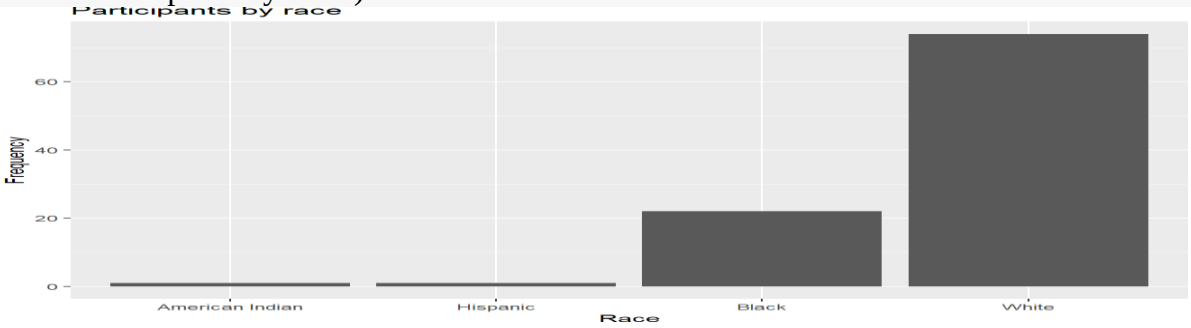


Figure: Sorted bar chart

The graph bars are sorted in ascending order. Use `reorder(race, -n)` to sort in descending order.

5.4.1.1.3 Labeling bars

Finally, you may want to label each bar with its numerical value.

```
# plot the bars with numeric labels
ggplot(plotdata,
aes(x = race,
y = n)) +
geom_bar(stat ="identity") +
geom_text(aes(label = n),
vjust=-0.5) +
labs(x ="Race",
y ="Frequency",
```

```
title ="Participants by race")
```

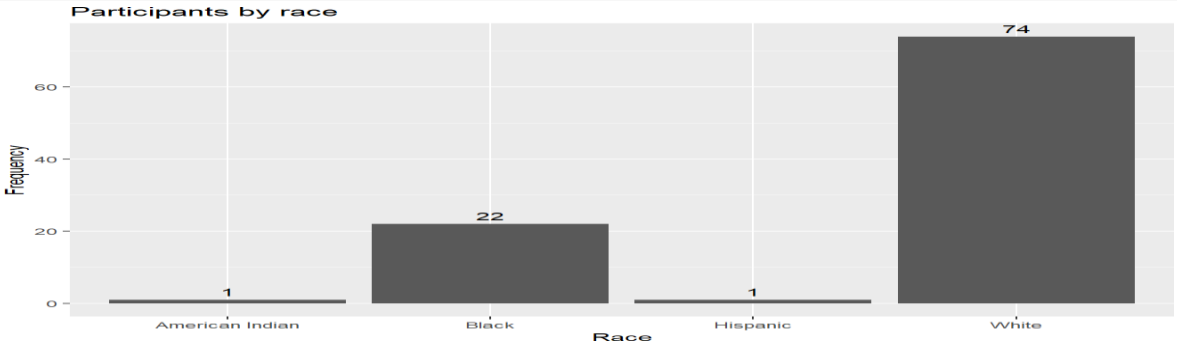


Figure: Bar chart with numeric labels

5.4.1.1.4 Overlapping labels

Category labels may overlap if (1) there are many categories or (2) the labels are long. Consider the distribution of marriage officials.

```
# basic bar chart with overlapping labels
ggplot(Marriage, aes(x =officialTitle)) +
geom_bar() +
labs(x ="Officiate",
y ="Frequency",
title ="Marriages by officiate")
```

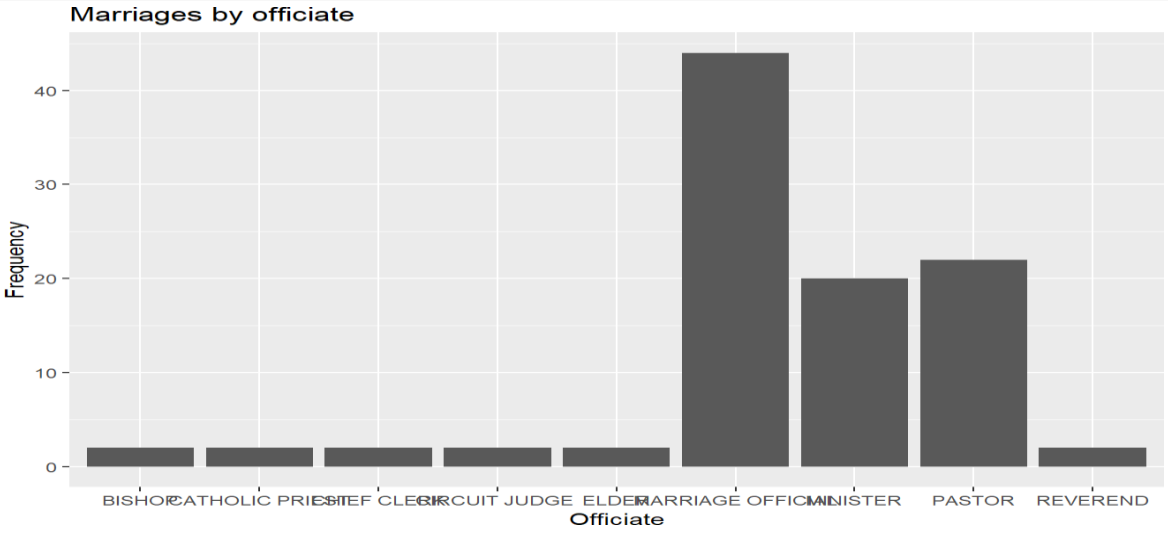


Figure: Barchart with problematic labels

5.4.1.2 Pie chart

Pie charts are controversial in statistics. If your goal is to compare the frequency of categories, you are better off with bar charts (humans are better at judging the length of bars than the volume of pie slices). If your goal is compare each category with the the whole (e.g., what portion of participants are Hispanic compared to all participants), and the number of categories is small, then pie charts may work for you. It takes a bit more code to make an attractive pie chart in R.

```
# create a basic ggplot2 pie chart
plotdata<-Marriage %>%
count(race) %>%
arrange(desc(race)) %>%
mutate(prop =round(n *100/sum(n), 1),
```

```
lab.ypos =cumsum(prop) -0.5*prop)

ggplot(plotdata,
aes(x ="",
y = prop,
fill = race)) +
geom_bar(width =1,
stat  ="identity",
color ="black") +
coord_polar("y",
start =0,
direction =-1) +
theme_void()
```

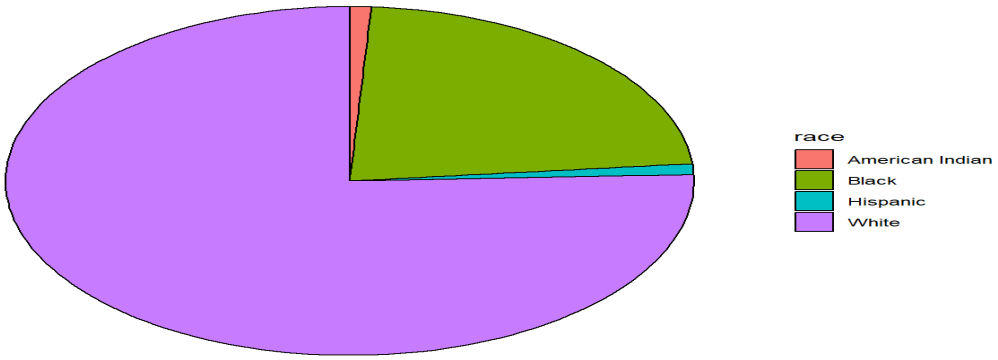


Figure: Basic pie chart

5.4.1.3 Tree map

An alternative to a pie chart is a tree map. Unlike pie charts, it can handle categorical variables that have *many* levels.

```
library(treemapify)

# create a treemap of marriage officials
plotdata<-Marriage %>%
count(officialTitle)

ggplot(plotdata,
aes(fill =officialTitle,
area = n)) +
geom_treemap() +
labs(title ="Marriages by officiate")
```

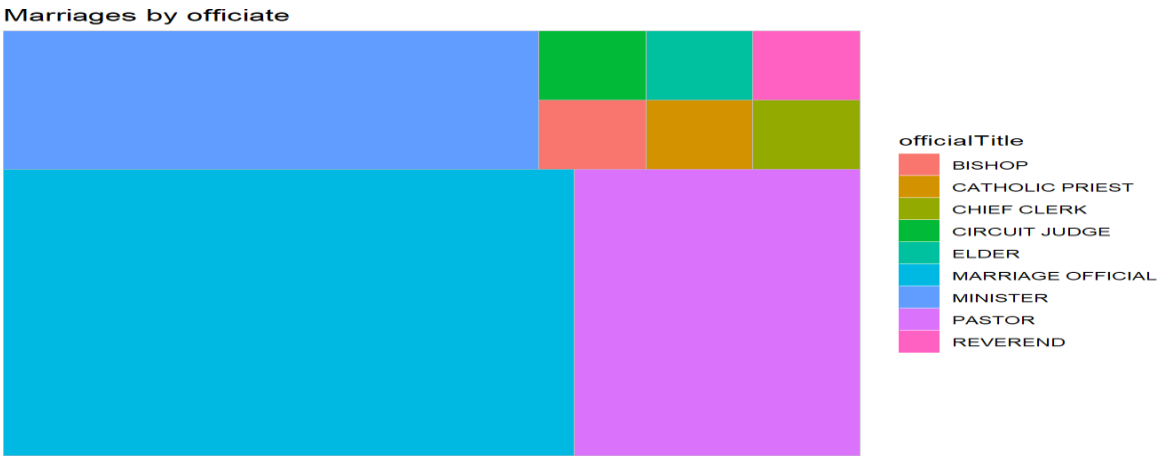


Figure: Basic treemap

5.4.2 Quantitative

The distribution of a single quantitative variable is typically plotted with a histogram, kernel density plot, or dot plot.

5.4.2.1 Histogram

Using the Marriage dataset, let's plot the ages of the wedding participants.

```
library(ggplot2)

# plot the age distribution using a histogram
ggplot(Marriage, aes(x = age)) +
geom_histogram() +
labs(title ="Participants by age",
x ="Age")
```

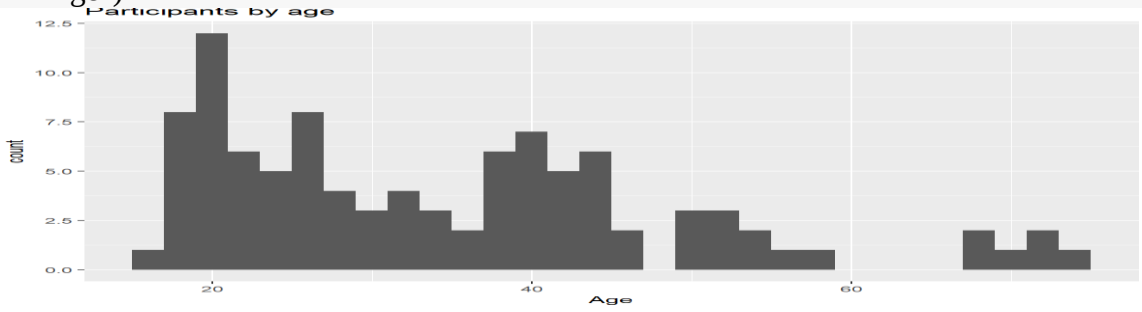


Figure: Basic histogram

5.4.2.1.1 Bins and bandwidths

One of the most important histogram options is bins, which controls the number of bins into which the numeric variable is divided (i.e., the number of bars in the plot). The default is 30, but it is helpful to try smaller and larger numbers to get a better impression of the shape of the distribution.

```
# plot the histogram with 20 bins
ggplot(Marriage, aes(x = age)) +
geom_histogram(fill ="cornflowerblue",
color ="white",
bins =20) +
labs(title="Participants by age",
```

```
subtitle ="number of bins = 20",
x ="Age")
```

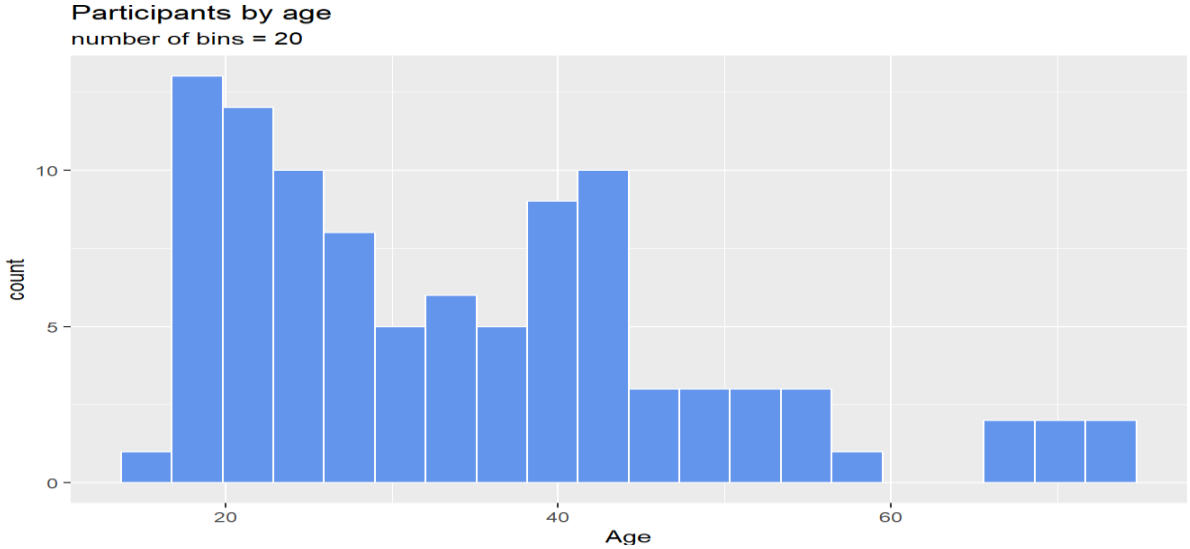


Figure: Histogram with a specified number of bins

5.4.2.2 Kernel Density plot

An alternative to a histogram is the kernel density plot. Technically, kernel density estimation is a nonparametric method for estimating the probability density function of a continuous random variable. (What??) Basically, we are trying to draw a smoothed histogram, where the area under the curve equals one.

```
# Create a kernel density plot of age
ggplot(Marriage, aes(x = age)) +
geom_density() +
labs(title ="Participants by age")
```

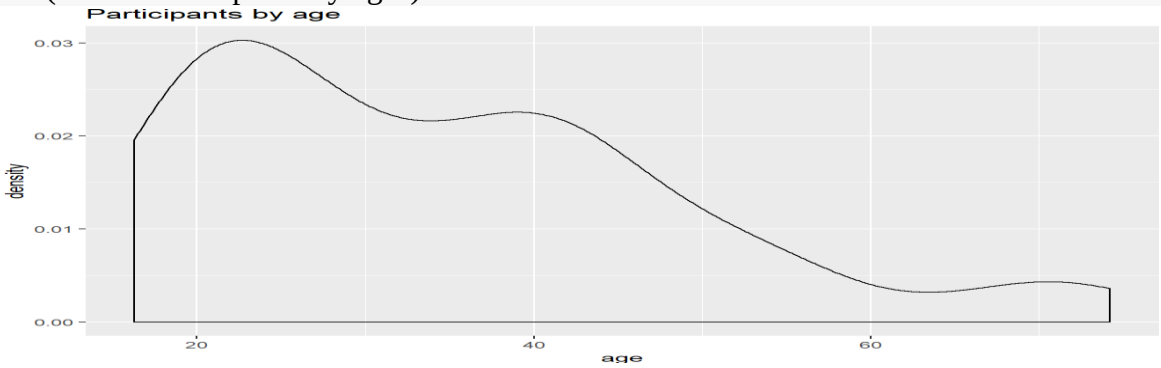


Figure: Basic kernel density plot

5.4.2.2.1 Smoothing parameter

The degree of smoothness is controlled by the bandwidth parameter `bw`. To find the default value for a particular variable, use the `bw.nrd0` function. Values that are larger will result in more smoothing, while values that are smaller will produce less smoothing.

```
# default bandwidth for the age variable
bw.nrd0(Marriage$age)
## [1] 5.181946
# Create a kernel density plot of age
```

```
ggplot(Marriage, aes(x = age)) +
geom_density(fill ="deepskyblue",
bw =1) +
labs(title ="Participants by age",
subtitle ="bandwidth = 1")
```

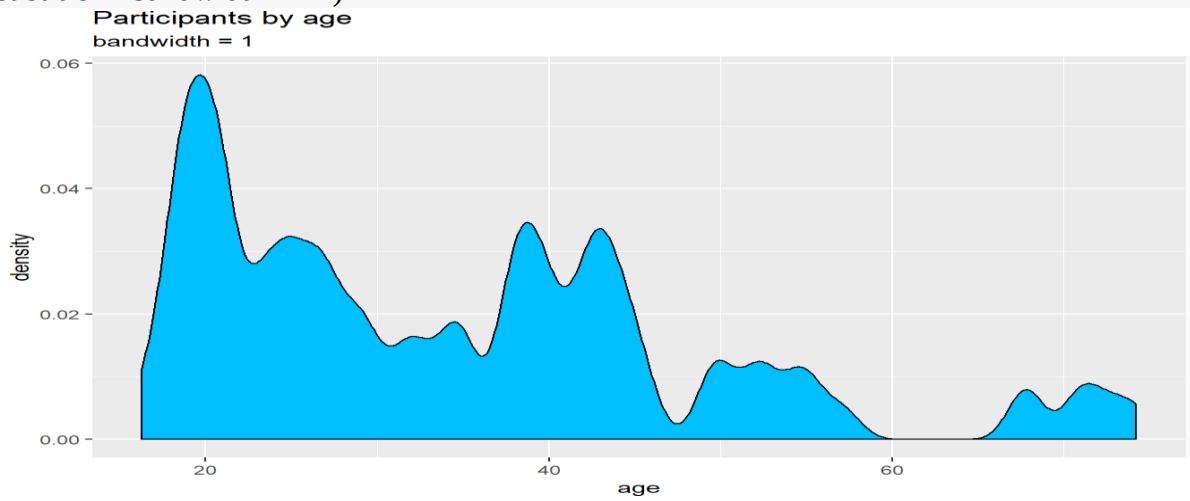


Figure: Kernel density plot with a specified bandwidth

5.4.2.3 Dot Chart

Another alternative to the histogram is the dot chart. Again, the quantitative variable is divided into bins, but rather than summary bars, each observation is represented by a dot. By default, the width of a dot corresponds to the bin width, and dots are stacked, with each dot representing one observation. This works best when the number of observations is small (say, less than 150).

```
# plot the age distribution using a dotplot
ggplot(Marriage, aes(x = age)) +
geom_dotplot() +
labs(title ="Participants by age",
y ="Proportion",
x ="Age")
```

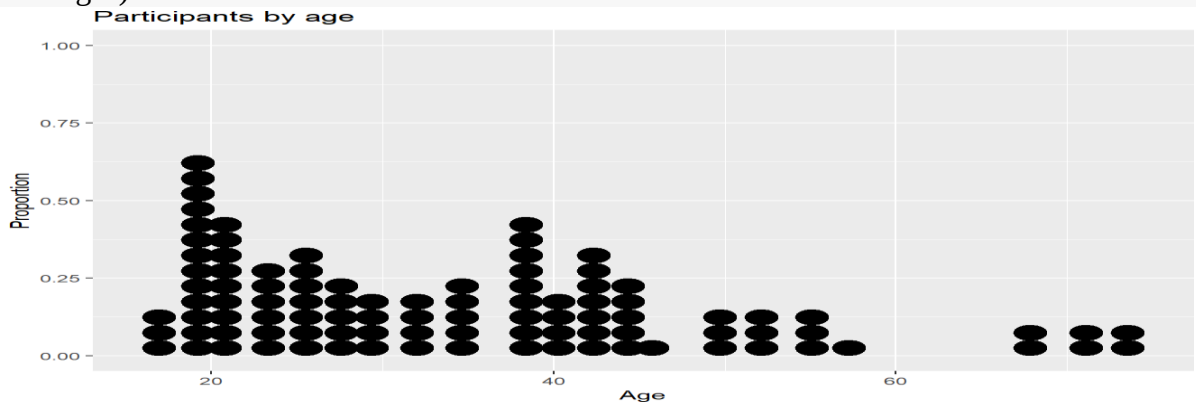


Figure: Basic dotplot

5.5 Bivariate Graphs:

Bivariate graphs display the relationship between two variables. The type of graph will depend on the measurement level of the variables (categorical or quantitative).

5.5.1 Categorical vs. Categorical

When plotting the relationship between two categorical variables, stacked, grouped, or segmented bar charts are typically used. A less common approach is the mosaic chart.

5.5.1.1 Stacked bar chart

Let's plot the relationship between automobile class and drive type (front-wheel, rear-wheel, or 4-wheel drive) for the automobiles in the Fuel economy dataset.

```
library(ggplot2)

# stacked bar chart
ggplot(mpg,
aes(x = class,
fill =drv)) +
geom_bar(position ="stack")
```

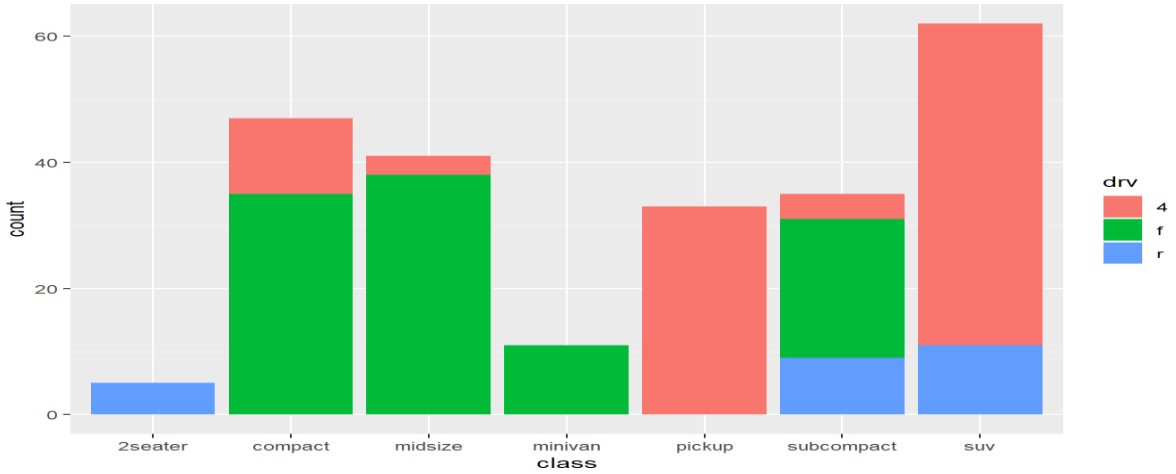


Figure: Stacked bar chart

Stacked is the default, so the last line could have also been written as geom_bar().

5.5.1.2 Grouped bar chart

Grouped bar charts place bars for the second categorical variable side-by-side. To create a grouped bar plot use the position = "dodge" option.

```
library(ggplot2)

# grouped bar plot
ggplot(mpg,
aes(x = class,
fill =drv)) +
geom_bar(position ="dodge")
```

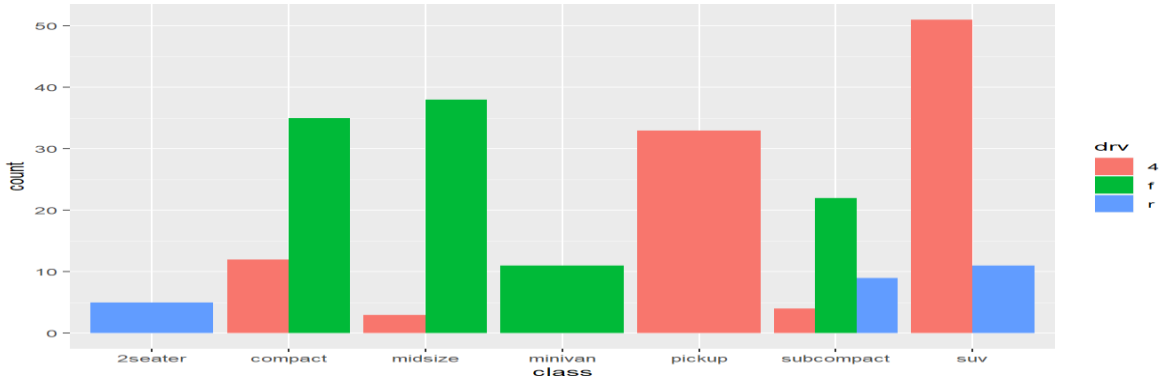


Figure: Side-by-side bar chart

5.5.1.3 Segmented bar chart

A segmented bar plot is a stacked bar plot where each bar represents 100 percent. You can create a segmented bar chart using the position = "filled" option.

```
library(ggplot2)

# bar plot, with each bar representing 100%
ggplot(mpg,
aes(x = class,
fill =drv)) +
geom_bar(position ="fill") +
labs(y ="Proportion")
```

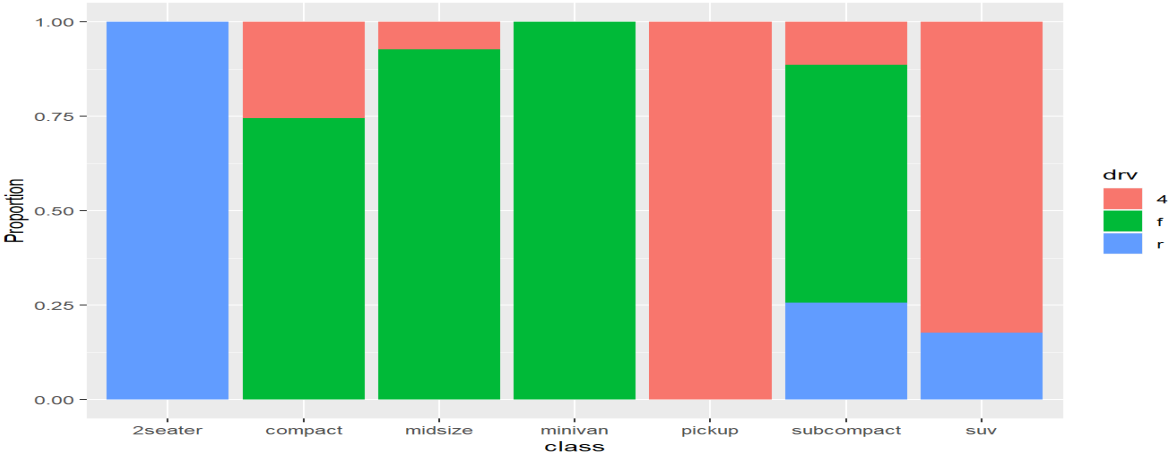


Figure: Segmented bar chart

5.5.1.4 Improving the color and labeling

You can use additional options to improve color and labeling. In the graph below

- factor modifies the order of the categories for the class variable and both the order and the labels for the drive variable
- scale_y_continuous modifies the y-axis tick mark labels
- labs provides a title and changed the labels for the x and y axes and the legend
- scale_fill_brewer changes the fill color scheme
- theme_minimal removes the grey background and changed the grid color

```
library(ggplot2)
```

```
# bar plot, with each  bar representing  100%,
# reordered bars, and better labels and colors
library(scales)
ggplot(mpg,
aes(x =factor(class,
levels =c("2seater",  "subcompact",
"compact", "midsize",
"minivan",  "suv",  "pickup")),
fill =factor(drv,
levels =c("f", "r", "4"),
labels =c("front-wheel",
"rear-wheel",
"4-wheel")))) +
geom_bar(position ="fill") +
scale_y_continuous(breaks =seq(0, 1, .2),
label =  percent) +
scale_fill_brewer(palette ="Set2")  +
labs(y ="Percent",
fill ="Drive Train",
x ="Class",
title ="Automobile Drive by Class") +
theme_minimal()
```

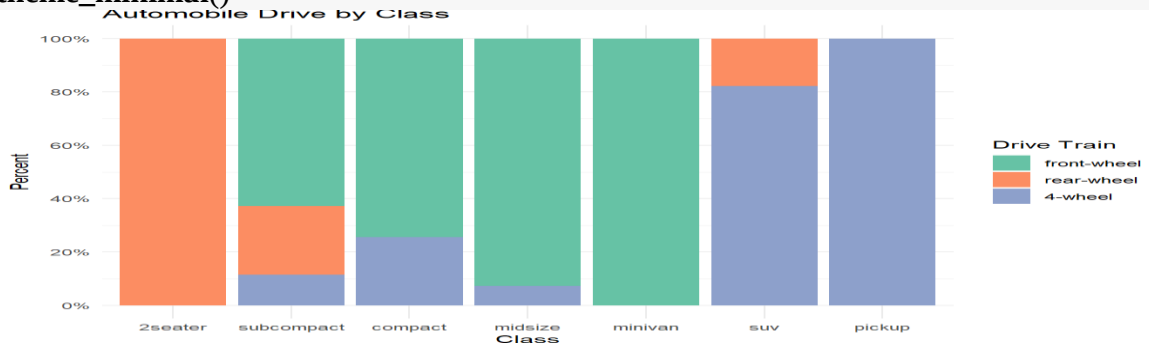


Figure: Segmented bar chart with improved labeling and color

5.5.1.5 Other plots

Mosaic plots provide an alternative to stacked bar charts for displaying the relationship between categorical variables. They can also provide more sophisticated statistical information.

5.5.2 Quantitative vs. Quantitative

The relationship between two quantitative variables is typically displayed using scatterplots and line graphs.

5.5.2.1 Scatterplot

The simplest display of two quantitative variables is a scatterplot, with each variable represented on an axis. For example, using the Salaries dataset, we can plot experience (*yrs.since.phd*) vs. academic salary (*salary*) for college professors.

```
library(ggplot2)
data(Salaries,  package="carData")
```

```
# simple scatterplot
ggplot(Salaries,
aes(x =yrs.since.phd,
y = salary)) +
geom_point()
```

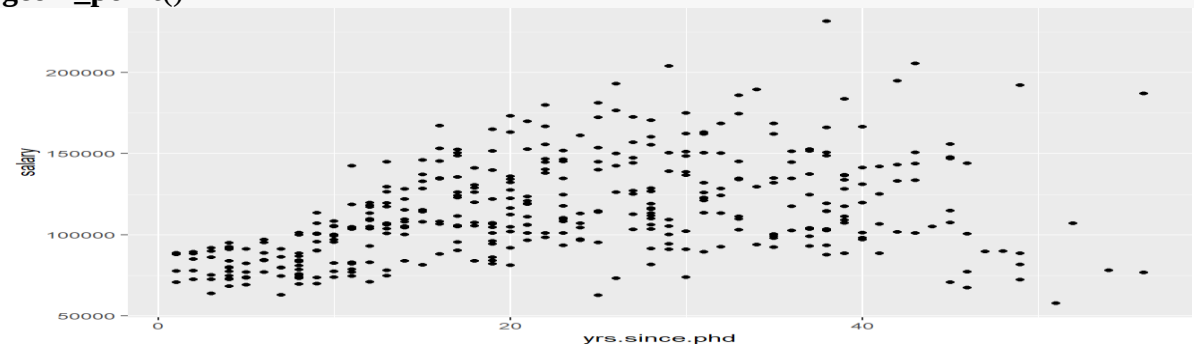


Figure: Simple scatterplot

5.5.2.1.1 Adding best fit lines

It is often useful to summarize the relationship displayed in the scatterplot, using a best fit line. Many types of lines are supported, including linear, polynomial, and nonparametric (loess). By default, 95% confidence limits for these lines are displayed.

```
# scatterplot with linear fit line
ggplot(Salaries,
aes(x =yrs.since.phd,
y = salary)) +
geom_point(color="steelblue") +
geom_smooth(method ="lm")
```

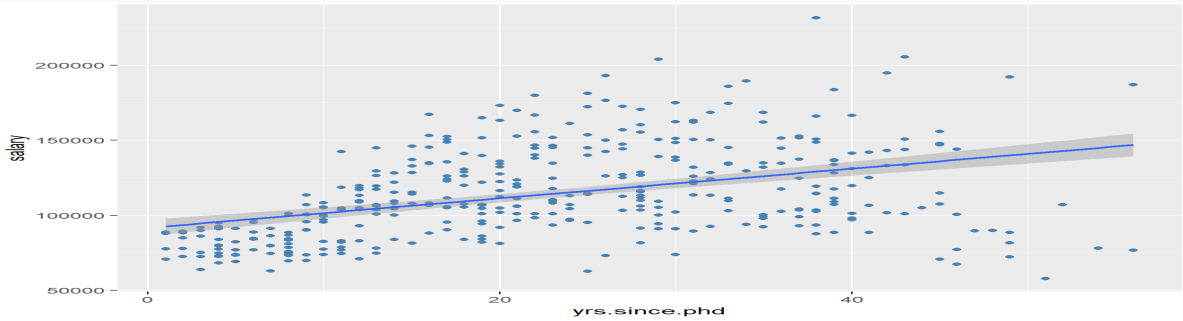


Figure: Scatterplot with linear fit line

5.5.2.2 Line plot

When one of the two variables represents time, a line plot can be an effective method of displa y-ing relationship. For example, the code below displays the relationship between time (*year*) and life expectancy (*lifeExp*) in the United States between 1952 and 2007. The data comes from the gapminder dataset.

```
data(gapminder, package="gapminder")

# Select US cases
library(dplyr)
plotdata<-filter(gapminder,
                 country == "United States")
```

```
# simple line plot
ggplot(plotdata,
aes(x = year,
y =lifeExp)) +
geom_line()
```

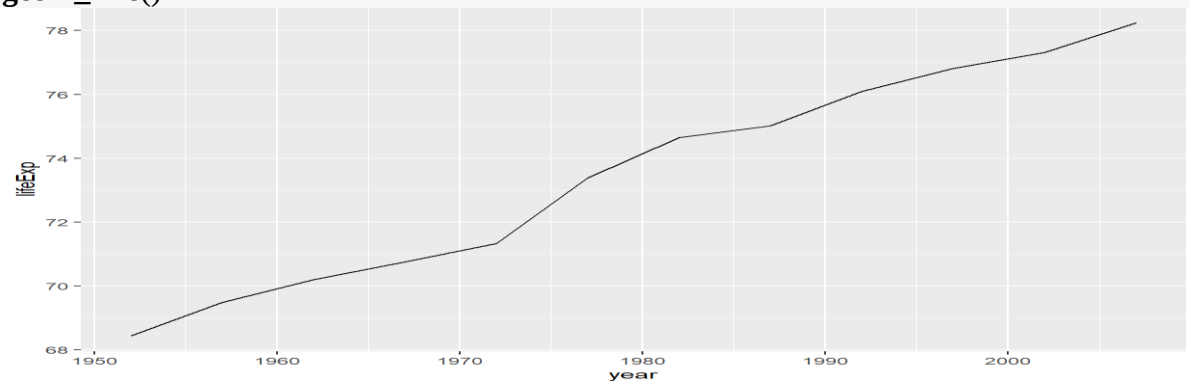


Figure: Simple line plot

5.5.3 Categorical vs. Quantitative

When plotting the relationship between a categorical variable and a quantitative variable, a large number of graph types are available. These include bar charts using summary statistics, grouped kernel density plots, side-by-side box plots, side-by-side violin plots, mean/sem plots, ridgeline plots, and Cleveland plots.

5.5.3.1 Bar chart (on summary statistics)

In previous sections, bar charts were used to display the number of cases by category for a single variable or for two variables. You can also use bar charts to display other summary statistics (e.g., means or medians) on a quantitative variable for each level of a categorical variable.

For example, the following graph displays the mean salary for a sample of university professors by their academic rank.

```
data(Salaries, package="carData")

# calculate mean salary for each rank
library(dplyr)
plotdata<-Salaries %>%
group_by(rank) %>%
summarize(mean_salary =mean(salary))

# plot mean salaries
ggplot(plotdata,
aes(x = rank,
y =mean_salary)) +
geom_bar(stat ="identity")
```

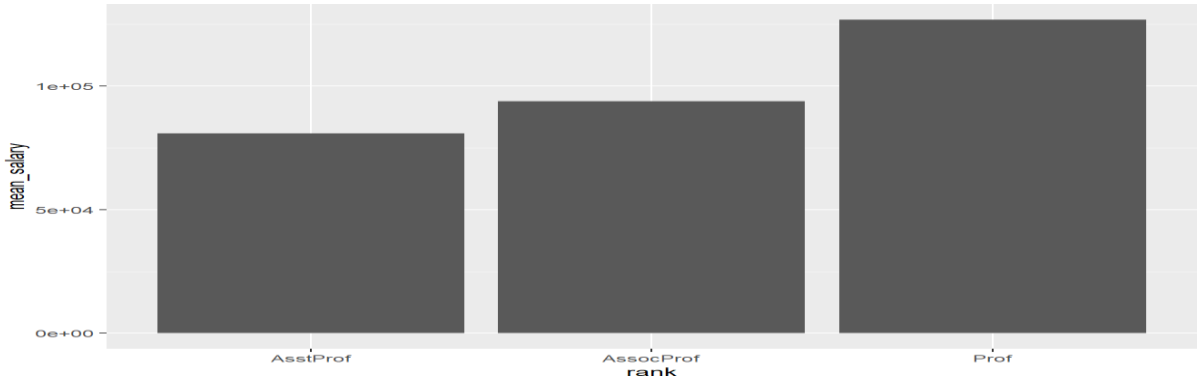


Figure: Bar chart displaying means

5.5.3.2 Grouped kernel density plots

One can compare groups on a numeric variable by superimposing kernel density plots in a single graph.

```
# plot the distribution of salaries
# by rank using kernel density plots
ggplot(Salaries,
aes(x = salary,
fill = rank)) +
geom_density(alpha =0.4) +
labs(title ="Salary distribution by rank")
```

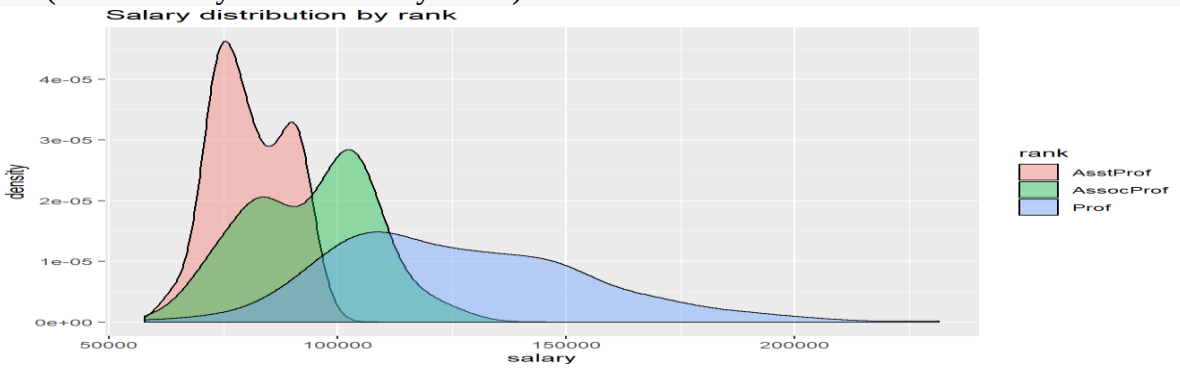
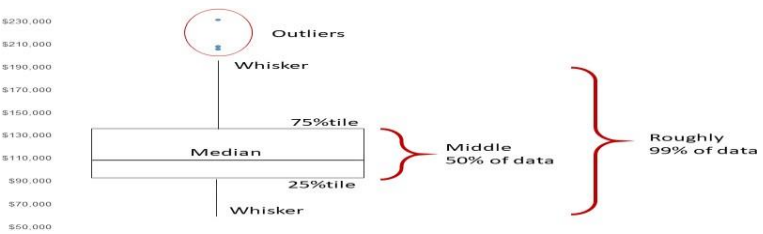


Figure: Grouped kernel density plots

5.5.3.3 Box plots

A boxplot displays the 25th percentile, median, and 75th percentile of a distribution. The whiskers (vertical lines) capture roughly 99% of a normal distribution, and observations outside this range are plotted as points representing outliers (see the figure below).



Side-by-side box plots are very useful for comparing groups (i.e., the levels of a categorical variable) on a numerical variable.

```
# plot the distribution of salaries by rank using boxplots
ggplot(Salaries,
aes(x = rank,
y = salary)) +
geom_boxplot() +
labs(title ="Salary distribution by rank")
```

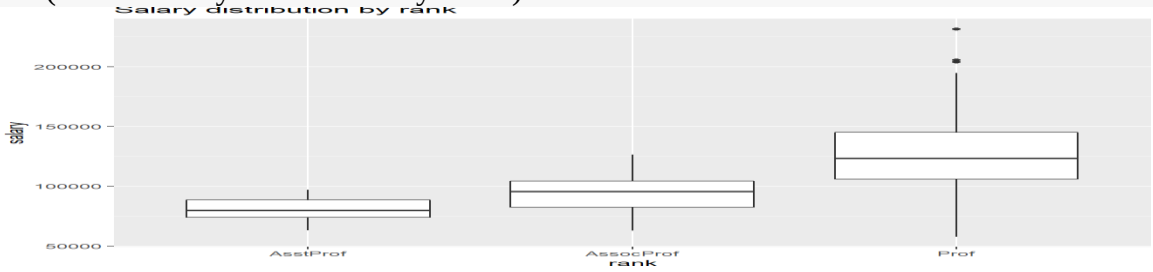


Figure: Side-by-side boxplots

5.5.3.4 Violin plots

Violin plots are similar to kernel density plots, but are mirrored and rotated 90°.

```
# plot the distribution of salaries
# by rank using violin plots
ggplot(Salaries,
aes(x = rank,
y = salary)) +
geom_violin() +
labs(title ="Salary distribution by rank")
```

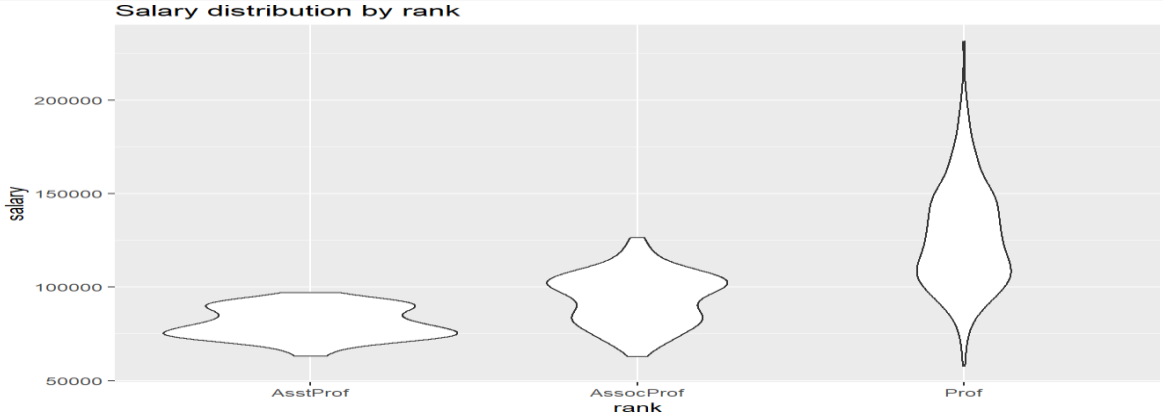


Figure: Side-by-side violin plots

5.5.3.5 Ridgeline plots

A ridgeline plot (also called a joyplot) displays the distribution of a quantitative variable for several groups. They're similar to kernel density plots with vertical faceting, but take up less room. Ridgeline plots are created with the ggridges package.
Using the Fuel economy dataset, let's plot the distribution of city driving miles per gallon by car class.

```
# create ridgeline graph
library(ggplot2)
library(ggridges)

ggplot(mpg,
```

```
aes(x =cty,
y = class,
fill = class)) +
geom_density_ridges() +
theme_ridges() +
labs("Highway mileage by auto class") +
theme(legend.position ="none")
```

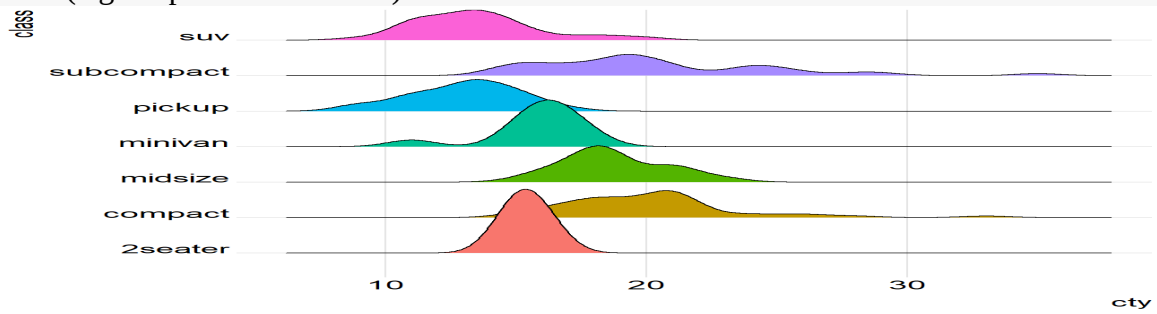


Figure: Ridgeline graph with color fill

5.5.3.6 Mean/SEM plots

A popular method for comparing groups on a numeric variable is the mean plot with error bars. Error bars can represent standard deviations, standard error of the mean, or confidence intervals. In this section, we'll plot means and standard errors.

```
# calculate means, standard deviations,
#  standard errors, and 95% confidence
# intervals by rank
library(dplyr)
plotdata<-Salaries %>%
group_by(rank) %>%
summarize(n =n(),
mean =mean(salary),
sd =sd(salary),
se =sd/sqrt(n),
ci =qt(0.975, df = n -1) *sd/sqrt(n))
```

The resulting dataset is given below.

Table 4.1: Plot data

rank	n	mean	sd	se	ci
AsstProf	67	80775.99	8174.113	998.6268	1993.823
AssocProf	64	93876.44	13831.700	1728.9625	3455.056
Prof	266	126772.11	27718.675	1699.5410	3346.322

```
# plot the means and standard errors
ggplot(plotdata,
aes(x = rank,
y = mean,
```

```
group =1)) +
geom_point(size =3) +
geom_line() +
geom_errorbar(aes(ymin = mean -se,
ymax = mean +se),
width = .1)
```

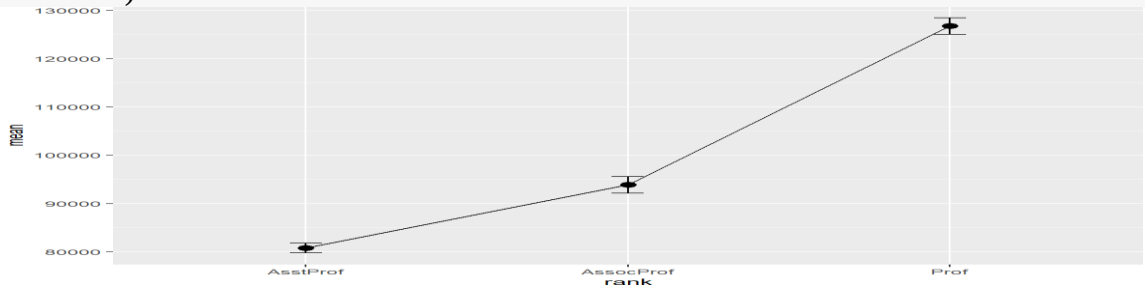


Figure: Mean plots with standard error bars

5.5.3.7 Strip plots

The relationship between a grouping variable and a numeric variable can be displayed with a scatter plot. For example

```
# plot the distribution of salaries
# by rank using strip plots
ggplot(Salaries,
aes(y = rank,
x = salary)) +
geom_point() +
labs(title ="Salary distribution by rank")
```

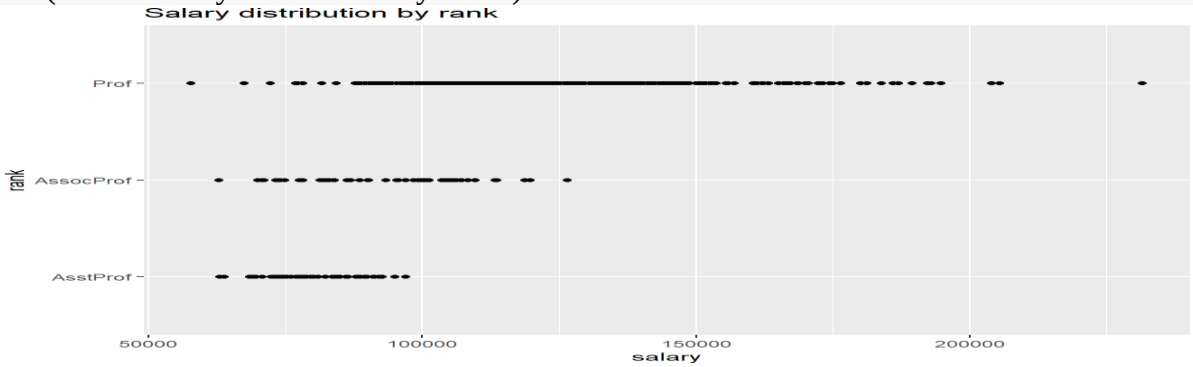


Figure: Categorical by quantiative scatterplot

5.5.3.7.1 Combining jitter and boxplots

It may be easier to visualize distributions if we add boxplots to the jitter plots.

```
# plot the distribution of salaries
# by rank using jittering
library(scales)
ggplot(Salaries,
aes(x =factor(rank,
labels =c("Assistant\nProfessor",
"Associate\nProfessor",
"Full\nProfessor")),
y = salary,
```

```
color = rank)) +
geom_boxplot(size=1,
outlier.shape =1,
outlier.color ="black",
outlier.size =3) +
geom_jitter(alpha =0.5,
width=.2) +
scale_y_continuous(label = dollar) +
labs(title ="Academic Salary by Rank",
subtitle ="9-month salary for 2008-2009",
x ="",
y ="") +
theme_minimal() +
theme(legend.position ="none") +
coord_flip()
```

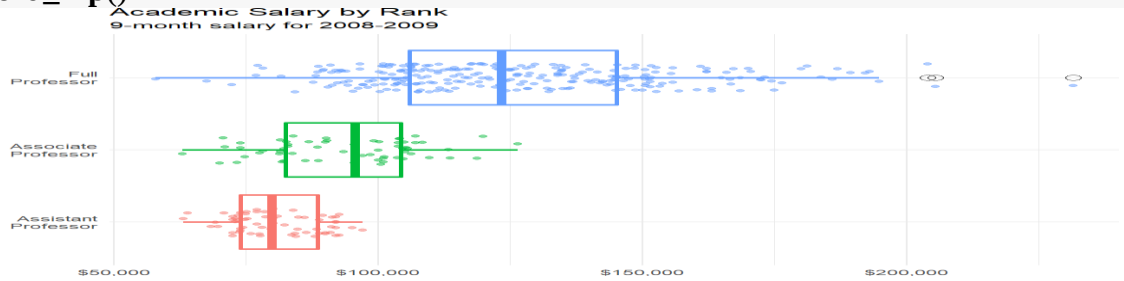


Figure: Jitter plot with superimposed box plots

5.5.3.8 Beeswarm Plots

Beeswarm plots (also called violin scatter plots) are similar to jittered scatterplots, in that they display the distribution of a quantitative variable by plotting points in way that reduces overlap. In addition, they also help display the density of the data at each point (in a manner that is similar to a violin plot). Continuing the previous example

```
# plot the distribution of salaries
# by rank using beewarm-syle plots
library(ggbeeswarm)
library(scales)
ggplot(Salaries,
aes(x =factor(rank,
labels =c("Assistant\nProfessor",
"Associate\nProfessor",
"Full\nProfessor")),
y = salary,
color = rank)) +
geom_quasirandom(alpha =0.7,
size =1.5) +
scale_y_continuous(label = dollar) +
labs(title ="Academic Salary by Rank",
subtitle ="9-month salary for 2008-2009",
x ="",
y ="") +
theme_minimal() +
theme(legend.position ="none")
```

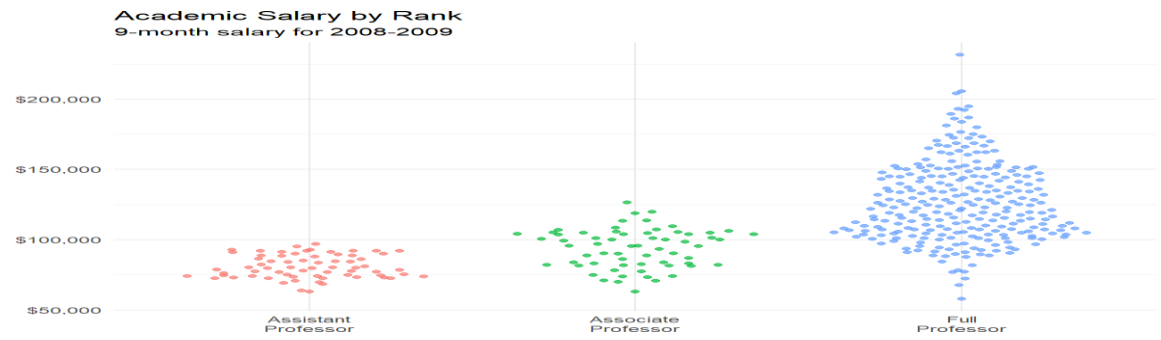


Figure: Beeswarm plot

.5.5.3.9 Cleveland Dot Charts

Cleveland plots are useful when you want to compare a numeric statistic for a large number of groups. For example, say that you want to compare the 2007 life expectancy for Asian country using the gapminder dataset.

```
data(gapminder, package="gapminder")

# subset Asian countries in 2007
library(dplyr)
plotdata<-gapminder%>%
filter(continent == "Asia"&
year ==2007)

# basic Cleveland plot of life expectancy by country
ggplot(plotdata,
aes(x=lifeExp, y = country)) +
geom_point()
```

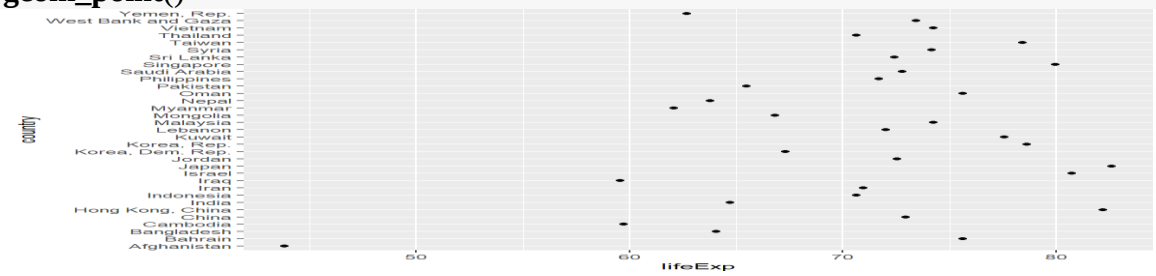


Figure: Basic Cleveland dot plot

5.6 Multivariate Graphs:

Multivariate graphs display the relationships among three or more variables. There are two common methods for accommodating multiple variables: grouping and faceting.

5.6.1 Grouping

In grouping, the values of the first two variables are mapped to the *x* and *y* axes. Then additional variables are mapped to other visual characteristics such as color, shape, size, line type, and transparency. Grouping allows you to plot the data for multiple groups in a single graph.

Using the Salaries dataset, let's display the relationship between *yrs.since.phd* and *salary*.

```
library(ggplot2)
data(Salaries, package="carData")
```

```
# plot experience vs. salary
ggplot(Salaries,
aes(x =yrs.since.phd,
y = salary)) +
geom_point() +
labs(title ="Academic salary by years since degree")
```

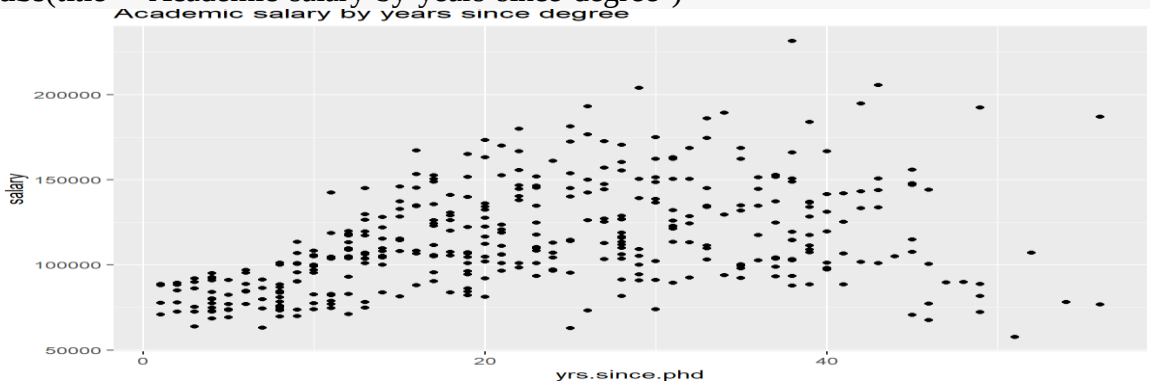


Figure: Simple scatterplot

Next, let's include the rank of the professor, using color.

```
# plot experience vs. salary (color represents rank)
ggplot(Salaries, aes(x =yrs.since.phd,
y = salary,
color=rank)) +
geom_point() +
labs(title ="Academic salary by rank and years since degree")
```

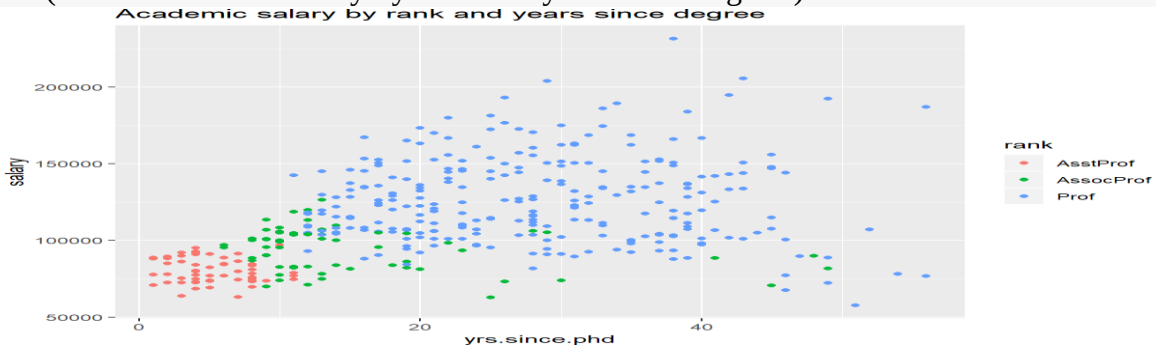


Figure: Scatterplot with color mapping

5.6.2 Faceting

In **faceting**, a graph consists of several separate plots or *small multiples*, one for each level of a third variable, or combination of variables. It is easiest to understand this with an example.

```
# plot salary histograms by rank
ggplot(Salaries, aes(x = salary)) +
geom_histogram(fill ="cornflowerblue",
color ="white") +
facet_wrap(~rank, ncol =1) +
labs(title ="Salary histograms by rank")
```

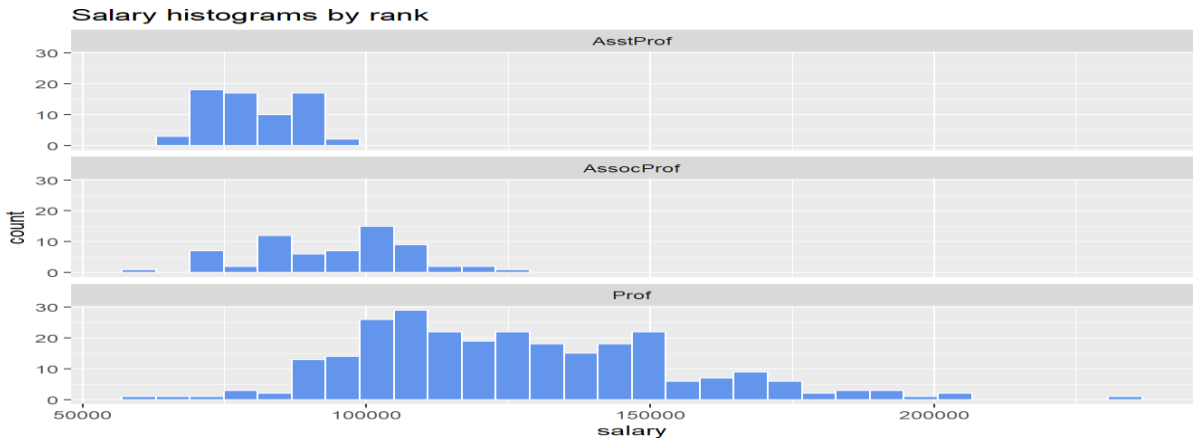


Figure: Salary distribution by rank

The facet_wrap function creates a separate graph for each level of rank. The ncol option controls the number of columns.

*****