DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DIGITAL NOTES

ON

DATA SCIENCE &ITS MODELING METHODS

(R22A6706)

B.TECH IV YEAR–I SEM
(R22) REGULATION
(2025-26)

Prepared by Mrs AGNISHA MANDAVA

MALLAREDDY COLLEGE OF ENGINEERING &TECHNOLOGY
(AutonomousInstitution–UGC,Govt.ofIndia)
Recognizedunder2(f)and12(B) ofUGC ACT1956
(AffiliatedtoJNTUH,Hyderabad,ApprovedbyAICTE-AccreditedbyNBA&NAAC–'A'Grade-ISO9001:2015Certified)

Maisammaguda,Dhulapally(PostVia.Hakimpet),Secunderabad–500100,TelanganaState,India

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Vision

To acknowledge quality education and instill high patterns of discipline making the students technologically superior and ethically strong which involves the improvement in the quality of life in human race.

Mission

- To achieve and impart holistic technical education using the best of infrastructure, outstanding technical and teaching expertise to establish the students into competent and confident engineers.
- Evolving the center of excellence through creative and innovative teaching learning practicesforpromotingacademicachievementtoproduceinternationallyacceptedcompetitiveand world class professionals.

PROGRAMME EDUCATIONAL OBJECTIVES (PEOs)

PEO1–ANALYTICALSKILLS

- To facilitate the graduates with the ability to visualize, gather information, articulate, analyze, solve complex problems, and make decisions. These are essential to address the challenges of complex and computation intensive problems increasing their productivity.

PEO2–TECHNICALSKILLS

- Tofacilitatethegraduateswiththetechnicalskillsthatpreparethemforimmediateemploymentandpursue certification providing a deeper understanding of the technology in advanced areas of computer science and related fields, thus encouraging pursuing higher education and research based on their interest.

PEO3–SOFTSKILLS

- To facilitate the graduates with the soft skills that include fulfilling the mission, setting goals, showing self confidence by communicating effectively, having a positive attitude, get involved in team-work, being a leader, managing their career and their life.

PEO4–PROFESSIONALETHICS

- To facilitate the graduates with the knowledge of professional and ethical responsibilities by paying attention to grooming, being conservative with style, following dress codes, safety codes, and adapting them to technological advancements.

PROGRAM SPECIFIC OUTCOMES (PSOs)

After the completion of the course, B.Tech Computer Science and Engineering, the graduates will have the following Program Specific Outcomes:

1.FundamentalsandcriticalknowledgeoftheComputerSystem:-
AbletoUnderstandtheworkingprinciples of the computer System and its components, Apply the knowledge to build, asses, and analyze the software and hardware aspects of it.

2.The comprehensive and Applicative knowledge of Software Development: Comprehensive skills of Programming Languages, Software process models, methodologies, and able to plan, develop, test, analyze, and manage the software and hardware intensive systems in heterogeneous platforms individually or working in teams.

3.Applications of Computing Domain & Research: Able to use the professional, managerial, interdisciplinary skill set, and domain specific tools in development processes, identify their search gaps, and provide innovative solutions to them.

PROGRAM OUTCOMES (POs)

Engineering Graduates should possess the following:

1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals,andanengineeringspecializationtothesolutionofcomplexengineeringproblems.

2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

3. Design / development of solutions: Design solutions for complex engineeringproblemsanddesignsystemcomponentsorprocessesthatmeetthespecifiedneedswithappropriateconsideration for thepublic health and safety, and the cultural, societal, and environmental considerations.

4. Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5. Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6. The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

7. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. Individual and team work: Function effectively as an individual, and as member or leader in diverse teams, and in multidisciplinary settings.

10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

MALLA REDDY COLLEGE OF ENGINEERING AND TECHNOLOGY

IV Year B.Tech. CSE- II Sem **L/T/P/C**

3 -/-/3

(R22A6706) DATA SCIENCE&ITS MODELING

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 and Overview of R Data Science Process: Roles in a data science project, Stages in a data science project, Setting expectations. 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 NA Values. Control Structures: 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:Modelling Methods-I: Choosing and evaluating Models Mapping problems to machine learning tasks: Classification problems, Scoring problems, Grouping: working without known targets, Problem-to-method mapping, Evaluating models: Over fitting, Measures of model performance, Evaluating classification models, Evaluating scoring models, Evaluating probability model.

UNIT IV: **Modelling Methods-II**: Linear and logistic regression Using linear regression: Understanding linear regression, Building a linear regression model, making predictions. Using logistic regression: Understanding logistic regression, Building a logistic regression model, making predictions.

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.

TEXT BOOKS:

1. Practical Data Science with R, Nina Zumel & John Mount , Manning PublicationsNY, 2014.
2. Beginning Data Science in R-Data Analysis, Visualization, and Modellingfor 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 OReilly 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:

1. Analyze the basics in R programming in terms of constructs, control statements,Functions.
2. Implement Data Preprocessing using R Libraries.
3. Apply the R programming from a statistical perspective and ModelingMethods.
4. Build regression models for a given problem.
5. 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. 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 NA Values. 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 how products 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 betterunderstand 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'regoing 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 goodvector 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 by row 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 complexdata 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

We saw basic indexing in Chapter 1, but there is much more to indexing in R that that. Type ?`[[` into the R prompt and prepare to be amazed. We have already seen the basic indexing. If you want the nth element of a vector v, you use v[n]:

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

You 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,

1. we create a vector.
2. 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 offactor() function

```
factor_data<- factor(vector)
```

```
data<-
c("Shubham","Nishka","Arpita","Nishka","Shubham","Sumit","Nishka","Shubham","Sumit
","Arpita","Sumit")
print(data)
print(is.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","Arpit
a","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,T
RUE)])
```

[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)
```

[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 10if(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 think

about the body of a function as an expression, but typically it is a sequence of statements enclosed by curly

brackets:

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

In the above function calls, the argument matching of formal argument to the actual arguments takes place in positional order.
This means that, in the call pow(8,2), the formal arguments x and y are assigned 8 and 2 respectively.
We can also call the function using named arguments.
When calling a function in this way, the order of the actual arguments doesn't matter.
For example, all of the function calls given below are equivalent.

```
> pow(8, 2)
[1] "8 raised to the power 2 is 64"
> pow(x = 8, y = 2)
[1] "8 raised to the power 2 is 64"
> pow(y = 2, x = 8)
[1] "8 raised to the power 2 is 64"
```

Furthermore, we can use named and unnamed arguments in a single call.
In such case, all the named arguments are matched first and then the remaining unnamed arguments are matched in a positional order.

```
> pow(x=8, 2)
[1] "8 raised to the power 2 is 64"
> pow(2, x=8)
[1] "8 raised to the power 2 is 64"
```

In all the examples above, x gets the value 8 and y gets the value 2.
Default Values for Arguments
We can assign default values to arguments in a function in R.
This is done by providing an appropriate value to the formal argument in the function declaration.
Here is the above function with a default value for y.

```
pow <- function(x, y = 2) {
 # function to print x raised to the power y
 result <- x^y
 print(paste(x,"raised to the power", y, "is", result))
}
```

The use of default value to an argument makes it optional when calling the function.

```
> pow(3)
[1] "3 raised to the power 2 is 9"
```

```
> pow(3,1)
[1] "3 raised to the power 1 is 3"
```

Here, y is optional and will take the value 2 when not provided.

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,

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.

- 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]
```

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 orten 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 importreader package.

Syntax:

```
read_lines(file)file:
the file path
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. Thedata 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 thedata 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 pointand 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 helpof 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 commandwith 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 Michelle611     11/15/2014 IT
4Ryan       729     5/11/2014 HR
5Gary       843.253/27/2015Finance
6Nina        5785/21/2013  IT
7Simon 632.87/30/2013Operations
8Guru722.5 6/17/2014Finance
```

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")
```

To 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"
```

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.

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 issue SQL queries from R. SQL is the most common database query language and allows us to specify arbitrary joins and aggregations. SQL is called a declarative language (as opposed to a procedural language) because in SQL we specify what relations we would like our data sample to have, not how to

compute them. For our example, we load a
sample 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 files into our database and a useful sample has been loaded into R for analysis. We have actually accomplished a lot. Generating, as we have, a uniform sample of households and 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 (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
..	...	...	...	...	...	...	...	...	...	...	...

> myirisdata

	Sepal.Length (dbl)	Sepal.Width (dbl)	Petal.Length (dbl)	Petal.Width (dbl)	Species (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 (dbl)	hp (dbl)	drat (dbl)	wt (dbl)	qsec (dbl)	vs (dbl)	am (dbl)	gear (dbl)	carb (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 (dbl)	hp (dbl)	drat (dbl)	wt (dbl)	qsec (dbl)	vs (dbl)	am (dbl)	gear (dbl)	carb (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 (dbl)	disp (dbl)	hp (dbl)	drat (dbl)	wt (dbl)	qsec (dbl)	vs (dbl)	am (dbl)	gear (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 (fctr)	Average (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 (fctr)	Sepal.Length_mean (dbl)	Sepal.Width_mean (dbl)	Sepal.Length_n (int)	Sepal.Width_n (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 5
 [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 7
 [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 8
 [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 7
[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 7
[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.
 [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.
 [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.
 [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.
 [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.
[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.
[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.
[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
```

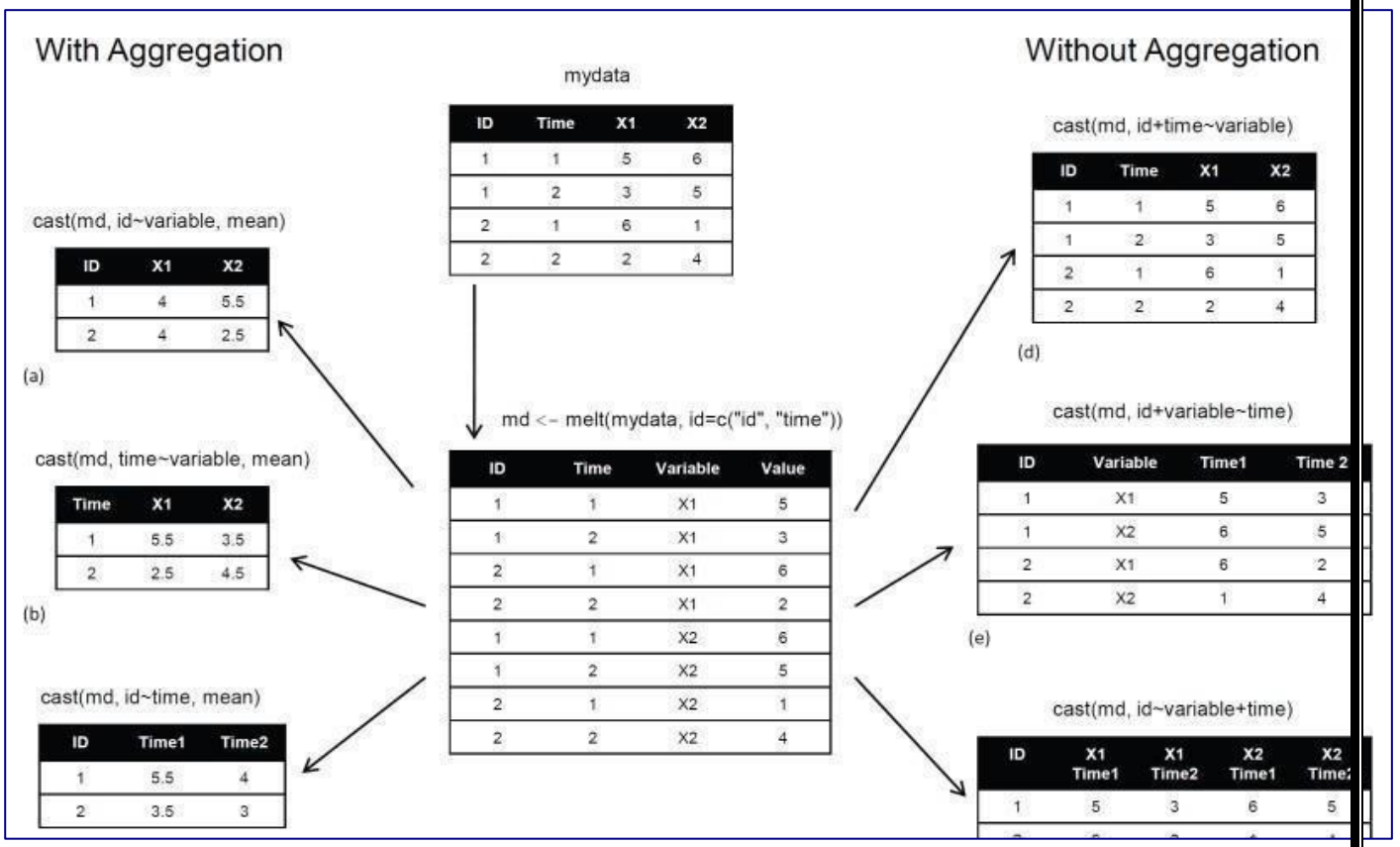
	ID	Names	variable	value
1	1	Joseph	DateofBirth	1993
2	2	Matrin	DateofBirth	1992
3	3	Joseph	DateofBirth	1993
4	4	James	DateofBirth	1994
5	5	Matrin	DateofBirth	1992
6	1	Joseph	Subject	Maths
7	2	Matrin	Subject	Biology
8	3	Joseph	Subject	Science
9	4	James	Subject	Psycology
10	5	Matrin	Subject	Physics

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)
```

n_time.hour	n_time.minute	n_time.second	n_time.month	n_time.year
13	27	41.65723	12	2015

```
> new_data
```

UNIT-III

Modelling Methods-I: Choosing and evaluating Models

Mapping problems to machine learning tasks: Classification problems, Scoring problems, Grouping: working without known targets, Problem-to-method mapping, Evaluating models: Over fitting, Measures of model performance, Evaluating classification models, Evaluating scoring models, Evaluating probability model.

There are a number of business problems that your team might be called on to address:

- Predicting what customers might buy, based on past transactions
- Identifying fraudulent transactions
- Determining price elasticity (the rate at which a price increase will decrease
- sales, and vice versa) of various products or product classes
- Determining the best way to present product listings when a customer searches

for an item Customer segmentation: grouping customers with similar purchasing behavior

- AdWord valuation: how much the company should spend to buy certain AdWords on search engines
- Evaluating marketing campaigns
- Organizing new products into a product catalog

Your intended uses of the model have a big influence on what methods you should use. If you want to know how small variations in input variables affect outcome, then you likely want to use a regression method. If you want to know what single variable drives most of a categorization, then decision trees might be a good choice. Also, each business problem suggests a statistical approach to try. If you're trying to predict scores, some sort of regression is likely a good choice; if you're trying to predict categories, then something like random forests is probably a good choice.

Solving classification problems

Suppose your task is to automate the assignment of new products to your company's product categories, This can be more complicated than it sounds. Products that come from different sources may have their own product classification that doesn't coincide with the one that you use on your retail site, or they may come without any classification at all. Many large online retailers use teams of human taggers to hand-categorize their products. This is not only labor-intensive, but inconsistent and error-prone. Automation is an attractive option; it's labor-saving, and can improve the quality of the retail site. Product categorization based on product attributes and/or text descriptions of the product is an example of classification: deciding how to assign (known) labels to an object. Classification itself is an example of what is called supervised learning: in order to learn how to classify objects, you need a dataset of objects that have already been classified (called the training set). Building training data is the major expense for most classification tasks, especially text-related ones.

Naive Bayes:

Naive Bayes classifiers are especially useful for problems with many input variables, categorical

input variables with a very large number of possible values, and text classification. Naive Bayes would be a good first attempt at solving the product categorization problem.

Decision trees:

Decision trees are useful when input variables interact with the output in "if-then" kinds of ways (such as IF age > 65, THEN has.health.insurance=T). They are also suitable when inputs have an AND relationship to each other (such as IF age < 25 AND student=T, THEN...) or when input variables are redundant or correlated. The decision rules that come from a decision tree are in principle easier for nontechnical users to understand than the decision processes that come from other classifiers.

Logistic regression:

Logistic regression is appropriate when you want to estimate class probabilities (the probability that an object is in a given class) in addition to class assignments. a An example use of a logistic regression–based classifier is estimating the probability of fraud in credit card purchases. Logistic regression is also a good choice when you want an idea of the relative impact of different input variables on the output. For example, you might find out that a $100 increase in transaction size increases the odds that the transaction is fraud by 2%, all else being equal.

Support vector machines:

Support vector machines (SVMs) are useful when there are very many input variables or when input variables interact with the outcome or with each other in complicated (nonlinear) ways. SVMs make fewer assumptions about variable distribution than do many other methods, which makes them especially useful when the training data isn't completely representative of the way the data is distributed in production.

Solving scoring problems

For a scoring example, suppose that your task is to help evaluate how different marketing campaigns can increase valuable traffic to the website. The goal is not only to bring more people to the site, but to bring more people who buy. You're looking at a number of different factors: the communication channel (ads on websites, YouTube videos, print media, email, and so on); the traffic source (Facebook, Google, radio stations, and so on); the demographic targeted; the time of year; and so on.Predicting the increase in sales from a particular marketing campaign is an example of regression, or scoring. Fraud detection can be considered scoring, too, if you're trying to estimate the probability that a given transaction is a fraudulent one (rather than just returning a yes/no answer).

Scoring is also an instance of supervised learning.

Linear regression

Linear regression builds a model such that the predicted numerical output is a linear additive function of the inputs. This can be a very effective approximation, even when the underlying situation is in fact nonlinear. The resulting model also gives an indication of the relative impact of each input variable on the output. Linear regression is often a good first model to try when trying to predict a numeric value.

Logistic regression

Logistic regression always predicts a value between 0 and 1, making it suitable for predicting

probabilities (when the observed outcome is a categorical value) and rates (when the observed outcome is a rate or ratio). As we mentioned, logistic regression is an appropriate approach to the fraud detection problem, if what you want to estimate is the probability that a given transaction is fraudulent or legitimate.

Working without known targets

The preceding methods require that you have a training dataset of situations with known outcomes. In some situations, there's not (yet) a specific outcome that you want to predict. Instead, you may be looking for patterns and relationships in the data that will help you understand your customers or your business better.These situations correspond to a class of approaches called unsupervised learning: rather than predicting outputs based on inputs, the objective of unsupervised learning is to discover similarities and relationships in the data.

Some common clustering methods include these:

- K-means clustering
- Apriori algorithm for finding association rules
- Nearest neighbor

But these methods make more sense when we provide some context and explain their use, as we do next.

Example tasks	Machine learning terminology	Typical algorithms
Predicting the value of AdWords Estimating the probability that a loan will default Predicting how much a marketing campaign will increase traffic or sales	Regression: predicting or forecasting numerical values	Linear regression Logistic regression
Finding products that are purchased together Identifying web pages that are often visited in the same session Identifying successful (much-clicked) combinations of web pages and AdWords	Association rules: finding objects that tend to appear in the data together	Apriori
Identifying groups of customers with the same buying patterns Identifying groups of products that are popular in the same regions or with the same customer clusters Identifying news items that are all discussing similar events	Clustering: finding groups of objects that are more similar to each other than to objects in other groups	K-means
Making product recommendations for a customer based on the purchases of other similar customers Predicting the final price of an auction item based on the final prices of similar products that have been auctioned in the past	Nearest neighbor: predicting a property of a datum based on the datum or data that are most similar to it	Nearest neighbor

Evaluating models

For most model evaluations, we just want to compute one or two summary scores that tell us if the

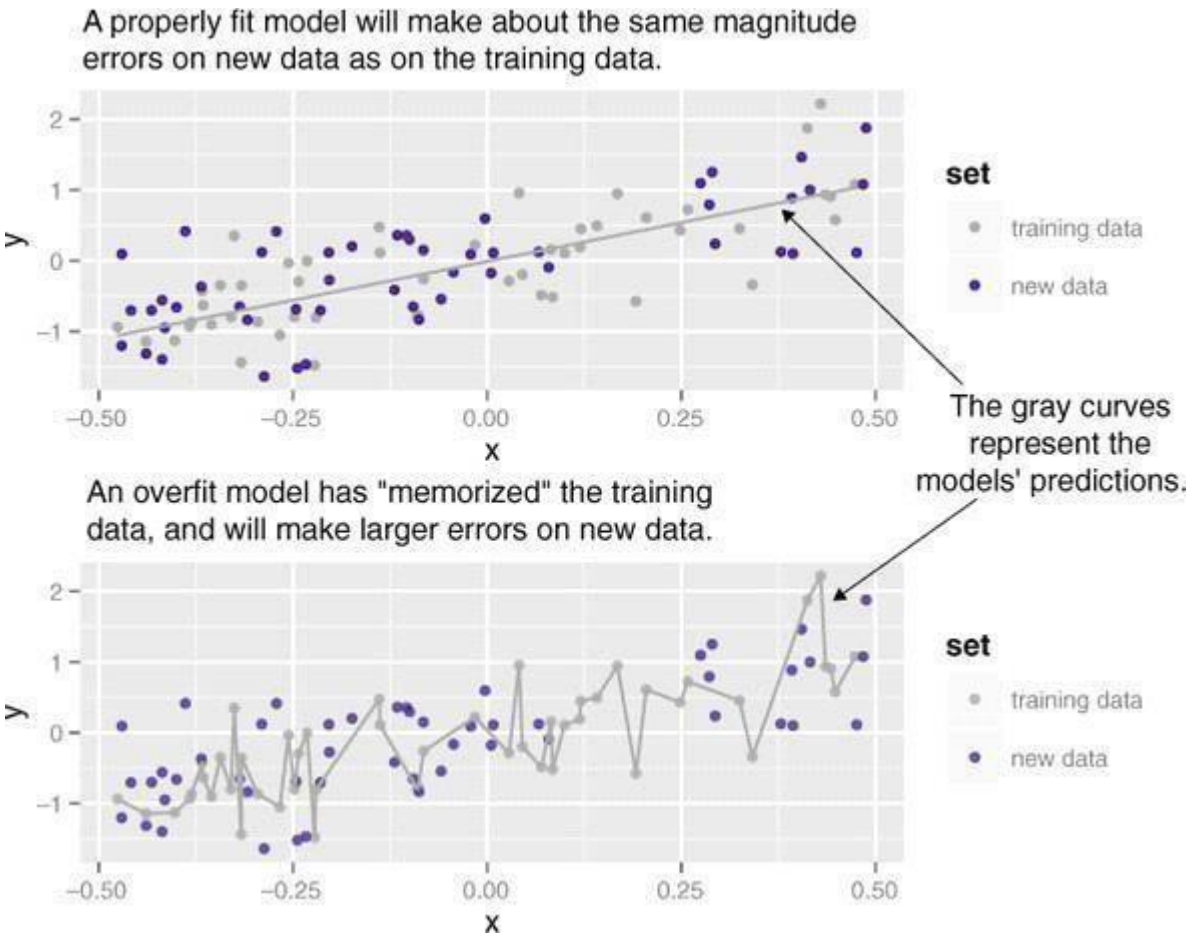
model is effective. To decide if a given score is high or low, we have to appeal to a few ideal models: a null model (which tells us what low performance looks like), a Bayes rate model (which tells us what high performance looks like), and the best single-variable model (which tells us what a simple model can achieve).

Ideal model	Purpose
Null model	A null model is the best model of a very simple form you're trying to outperform. The two most typical null model choices are a model that is a single constant (returns the same answer for all situations) or a model that is independent (doesn't record any important relation or interaction between inputs and outputs). We use null models to lower-bound desired performance, so we usually compare to a best null model. For example, in a categorical problem, the null model would always return the most popular category (as this is the easy guess that is least often wrong); for a score model, the null model is often the average of all the outcomes (as this has the least square deviation from all of the outcomes); and so on. The idea is this: if you're not out-performing the null model, you're not delivering value. Note that it can be hard to do as good as the best null model, because even though the null model is simple, it's privileged to know the overall distribution of the items it will be quizzed on. We always assume the null model we're comparing to is the best of all possible null models.
Bayes rate model	A Bayes rate model (also sometimes called a *saturated model*) is a best possible model given the data at hand. The Bayes rate model is the perfect model and it only makes mistakes when there are multiple examples with the exact same set of known facts (same *x*s) but different outcomes (different *y*s). It isn't always practical to construct the Bayes rate model, but we invoke it as an upper bound on a model evaluation score. If we feel our model is performing significantly above the null model rate and is approaching the Bayes rate, then we can stop tuning. When we have a lot of data and very few modeling features, we can estimate the Bayes error rate. Another way to estimate the Bayes rate is to ask several different people to score the same small sample of your data; the found inconsistency rate can be an estimate of the Bayes rate.[a]

Ideal model	Purpose
Single-variable models	We also suggest comparing any complicated model against the best single-variable model you have available A complicated model can't be justified if it doesn't outperform the best single-variable model available from your training data. Also, business analysts have many tools for building effective single-variable models (such as pivot tables), so if your client is an analyst, they're likely looking for performance above this level.

Overfitting

An overfit model looks great on the training data and then performs poorly on new data. A model's prediction error on the data that it trained from is called *training error*. A model's prediction error on new data is called *generalization error*. Usually, training error will be smaller than generalization error (no big surprise). Ideally, though, the two error rates should be close. If generalization error is large, and your model's test performance is poor, then your model has probably *overfit*—it's memorized the training data instead of discovering generalizable rules or patterns. You want to avoid overfitting by preferring (as long as possible) simpler models which do in fact tend to generalize better



If you do not split your data, but instead use all available data to both train and evaluate each model, then you might think that you will pick the better model, because the model evaluation has seen more data. However, the data used to build a model is not the best data for evaluating the model's performance. This is because there's an optimistic *measurement bias* in this data, because this data was seen during model construction. Model construction is optimizing your performance measure (or at least something related to your performance measure), so you tend to get exaggerated estimates of performance on your training data.

In addition, data scientists naturally tend to tune their models to get the best possible performance out of them. This also leads to exaggerated measures of performance. This is often called *multiple comparison bias*. And since this tuning might sometimes take advantage of quirks in the training data, it can potentially lead to overfit.

A recommended precaution for this optimistic bias is to split your available data into test and training. Perform all of your clever work on the training data alone, and delay measuring your performance with respect to your test data until as late as possible in your project (as all choices you make after seeing your test or holdout performance introduce a modeling bias). The desire to keep the test data secret for as long as possible is why we often actually split data into training, calibration, and test sets

When partitioning your data, you want to balance the trade-off between keeping enough data to fit a good model, and holding out enough data to make good estimates of the model's performance.

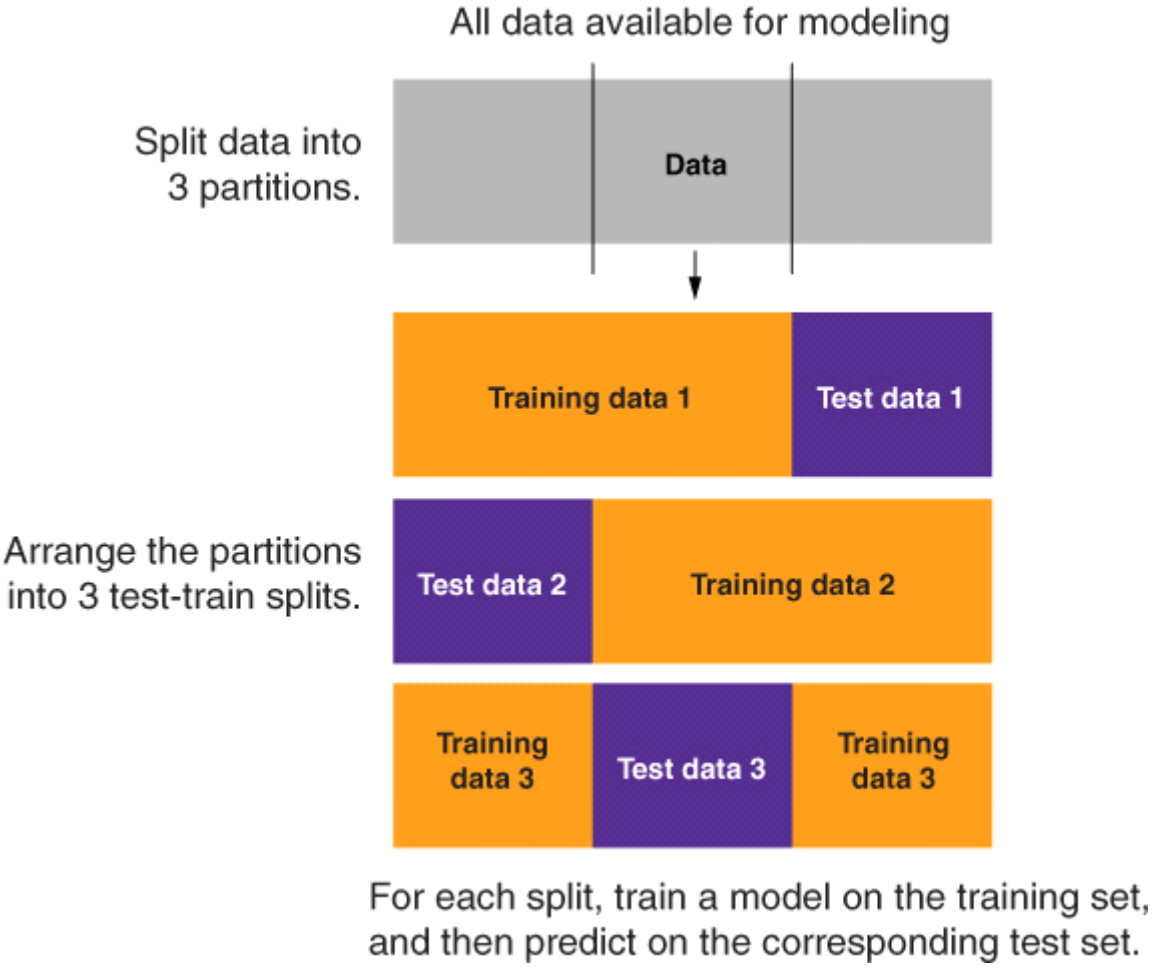
Some common splits are 70% training to 30% test, or 80% training to 20% test. For large datasets, you may even sometimes see a 50–50 split.

K-fold cross-validation

Testing on holdout data, while useful, uses each example only once: either as part of the model construction or as part of the held-out model evaluation set. This is not *statistically efficient*, because the test set is often much smaller than our whole dataset. This means we are losing some precision in our estimate of model performance by partitioning our data so simply. In our example scenario, suppose you were not able to collect a very large dataset of historical used car prices. Then you might feel that you do not have enough data to split into training and test sets that are large enough to both build good models *and* evaluate them properly. In this situation, you might choose to use a more thorough partitioning scheme called *k-fold cross-validation*.

An estimator is called statistically efficient when it has minimal variance for a given dataset size.The idea behind k-fold cross-validation is to repeat the construction of a model on different subsets of the available training data and then evaluate that model only on data not seen during construction. This allows us to use each and every example in both training and evaluating models (just never the same example in both roles at the same time). The idea is shown in figure for $k = 3$.

Figure : Partitioning data for 3-fold cross-validation



In the figure, the data is split into three non-overlapping partitions, and the three partitions are arranged to form three test-train splits. For each split, a model is trained on the training set and then applied to the corresponding test set. The entire set of predictions is then evaluated, using the

appropriate evaluation scores that we will discuss later in the chapter. This simulates training a model and then evaluating it on a holdout set that is the same size as the entire dataset. Estimating the model's performance on all the data gives us a more precise estimate of how a model of a given type would perform on new data. Assuming that this performance estimate is satisfactory, then you would go back and train a final model, using *all* the training data.

Measures of model performance

For most model evaluations, we just want to compute one or two summary scores that tell us if the model is effective. To decide if a given score is high or low, we generally compare our model's performance to a few baseline models.

The null model

The null model is the best version of a very simple model you're trying to outperform. The most typical null model is a model that returns the same answer for all situations (a constant model). We use null models as a lower bound on desired performance. For example, in a categorical problem, the null model would always return the most popular category, as this is the easy guess that is least often wrong. For a score model, the null model is often the average of all the outcomes, as this has the least square deviation from all the outcomes.

The idea is that if you're not outperforming the null model, you're not delivering value. Note that it can be hard to do as good as the best null model, because even though the null model is simple, it's privileged to know the overall distribution of the items it will be quizzed on. We always assume the null model we're comparing to is the best of all possible null models

Single-variable models

We also suggest comparing any complicated model against the best single-variable model you have available A complicated model can't be justified if it doesn't outperform the best single-variable model available from your training data. Also, business analysts have many tools for building effective single-variable models (such as pivot tables), so if your client is an analyst, they're likely looking for performance above this level.

Evaluating classification models

A classification model places examples into two or more categories. The most common measure of classifier quality is accuracy. For measuring classifier performance,we'll first introduce the incredibly useful tool called the confusion matrix and show how it can be used to calculate many important evaluation scores. The first score we'll discuss is accuracy, and then we'll move on to better and more detailed measures such as precision and recall.

Let's use the example of classifying email into spam (email we in no way want) and non-spam (email we want). A ready-to-go example (with a good description) is the Spambase dataset (http://mng.bz/e8Rh). Each row of this dataset is a set of featuresmeasured for a specific email and an additional column telling whether the mail was spam (unwanted) or non-spam (wanted). We'll quickly build a spam classification

model so we have results to evaluate. To do this, download the file Spambase/spamD.tsv from the book's GitHub site (https://github.com/WinVector/zmPDSwR/

tree/master/Spambase) and then perform the steps shown in the following listing.

Building and applying a logistic regression spam model

```
spamD <- read.table('spamD.tsv',header=T,sep='\t')
spamTrain <- subset(spamD,spamD$rgroup>=10)
spamTest <- subset(spamD,spamD$rgroup<10)
```

```
spamVars <- setdiff(colnames(spamD),list('rgroup','spam'))
spamFormula <- as.formula(paste('spam==''spam'"',
paste(spamVars,collapse=' + '),sep=' ~ '))
spamModel <- glm(spamFormula,family=binomial(link='logit'),
data=spamTrain)
spamTrain$pred <- predict(spamModel,newdata=spamTrain,
type='response')
spamTest$pred <- predict(spamModel,newdata=spamTest,
type='response')
print(with(spamTest,table(y=spam,glmPred=pred>0.5)))
##
glmPred
## y
FALSE TRUE
##
non-spam
264
14
##
spam
22 158
```

A sample of the results of our simple spam classifier is shown in the next listing.

Spam classifications

```
> sample <- spamTest[c(7,35,224,327),c('spam','pred')]
> print(sample)
spam
pred
115
spam 0.9903246227
361
spam 0.4800498077
2300 non-spam 0.0006846551
3428 non-spam 0.0001434345
```

CONFUSION MATRIX

The absolute most interesting summary of classifier performance is the confusion matrix. This matrix is just a table that summarizes the classifier's predictions against the actual known data categories.The confusion matrix is a table counting how often each combination of known outcomes (the truth) occurred in combination with each prediction type. For our email spam example, the confusion matrix is given by the following R command.

```
cM <- table(truth=spamTest$spam,prediction=spamTest$pred>0.5)
> print(cM)
prediction
truth          FALSE      TRUE
non-spam  264           14
```

CHANGING A SCORE TO A CLASSIFICATION

Note that we converted the numerical prediction score into a decision by checking if the score was above or below 0.5. For some scoring models (like logistic regression) the 0.5 score is likely a high accuracy value. However, accuracy isn't always the end goal, and for unbalanced training data the

0.5 threshold won't be good. Picking thresholds other than 0.5 can allow the data scientist to trade precision for recall we can start at 0.5, but considertrying other thresholds and looking at the ROC curve.Most of the performance measures of a classifier can be read off the entries of this confusion matrix. We start with the most common measure: accuracy.

A CCURACY

Accuracy is by far the most widely known measure of classifier performance. For a classifier, accuracy is defined as the number of items categorized correctly divided by the total number of items. It's simply what fraction of the time the classifier is correct. At the very least, you want a classifier to be accurate. In terms of our confusion matrix, accuracy is (TP+TN)/(TP+FP+TN+FN)=(cM[1,1]+cM[2,2])/sum(cM) or 92% accurate.The error of around 8% is unacceptably high for a spam filter, but good for illustrating different sorts of model evaluation criteria.

ACCURACY IS AN INAPPROPRIATE MEASURE FOR UNBALANCED CLASSES

Suppose we have a situation where we have a rare event (say, severe complications during childbirth). If the event we're trying to predict is rare (say, around 1% of the population), the null model—the rare event never happens— is very accurate. The null model is in fact more accurate than a useful (but not perfect model) that identifies 5% of the population as being "at risk" and captures all of the bad events in the 5%. This is not any sort of paradox. It's just that accuracy is not a good measure for events that have unbalanced distribution or unbalanced costs (different costs of "type 1" and "type 2" errors).

PRECISION AND RECALL

Another evaluation measure used by machine learning researchers is a pair of numbers called precision and recall. These terms come from the field of information retrieval and are defined as follows. Precision is what fraction of the items the classifier flags as being in the class actually are in the class. So precision is TP/(TP+FP) , which is cM[2,2]/(cM[2,2]+cM[1,2]) , or about 0.92 (it is only a coincidence that this is so close to the accuracy number we reported earlier). Again, precision is how often a positive indication turns out to be correct. It's important to remember that precision is a function of the combination of the classifier and the dataset. It doesn't make sense to ask how precise a classifier is in isolation; it's only sensible to ask how precise a clas- sifier is for a given dataset. In our email spam example, 93% precision means 7% of what was flagged as spam was in fact not spam. This is an unacceptable rate for losing possibly important messages. Akismet, on the other hand, had a precision of t[2,2]/(t[2,2]+t[1,2]) , or over 99.99%, so in addition to having high accuracy, Akismet has even higher precision (very important in a spam filtering application).The companion score to precision is recall. Recall is what fraction of the things that are in the class are detected by the classifier, or TP/(TP+FN)=cM[2,2]/(cM[2,2]+cM[2,1]) .For our email spam example this is 88%, and for the Akismet example it is 99.87%. In both cases most spam is in fact tagged (we have high recall) and precision is emphasized over recallIt's important to remember this: precision is a measure of confirmation (when the classifier indicates positive, how often it is in fact correct), and recall is a measure of utility (how much the classifier finds of what there actually is to find). Precision and recall tend to be relevant to business needs and are good measures to discuss with your project sponsor and client.

F1

The F1 score is a useful combination of precision and recall. If either precision or recall is very small, then F1 is also very small. F1 is defined as 2*precision*recall/(precision+recall) . So our email spam example with 0.93 precision and 0.88 recall has an F1 score of 0.90. The idea is that a classifier that improves precision or recall by sacrificing a lot of the complementary measure will

have a lower F1.

SENSITIVITY AND SPECIFICITY

Scientistsand doctors tend to use a pair of measures called sensitivity and specificity. Sensitivity is also called the true positive rate and is exactly equal to recall. Specificity is also called the true negative rate and is equal to TN/(TN+FP)=cM[1,1]/(cM[1,1]+cM[1,2]) or about 95%.

Table **Example classifier performance measures**

Measure	Formula	Email spam example	Akismet spam example
Accuracy	`(TP+TN) / (TP+FP+TN+FN)`	0.9214	0.9987
Precision	`TP/(TP+FP)`	0.9187	0.9999
Recall	`TP/(TP+FN)`	0.8778	0.9988
Sensitivity	`TP/(TP+FN)`	0.8778	0.9988
Specificity	`TN/(TN+FP)`	0.9496	0.9965

Table **Classifier performance measures business stories**

Measure	Typical business need	Follow-up question
Accuracy	"We need most of our decisions to be correct."	"Can we tolerate being wrong 5% of the time? And do users see mistakes like spam marked as non-spam or non-spam marked as spam as being equivalent?"
Precision	"Most of what we marked as spam had darn well better be spam."	"That would guarantee that most of what is in the spam folder is in fact spam, but it isn't the best way to measure what fraction of the user's legitimate email is lost. We could cheat on this goal by sending all our users a bunch of easy-to-identify spam that we correctly identify. Maybe we really want good specificity."
Recall	"We want to cut down on the amount of spam a user sees by a factor of 10 (eliminate 90% of the spam)."	"If 10% of the spam gets through, will the user see mostly non-spam mail or mostly spam? Will this result in a good user experience?"
Sensitivity	"We have to cut a lot of spam, otherwise the user won't see a benefit."	"If we cut spam down to 1% of what it is now, would that be a good user experience?"
Specificity	"We must be at least *three nines* on legitimate email; the user must see at least 99.9% of their non-spam email."	"Will the user tolerate missing 0.1% of their legitimate email, and should we keep a spam folder the user can look at?"

One conclusion for this dialogue process on spam classification would be to recommend writing the business goals as maximizing sensitivity while maintaining a specificity of at least 0.999.

Evaluating scoring models

Evaluating models that assign scores can be a somewhat visual task. The main concept is looking at what is called the *residuals* or the difference between our predictions.

f(x[i,]) and actual outcomes y[i].

d <- data.frame(y=(1:10)^2,x=1:10)

```
model <- lm(y~x,data=d)

d$prediction <- predict(model,newdata=d)
library('ggplot2')
ggplot(data=d) + geom_point(aes(x=x,y=y)) +
geom_line(aes(x=x,y=prediction),color='blue') +
geom_segment(aes(x=x,y=prediction,yend=y,xend=x)) +
scale_y_continuous('')
```

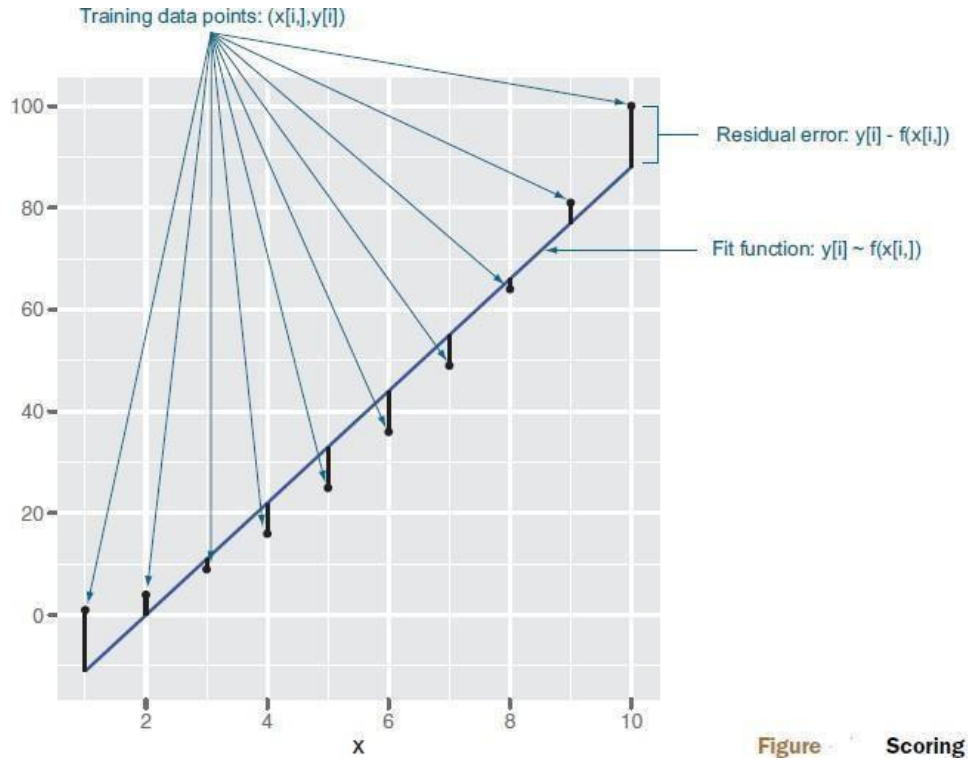


Figure Scoring residuals

ROOT MEAN SQUARE ERROR

The most common goodness-of-fit measure is called *root mean square error (RMSE)*. This is the square root of the average square of the difference between our prediction and actual values. Think of it as being like a standard deviation: how much your prediction is typically off.

R-SQUARED

Another important measure of fit is called R-squared (or R2, or the coefficient of determination). It's defined as 1.0 minus how much unexplained variance your model leaves (measured relative to a null model of just using the average *y* as a prediction).

CORRELATION

Correlation is very helpful in checking if variables are potentially useful in a model. Be advised that there are at least three calculations that go by the name of correlation:Pearson, Spearman, and Kendall (see help(cor)). The Pearson coefficient checks for linear relations, the Spearman coefficient checks for rank or ordered relations, and the Kendall coefficient checks for degree of voting agreement. Each of these coefficients performs a progressively more drastic transform than the one before and has well-known direct significance tests (see help(cor.test)).

DON'T USE CORRELATION TO EVALUATE MODEL QUALITY IN PRODUCTION

It's tempting to use correlation to measure model quality, but we advise against it. The problem is

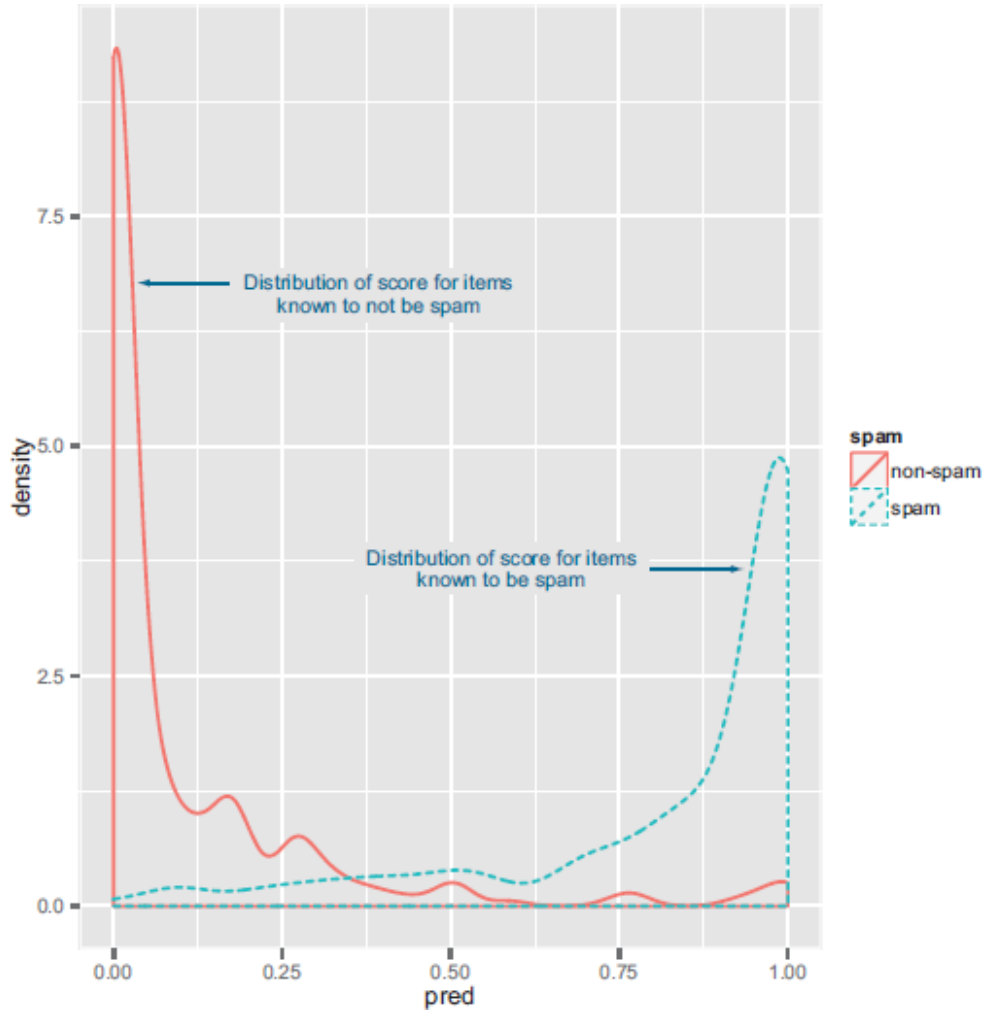
this: correlation ignores shifts and scaling factors. So correlation is actually computing if there is any shift and rescaling of your predictor that is a good predictor. This isn't a problem for training data (as these predictions tend to not have a systematic bias in shift or scaling by design) but can mask systematic errors that may arise when a model is used in production.

ABSOLUTE ERROR

For many applications (especially those involving predicting monetary amounts), measures such as absolute error (sum(abs(d$prediction-d$y))), mean absolute error (sum(abs(d$prediction d$y))/length(d$y)), and relative absolute error (sum(abs(d$prediction-d$y))/sum(abs(d$y))) are tempting measures. It does make sense to check and report these measures, but it's usually not advisable to make these measures the project goal or to attempt to directly optimize them. This is because absolute error measures tend not to "get aggregates right" or "roll up reasonably" as most of the squared errors do. As an example, consider an online advertising company with three advertisement purchases returning $0, $0, and $25 respectively. Suppose our modeling task is as simple as picking a single summary value not too far from the original three prices. The price minimizing absolute error is the median, which is $0, yielding an absolute error of sum(abs(c(0,0,25)-20)), or $25. The price minimizing square error is the mean, which is $8.33 (which has a worse absolute error of $33.33). However the median price of $0 misleadingly values the entire campaign at $0. One great advantage of the mean is this: aggregating a mean prediction gives an unbiased prediction of the aggregate in question. It is *often* an unstated project need that various totals or roll-ups of the predicted amounts be close to the roll-ups of the unknown values to be predicted. For monetary applications, predicting the totals or aggregates accurately is often more important than getting individual values right. In fact, most statistical modeling techniques are designed for *regression*, which is the unbiased prediction of means or
expected values.

Evaluating probability models

Probability models are useful for both classification and scoring tasks. Probability models are models that both decide if an item is in a given class and return an estimated probability (or confidence) of the item being in the class. The modeling techniques of logistic regression and decision trees are fairly famous for being able to return good probability estimates. Such models can be evaluated on their final decisions, most of the measures for probability models are very technical and very good at comparing the qualities of different models on the same dataset. But these criteria aren't easy to precisely translate into businesses needs. So we recommend tracking them, but not using them with your project sponsor or client.



Distribution of score broken up by known classes

THE RECEIVER OPERATING CHARACTERISTIC CURVE

The *receiver operating characteristic curve* (or *ROC curve*) is a popular alternative to the double density plot. For each different classifier we'd get by picking a different score threshold between positive and negative determination, we plot both the true positive rate and the false positive rate. This curve represents every possible trade-off between sensitivity and specificity that is available for this classifier. The steps to produced the ROC plot.

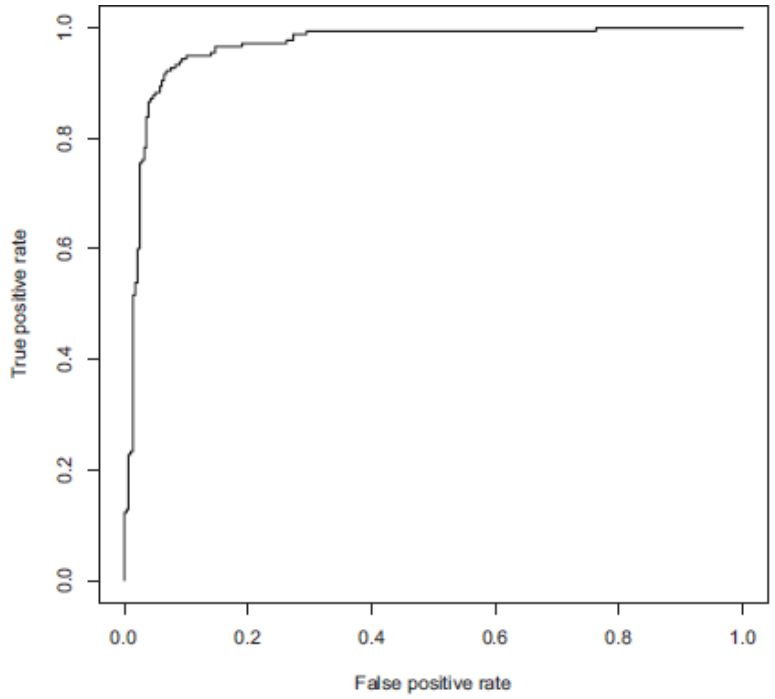


Figure ROC curve for the email spam example

```
library('ROCR')
eval <- prediction(spamTest$pred,spamTest$spam)
plot(performance(eval,"tpr","fpr"))
print(attributes(performance(eval,'auc'))$y.values[[1]])
[1] 0.9660072
```

LOG LIKELIHOOD

An important evaluation of an estimated probability is the log likelihood. The log likelihood is the logarithm of the product of the probability the model assigned to each example.2 For a spam email with an estimated likelihood of 0.9 of being spam, the log likelihood is log(0.9); for a non-spam email, the same score of 0.9 is a log likelihood of log(1-0.9) (or just the log of 0.1, which was the estimated probability of not being spam). The principle is this: if the model is a good explanation, then the data should look likely (not implausible) under the model. The following listing shows how the log likelihood of our example is derived.

```
> sum(ifelse(spamTest$spam=='spam',
log(spamTest$pred),
log(1-spamTest$pred)))
[1] -134.9478
> sum(ifelse(spamTest$spam=='spam',
log(spamTest$pred),
log(1-spamTest$pred)))/dim(spamTest)[[1]]
[1] -0.2946458
```

The first term (-134.9478) is the model log likelihood the model assigns to the test data. This number will always be negative, and is better as we get closer to 0. The second expression is the log likelihood rescaled by the number of data points to give us a rough average surprise per data point. Now a good null model in this case would be always returning the probability of 180/458 (the number of known spam emails over
the total number of emails as the best single-number estimate of the chance of spam). This null model gives the log likelihood shown in the next listing.

DEVIANCE

Another common measure when fitting probability models is the *deviance*. The deviance is defined as -2*(logLikelihood-S), where S is a technical constant called "the log likelihood of the saturated model." The lower the residual deviance, the better the model. In most cases, the saturated model is a perfect model that returns probability 1 for items in the class and probability 0 for items not in the class (so S=0).

AIC

An important variant of deviance is the *Akaike information criterion* (*AIC*). This is equivalent to deviance + 2*numberOfParameters used in the model used to make the prediction. Thus, AIC is deviance penalized for model complexity. A nice trick is to do as the Bayesians do: use *Bayesian information criterion* (*BIC*) (instead of AIC) where an empirical estimate of the model complexity (such as 2*2^entropy, instead of 2*numberOfParameters) is used as the penalty.

ENTROPY

Entropy is a fairly technical measure of information or surprise, and is measured in a unit called *bits*. If *p* is a vector containing the probability of each possible outcome, then the entropy of the outcomes is calculated as sum(-p*log(p,2)) (with the convention that 0*log(0) = 0). As entropy measures surprise, you want what's called the *conditional entropy* of your model to be appreciably lower than the original entropy. The conditional entropy is a measure that gives an indication of how good the prediction is on different categories, tempered by how often it predicts different categories.

Calculating entropy and conditional entropy

```
> entropy <- function(x) {
    xpos <- x[x>0]
    scaled <- xpos/sum(xpos)
    sum(-scaled*log(scaled,2))
  }

> print(entropy(table(spamTest$spam)))
[1] 0.9667165

> conditionalEntropy <- function(t) {
     (sum(t[,1])*entropy(t[,1]) + sum(t[,2])*entropy(t[,2]))/sum(t)
  }

> print(conditionalEntropy(cM))
[1] 0.3971897
```

Define function that computes the entropy from list of outcome counts

Calculate entropy of spam/non-spam distribution

Function to calculate conditional or remaining entropy of spam distribution (rows) given prediction (columns)

Calculate conditional or remaining entropy of spam distribution given prediction

We see the initial entropy is 0.9667 bits per example (so a lot of surprise is present), and the conditional entropy is only 0.397 bits per example.

UNIT – IV

Modelling Methods-II: Linear and logistic regression
Using linear regression: Understanding linear regression, Building a linear regression model, making predictions.
Using logistic regression: Understanding logistic regression, Building a logistic regression model, making predictions.

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

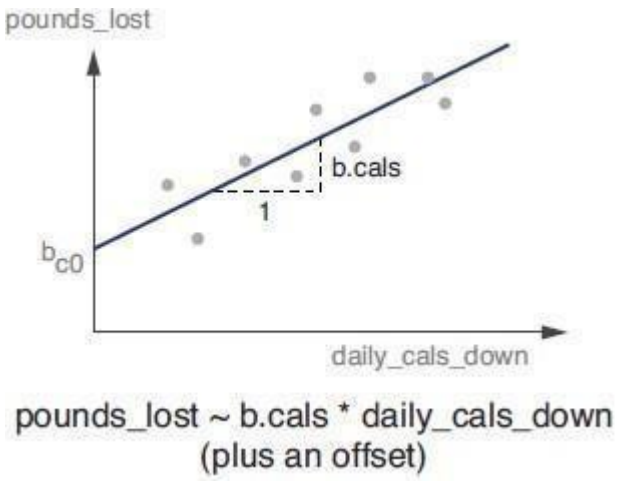


Figure 7.2 The linear relationship between `daily_cals_down` and `pounds_lost`

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."

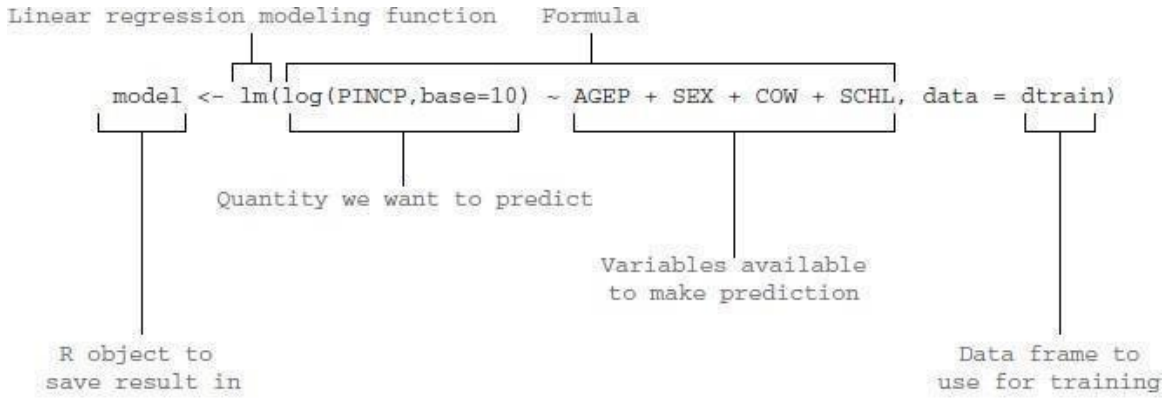


Figure 7.4 Building a linear model using `lm()`

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.

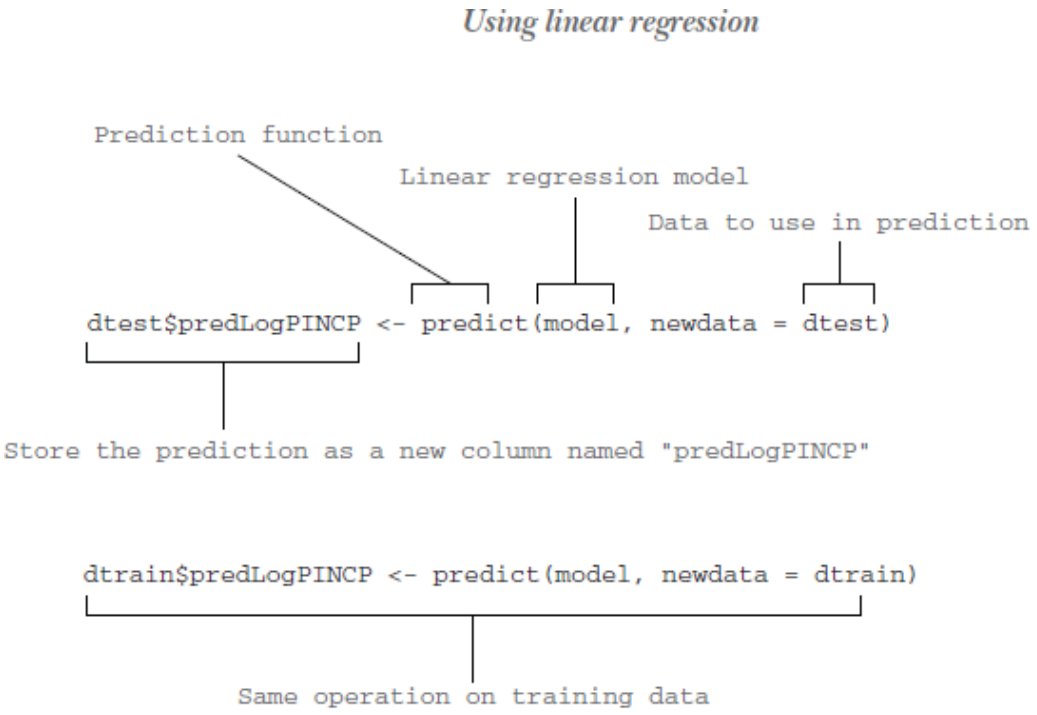


Figure 7.5 Making predictions with a linear regression model

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
```

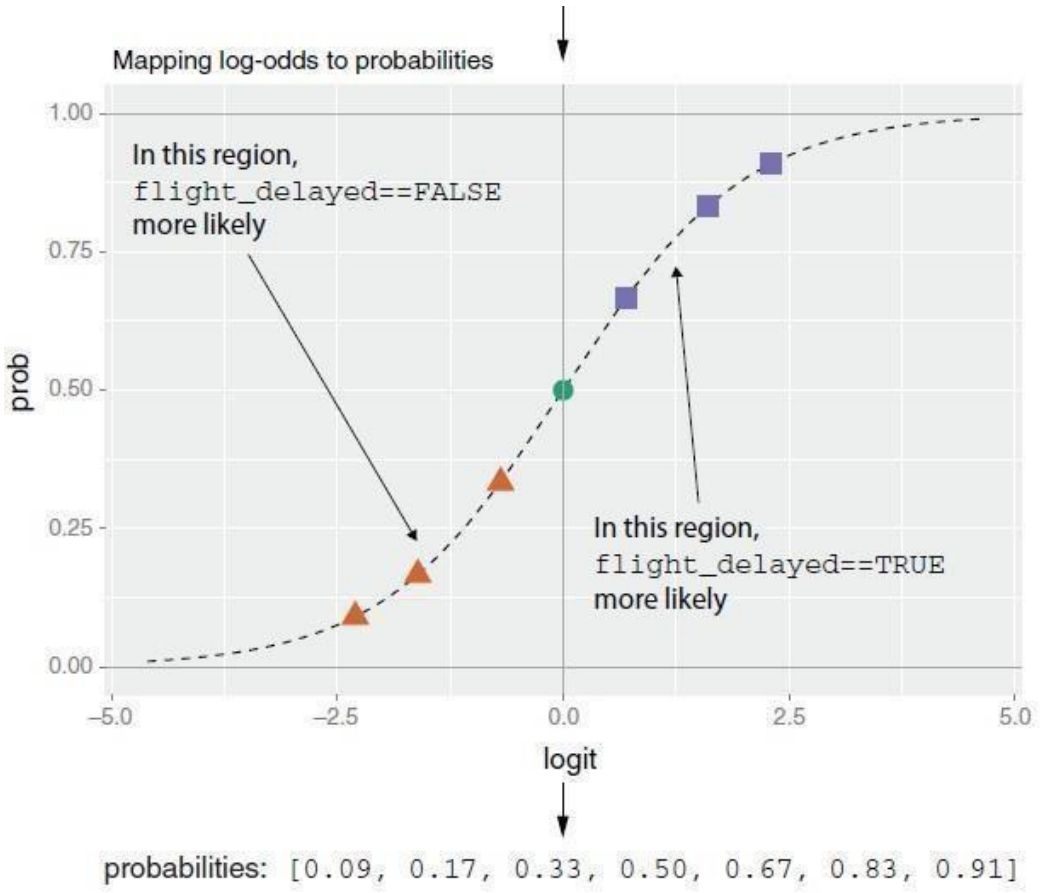


Figure 7.13 Mapping log-odds to the probability of a flight delay via the sigmoid function

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.

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:

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.

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))
```

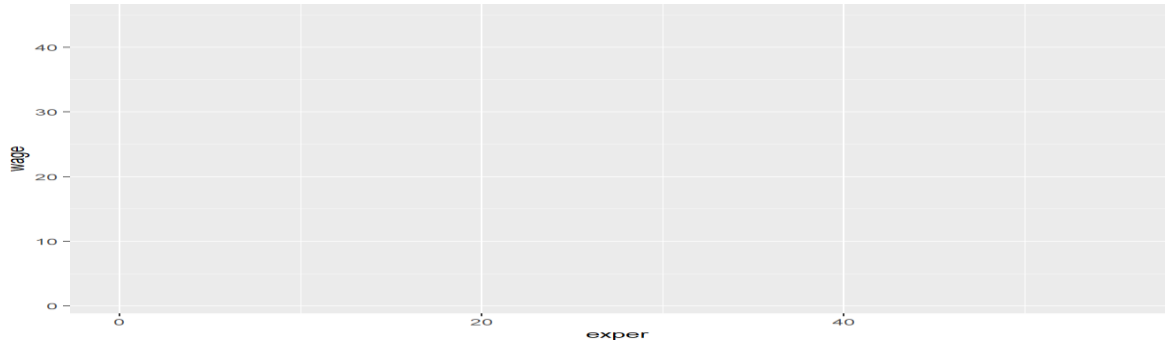


Figure: Map variables

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.

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()
```

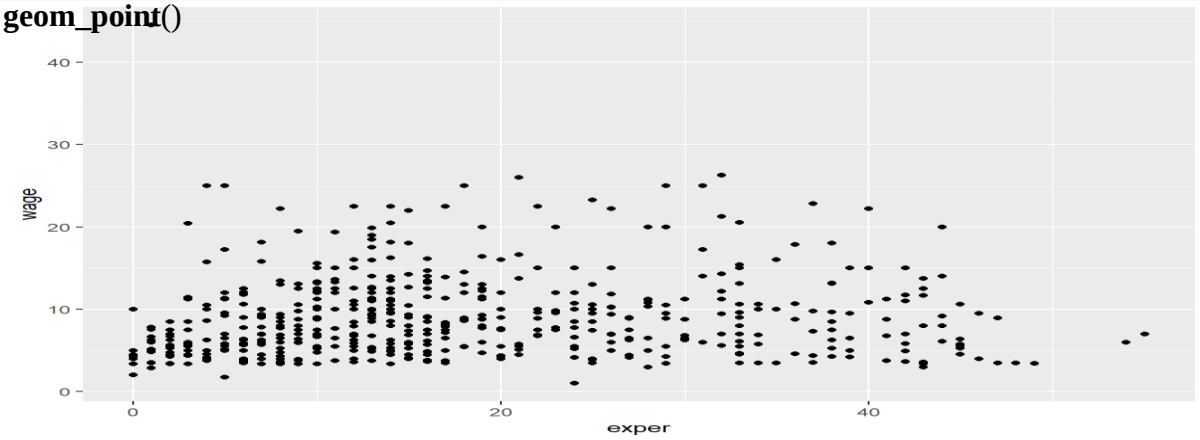


Figure: Add points

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()
```

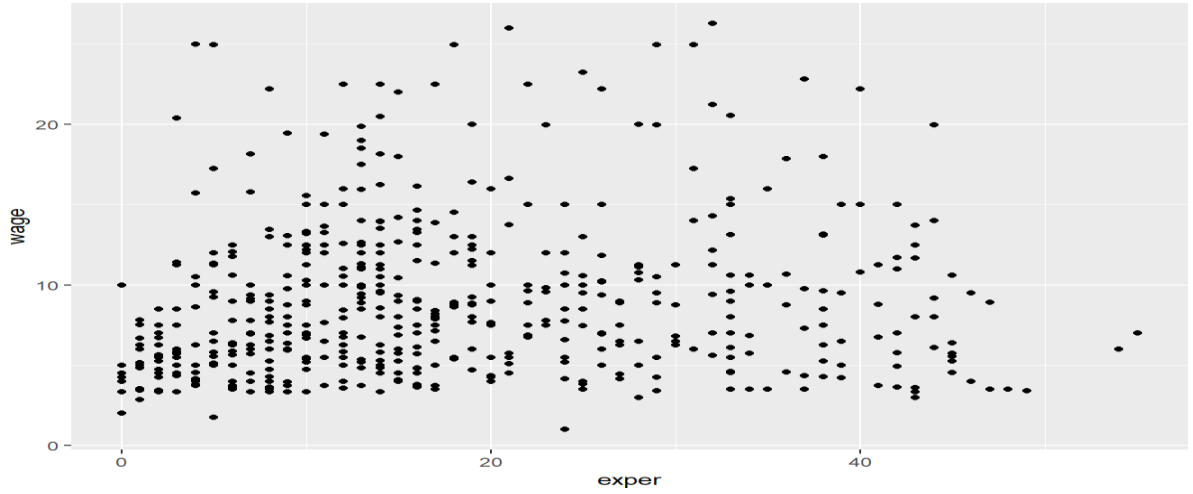


Figure: Remove outlier

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)
```

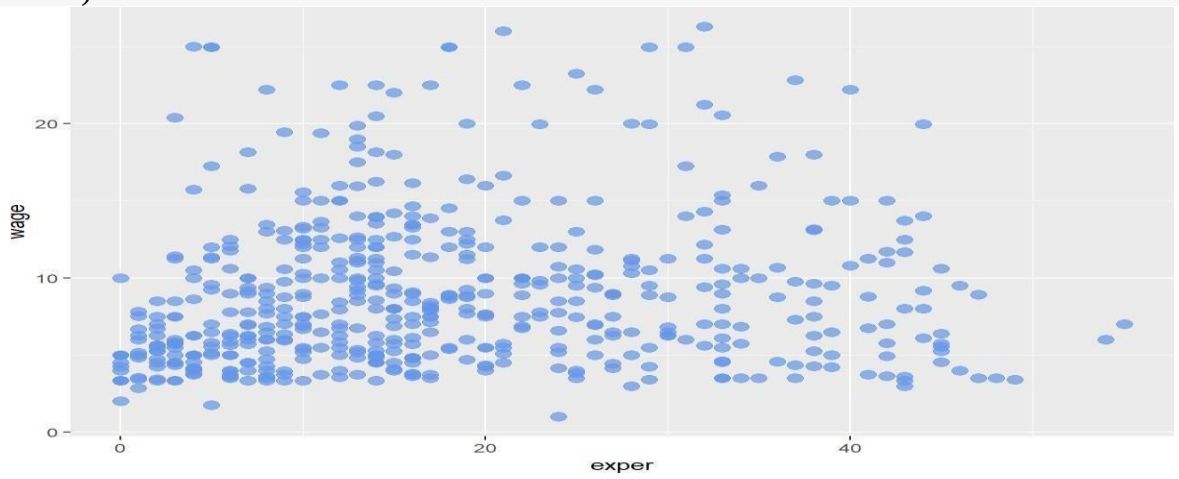


Figure: Modify point color, transparency, and size

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 =

```
# 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")
```

lm) line (where *lm* stands for linear model).

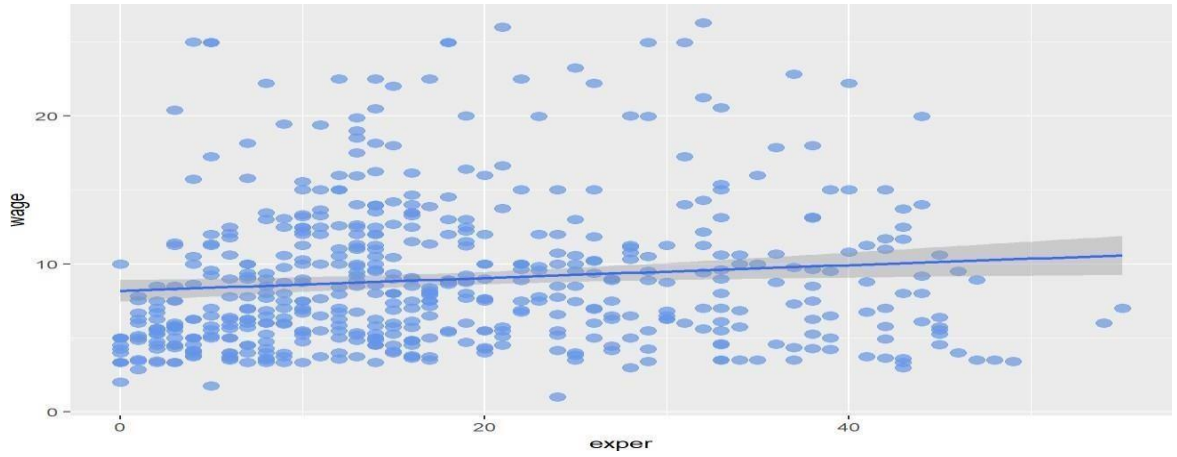


Figure: Add line of best fit

Wages appears to increase with experience.

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)
```

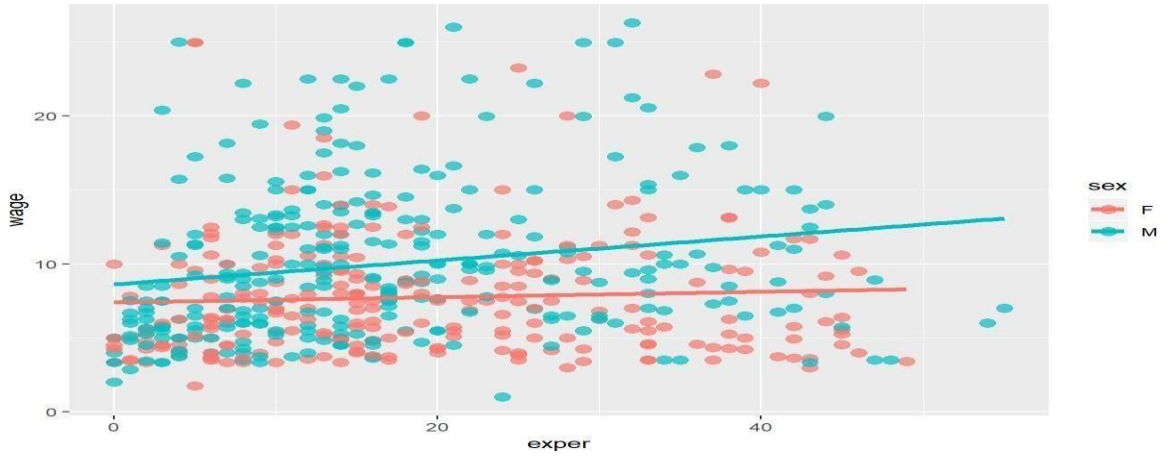


Figure: Include sex, using color

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 inter-vals.

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.

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

```
# 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) +
```

the *x* and *y* axis scaling, and the colors employed.

```
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"))
```

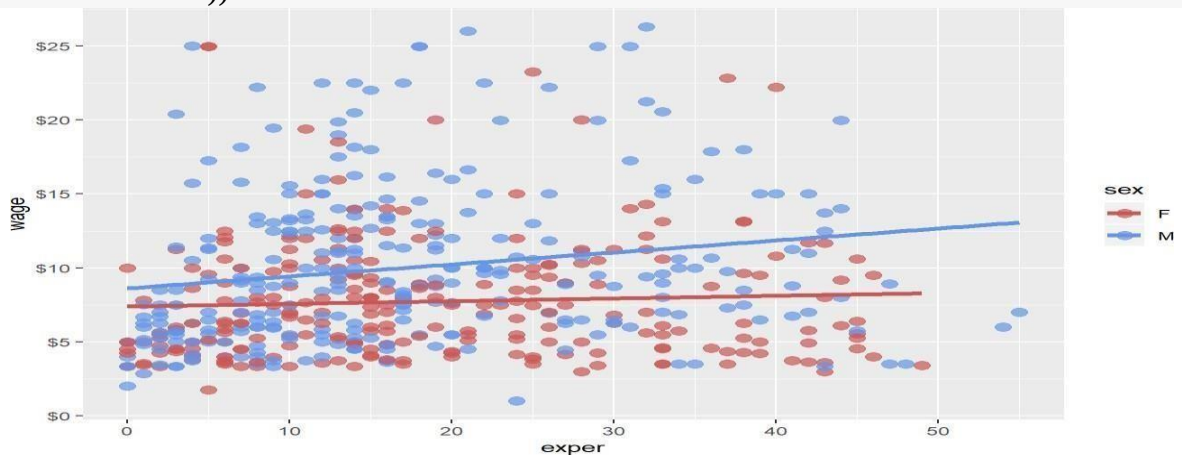


Figure: Change colors and axis labels

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.

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)
```

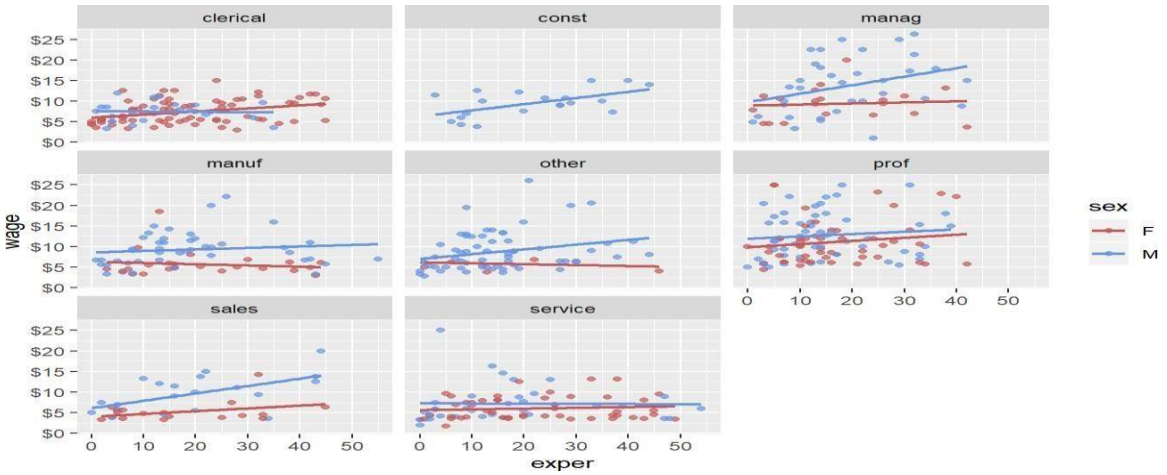


Figure: Add job sector, using faceting

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

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 cus-

```
# 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")
```

tom title, subtitle, and caption can be added.

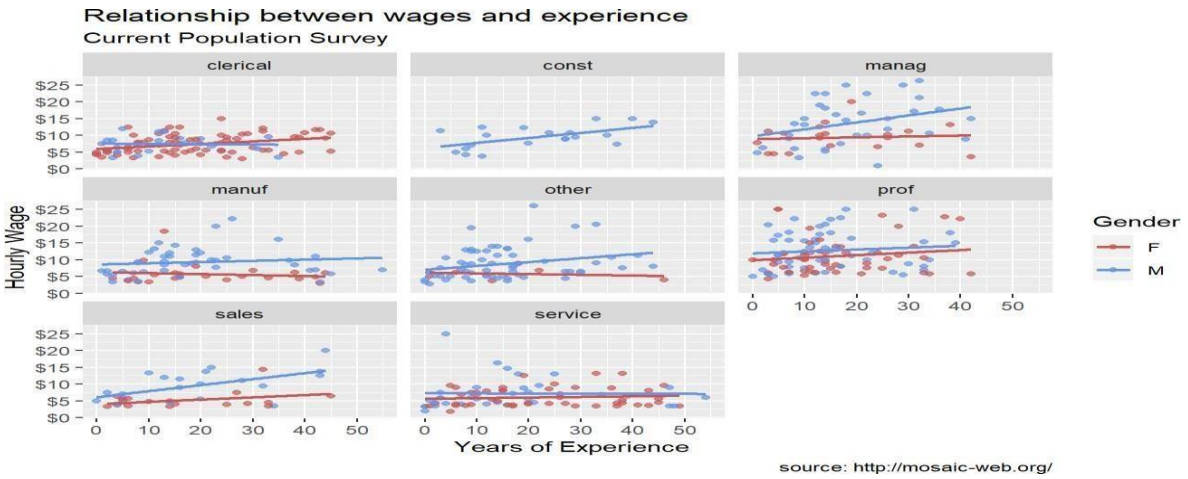


Figure: Add informative titles and labels

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

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()
```

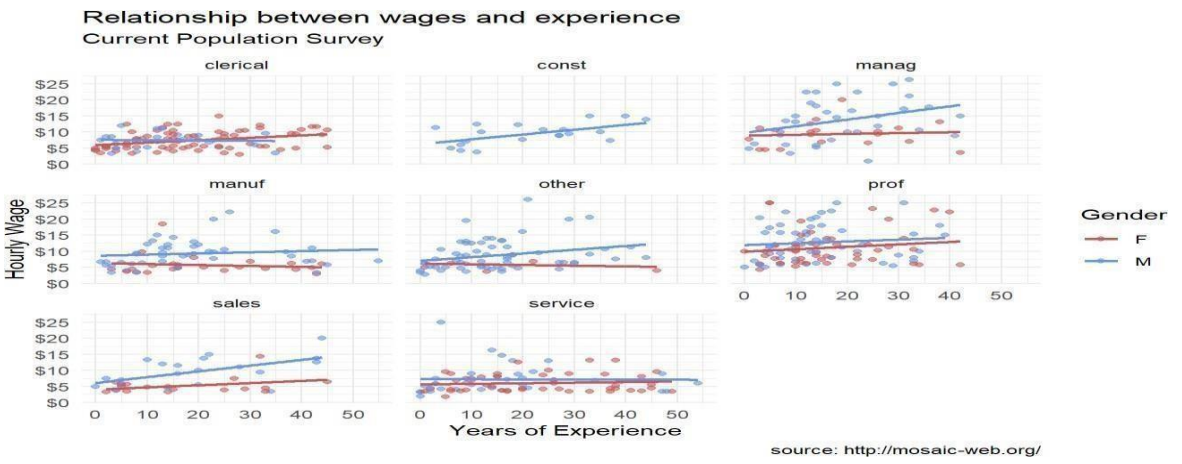


Figure: Use a simpler theme

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.

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)
```

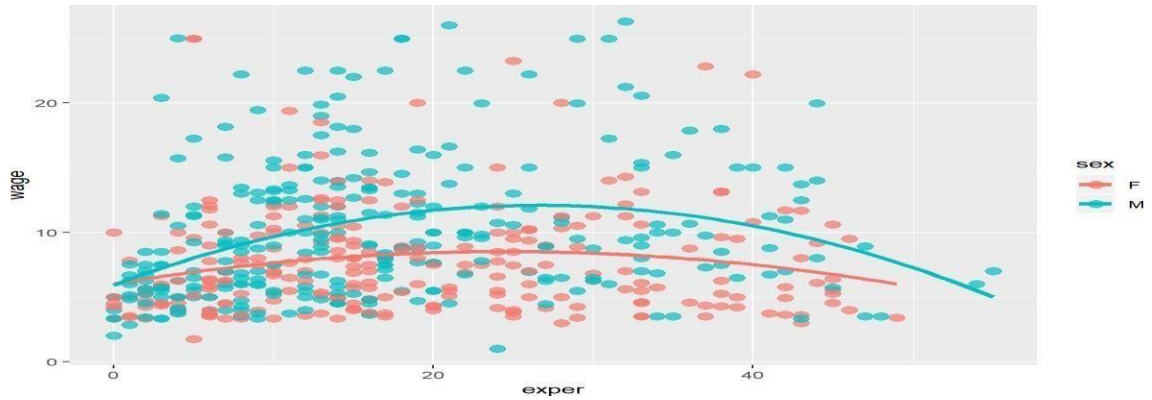


Figure: Color mapping in ggplot function

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)
```

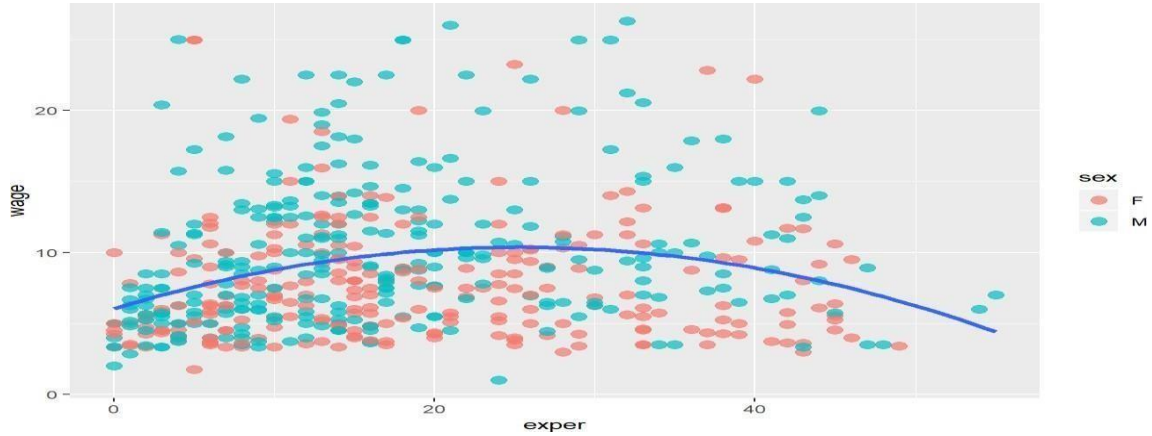


Figure12: Color mapping in ggplot function

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.

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()
```

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).

Categorical

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

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()
```

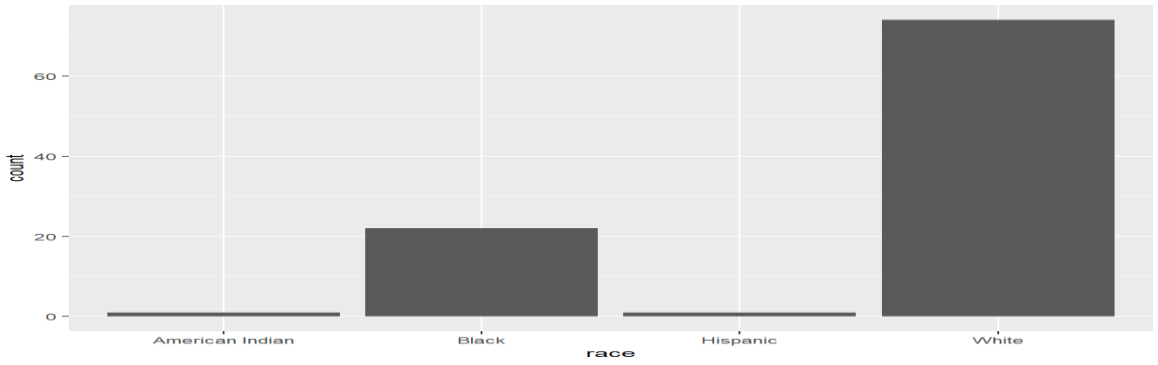


Figure: Simple barchart

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

```
# 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)
```

the y variable explicitly.

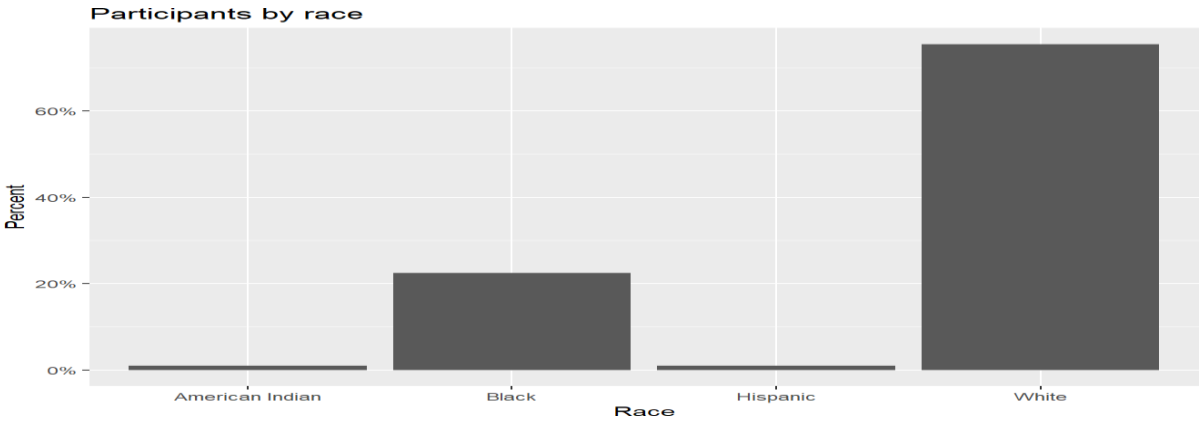


Figure: Barchart with percentages

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

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

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

directly.

```
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")
```

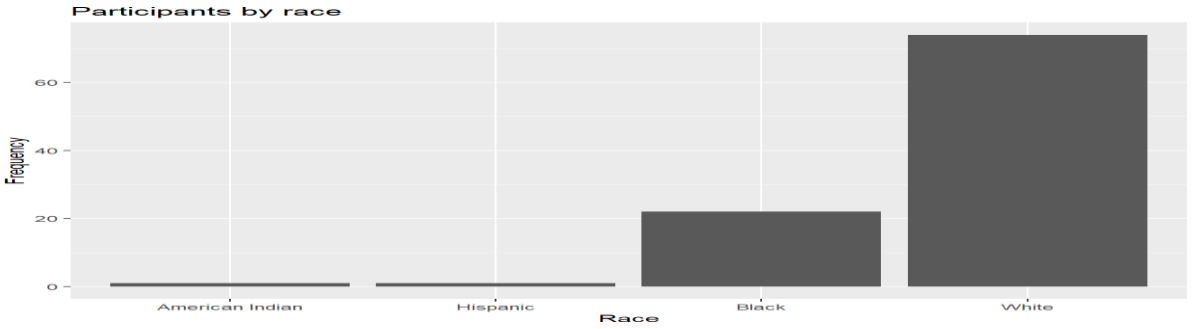


Figure: Sorted bar chart

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

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")
```

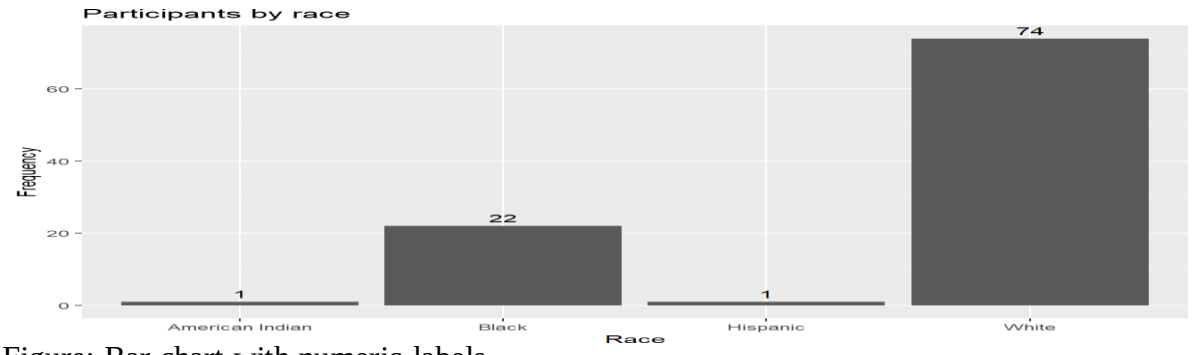


Figure: Bar chart with numeric labels

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")
```

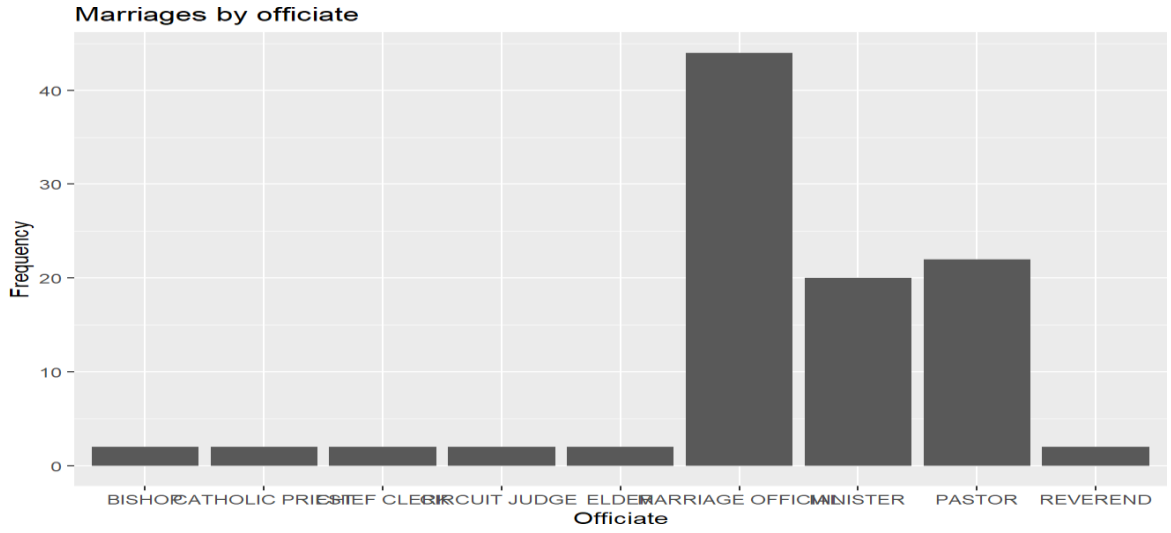


Figure: Barchart with problematic labels

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()
```

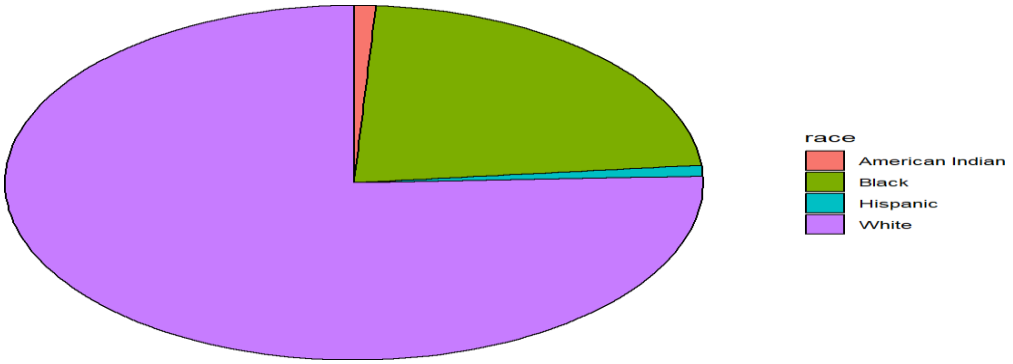


Figure: Basic pie chart

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")
```

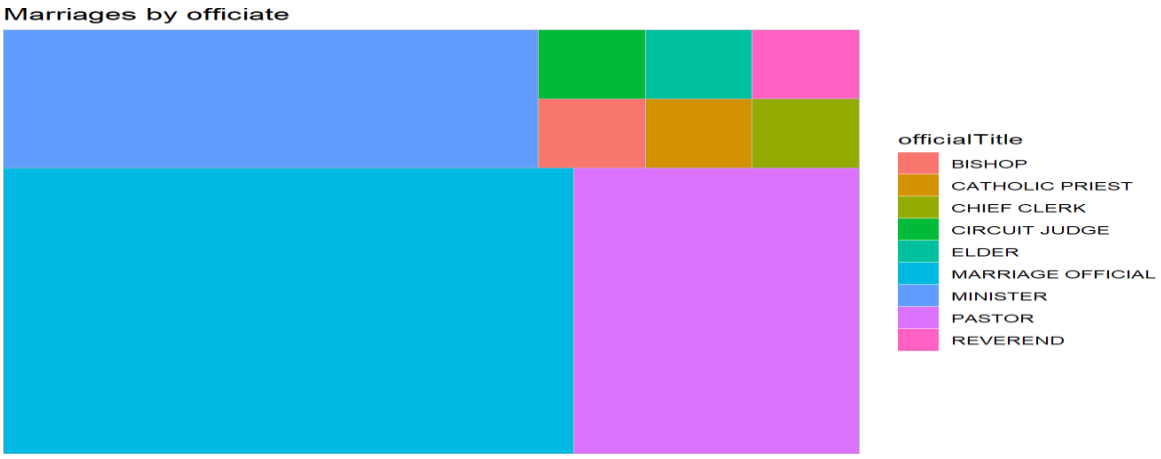


Figure: Basic treemap

Quantitative

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

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")
```

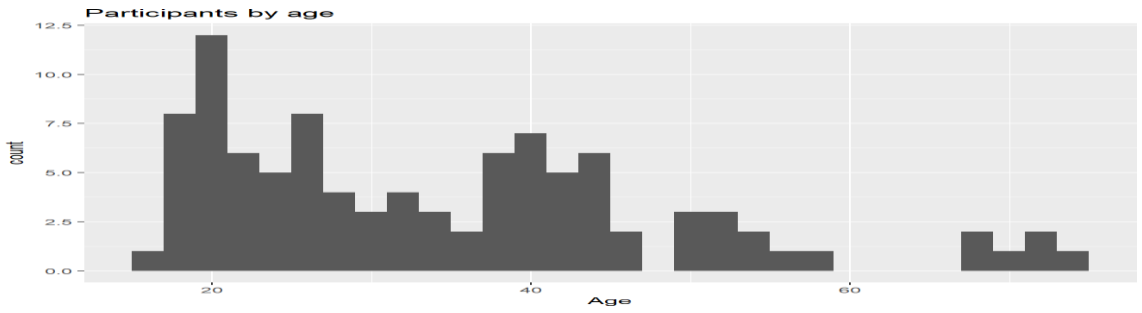


Figure: Basic histogram

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 dis-

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

tribution.

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

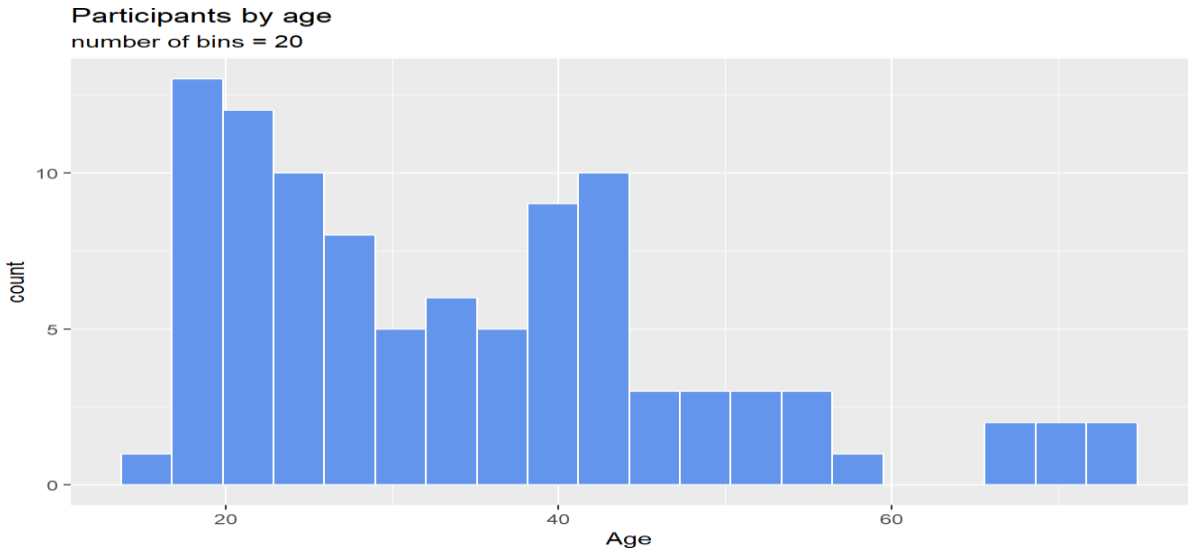


Figure: Histogram with a specified number of bins

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")
```

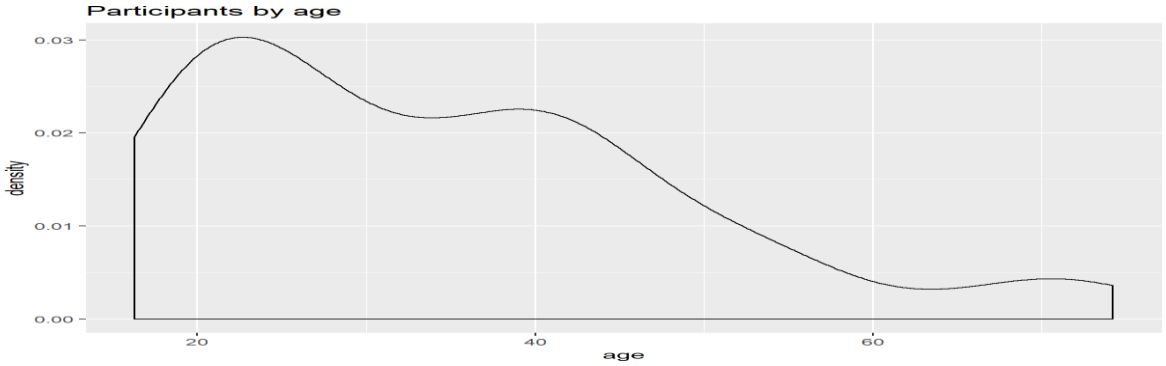


Figure: Basic kernel density plot

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

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

smoothing, while values that are smaller will produce less smoothing.

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

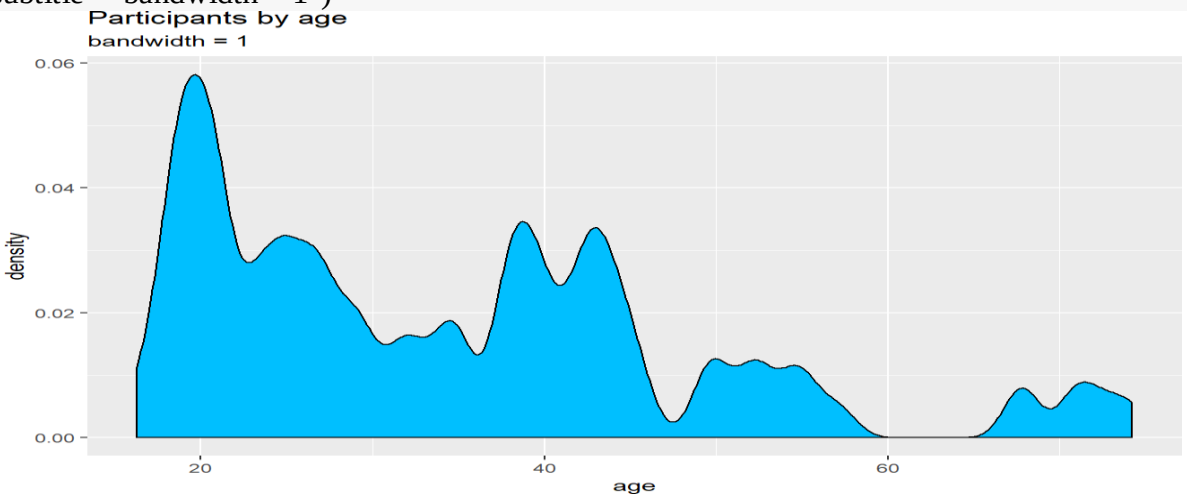


Figure: Kernel density plot with a specified bandwidth

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")
```

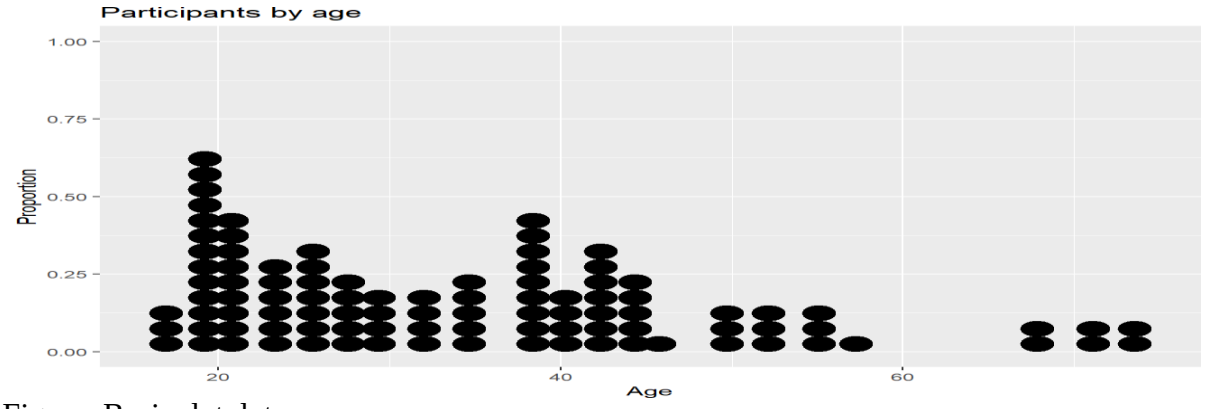


Figure: Basic dotplot

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).

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.

Stacked bar chart

Let's plot the relationship between automobile class and drive type (front-wheel, rear-wheel, or4-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")
```

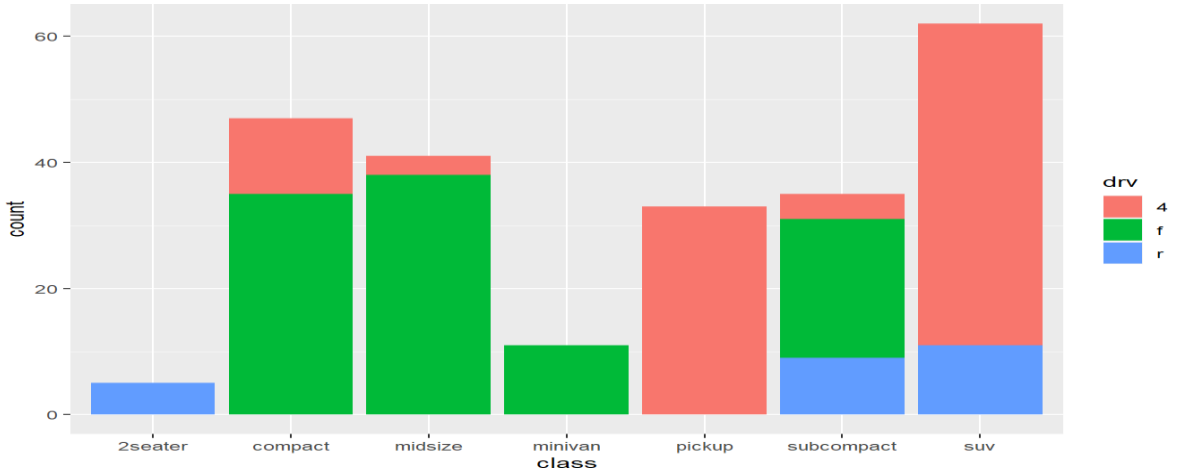


Figure: Stacked bar chart

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

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")
```

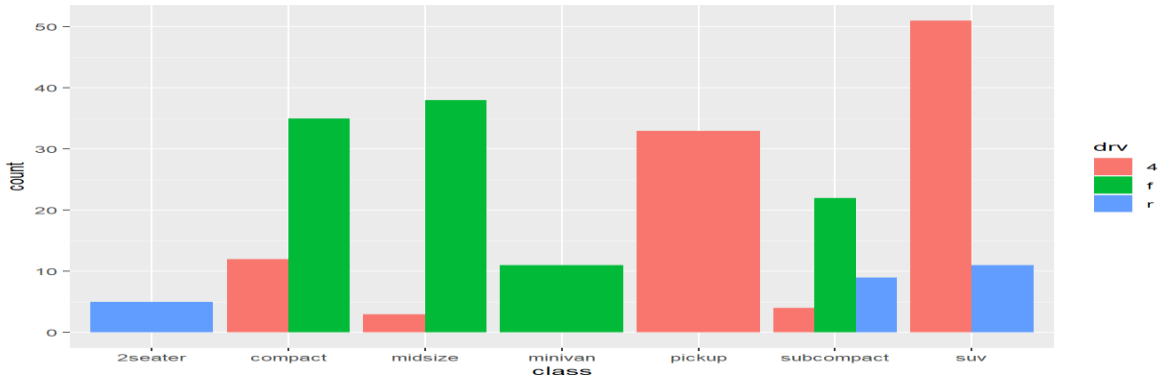


Figure: Side-by-side bar chart

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")
```

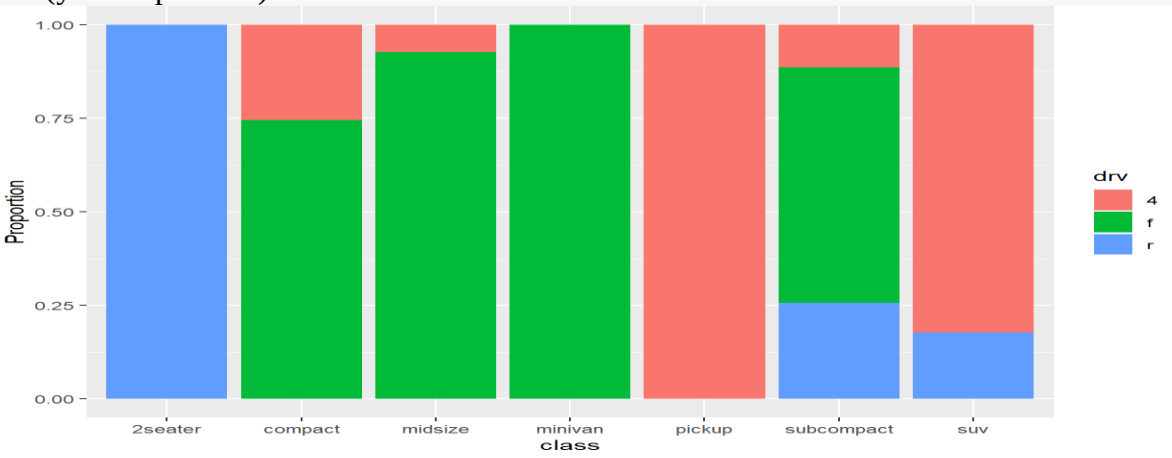


Figure: Segmented bar chart

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()
```

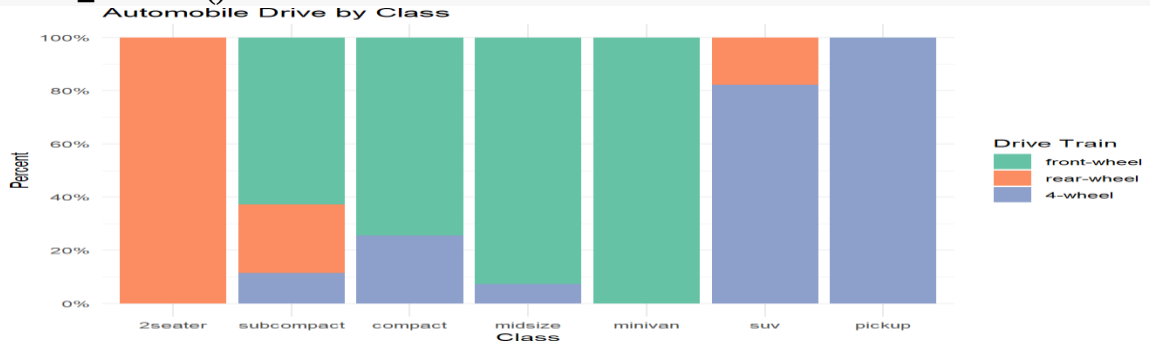


Figure: Segmented bar chart with improved labeling and color

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.

Quantitative vs. Quantitative

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

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()
```

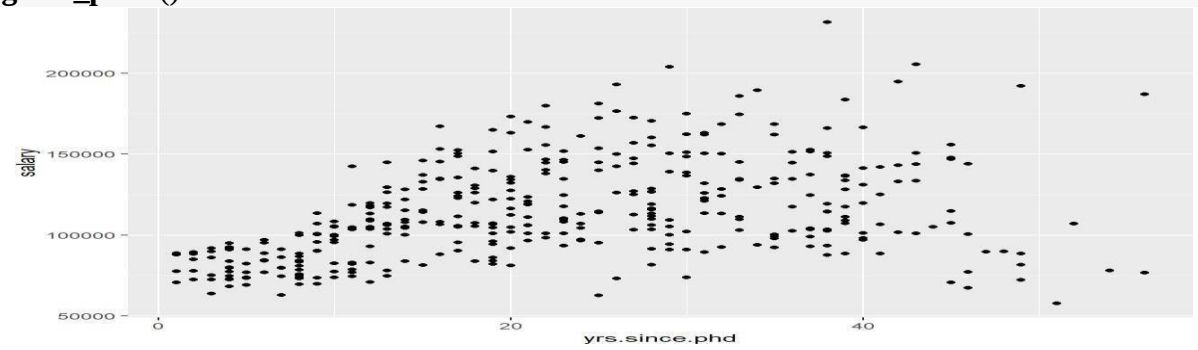


Figure: Simple scatterplot

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")
```

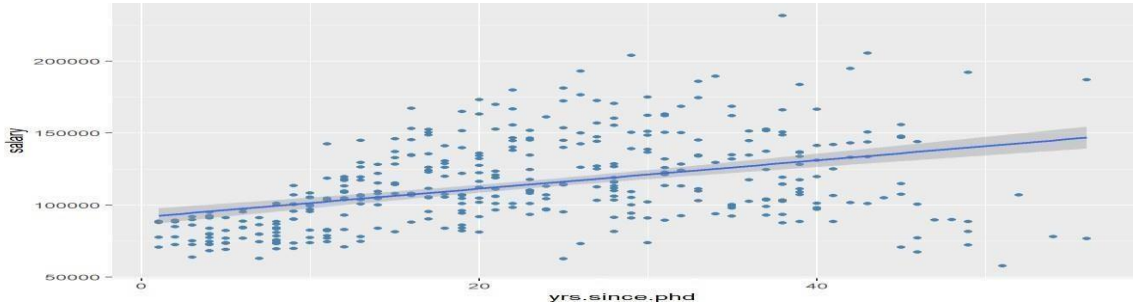


Figure: Scatterplot with linear fit line

Line plot

When one of the two variables represents time, a line plot can be an effective method of displaying 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()
```

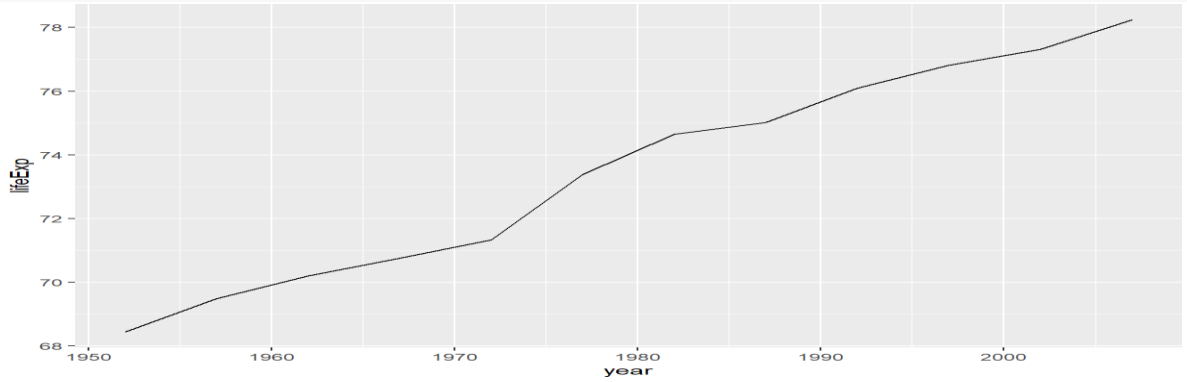


Figure: Simple line plot

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.

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")
```

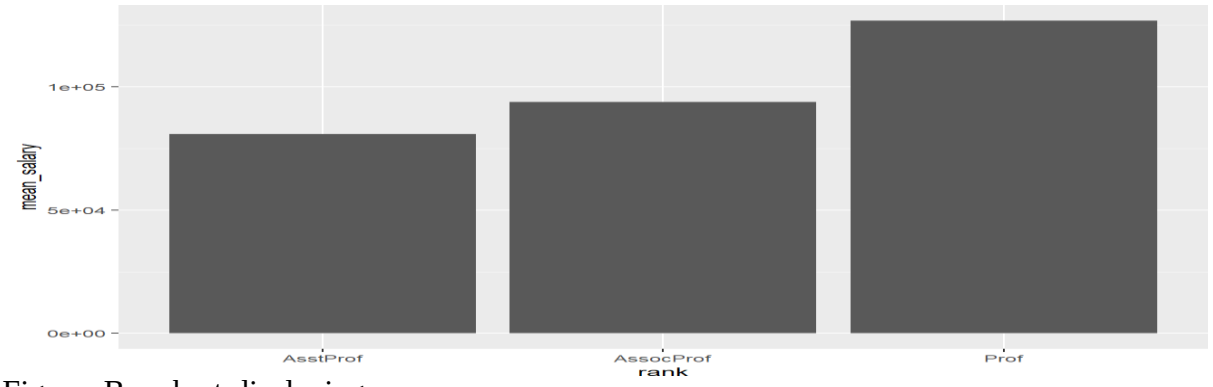


Figure: Bar chart displaying means

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")
```

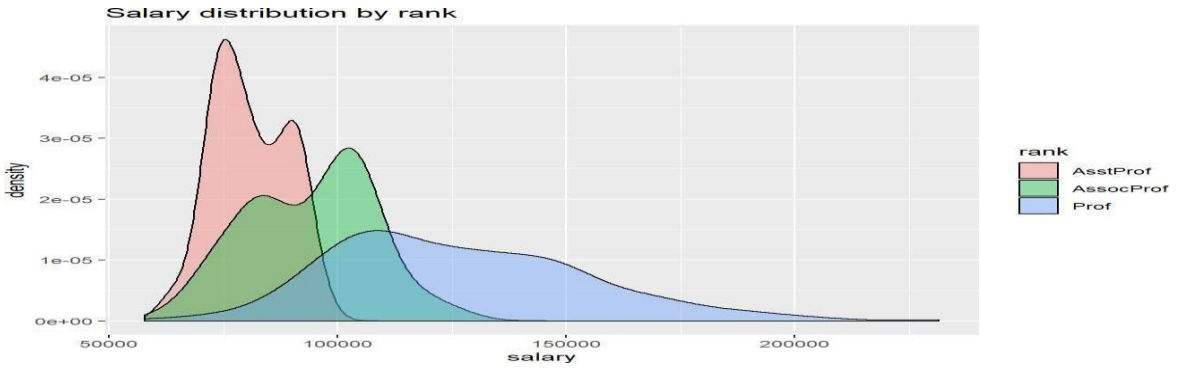
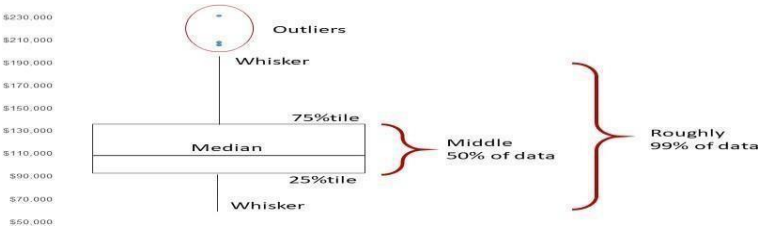


Figure: Grouped kernel density plots

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")
```

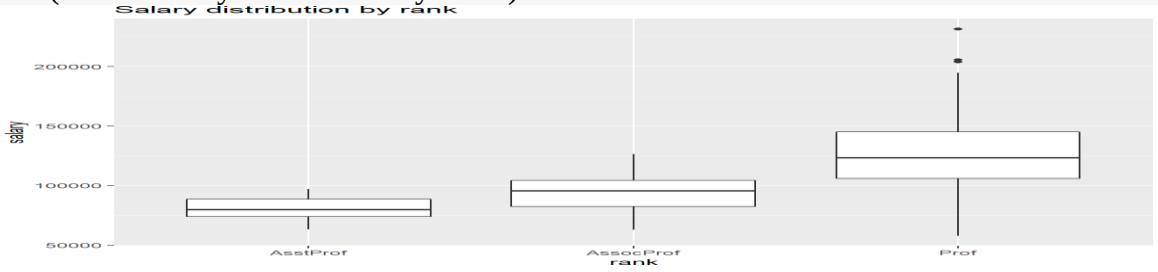


Figure: Side-by-side boxplots

Violin plots

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

```
# 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")
```

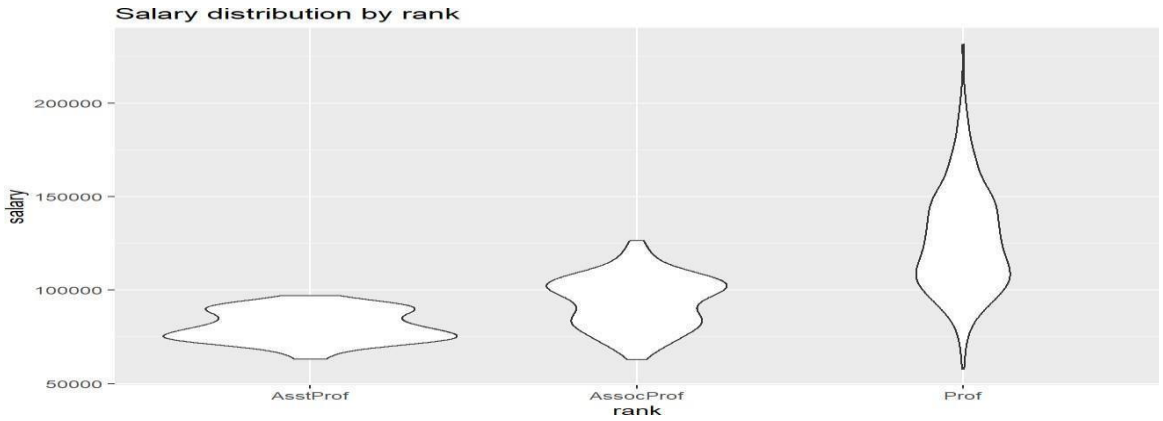


Figure: Side-by-side violin plots

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")
```

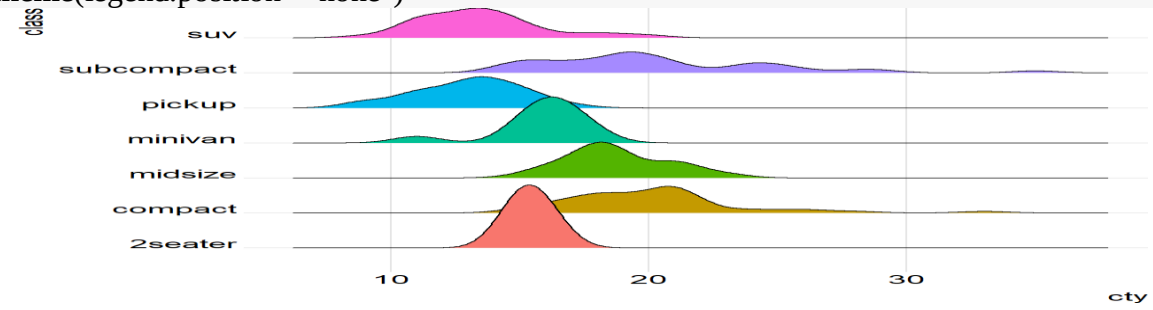


Figure: Ridgeline graph with color fill

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),
w idth = .1)
```

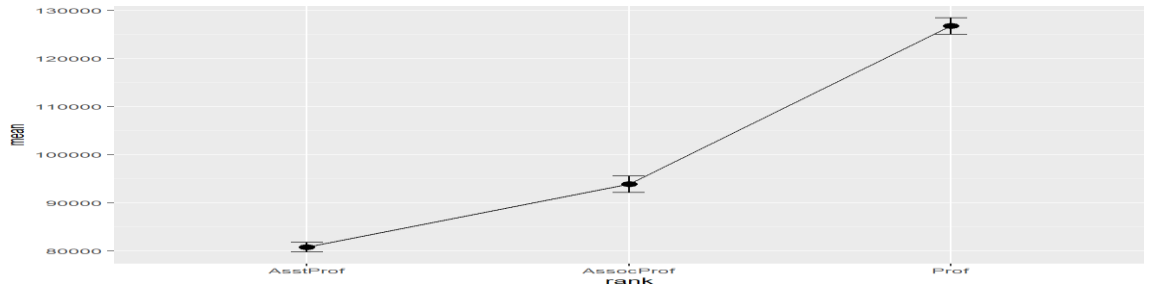


Figure: Mean plots with standard error bars

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() +
```

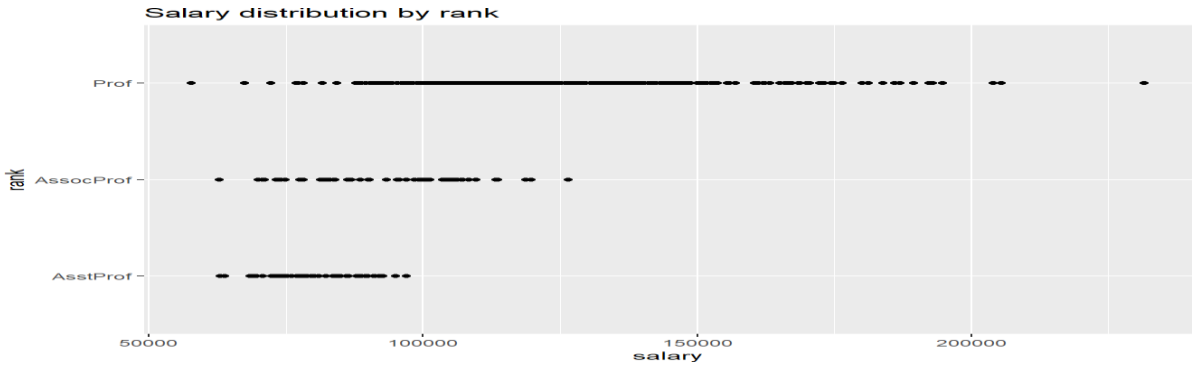


Figure: Categorical by quantiative scatterplot

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")),
```

```
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() +
```

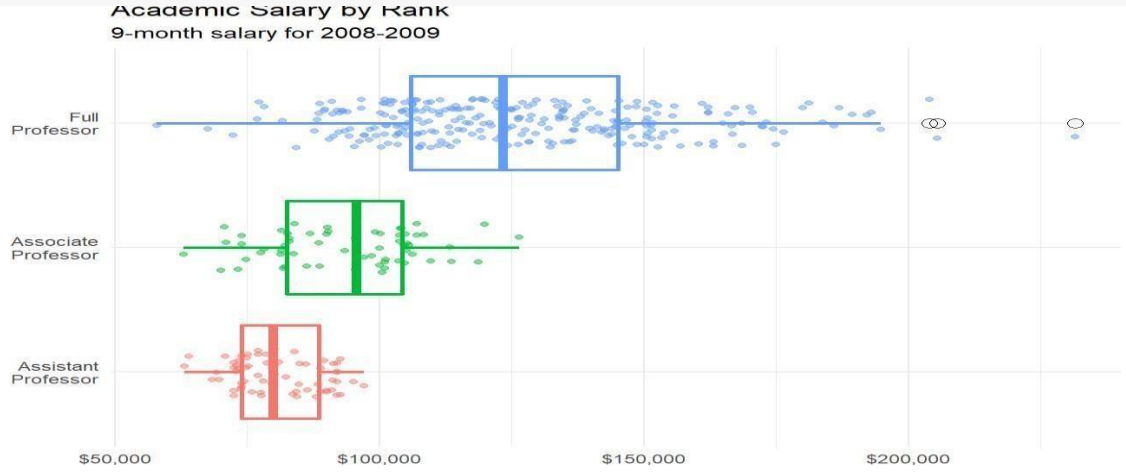


Figure: Jitter plot with superimposed box plots

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

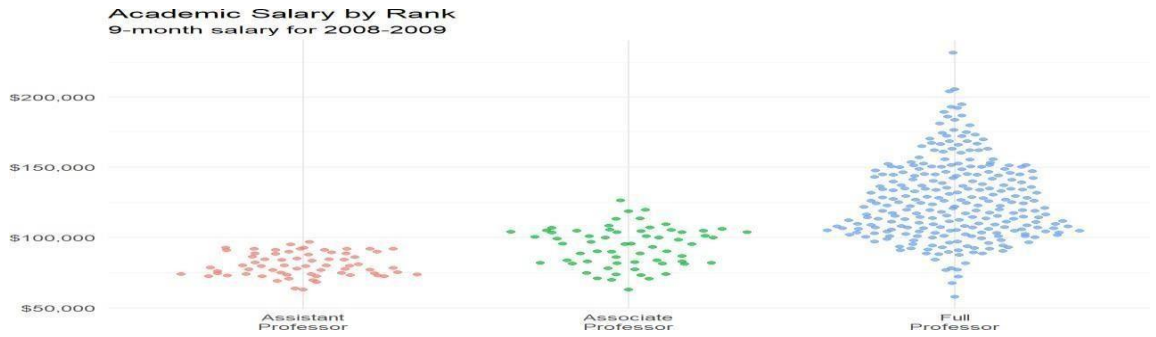


Figure: Beeswarm plot

.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()
```

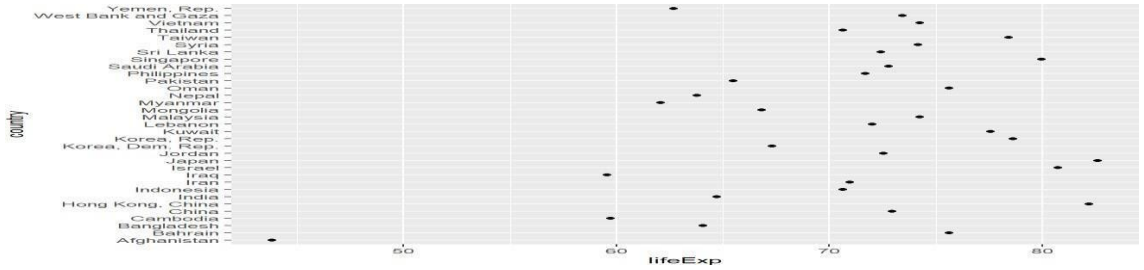


Figure: Basic Cleveland dot plot

Multivariate Graphs:

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

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")
```

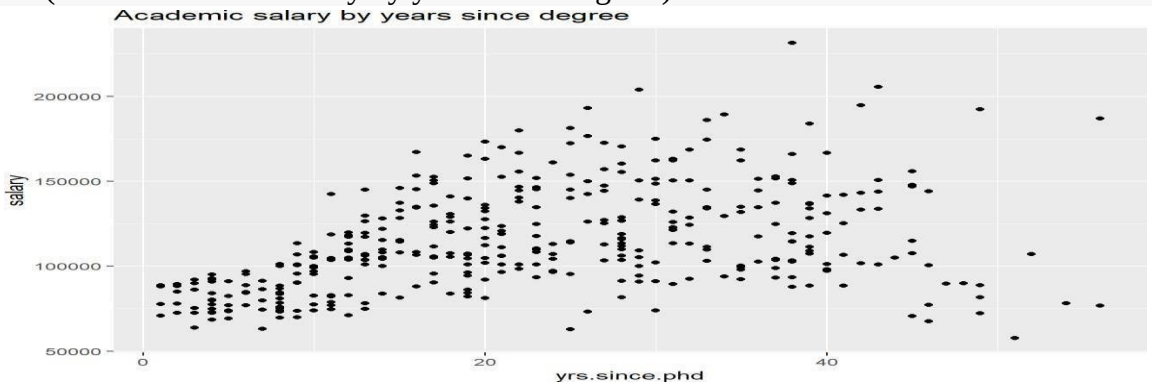


Figure: Simple scatterplot

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")
```

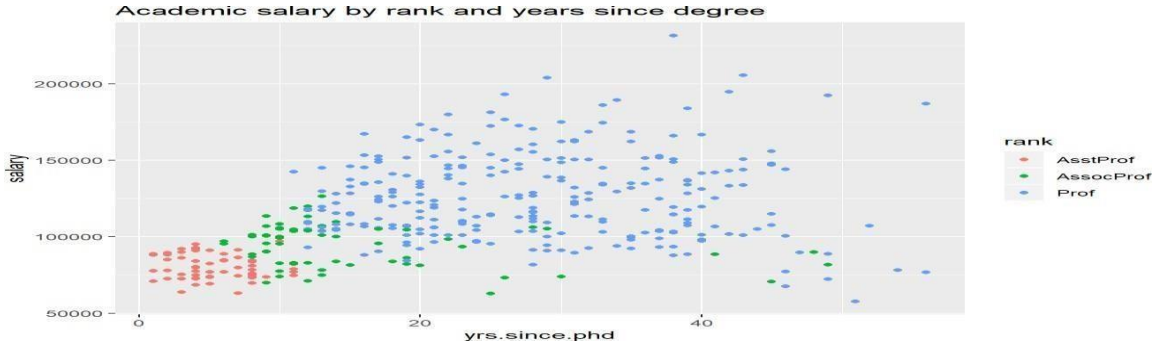


Figure: Scatterplot with color mapping

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")
```

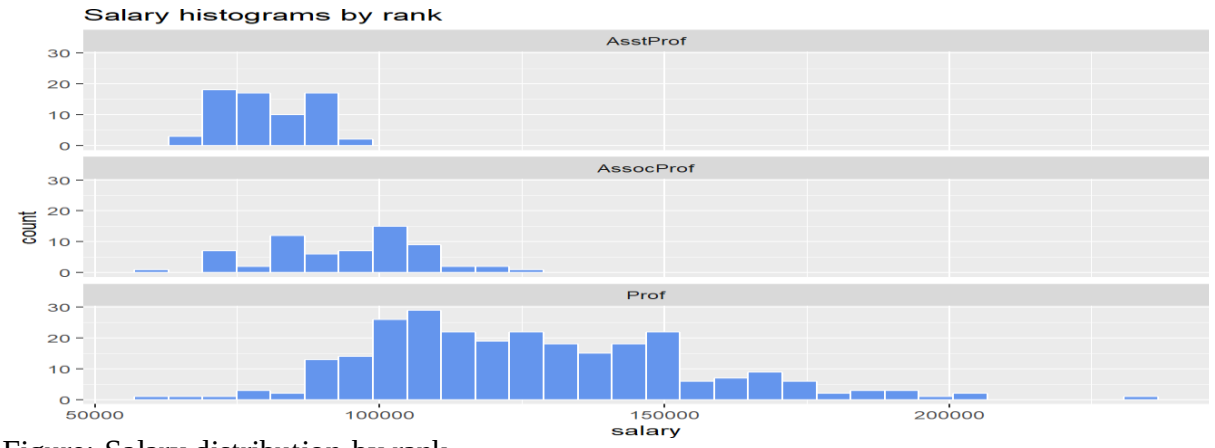


Figure: Salary distribution by rank

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

*****