BCA

P Veera Venkata Durga PraSad
Department of ComputerScience
Aditya Degree College for Women,KKD

INTRODUCTION TO DATA SCIENCE AND R PROGRAMMING

UNIT I:

Defining Data Science and Big data, Benefits and Uses, facets of Data, Data Science Process. History and Overview of R, Getting Started with R, R Nuts and Bolts

UNIT II:

The Data Science Process: Overview of the Data Science Process-Setting the research goal, Retrieving Data, Data Preparation, Exploration, Modeling, data Presentation and Automation. Getting Data in and out of R, Using reader package, Interfaces to the outside world.

UNIT III:

Machine Learning: Understanding why data scientists use machine learning-What is machine learning and why we should care about, Applications of machine learning in data science, Where it is used in data science, The modeling process, Types of Machine Learning-Supervised and Unsupervised.

UNIT IV:

Handling large Data on a Single Computer: The problems we face when handling large data, General Techniques for handling large volumes of data, Generating programming tips for dealing with large datasets. Case study-Predicting malicious URLs(This can be implemented in R).

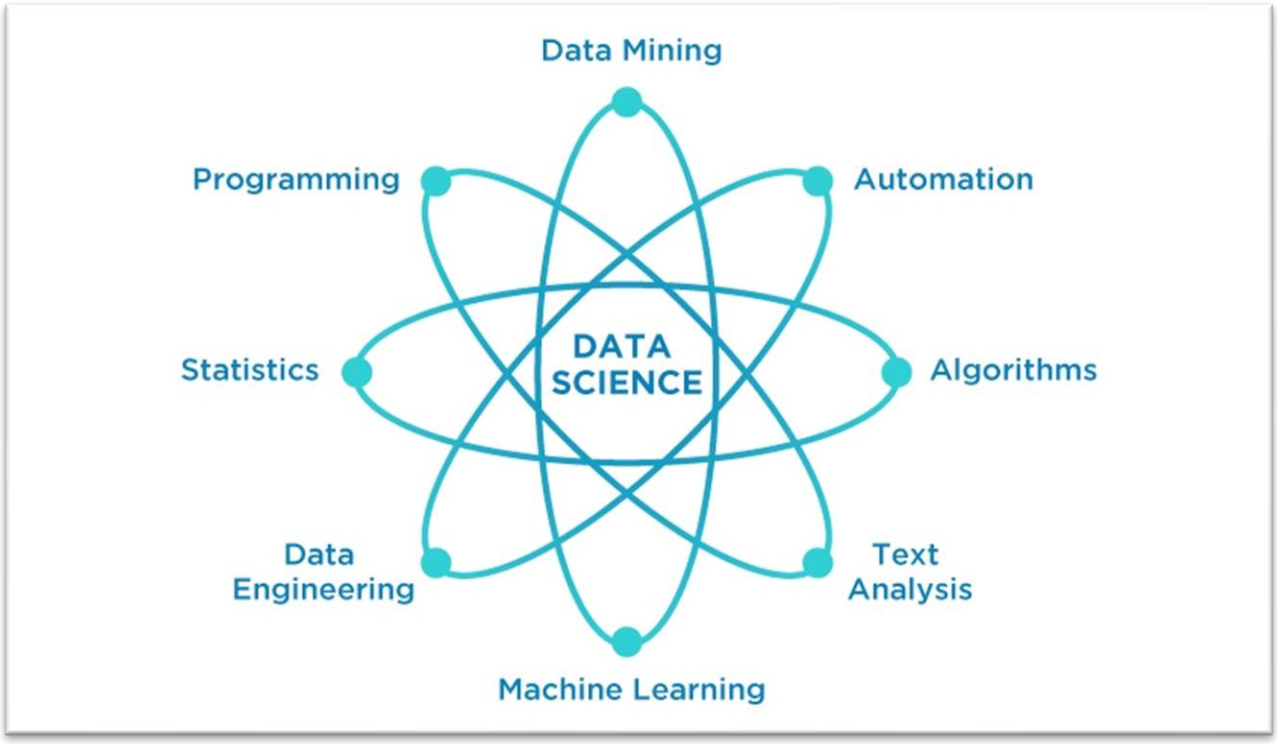
UNIT V:

Sub setting R objects, Vectorised Operations, Managing Data Frames with the dplyr, Control structures, functions, Scoping rules of R, Coding Standards in R, Loop Functions, Debugging, Simulation

UNIT- I

➢ Data Science

Data science combines math and statistics, specialized programming, advanced analytics, artificial intelligence (AI), and machine learning with specific subject matter expertise to uncover actionable insights hidden in an organization's data. These insights can be used to guide decision making and strategic planning.



The accelerating volume of data sources, and subsequently data, has made data science is one of the fastest growing field across every industry. As a result, it is no surprise that the role of the data scientist was dubbed the "sexiest job of the 21st century" by Harvard Business Review (link resides outside of IBM). Organizations are increasingly reliant on them to interpret data and provide actionable recommendations to improve business outcomes.

The data science lifecycle involves various roles, tools, and processes, which enables analysts to glean actionable insights. Typically, a data science project undergoes the following stages:

Data ingestion: The lifecycle begins with the data collection--both raw structured and unstructured data from all relevant sources using a variety of methods. These methods can include manual entry, web scraping, and real-time streaming data from systems and devices. Data sources can include structured data, such as customer data, along with unstructured data like log files, video, audio, pictures, the Internet of Things (IoT), social media, and more.
Data storage and data processing: Since data can have different formats and structures, companies need to consider different storage systems based on the type of data that needs to be captured. Data management teams help to set standards around data storage and structure, which facilitate workflows around analytics, machine learning and deep learning models. This stage includes cleaning data, deduplicating, transforming and combining the data using ETL (extract, transform, load) jobs or other data integration technologies. This data preparation is essential for promoting data quality before loading into a data warehouse, data lake, or other repository.
Data analysis: Here, data scientists conduct an exploratory data analysis to examine biases, patterns, ranges, and distributions of values within the data. This data analytics exploration drives hypothesis generation for a/b testing. It also allows analysts to determine the data's relevance for use within modeling efforts for predictive analytics, machine learning, and/or deep learning. Depending on a model's accuracy, organizations can become reliant on these insights for business decision making, allowing them to drive more scalability.
Communicate: Finally, insights are presented as reports and other data visualizations that make the insights—and their impact on business—easier for business analysts and other decision-makers to understand. A data science programming language such as R or Python includes components for generating visualizations; alternately, data scientists can use dedicated visualization tools.

➢ Big Data

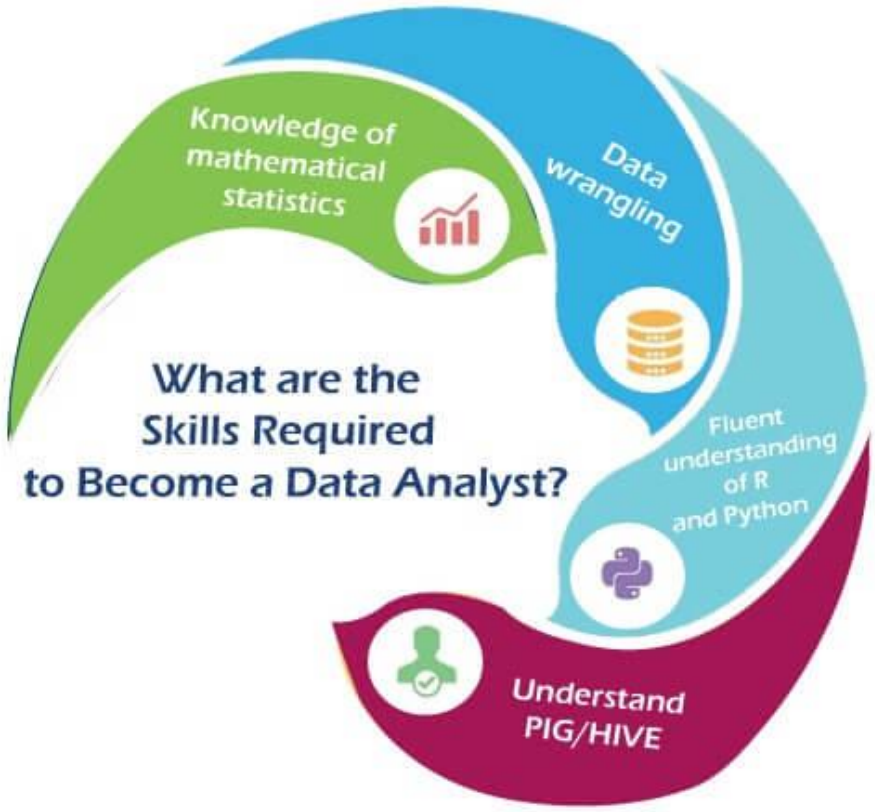
Big data is huge, large, or voluminous data, information, or the relevant statistics acquired by large organizations that are difficult to process by traditional tools. Big data can analyze structured, unstructured or semi-structured. Data is one of the key players to run any business, and it is exponentially increasing with passes of time. Before a decade, organizations were capable of dealing with gigabytes of

data only and suffered problems with data storage, but after emerging Big data, organizations are now capable of handling petabytes and exabytes of data as well as able to store huge volumes of data using cloud and big data frameworks such as Hadoop, etc.

Big Data is used to store, analyze and organize the huge volume of structured as well as unstructured datasets. Big Data can be described mainly with 5 V's as follows:

- Volume
- Variety
- Velocity
- Value
- Veracity

Skills required for Big Data



- Strong knowledge of Machine Learning concepts
- Understand the Database such as SQL, NoSQL, etc.
- In-depth knowledge of various programming languages such as Hadoop, Java, Python, etc.
- Knowledge of Apache Kafka, Scala, and cloud computing
- Knowledge of database warehouses such as Hive.

➢ Big data examples

Data can be a company's most valuable asset. Using big data to reveal insights can help you understand the areas that affect your business—from market conditions and customer purchasing behaviors to your business processes.

Here are some big data examples that are helping transform organizations across every industry:

- Tracking consumer behavior and shopping habits to deliver hyper-personalized retail product recommendations tailored to individual customers
- Monitoring payment patterns and analyzing them against historical customer activity to detect fraud in real time
- Combining data and information from every stage of an order's shipment journey with hyperlocal traffic insights to help fleet operators optimize last-mile delivery
- Using AI-powered technologies like natural language processing to analyze unstructured medical data (such as research reports, clinical notes, and lab results) to gain new insights for improved treatment development and enhanced patient care

- Using image data from cameras and sensors, as well as GPS data, to detect potholes and improve road maintenance in cities
- Analyzing public datasets of satellite imagery and geospatial datasets to visualize, monitor, measure, and predict the social and environmental impacts of supply chain operations

These are just a few ways organizations are using big data to become more data-driven so they can adapt better to the needs and expectations of their customers and the world around them.

The Vs of big data

Big data definitions may vary slightly, but it will always be described in terms of volume, velocity, and variety. These big data characteristics are often referred to as the "3 Vs of big data" and were first defined by Gartner in 2001.

Volume As its name suggests, the most common characteristic associated with big data is its high volume. This describes the enormous amount of data that is available for collection and produced from a variety of sources and devices on a continuous basis.
Velocity Big data velocity refers to the speed at which data is generated. Today, data is often produced in real time or near real time, and therefore, it must also be processed, accessed, and analyzed at the same rate to have any meaningful impact.
Variety Data is heterogeneous, meaning it can come from many different sources and can be structured, unstructured, or semi-structured. More traditional structured data (such as data in spreadsheets or relational databases) is now supplemented by

unstructured text, images, audio, video files, or semi-structured formats like sensor data that can't be organized in a fixed data schema.

In addition to these three original Vs, three others that are often mentioned in relation to harnessing the power of big data: **veracity**, **variability**, and **value**.

- **Veracity**: Big data can be messy, noisy, and error-prone, which makes it difficult to control the quality and accuracy of the data. Large datasets can be unwieldy and confusing, while smaller datasets could present an incomplete picture. The higher the veracity of the data, the more trustworthy it is.
- **Variability:** The meaning of collected data is constantly changing, which can lead to inconsistency over time. These shifts include not only changes in context and interpretation but also data collection methods based on the information that companies want to capture and analyze.
- **Value:** It's essential to determine the business value of the data you collect. Big data must contain the right data and then be effectively analyzed in order to yield insights that can help drive decision-making.

Big Data Works

The central concept of big data is that the more visibility you have into anything, the more effectively you can gain insights to make better decisions, uncover growth opportunities, and improve your business model.

Making big data work requires three main actions:

- **Integration:** Big data collects terabytes, and sometimes even petabytes, of raw data from many sources that must be received, processed, and transformed into the format that business users and analysts need to start analyzing it.

- **Management:** Big data needs big storage, whether in the cloud, on-premises, or both. Data must also be stored in whatever form required. It also needs to be processed and made available in real time. Increasingly, companies are turning to cloud solutions to take advantage of the unlimited compute and scalability.
- **Analysis:** The final step is analyzing and acting on big data—otherwise, the investment won't be worth it. Beyond exploring the data itself, it's also critical to communicate and share insights across the business in a way that everyone can understand. This includes using tools to create data visualizations like charts, graphs, and dashboards.

➢ Benefits And Uses of Data Science and Big Data

Data science and big data are used almost everywhere in both commercial and noncommercial settings. The number of use cases is vast, and the examples we'll provide throughout this book only scratch the surface of the possibilities.

Commercial companies in almost every industry use data science and big data to gain insights into their customers, processes, staff, completion, and products. Many companies use data science to offer customers a better user experience, as well as to cross-sell, up-sell, and personalize their offerings. A good example of this is Google AdSense, which collects data from internet users so relevant commercial messages can be matched to the person browsing the internet. MaxPoint (http://maxpoint.com/us) is another example of real-time personalized advertising. Human resource professionals use people analytics and text mining to screen candidates, monitor the mood of employees, and study informal networks among coworkers.

Governmental organizations are also aware of data's value. Many governmental organizations not only rely on internal data scientists to discover valuable information, but also share their data with the public. You can use this data to gain insights or build data-driven applications. *Data.gov* is but one example; it's the home of the US Government's open data. A data scientist in a governmental organization gets to work on diverse projects such as detecting fraud and other criminal activity or optimizing project funding. A well-known example was provided by Edward Snowden, who leaked internal documents of the American National Security Agency and the British Government Communications Headquarters that show clearly how they used data science and big data to monitor millions of individuals. Those organizations collected 5 billion data records from widespread applications such as Google Maps, Angry Birds, email, and text messages, among many other data sources. Then they applied data science techniques to distill information.

Nongovernmental organizations (NGOs) are also no strangers to using data. They use it to raise money and defend their causes. The World Wildlife Fund (WWF), for instance, employs data scientists to increase the effectiveness of their fundraising efforts. Many data scientists devote part of their time to helping NGOs, because NGOs often lack the resources to collect data and employ data scientists. DataKind is one such data scientist group that devotes its time to the benefit of mankind.

Universities use data science in their research but also to enhance the study experience of their students. The rise of massive open online courses (MOOC) produces a lot of data, which allows universities to study how this type of learning can complement traditional classes. MOOCs are an invaluable asset if you want to become a data scientist and big data professional, so definitely look at a few of the better-known ones: Coursera, Udacity, and edX. The big data and

data science landscape changes quickly, and MOOCs allow you to stay up to date by following courses from top universities. If you aren't acquainted with them yet, take time to do so now; you'll come to love them as we have.

➢ Facets of data

In data science and big data you'll come across many different types of data, and each of them tends to require different tools and techniques. The main categories of data are these:

- Structured
- Unstructured
- Natural language
- Machine-generated
- Graph-based
- Audio, video, and images
- Streaming

Let's explore all these interesting data types.

1.2.1. Structured data

Structured data is data that depends on a data model and resides in a fixed field within a record. As such, it's often easy to store structured data in tables within databases or Excel file. SQL, or Structured Query Language, is the preferred way to manage and query data that resides in databases. You may also come across structured data that might give you a hard time storing it in a traditional relational database. Hierarchical data such as a family tree is one such example.

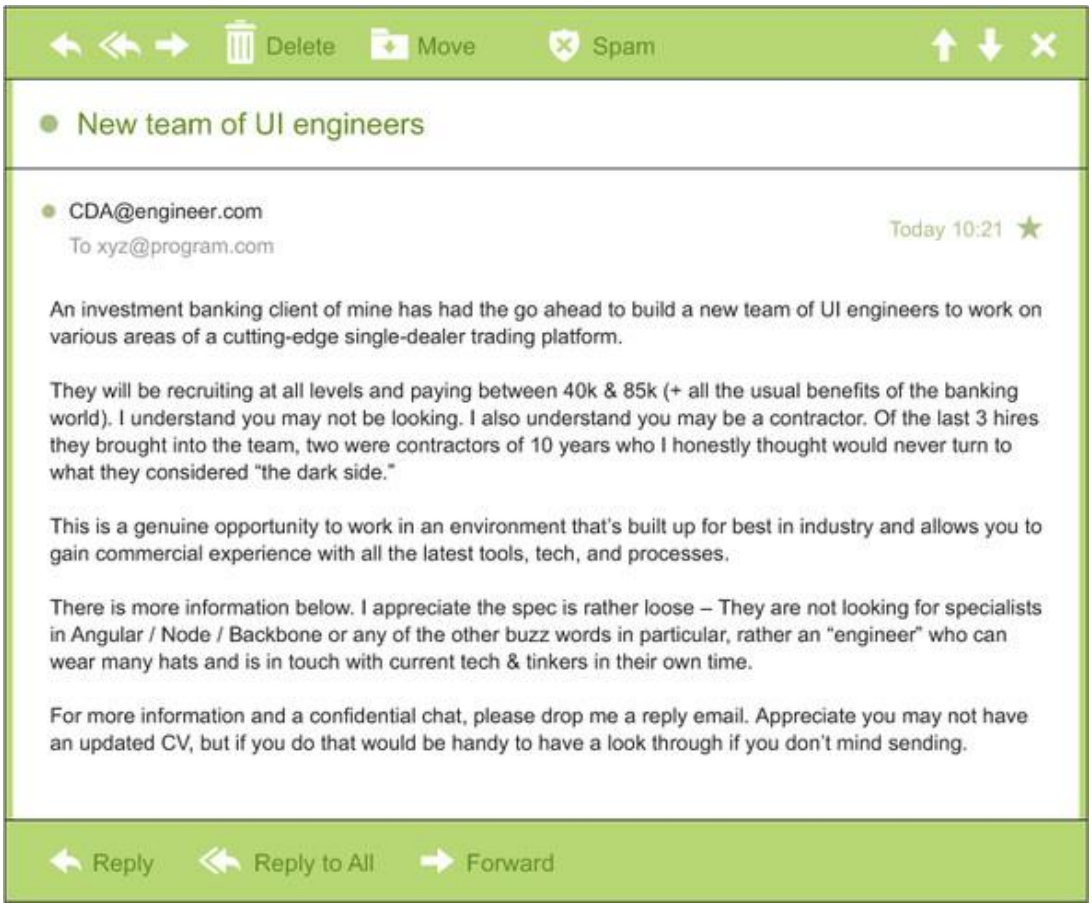
An Excel table is an example of structured data.

	Indicator ID	Dimension List	Timeframe	Numeric Value	Missing Value Flag	Confidence Inte
2	214390830	Total (Age-adjusted)	2008	74.6%		73.8%
3	214390833	Aged 18-44 years	2008	59.4%		58.0%
4	214390831	Aged 18-24 years	2008	37.4%		34.6%
5	214390832	Aged 25-44 years	2008	66.9%		65.5%
6	214390836	Aged 45-64 years	2008	88.6%		87.7%
7	214390834	Aged 45-54 years	2008	86.3%		85.1%
8	214390835	Aged 55-64 years	2008	91.5%		90.4%
9	214390840	Aged 65 years and over	2008	94.6%		93.8%
10	214390837	Aged 65-74 years	2008	93.6%		92.4%
11	214390838	Aged 75-84 years	2008	95.6%		94.4%
12	214390839	Aged 85 years and over	2008	96.0%		94.0%
13	214390841	Male (Age-adjusted)	2008	72.2%		71.1%
14	214390842	Female (Age-adjusted)	2008	76.8%		75.9%
15	214390843	White only (Age-adjusted)	2008	73.8%		72.9%
16	214390844	Black or African American only (Age-adjusted)	2008	77.0%		75.0%
17	214390845	American Indian or Alaska Native only (Age-adjusted)	2008	66.5%		57.1%
18	214390846	Asian only (Age-adjusted)	2008	80.5%		77.7%
19	214390847	Native Hawaiian or Other Pacific Islander only (Age-adjusted)	2008	DSU		
20	214390848	2 or more races (Age-adjusted)	2008	75.6%		69.6%

The world isn't made up of structured data, though; it's imposed upon it by humans and machines. More often, data comes unstructured.

1.2.2. Unstructured data

Unstructured data is data that isn't easy to fit into a data model because the content is context-specific or varying. One example of unstructured data is your regular email. Although email contains structured elements such as the sender, title, and body text, it's a challenge to find the number of people who have written an email complaint about a specific employee because so many ways exist to refer to a person, for example. The thousands of different languages and dialects out there further complicate this.



Email is simultaneously an example of unstructured data and natural language data.

A human-written email, as shown in above , is also a perfect example of natural language data.

1.2.3. Natural language

Natural language is a special type of unstructured data; it's challenging to process because it requires knowledge of specific data science techniques and linguistics.

The natural language processing community has had success in entity recognition, topic recognition, summarization, text completion, and sentiment analysis, but models trained in one domain don't generalize well to other

domains. Even state-of-the-art techniques aren't able to decipher the meaning of every piece of text. This shouldn't be a surprise though: humans struggle with natural language as well. It's ambiguous by nature. The concept of meaning itself is questionable here. Have two people listen to the same conversation. Will they get the same meaning? The meaning of the same words can vary when coming from someone upset or joyous.

1.2.4. Machine-generated data

Machine-generated data is information that's automatically created by a computer, process, application, or other machine without human intervention. Machine-generated data is becoming a major data resource and will continue to do so. Wikibon has forecast that the market value of the *industrial Internet* will be approximately $540 billion in 2020. IDC (International Data Corporation) has estimated there will be 26 times more connected things than people in 2020. This network is commonly referred to as *the internet of things*.

Examples of machine data are web server logs, call detail records, network event logs, and telemetry.

Example of machine-generated data

```
CSIPERF:TXCOMMIT;313236
2014-11-28 11:36:13, Info                  CSI    00000153 Creating NT transaction (seq
69), objectname [6]"(null)"
2014-11-28 11:36:13, Info                  CSI    00000154 Created NT transaction (seq 69)
result 0x00000000, handle @0x4e54
2014-11-28 11:36:13, Info                  CSI    00000155@2014/11/28:10:36:13.471
Beginning NT transaction commit...
2014-11-28 11:36:13, Info                  CSI    00000156@2014/11/28:10:36:13.705 CSI perf
trace:
CSIPERF:TXCOMMIT;273983
2014-11-28 11:36:13, Info                  CSI    00000157 Creating NT transaction (seq
70), objectname [6]"(null)"
2014-11-28 11:36:13, Info                  CSI    00000158 Created NT transaction (seq 70)
result 0x00000000, handle @0x4e5c
2014-11-28 11:36:13, Info                  CSI    00000159@2014/11/28:10:36:13.764
Beginning NT transaction commit...
2014-11-28 11:36:14, Info                  CSI    0000015a@2014/11/28:10:36:14.094 CSI perf
trace:
CSIPERF:TXCOMMIT;386259
2014-11-28 11:36:14, Info                  CSI    0000015b Creating NT transaction (seq
71), objectname [6]"(null)"
2014-11-28 11:36:14, Info                  CSI    0000015c Created NT transaction (seq 71)
result 0x00000000, handle @0x4e5c
2014-11-28 11:36:14, Info                  CSI    0000015d@2014/11/28:10:36:14.106
Beginning NT transaction commit...
2014-11-28 11:36:14, Info                  CSI    0000015e@2014/11/28:10:36:14.428 CSI perf
trace:
CSIPERF:TXCOMMIT;375581
```

The machine data shown in would fit nicely in a classic table-structured database. This isn't the best approach for highly interconnected or "networked" data, where the relationships between entities have a valuable role to play.

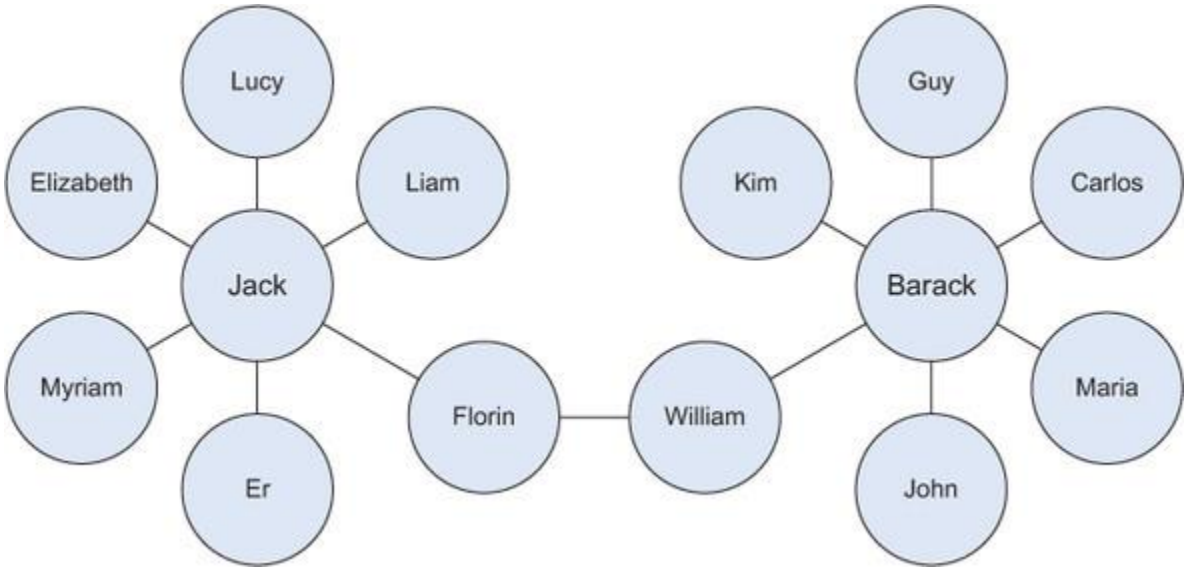
1.2.5. Graph-based or network data

"Graph data" can be a confusing term because any data can be shown in a graph. "Graph" in this case points to mathematical *graph theory*. In graph theory, a graph is a mathematical structure to model pair-wise relationships between objects. Graph or network data is, in short, data that focuses on the relationship or adjacency of objects. The graph structures use nodes, edges, and properties to represent and store graphical data. Graph-based data is a natural way to represent social networks, and its structure allows you to calculate specific metrics such as the influence of a person and the shortest path between two people.

Examples of graph-based data can be found on many social media websites. For instance, on LinkedIn you can see who you know at which company. Your

follower list on Twitter is another example of graph-based data. The power and sophistication comes from multiple, overlapping graphs of the same nodes. For example, imagine the connecting edges here to show “friends” on Facebook. Imagine another graph with the same people which connects business colleagues via LinkedIn. Imagine a third graph based on movie interests on Netflix. Overlapping the three different-looking graphs makes more interesting questions possible.

Friends in a social network are an example of graph-based data.



Graph databases are used to store graph-based data and are queried with specialized query languages such as SPARQL.

Graph data poses its challenges, but for a computer interpreting additive and image data, it can be even more difficult.

1.2.6. Audio, image, and video

Audio, image, and video are data types that pose specific challenges to a data scientist. Tasks that are trivial for humans, such as recognizing objects in pictures, turn out to be challenging for computers. MLBAM (Major League

Baseball Advanced Media) announced in 2014 that they'll increase video capture to approximately 7 TB per game for the purpose of live, in-game analytics. High-speed cameras at stadiums will capture ball and athlete movements to calculate in real time, for example, the path taken by a defender relative to two baselines.

Recently a company called DeepMind succeeded at creating an algorithm that's capable of learning how to play video games. This algorithm takes the video screen as input and learns to interpret everything via a complex process of deep learning. It's a remarkable feat that prompted Google to buy the company for their own Artificial Intelligence (AI) development plans. The learning algorithm takes in data as it's produced by the computer game; it's streaming data.

1.2.7. Streaming data

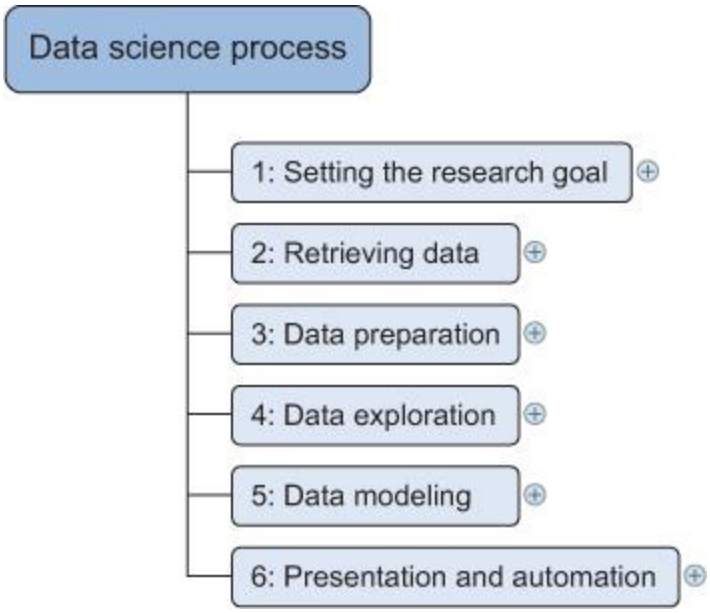
While streaming data can take almost any of the previous forms, it has an extra property. The data flows into the system when an event happens instead of being loaded into a data store in a batch. Although this isn't really a different type of data, we treat it here as such because you need to adapt your process to deal with this type of information.

Examples are the "What's trending" on Twitter, live sporting or music events, and the stock market.

➢ The Data Science Process

The data science process typically consists of six steps, as you can see in the mind map in below.

The data science process



1.3.1. Setting the research goal

Data science is mostly applied in the context of an organization. When the business asks you to perform a data science project, you'll first prepare a project charter. This charter contains information such as what you're going to research, how the company benefits from that, what data and resources you need, a timetable, and deliverables. Throughout this book, the data science process will be applied to bigger case studies and you'll get an idea of different possible research goals.

1.3.2. Retrieving data

The second step is to collect data. You've stated in the project charter which data you need and where you can find it. In this step you ensure that you can use the data in your program, which means checking the existence of, quality, and access to the data. Data can also be delivered by third-party companies and takes many forms ranging from Excel spreadsheets to different types of databases.

1.3.3. Data preparation

Data collection is an error-prone process; in this phase you enhance the quality of the data and prepare it for use in subsequent steps. This phase consists of three subphases: *data cleansing* removes false values from a data source and inconsistencies across data sources, *data integration* enriches data sources by combining information from multiple data sources, and *data transformation* ensures that the data is in a suitable format for use in your models.

1.3.4. Data exploration

Data exploration is concerned with building a deeper understanding of your data. You try to understand how variables interact with each other, the distribution of the data, and whether there are outliers. To achieve this you mainly use descriptive statistics, visual techniques, and simple modeling. This step often goes by the abbreviation EDA, for Exploratory Data Analysis.

1.3.5. Data modeling or model building

In this phase you use models, domain knowledge, and insights about the data you found in the previous steps to answer the research question. You select a technique from the fields of statistics, machine learning, operations research, and so on. Building a model is an iterative process that involves selecting the variables for the model, executing the model, and model diagnostics.

1.3.6. Presentation and automation

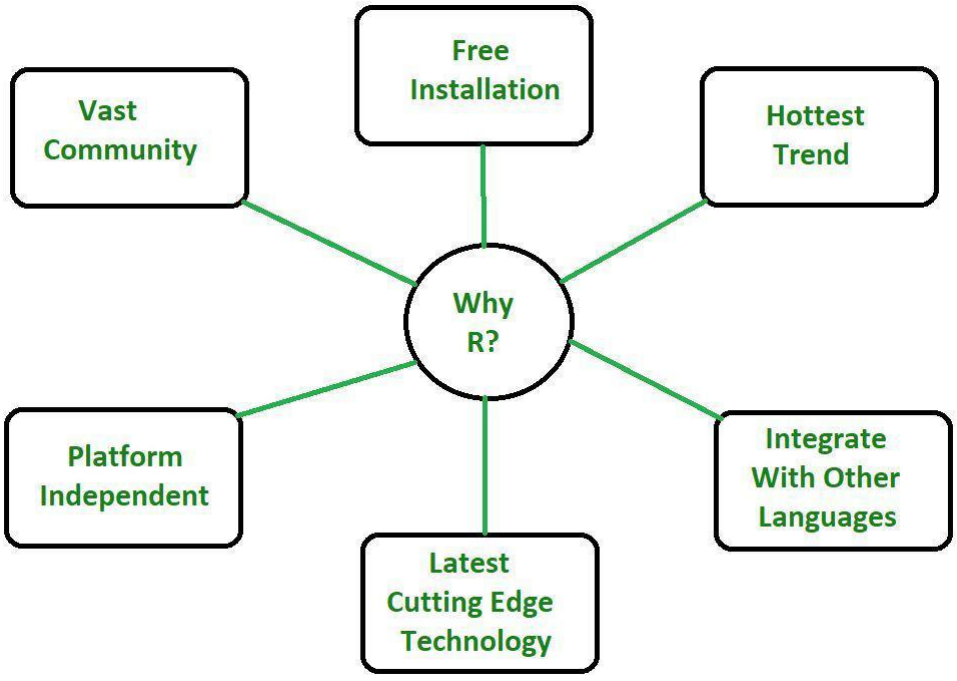
Finally, you present the results to your business. These results can take many forms, ranging from presentations to research reports. Sometimes you'll need to automate the execution of the process because the business will want to use the insights you gained in another project or enable an operational process to use the outcome from your model.

R Programming Language

R is an open-source programming language that is widely used as a statistical software and data analysis tool. R generally comes with the Command-line interface. R is available across widely used platforms like Windows, Linux, and macOS. Also, the R programming language is the latest cutting-edge tool.

It was designed by **Ross Ihaka and Robert Gentleman** at the University of Auckland, New Zealand, and is currently being developed by the R Development Core Team.

R programming language is an implementation of the S programming language. It also combines with lexical scoping semantics inspired by Scheme. Moreover, the project was conceived in 1992, with an initial version released in 1995 and a stable beta version in 2000.



R Programming Language

- R programming is used as a leading tool for machine learning, statistics, and data analysis. Objects, functions, and packages can easily be created by R.

- It's a platform-independent language. This means it can be applied to all operating systems.
- It's an open-source free language. That means anyone can install it in any organization without purchasing a license.
- R programming language is not only a statistic package but also allows us to integrate with other languages (C, C++). Thus, you can easily interact with many data sources and statistical packages.
- The R programming language has a vast community of users and it's growing day by day.
- R is currently one of the most requested programming languages in the Data Science job market which makes it the hottest trend nowadays

Using in R

- **Statistical Analysis:** R is designed for analysis and It provides an extensive collection of graphical and statistical techniques, By making a preferred choice for statisticians and data analysts.
- **Open Source:** R is an open – source software, which means it is freely available to anyone. It can be accessble by a vibrant community of users and developers.
- **Data Visulaization :** R boasts an array of libraries like ggplot2 that enable the creation of high-quality, customizable data visualizations.
- **Data Manipulation :** R offers tools that are for data manipulation and transformation. For example: IT simplifies the process of filtering , summarizing and transforming data.
- **Integration :** R can be easily integrate with other programming languages and data sources. IT has connectors to various databases and can be used in conjunction with python, SQL and other tools.
- **Community and Packages:** R has vast ecosystem of packages that extend its functionality. There are packages that can help you accomplish needs of analytics.

Features of R Programming Language

- **R Packages:** One of the major features of R is it has a wide availability of libraries. R has CRAN(**Comprehensive R Archive Network**), which is a repository holding more than 10, 0000 packages.
- **Distributed Computing:** Distributed computing is a model in which components of a software system are shared among multiple computers to improve efficiency

and performance. Two new packages ddR and multidplyr used for distributed programming in R were released in November 2015.

Statistical Features of R

- **Basic Statistics:** The most common basic statistics terms are the mean, mode, and median. These are all known as “Measures of Central Tendency.” So using the R language we can measure central tendency very easily.
- **Static graphics:** R is rich with facilities for creating and developing interesting static graphics. R contains functionality for many plot types including graphic maps, mosaic plots, biplots, and the list goes on.
- **Probability distributions:** Probability distributions play a vital role in statistics and by using R we can easily handle various types of probability distributions such as Binomial Distribution, Normal Distribution, Chi-squared Distribution, and many more.
- **Data analysis:** It provides a large, coherent, and integrated collection of tools for data analysis.

Advantages of R

- R is the most comprehensive statistical analysis package. As new technology and concepts often appear first in R.
- As R programming language is an open source. Thus, you can run R anywhere and at any time.
- R programming language is suitable for GNU/Linux and Windows operating systems.
- R programming is cross-platform and runs on any operating system.
- In R, everyone is welcome to provide new packages, bug fixes, and code enhancements.

Disadvantages of R

- In the R programming language, the standard of some packages is less than perfect.
- Although, R commands give little pressure on memory management. So R programming language may consume all available memory.
- In R basically, nobody to complain if something doesn’t work.
- R programming language is much slower than other programming languages such as Python and MATLAB.

Applications of R

- We use R for Data Science. It gives us a broad variety of libraries related to statistics. It also provides the environment for statistical computing and design.
- R is used by many quantitative analysts as its programming tool. Thus, it helps in data importing and cleaning.
- R is the most prevalent language. So many data analysts and research programmers use it. Hence, it is used as a fundamental tool for finance.
- Tech giants like Google, Facebook, Bing, Twitter, Accenture, Wipro, and many more using R nowadays.

➢ Getting Started With R

R is an **interpreted programming language**. It also allows you to carry out modular programming with the help of functions. It is widely used to analyze statistical information as well as graphical representation.
R allows you to integrate with programming procedures written in C, C++, Python, .Net, etc. Today, R is widely used in the field of data science by data analysts, researchers, statisticians, etc. It is used to retrieve data from datasets, clean it, analyze and visualize it, and present it in the most suitable way.

Install R in Your Local Machine

Before installing R on your computer, you first need to determine the operating system that you are using. R has binaries for all the major operating systems including Windows, MacOS, and Linux.

Running Your First R Program

Now that you have installed R and RStudio successfully, let's try to create your first R program. We will try to create a simple `Hello World` program.
A Hello World program is a simple program that simply prints a `"Hello World!"` message on the screen. It's generally used to introduce a new language to learners.
Consider the program below.

```
message <-"Hello World!"
print(message)
```

Output

```
[1] "Hello World!"
```

Here, we have created a simple variable called `message`. We have initialized this variable with a simple message string called `"Hello World!"`. On execution, this program prints the message stored inside the variable.

R Nuts and Bolts

1 Entering Input

At the R prompt we type expressions. The `<-` symbol is the assignment operator.

```
> x <- 1
> print(x)
[1] 1
> x
[1] 1
> msg <- "hello"
```

The grammar of the language determines whether an expression is complete or not.

```
x <-  ## Incomplete expression
```

The # character indicates a comment. Anything to the right of the # (including the # itself) is ignored. This is the only comment character in R. Unlike some other languages, R does not support multi-line comments or comment blocks.

2 Evaluation

When a complete expression is entered at the prompt, it is evaluated and the result of the evaluated expression is returned. The result may be *auto-printed*.

```
> x <- 5  ## nothing printed
> x      ## auto-printing occurs
[1] 5
> print(x)  ## explicit printing
[1] 5
```

The [1] shown in the output indicates that x is a vector and 5 is its first element.
Typically with interactive work, we do not explicitly print objects with the print function; it is much easier to just auto-print them by typing the name of the object and hitting return/enter. However, when writing scripts, functions, or longer programs, there is sometimes a need to explicitly print objects because auto-printing does not work in those settings.
When an R vector is printed you will notice that an index for the vector is printed in square brackets [] on the side. For example, see this integer sequence of length 20.

```
> x <- 11:30
> x
 [1] 11 12 13 14 15 16 17 18 19 20 21 22
[13] 23 24 25 26 27 28 29 30
```

The numbers in the square brackets are not part of the vector itself, they are merely part of the *printed output*.

With R, it's important that one understand that there is a difference between the actual R object and the manner in which that R object is printed to the console. Often, the printed output may have additional bells and whistles to make the output more friendly to the users. However, these bells and whistles are not inherently part of the object.

Note that the : operator is used to create integer sequences.

3 R Objects

Watch a video of this section

R has five basic or “atomic” classes of objects:

- character
- numeric (real numbers)
- integer
- complex
- logical (True/False)

The most basic type of R object is a vector. Empty vectors can be created with the `vector()` function. There is really only one rule about vectors in R, which is that **A vector can only contain objects of the same class**.
But of course, like any good rule, there is an exception, which is a *list*, which we will get to a bit later. A list is represented as a vector but can contain objects of different classes. Indeed, that’s usually why we use them.

There is also a class for “raw” objects, but they are not commonly used directly in data analysis and I won’t cover them here.

4 Numbers

Numbers in R are generally treated as numeric objects (i.e. double precision real numbers). This means that even if you see a number like “1” or “2” in R, which you might think of as integers, they are likely represented behind the scenes as numeric objects (so something like “1.00” or “2.00”). This isn’t important most of the time…except when it is.

If you explicitly want an integer, you need to specify the `L` suffix. So entering `1` in R gives you a numeric object; entering `1L` explicitly gives you an integer object.
There is also a special number `Inf` which represents infinity. This allows us to represent entities like `1 / 0`. This way, `Inf` can be used in ordinary calculations; e.g. `1 / Inf` is 0.
The value `NaN` represents an undefined value (“not a number”); e.g. `0 / 0`; `NaN` can also be thought of as a missing value (more on that later)

5 Attributes

R objects can have attributes, which are like metadata for the object. These metadata can be very useful in that they help to describe the object. For example, column names on a data frame help to tell us what data are contained in each of the columns. Some examples of R object attributes are

- names, dimnames
- dimensions (e.g. matrices, arrays)
- class (e.g. integer, numeric)
- length
- other user-defined attributes/metadata

Attributes of an object (if any) can be accessed using the attributes() function. Not all R objects contain attributes, in which case the attributes() function returns NULL.

6 Creating Vectors

Watch a video of this section

The c() function can be used to create vectors of objects by concatenating things together.

```
> x <- c(0.5, 0.6)      ## numeric
> x <- c(TRUE, FALSE)   ## logical
> x <- c(T, F)          ## logical
> x <- c("a", "b", "c") ## character
> x <- 9:29             ## integer
> x <- c(1+0i, 2+4i)    ## complex
```

Note that in the above example, T and F are short-hand ways to specify TRUE and FALSE. However, in general one should try to use the explicit TRUE and FALSE values when indicating logical values.
The T and F values are primarily there for when you're feeling lazy.
You can also use the vector() function to initialize vectors.

```
> x <- vector("numeric", length = 10)
> x
 [1] 0 0 0 0 0 0 0 0 0 0
```

7 Mixing Objects

There are occasions when different classes of R objects get mixed together. Sometimes this happens by accident but it can also happen on purpose. So what happens with the following code?

```
> y <- c(1.7, "a")   ## character
> y <- c(TRUE, 2)    ## numeric
> y <- c("a", TRUE)  ## character
```

In each case above, we are mixing objects of two different classes in a vector. But remember that the only rule about vectors says this is not allowed. When different objects are mixed in a vector, *coercion* occurs so that every element in the vector is of the same class.

In the example above, we see the effect of *implicit coercion*. What R tries to do is find a way to represent all of the objects in the vector in a reasonable fashion. Sometimes this does exactly what you want and…sometimes not. For example, combining a numeric object with a character object will create a character vector, because numbers can usually be easily represented as strings.

8 Explicit Coercion

Objects can be explicitly coerced from one class to another using the as.* functions, if available.

```
> x <- 0:6
> class(x)
[1] "integer"
> as.numeric(x)
[1] 0 1 2 3 4 5 6
> as.logical(x)
[1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
> as.character(x)
[1] "0" "1" "2" "3" "4" "5" "6"
```

Sometimes, R can't figure out how to coerce an object and this can result in NAs being produced.

```
> x <- c("a", "b", "c")
> as.numeric(x)
Warning: NAs introduced by coercion
[1] NA NA NA
> as.logical(x)
[1] NA NA NA
> as.complex(x)
Warning: NAs introduced by coercion
```

```
[1] NA NA NA
```

When nonsensical coercion takes place, you will usually get a warning from R.

9 Matrices

Matrices are vectors with a *dimension* attribute. The dimension attribute is itself an integer vector of length 2 (number of rows, number of columns)

```
> m <- matrix(nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
> dim(m)
[1] 2 3
> attributes(m)
$dim
[1] 2 3
```

Matrices are constructed *column-wise*, so entries can be thought of starting in the “upper left” corner and running down the columns.

```
> m <- matrix(1:6, nrow = 2, ncol = 3)
> m
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Matrices can also be created directly from vectors by adding a dimension attribute.

```
> m <- 1:10
> m
 [1]  1  2  3  4  5  6  7  8  9 10
> dim(m) <- c(2, 5)
> m
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
```

Matrices can be created by *column-binding* or *row-binding* with the cbind() and rbind() functions.

```
> x <- 1:3
> y <- 10:12
> cbind(x, y)
```

```
     x  y
[1,] 1 10
[2,] 2 11
[3,] 3 12
> rbind(x, y)
  [,1] [,2] [,3]
x    1    2    3
y   10   11   12
```

10 Lists

Lists are a special type of vector that can contain elements of different classes. Lists are a very important data type in R and you should get to know them well. Lists, in combination with the various “apply” functions discussed later, make for a powerful combination.

Lists can be explicitly created using the list() function, which takes an arbitrary number of arguments.

```
> x <- list(1, "a", TRUE, 1 + 4i)
> x
[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
[1] TRUE

[[4]]
[1] 1+4i
```

We can also create an empty list of a prespecified length with the vector() function

```
> x <- vector("list", length = 5)
> x
[[1]]
NULL

[[2]]
NULL

[[3]]
```

```
NULL

[[4]]
NULL

[[5]]
NULL
```

11 Factors

Watch a video of this section

Factors are used to represent categorical data and can be unordered or ordered. One can think of a factor as an integer vector where each integer has a *label*. Factors are important in statistical modeling and are treated specially by modelling functions like lm() and glm().
Using factors with labels is *better* than using integers because factors are self-describing. Having a variable that has values “Male” and “Female” is better than a variable that has values 1 and 2.

Factor objects can be created with the factor() function.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"))
> x
[1] yes yes no  yes no
Levels: no yes
> table(x)
x
 no yes
  2   3
> ## See the underlying representation of factor
> unclass(x)
[1] 2 2 1 2 1
attr(,"levels")
[1] "no"  "yes"
```

Often factors will be automatically created for you when you read a dataset in using a function like read.table(). Those functions often default to creating factors when they encounter data that look like characters or strings.
The order of the levels of a factor can be set using the levels argument to factor(). This can be important in linear modelling because the first level is used as the baseline level.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"))
> x  ## Levels are put in alphabetical order
```

```
[1] yes yes no  yes no
Levels: no yes
> x <- factor(c("yes", "yes", "no", "yes", "no"),
+             levels = c("yes", "no"))
> x
[1] yes yes no  yes no
Levels: yes no
```

12 Missing Values

Missing values are denoted by NA or NaN for q undefined mathematical operations.

- is.na() is used to test objects if they are NA
- is.nan() is used to test for NaN
- NA values have a class also, so there are integer NA, character NA, etc.
- A NaN value is also NA but the converse is not true

```
> ## Create a vector with NAs in it
> x <- c(1, 2, NA, 10, 3)
> ## Return a logical vector indicating which elements are NA
> is.na(x)
[1] FALSE FALSE  TRUE FALSE FALSE
> ## Return a logical vector indicating which elements are NaN
> is.nan(x)
[1] FALSE FALSE FALSE FALSE FALSE
> ## Now create a vector with both NA and NaN values
> x <- c(1, 2, NaN, NA, 4)
> is.na(x)
[1] FALSE FALSE  TRUE  TRUE FALSE
> is.nan(x)
[1] FALSE FALSE  TRUE FALSE FALSE
```

13 Data Frames

Data frames are used to store tabular data in R. They are an important type of object in R and are used in a variety of statistical modeling applications. Hadley Wickham's package dplyr has an optimized set of functions designed to work efficiently with data frames.

Data frames are represented as a special type of list where every element of the list has to have the same length. Each element of the list can be thought of as a column and the length of each element of the list is the number of rows.

Unlike matrices, data frames can store different classes of objects in each column. Matrices must have every element be the same class (e.g. all integers or all numeric).

In addition to column names, indicating the names of the variables or predictors, data frames have a special attribute called `row.names` which indicate information about each row of the data frame.
Data frames are usually created by reading in a dataset using the `read.table()` or `read.csv()`. However, data frames can also be created explicitly with the `data.frame()` function or they can be coerced from other types of objects like lists.
Data frames can be converted to a matrix by calling `data.matrix()`. While it might seem that the `as.matrix()` function should be used to coerce a data frame to a matrix, almost always, what you want is the result of `data.matrix()`.

```
> x <- data.frame(foo = 1:4, bar = c(T, T, F, F))
> x
  foo   bar
1   1  TRUE
2   2  TRUE
3   3 FALSE
4   4 FALSE
> nrow(x)
[1] 4
> ncol(x)
[1] 2
```

14 Names

R objects can have names, which is very useful for writing readable code and self-describing objects. Here is an example of assigning names to an integer vector.

```
> x <- 1:3
> names(x)
NULL
> names(x) <- c("New York", "Seattle", "Los Angeles")
> x
  New York    Seattle Los Angeles
         1          2          3
> names(x)
[1] "New York"    "Seattle"     "Los Angeles"
```

Lists can also have names, which is often very useful.

```
> x <- list("Los Angeles" = 1, Boston = 2, London = 3)
> x
$`Los Angeles`
[1] 1

$Boston
[1] 2

$London
[1] 3
> names(x)
[1] "Los Angeles" "Boston"      "London"
```

Matrices can have both column and row names.

```
> m <- matrix(1:4, nrow = 2, ncol = 2)
> dimnames(m) <- list(c("a", "b"), c("c", "d"))
> m
  c d
a 1 3
b 2 4
```

Column names and row names can be set separately using the colnames() and rownames() functions.

```
> colnames(m) <- c("h", "f")
> rownames(m) <- c("x", "z")
> m
  h f
x 1 3
z 2 4
```

Note that for data frames, there is a separate function for setting the row names, the row.names() function. Also, data frames do not have column names, they just have names (like lists). So to set the column names of a data frame just use the names() function. Yes, I know its confusing. Here's a quick summary:

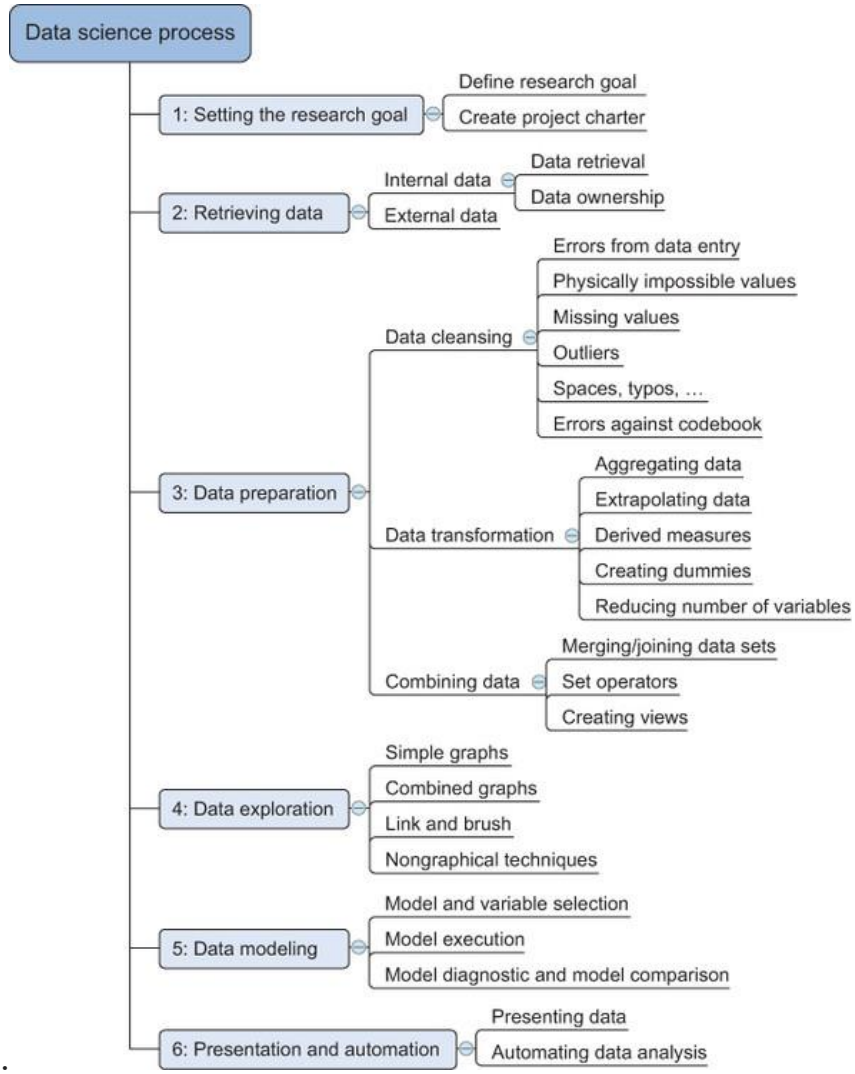
Object	Set column names	Set row names
data frame	names()	row.names()
matrix	colnames()	rownames()

UNIT-II

➢ Overview of the data science process

Following a structured approach to data science helps you to maximize your chances of success in a data science project at the lowest cost. It also makes it possible to take up a project as a team, with each team member focusing on what they do best. Take care, however: this approach may not be suitable for every type of project or be the only way to do good data science.

The typical data science process consists of six steps through which you'll iterate, as shown in figure



summarizes the data science process and shows the main steps and actions you'll take during a project. The following list is a short introduction; each of the steps will be discussed in greater depth throughout this chapter.

1. The first step of this process is setting a *research goal*. The main purpose here is making sure all the stakeholders understand the *what*, *how*, and *why* of the project. In every serious project this will result in a project charter.

2. The second phase is *data retrieval*. You want to have data available for analysis, so this step includes finding suitable data and getting access to the data from the data owner. The result is data in its raw form, which probably needs polishing and transformation before it becomes usable.

3. Now that you have the raw data, it's time to *prepare* it. This includes transforming the data from a raw form into data that's directly usable in your models. To achieve this, you'll detect and correct different kinds of errors in the data, combine data from different data sources, and transform it. If you have successfully completed this step, you can progress to data visualization and modeling.

4. The fourth step is *data exploration*. The goal of this step is to gain a deep understanding of the data. You'll look for patterns, correlations, and deviations based on visual and descriptive techniques. The insights you gain from this phase will enable you to start modeling.

5. Finally, we get to the sexiest part: *model building* (often referred to as "data modeling" throughout this book). It is now that you attempt to gain the insights or make the predictions stated in your project charter. Now is the time to bring out the heavy guns, but remember research has taught us that often (but not always) a combination of simple

models tends to outperform one complicated model. If you've done this phase right, you're almost done.

6. The last step of the data science model is *presenting your results and automating the analysis,* if needed. One goal of a project is to change a process and/or make better decisions. You may still need to convince the business that your findings will indeed change the business process as expected. This is where you can shine in your influencer role. The importance of this step is more apparent in projects on a strategic and tactical level. Certain projects require you to perform the business process over and over again, so automating the project will save time.

In reality you won't progress in a linear way from step 1 to step 6. Often you'll regress and iterate between the different phases.

Following these six steps pays off in terms of a higher project success ratio and increased impact of research results. This process ensures you have a well-defined research plan, a good understanding of the business question, and clear deliverables before you even start looking at data. The first steps of your process focus on getting high-quality data as input for your models. This way your models will perform better later on. In data science there's a well-known saying: *Garbage in equals garbage out*.

Another benefit of following a structured approach is that you work more in *prototype mode* while you search for the best model. When building a *prototype,* you'll probably try multiple models and won't focus heavily on issues such as program speed or writing code against standards. This allows you to focus on bringing business value instead.

Not every project is initiated by the business itself. Insights learned during analysis or the arrival of new data can spawn new projects. When the data science team generates an idea, work has already been done to make a proposition and find a business sponsor.

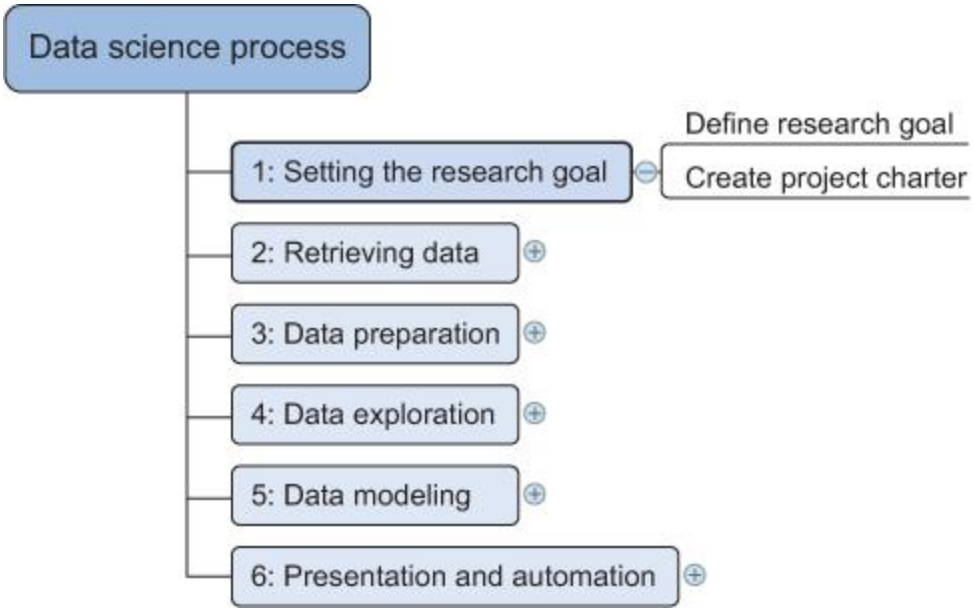
Dividing a project into smaller stages also allows employees to work together as a team. It's impossible to be a specialist in everything. You'd need to know how to upload all the data to all the different databases, find an optimal data scheme that works not only for your application but also for other projects inside your company, and then keep track of all the statistical and data-mining techniques, while also being an expert in presentation tools and business politics. That's a hard task, and it's why more and more companies rely on a team of specialists rather than trying to find one person who can do it all.

The process we described in this section is best suited for a data science project that contains only a few models. It's not suited for every type of project. For instance, a project that contains millions of real-time models would need a different approach than the flow we describe here. A beginning data scientist should get a long way following this manner of working, though.

➢ Step 1: Defining research goals and creating a project charter

A project starts by understanding the *what*, the *why*, and the *how* of your project (figure 2.2). What does the company expect you to do? And why does management place such a value on your research? Is it part of a bigger strategic picture or a "lone wolf" project originating from an opportunity someone detected? Answering these three questions (what, why, how) is the goal of the first phase, so that everybody knows what to do and can agree on the best course of action.

Figure 2.2. Step 1: Setting the research goal



The outcome should be a clear research goal, a good understanding of the context, well-defined deliverables, and a plan of action with a timetable. This information is then best placed in a project charter. The length and formality can, of course, differ between projects and companies. In this early phase of the project, people skills and business acumen are more important than great technical prowess, which is why this part will often be guided by more senior personnel.

1. Spend time understanding the goals and context of your research

An essential outcome is the research goal that states the purpose of your assignment in a clear and focused manner. Understanding the business goals and context is critical for project success. Continue asking questions and devising examples until you grasp the exact business expectations, identify how your project fits in the bigger picture, appreciate how your research is going to change the business, and understand how they'll use your results. Nothing is more frustrating than spending months researching something until you have that one moment of brilliance and solve the problem, but when you report your findings back to the organization, everyone immediately realizes that you misunderstood their question. Don't skim over this phase lightly. Many data scientists fail here: despite

their mathematical wit and scientific brilliance, they never seem to grasp the business goals and context.

2. Create a project charter

Clients like to know upfront what they're paying for, so after you have a good understanding of the business problem, try to get a formal agreement on the deliverables. All this information is best collected in a project charter. For any significant project this would be mandatory.

A project charter requires teamwork, and your input covers at least the following:

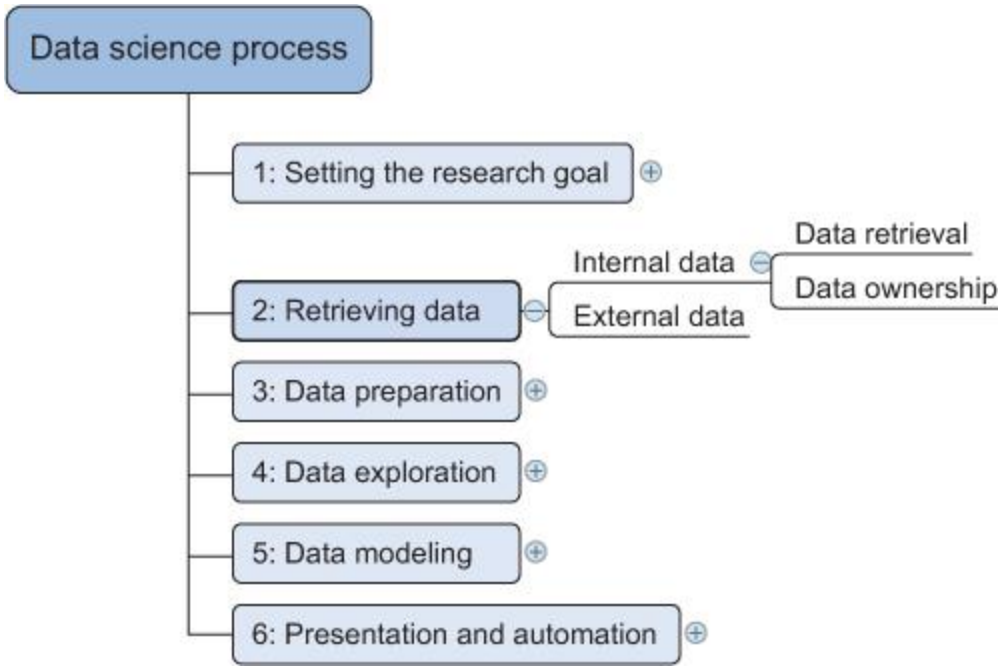
- A clear research goal
- The project mission and context
- How you're going to perform your analysis
- What resources you expect to use
- Proof that it's an achievable project, or proof of concepts
- Deliverables and a measure of success
- A timeline

Your client can use this information to make an estimation of the project costs and the data and people required for your project to become a success.

➢ Step 2: Retrieving data

The next step in data science is to retrieve the required data (figure 2.3). Sometimes you need to go into the field and design a data collection process yourself, but most of the time you won't be involved in this step. Many companies will have already collected and stored the data for you, and what they don't have can often be bought from third parties. Don't be afraid to look outside your organization for data, because more and more organizations are making even high-quality data freely available for public and commercial use.

Figure 2.3. Step 2: Retrieving data



Data can be stored in many forms, ranging from simple text files to tables in a database. The objective now is acquiring all the data you need. This may be difficult, and even if you succeed, data is often like a diamond in the rough: it needs polishing to be of any use to you.

1. Start with data stored within the company

Your first act should be to assess the relevance and quality of the data that's readily available within your company. Most companies have a program for maintaining key data, so much of the cleaning work may already be done. This data can be stored in official data repositories such as *databases*, *data marts*, *data warehouses*, and *data lakes* maintained by a team of IT professionals. The primary goal of a database is data storage, while a data warehouse is designed for reading and analyzing that data. A data mart is a subset of the data warehouse and geared toward serving a specific business unit. While data warehouses and data marts are home to preprocessed data, data lakes contains data in its natural or raw format. But the possibility exists that your data still resides in Excel files on the desktop of a domain expert.

Finding data even within your own company can sometimes be a challenge. As companies grow, their data becomes scattered around many places. Knowledge of the data may be dispersed as people change positions and leave the company. Documentation and metadata aren't always the top priority of a delivery manager, so it's possible you'll need to develop some Sherlock Holmes–like skills to find all the lost bits.

Getting access to data is another difficult task. Organizations understand the value and sensitivity of data and often have policies in place so everyone has access to what they need and nothing more. These policies translate into physical and digital barriers called *Chinese walls*. These "walls" are mandatory and well-regulated for customer data in most countries. This is for good reasons, too; imagine everybody in a credit card company having access to your spending habits. Getting access to the data may take time and involve company politics.

2. Don't be afraid to shop around

If data isn't available inside your organization, look outside your organization's walls. Many companies specialize in collecting valuable information. For instance, Nielsen and GFK are well known for this in the retail industry. Other companies provide data so that you, in turn, can enrich their services and ecosystem. Such is the case with Twitter, LinkedIn, and Facebook.

Although data is considered an asset more valuable than oil by certain companies, more and more governments and organizations share their data for free with the world. This data can be of excellent quality; it depends on the institution that creates and manages it. The information they share covers a broad range of topics such as the number of accidents or amount of drug abuse in a certain region and its demographics. This data is helpful when you want to enrich proprietary data but also convenient when training your data science skills at home. Table 2.1 shows only a small selection from the growing number of open-data providers.

Table 2.1. A list of open-data providers that should get you started

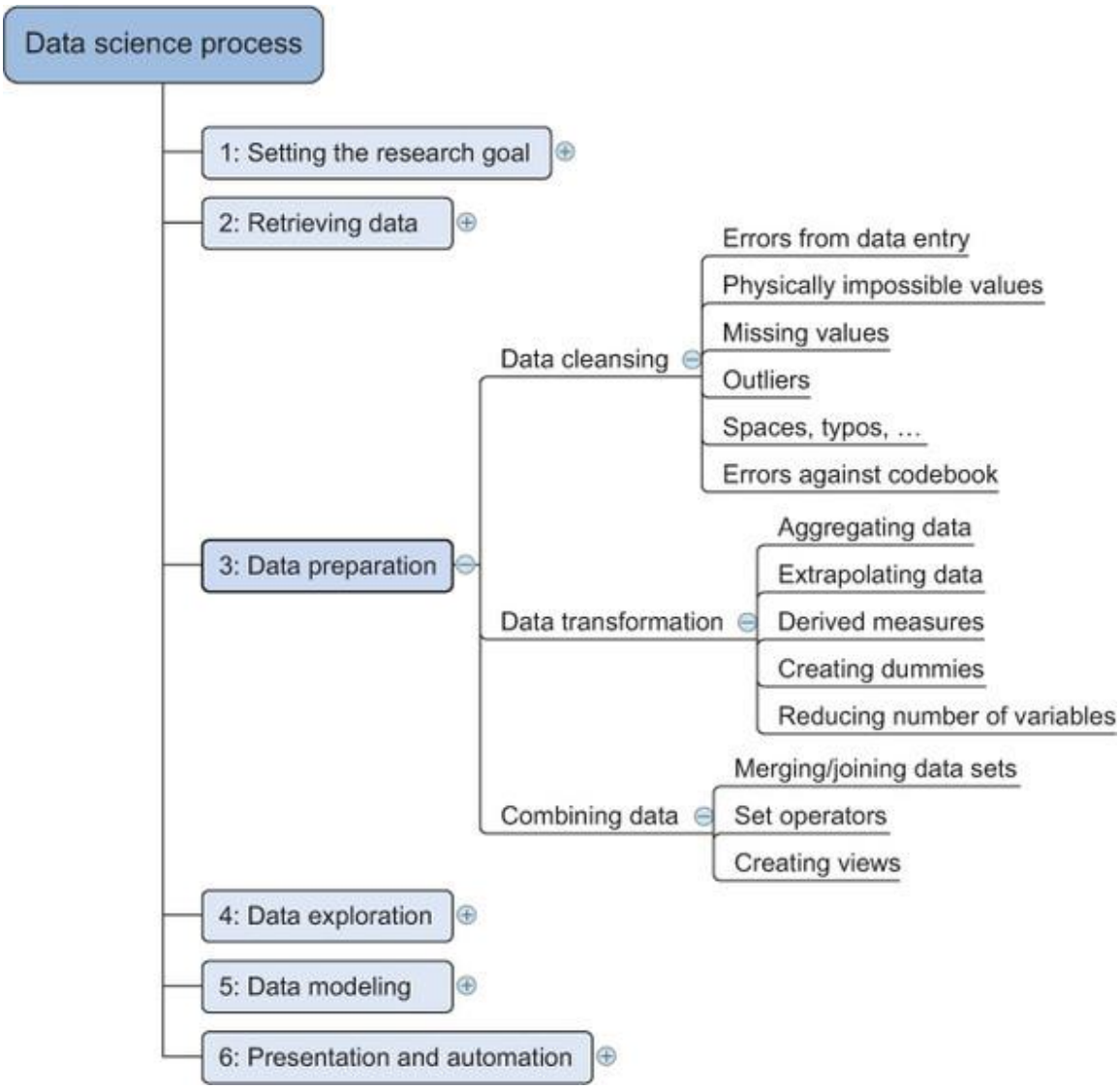
Open data site	Description
Data.gov	The home of the US Government's open data
https://open-data.europa.eu/	The home of the European Commission's open data
Freebase.org	An open database that retrieves its information from sites like Wikipedia, MusicBrains, and the SEC archive
Data.worldbank.org	Open data initiative from the World Bank
Aiddata.org	Open data for international development
Open.fda.gov	Open data from the US Food and Drug Administration

➢ Step 3: Cleansing, integrating, and transforming data

The data received from the data retrieval phase is likely to be "a diamond in the rough." Your task now is to sanitize and prepare it for use in the modeling and reporting phase. Doing so is tremendously important because your models will perform better and you'll lose less time trying to fix strange output. It can't be mentioned nearly enough times: garbage in equals garbage out. Your model needs the data in a specific format, so data transformation will always come into play. It's a good habit to correct data errors as early on in the process as possible. However, this isn't always possible in a realistic setting, so you'll need to take corrective actions in your program.

Figure 2.4 shows the most common actions to take during the data cleansing, integration, and transformation phase.

Figure 2.4. Step 3: Data preparation



This mind map may look a bit abstract for now, but we'll handle all of these points in more detail in the next sections. You'll see a great commonality among all of these actions.

➢ 1. Cleansing data

Data cleansing is a subprocess of the data science process that focuses on removing errors in your data so your data becomes a true and consistent representation of the processes it originates from.

By "true and consistent representation" we imply that at least two types of errors exist. The first type is the *interpretation error*, such as when you take the value in your data for

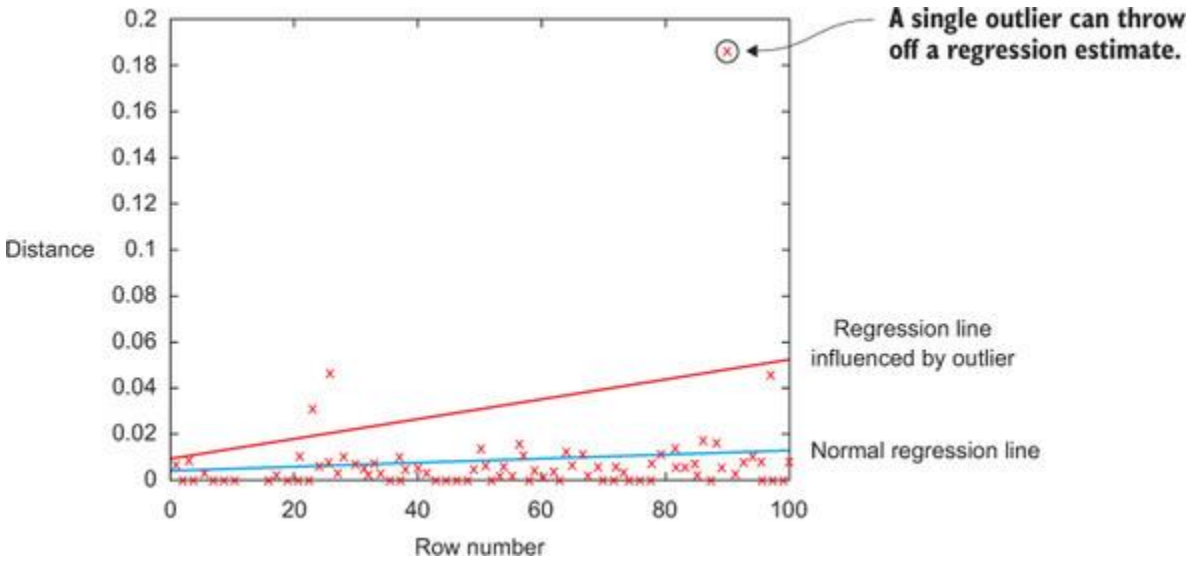
granted, like saying that a person's age is greater than 300 years. The second type of error points to *inconsistencies* between data sources or against your company's standardized values. An example of this class of errors is putting "Female" in one table and "F" in another when they represent the same thing: that the person is female. Another example is that you use Pounds in one table and Dollars in another. Too many possible errors exist for this list to be exhaustive, but table 2.2 shows an overview of the types of errors that can be detected with easy checks—the "low hanging fruit," as it were.

Table 2.2. An overview of common errors

General solution	
Try to fix the problem early in the data acquisition chain or else fix it in the program.	
Error description	**Possible solution**
Errors pointing to false values within one data set	
Mistakes during data entry	Manual overrules
Redundant white space	Use string functions
Impossible values	Manual overrules
Missing values	Remove observation or value
Outliers	Validate and, if erroneous, treat as missing value (remove or insert)
Errors pointing to inconsistencies between data sets	
Deviations from a code book	Match on keys or else use manual overrules

Sometimes you'll use more advanced methods, such as simple modeling, to find and identify data errors; diagnostic plots can be especially insightful. For example, in figure 2.5 we use a measure to identify data points that seem out of place. We do a regression to

get acquainted with the data and detect the influence of individual observations on the regression line. When a single observation has too much influence, this can point to an error in the data, but it can also be a valid point. At the data cleansing stage, these advanced methods are, however, rarely applied and often regarded by certain data scientists as overkill.



Now that we've given the overview, it's time to explain these errors in more detail.

a) Data entry errors

Data collection and data entry are error-prone processes. They often require human intervention, and because humans are only human, they make typos or lose their concentration for a second and introduce an error into the chain. But data collected by machines or computers isn't free from errors either. Errors can arise from human sloppiness, whereas others are due to machine or hardware failure. Examples of errors originating from machines are transmission errors or bugs in the extract, transform, and load phase (ETL).

For small data sets you can check every value by hand. Detecting data errors when the variables you study don't have many classes can be done by tabulating the data with counts. When you have a variable that can take only two values: "Good" and "Bad", you

can create a frequency table and see if those are truly the only two values present. In table 2.3, the values “Godo” and “Bade” point out something went wrong in at least 16 cases.

Table 2.3. Detecting outliers on simple variables with a frequency table

Value	Count
Good	1598647
Bad	1354468
Godo	15
Bade	1

Most errors of this type are easy to fix with simple assignment statements and if-then-else rules:

```
if x == "Godo":
   x = "Good"
if x == "Bade":
   x = "Bad"
```

copy

b) Redundant whitespace

Whitespaces tend to be hard to detect but cause errors like other redundant characters would. Who hasn’t lost a few days in a project because of a bug that was caused by whitespaces at the end of a string? You ask the program to join two keys and notice that observations are missing from the output file. After looking for days through the code, you finally find the bug. Then comes the hardest part: explaining the delay to the project stakeholders. The cleaning during the ETL phase wasn’t well executed, and keys in one table contained a whitespace at the end of a string. This caused a mismatch of keys such as “FR” – “FR”, dropping the observations that couldn’t be matched.

c) Impossible values and sanity checks

Sanity checks are another valuable type of data check. Here you check the value against physically or theoretically impossible values such as people taller than 3 meters or someone with an age of 299 years. Sanity checks can be directly expressed with rules:

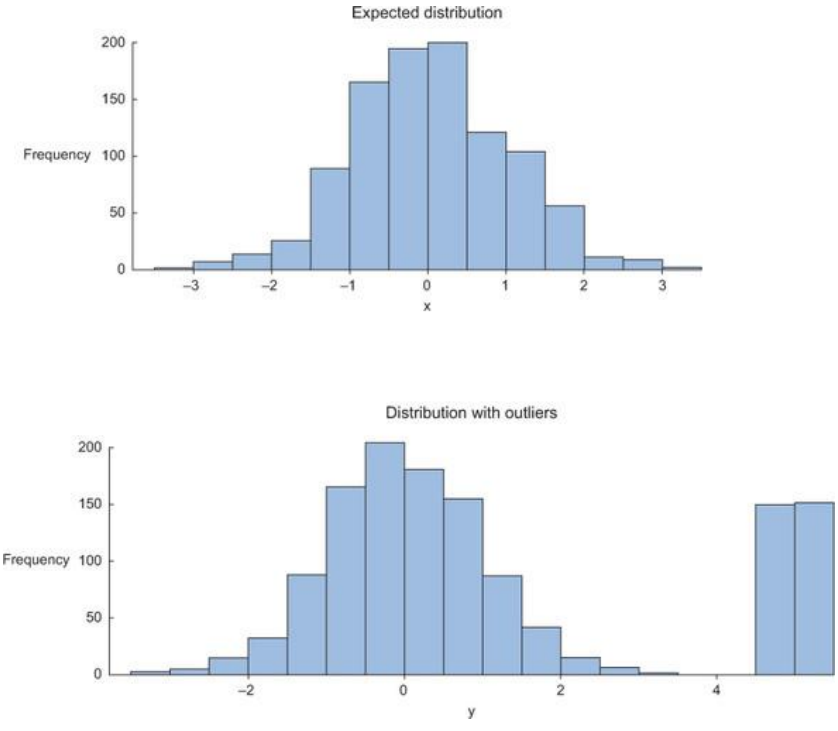
```
check = 0 <= age <= 120
```

copy

d) Outliers

An outlier is an observation that seems to be distant from other observations or, more specifically, one observation that follows a different logic or generative process than the other observations. The easiest way to find outliers is to use a plot or a table with the minimum and maximum values. An example is shown in figure 2.6.

Figure 2.6. Distribution plots are helpful in detecting outliers and helping you understand the variable.



e) Dealing with missing values

Missing values aren't necessarily wrong, but you still need to handle them separately; certain modeling techniques can't handle missing values. They might be an indicator that something went wrong in your data collection or that an error happened in the ETL process. Common techniques data scientists use are listed in table 2.4.

Table 2.4. An overview of techniques to handle missing data

Technique	Advantage	Disadvantage
Omit the values	Easy to perform	You lose the information from an observation
Set value to null	Easy to perform	Not every modeling technique and/or implementation can handle null values
Impute a static value such as 0 or the mean	Easy to perform You don't lose information from the other variables in the observation	Can lead to false estimations from a model
Impute a value from an estimated or theoretical distribution	Does not disturb the model as much	Harder to execute You make data assumptions

f) Deviations from a code book

Detecting errors in larger data sets against a code book or against standardized values can be done with the help of set operations. A code book is a description of your data, a form of metadata. It contains things such as the number of variables per observation, the number

of observations, and what each encoding within a variable means. (For instance “0” equals “negative”, “5” stands for “very positive”.) A code book also tells the type of data you’re looking at: is it hierarchical, graph, something else?

g) Different units of measurement

When integrating two data sets, you have to pay attention to their respective units of measurement. An example of this would be when you study the prices of gasoline in the world. To do this you gather data from different data providers. Data sets can contain prices per gallon and others can contain prices per liter. A simple conversion will do the trick in this case.

➢ 2. Combining data from different data sources

Your data comes from several different places, and in this substep we focus on integrating these different sources. Data varies in size, type, and structure, ranging from databases and Excel files to text documents.

We focus on data in table structures in this chapter for the sake of brevity. It’s easy to fill entire books on this topic alone, and we choose to focus on the data science process instead of presenting scenarios for every type of data. But keep in mind that other types of data sources exist, such as key-value stores, document stores, and so on, which we’ll handle in more appropriate places in the book.

The different ways of combining data

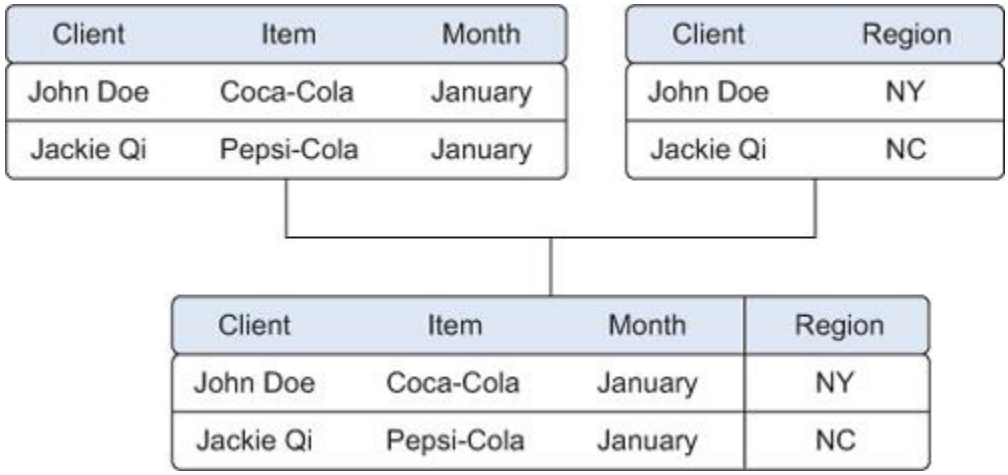
You can perform two operations to combine information from different data sets. The first operation is *joining*: enriching an observation from one table with information from another table. The second operation is *appending* or *stacking*: adding the observations of one table to those of another table.

When you combine data, you have the option to create a new physical table or a virtual table by creating a view. The advantage of a view is that it doesn't consume more disk space. Let's elaborate a bit on these methods.

Joining tables

Joining tables allows you to combine the information of one observation found in one table with the information that you find in another table. The focus is on enriching a single observation. Let's say that the first table contains information about the purchases of a customer and the other table contains information about the region where your customer lives. Joining the tables allows you to combine the information so that you can use it for your model, as shown in figure 2.7.

Figure 2.7. Joining two tables on the Item and Region keys



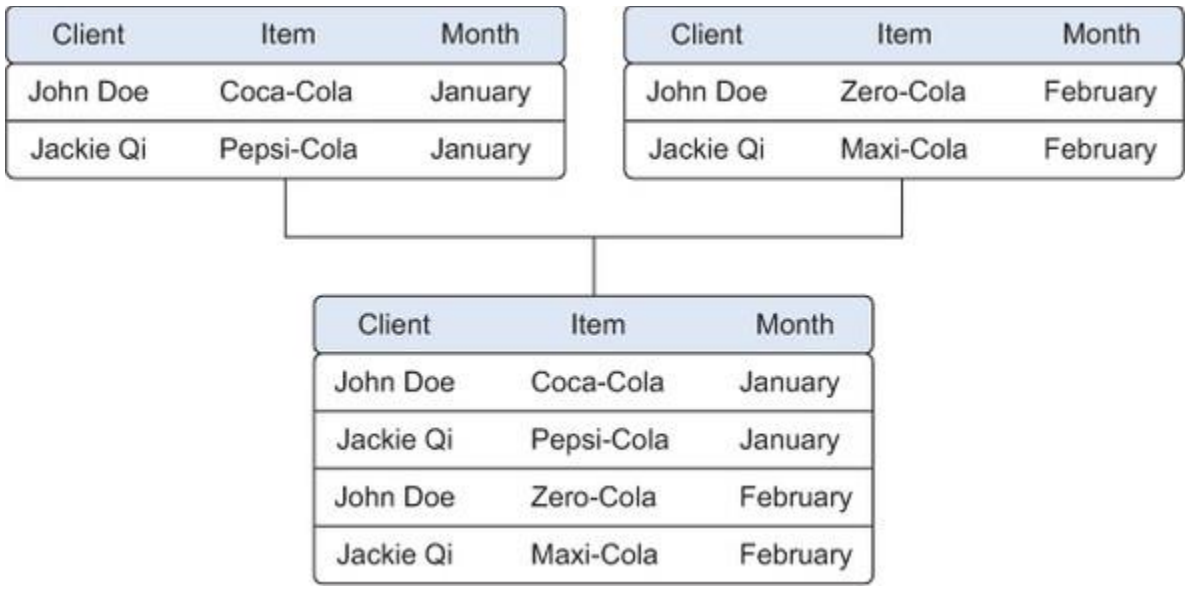
To join tables, you use variables that represent the same object in both tables, such as a date, a country name, or a Social Security number. These common fields are known as keys. When these keys also uniquely define the records in the table they are called *primary keys*. One table may have buying behavior and the other table may have demographic information on a person. In figure 2.7 both tables contain the client name, and this makes it easy to enrich the client expenditures with the region of the client. People who are acquainted with Excel will notice the similarity with using a lookup function.

The number of resulting rows in the output table depends on the exact join type that you use. We introduce the different types of joins later in the book.

Appending tables

Appending or stacking tables is effectively adding observations from one table to another table. Figure 2.8 shows an example of appending tables. One table contains the observations from the month January and the second table contains observations from the month February. The result of appending these tables is a larger one with the observations from January as well as February. The equivalent operation in set theory would be the union, and this is also the command in SQL, the common language of relational databases. Other set operators are also used in data science, such as set difference and intersection.

Figure 2.8. Appending data from tables is a common operation but requires an equal structure in the tables being appended.

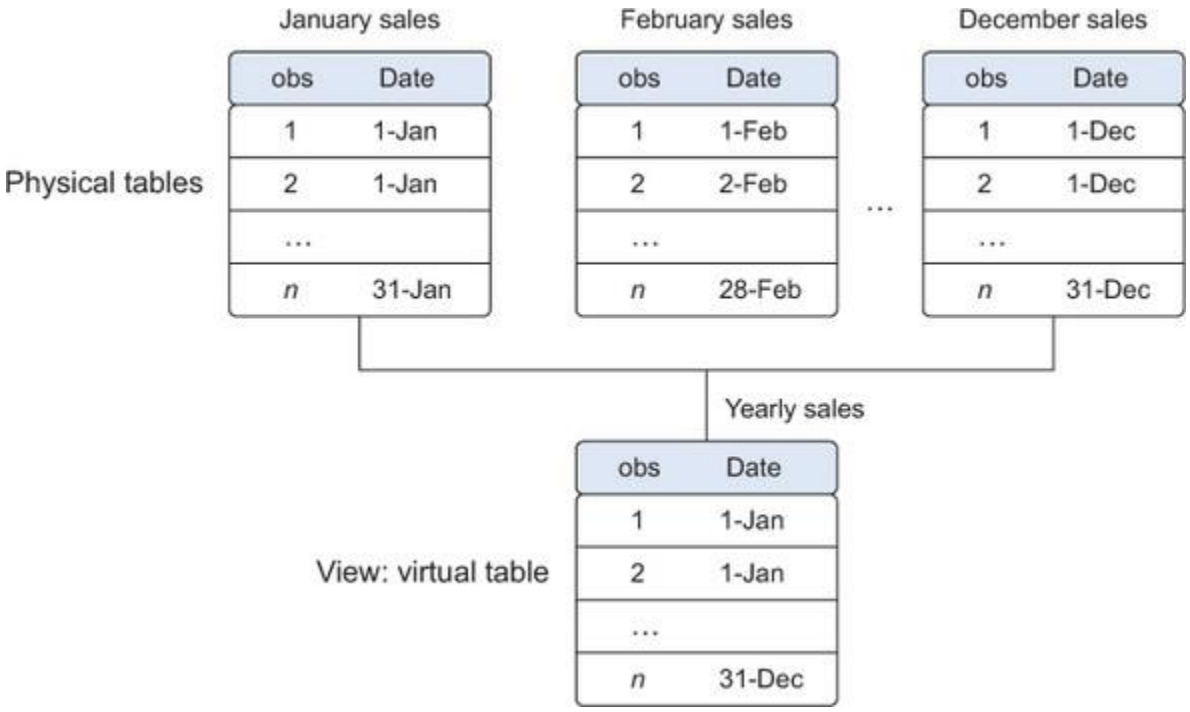


Using views to simulate data joins and appends

To avoid duplication of data, you virtually combine data with views. In the previous example we took the monthly data and combined it in a new physical table. The problem is that we duplicated the data and therefore needed more storage space. In the example

we're working with, that may not cause problems, but imagine that every table consists of terabytes of data; then it becomes problematic to duplicate the data. For this reason, the concept of a view was invented. A view behaves as if you're working on a table, but this table is nothing but a virtual layer that combines the tables for you. Figure 2.9 shows how the sales data from the different months is combined virtually into a yearly sales table instead of duplicating the data. Views do come with a drawback, however. While a table join is only performed once, the join that creates the view is recreated every time it's queried, using more processing power than a pre-calculated table would have.

Figure 2.9. A view helps you combine data without replication.



➢ 3 Transforming data

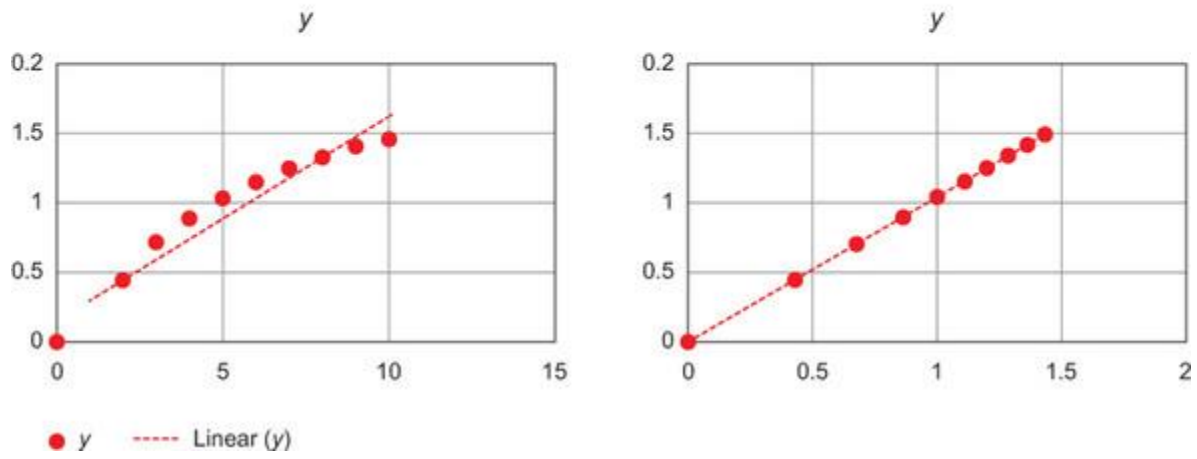
Certain models require their data to be in a certain shape. Now that you've cleansed and integrated the data, this is the next task you'll perform: transforming your data so it takes a suitable form for data modeling.

Transforming data

Relationships between an input variable and an output variable aren't always linear. Take, for instance, a relationship of the form $y = ae^{bx}$. Taking the log of the independent variables simplifies the estimation problem dramatically. Figure 2.11 shows how transforming the input variables greatly simplifies the estimation problem. Other times you might want to combine two variables into a new variable.

Figure 2.11. Transforming x to log x makes the relationship between x and y linear (right), compared with the non-log x (left).

x	1	2	3	4	5	6	7	8	9	10
log(x)	0.00	0.43	0.68	0.86	1.00	1.11	1.21	1.29	1.37	1.43
y	0.00	0.44	0.69	0.87	1.02	1.11	1.24	1.32	1.38	1.46



Reducing the number of variables

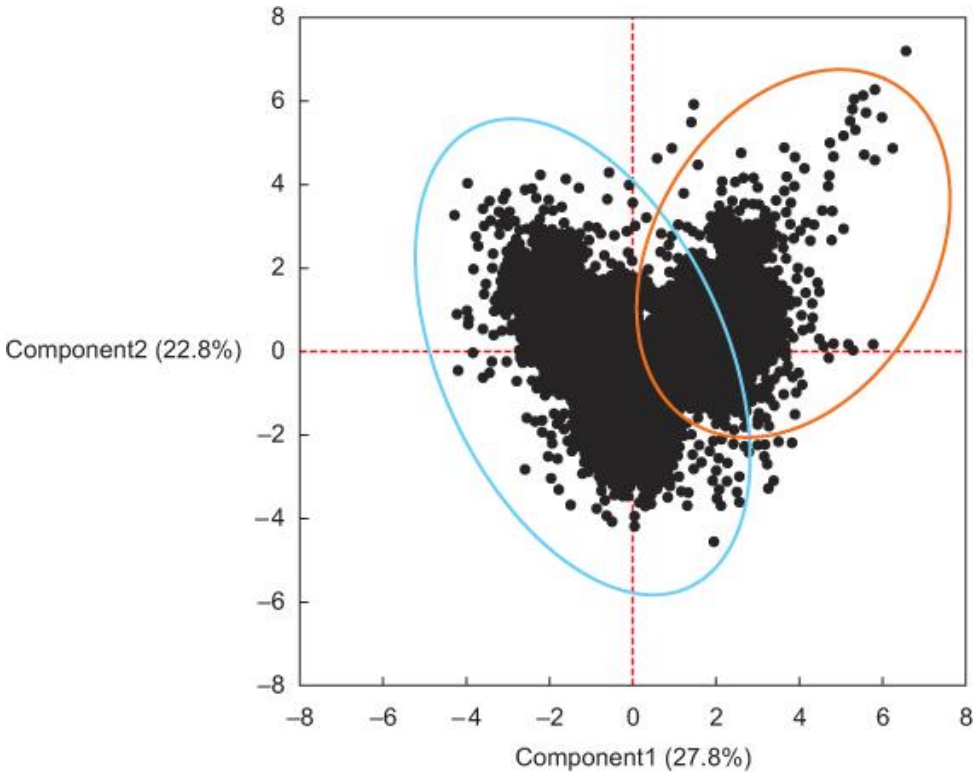
Sometimes you have too many variables and need to reduce the number because they don't add new information to the model. Having too many variables in your model makes the model difficult to handle, and certain techniques don't perform well when you overload them with too many input variables. For instance, all the techniques based on a Euclidean distance perform well only up to 10 variables.

EUCLIDEAN DISTANCE

Euclidean distance or "ordinary" distance is an extension to one of the first things anyone learns in mathematics about triangles (trigonometry): Pythagoras's leg theorem. If you

know the length of the two sides next to the 90° angle of a right-angled triangle you can easily derive the length of the remaining side (hypotenuse). The formula for this is hypotenuse = $\sqrt{(side1 + side2)^2}$. The Euclidean distance between two points in a two-dimensional plane is calculated using a similar formula: distance = $\sqrt{((x1 - x2)^2 + (y1 - y2)^2)}$. If you want to expand this distance calculation to more dimensions, add the coordinates of the point within those higher dimensions to the formula. For three dimensions we get distance = $\sqrt{((x1 - x2)^2 + (y1 - y2)^2 + (z1 - z2)^2)}$.

.

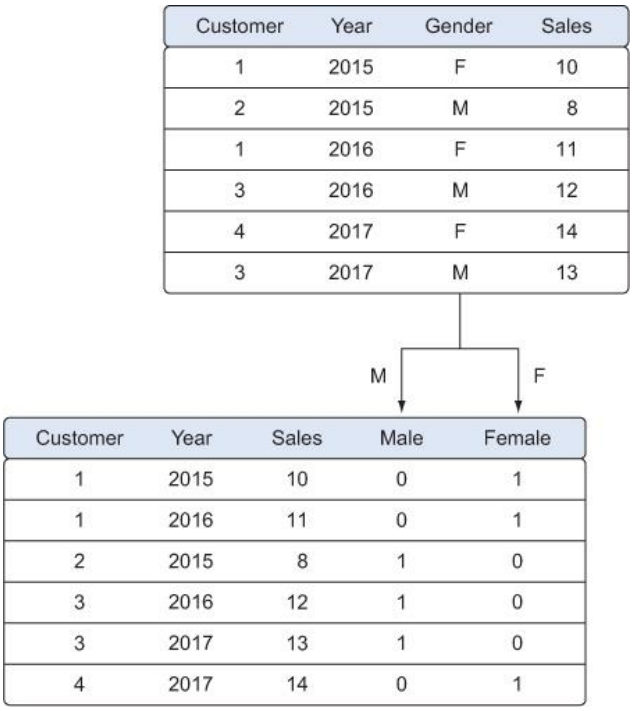


Turning variables into dummies

Variables can be turned into dummy variables (figure 2.13). *Dummy variables* can only take two values: true(1) or false(0). They're used to indicate the absence of a categorical effect that may explain the observation. In this case you'll make separate columns for the classes stored in one variable and indicate it with 1 if the class is present and 0 otherwise.

An example is turning one column named Weekdays into the columns Monday through Sunday. You use an indicator to show if the observation was on a Monday; you put 1 on Monday and 0 elsewhere. Turning variables into dummies is a technique that's used in modeling and is popular with, but not exclusive to, economists.

Figure 2.13. Turning variables into dummies is a data transformation that breaks a variable that has multiple classes into multiple variables, each having only two possible values: 0 or 1.

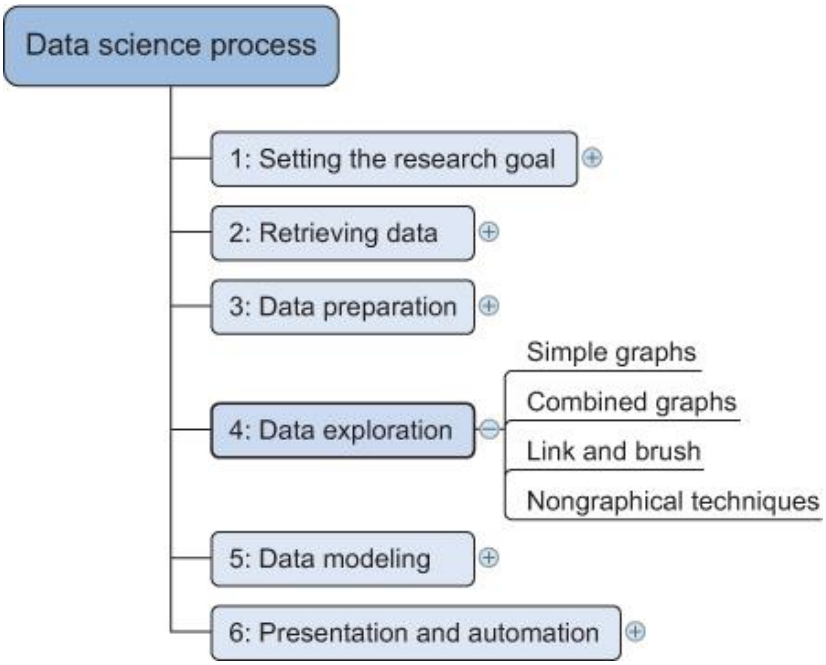


In this section we introduced the third step in the data science process—cleaning, transforming, and integrating data—which changes your raw data into usable input for the modeling phase. The next step in the data science process is to get a better understanding of the content of the data and the relationships between the variables and observations; we explore this in the next section.

➢ Step 4: Exploratory data analysis

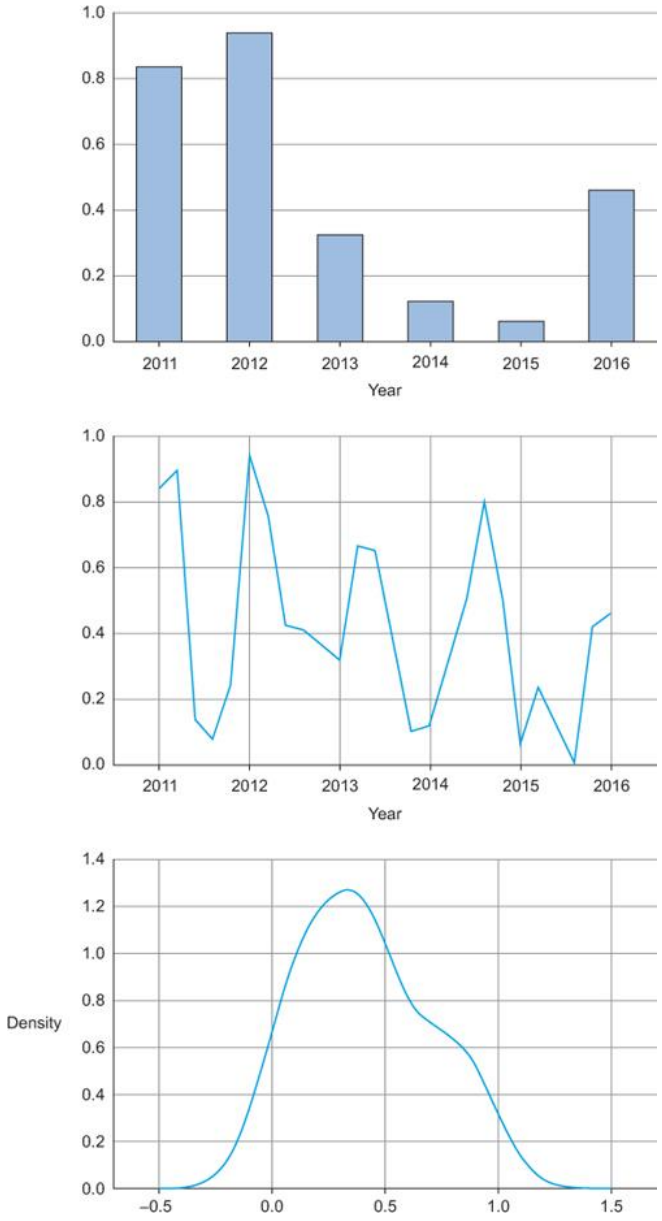
During exploratory data analysis you take a deep dive into the data (see figure 2.14). Information becomes much easier to grasp when shown in a picture, therefore you mainly use graphical techniques to gain an understanding of your data and the interactions between variables. This phase is about exploring data, so keeping your mind open and your eyes peeled is essential during the exploratory data analysis phase. The goal isn't to cleanse the data, but it's common that you'll still discover anomalies you missed before, forcing you to take a step back and fix them.

Figure 2.14. Step 4: Data exploration



The visualization techniques you use in this phase range from simple line graphs or histograms, as shown in figure 2.15, to more complex diagrams such as Sankey and network graphs. Sometimes it's useful to compose a composite graph from simple graphs to get even more insight into the data. Other times the graphs can be animated or made interactive to make it easier and, let's admit it, way more fun. An example of an interactive Sankey diagram can be found at http://bost.ocks.org/mike/sankey/.

Figure 2.15. From top to bottom, a bar chart, a line plot, and a distribution are some of the graphs used in exploratory analysis.



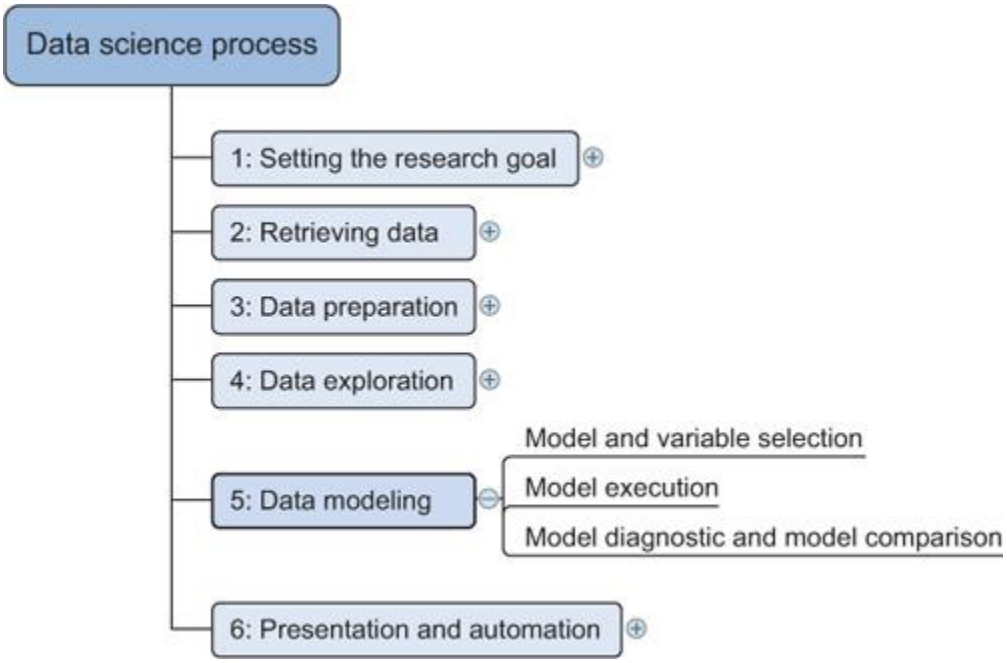
Mike Bostock has interactive examples of almost any type of graph. It's worth spending time on his website, though most of his examples are more useful for data presentation than data exploration.

Now that you've finished the data exploration phase and you've gained a good grasp of your data, it's time to move on to the next phase: building models.

- **Step 5: Build the models**

With clean data in place and a good understanding of the content, you're ready to build models with the goal of making better predictions, classifying objects, or gaining an understanding of the system that you're modeling. This phase is much more focused than the exploratory analysis step, because you know what you're looking for and what you want the outcome to be. Figure 2.21 shows the components of model building.

Figure 2.21. Step 5: Data modeling



The techniques you'll use now are borrowed from the field of machine learning, data mining, and/or statistics. In this chapter we only explore the tip of the iceberg of existing techniques, while chapter 3 introduces them properly. It's beyond the scope of this book to give you more than a conceptual introduction, but it's enough to get you started; 20% of the techniques will help you in 80% of the cases because techniques overlap in what they try to accomplish. They often achieve their goals in similar but slightly different ways.

Building a model is an iterative process. The way you build your model depends on whether you go with classic statistics or the somewhat more recent machine learning school, and the type of technique you want to use. Either way, most models consist of the following main steps:

1. Selection of a modeling technique and variables to enter in the model

2. Execution of the model

3. Diagnosis and model comparison

1. Model and variable selection

You'll need to select the variables you want to include in your model and a modeling technique. Your findings from the exploratory analysis should already give a fair idea of what variables will help you construct a good model. Many modeling techniques are available, and choosing the right model for a problem requires judgment on your part. You'll need to consider model performance and whether your project meets all the requirements to use your model, as well as other factors:

- Must the model be moved to a production environment and, if so, would it be easy to implement?
- How difficult is the maintenance on the model: how long will it remain relevant if left untouched?
- Does the model need to be easy to explain?

When the thinking is done, it's time for action.

2. Model execution

Once you've chosen a model you'll need to implement it in code.

Luckily, most programming languages, such as Python, already have libraries such as StatsModels or Scikit-learn. These packages use several of the most popular techniques. Coding a model is a nontrivial task in most cases, so having these libraries available can speed up the process. As you can see in the following code, it's fairly easy to use linear regression (2. figure 22) with StatsModels or Scikit-learn. Doing this yourself would require

much more effort even for the simple techniques. The following listing shows the execution of a linear prediction model.

We, however, created the target variable, based on the predictor by adding a bit of randomness. It shouldn't come as a surprise that this gives us a well-fitting model. The `results.summary()` outputs the table in figure 2.23. Mind you, the exact outcome depends on the random variables you got.

3. Model diagnostics and model comparison

You'll be building multiple models from which you then choose the best one based on multiple criteria. Working with a holdout sample helps you pick the best-performing model. A holdout sample is a part of the data you leave out of the model building so it can be used to evaluate the model afterward. The principle here is simple: the model should work on unseen data. You use only a fraction of your data to estimate the model and the other part, the holdout sample, is kept out of the equation. The model is then unleashed on the unseen data and error measures are calculated to evaluate it. Multiple error measures are available, and in figure 2.26 we show the general idea on comparing models. The error measure used in the example is the mean square error.

Figure 2.26. Formula for mean square error

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(\hat{Y}_i - Y_i)^2$$

Mean square error is a simple measure: check for every prediction how far it was from the truth, square this error, and add up the error of every prediction.

Figure 2.27 compares the performance of two models to predict the order size from the price. The first model is *size = 3 * price* and the second model is *size = 10*. To estimate the models, we use 800 randomly chosen observations out of 1,000 (or 80%), without showing

the other 20% of data to the model. Once the model is trained, we predict the values for the other 20% of the variables based on those for which we already know the true value, and calculate the model error with an error measure. Then we choose the model with the lowest error. In this example we chose model 1 because it has the lowest total error.

Figure 2.27. A holdout sample helps you compare models and ensures that you can generalize results to data that the model has not yet seen.

	n	Size	Price	Predicted model 1	Predicted model 2	Error model 1	Error model 2
	1	10	3				
	2	15	5				
	3	18	6				
	4	14	5				
	...	...					
80% train	800	9	3				
	801	12	4	12	10	0	2
	802	13	4	12	10	1	3
	...						
	999	21	7	21	10	0	11
20% test	1000	10	4	12	10	–2	0
				Total		5861	110225

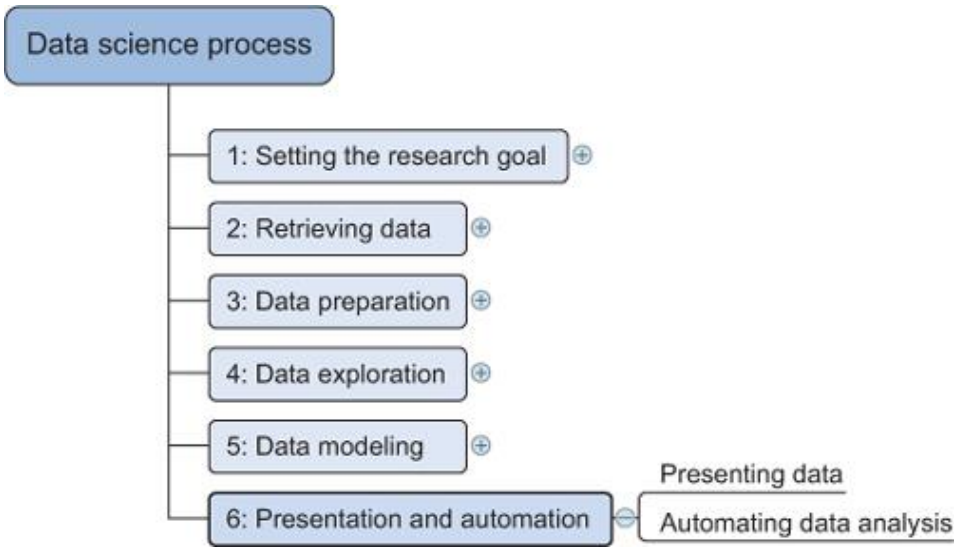
Many models make strong assumptions, such as independence of the inputs, and you have to verify that these assumptions are indeed met. This is called *model diagnostics*.

This section gave a short introduction to the steps required to build a valid model. Once you have a working model you're ready to go to the last step.

➢ Step 6: Presenting findings and building applications on top of them

After you've successfully analyzed the data and built a well-performing model, you're ready to present your findings to the world (figure 2.28). This is an exciting part; all your hours of hard work have paid off and you can explain what you found to the stakeholders.

Figure 2.28. Step 6: Presentation and automation



Sometimes people get so excited about your work that you'll need to repeat it over and over again because they value the predictions of your models or the insights that you produced. For this reason, you need to automate your models. This doesn't always mean that you have to redo all of your analysis all the time. Sometimes it's sufficient that you implement only the model scoring; other times you might build an application that automatically updates reports, Excel spreadsheets, or PowerPoint presentations. The last stage of the data science process is where your *soft skills* will be most useful, and yes, they're extremely important. In fact, we recommend you find dedicated books and other information on the subject and work through them, because why bother doing all this tough work if nobody listens to what you have to say?

If you've done this right, you now have a working model and satisfied stakeholders, so we can conclude this chapter here.

➢ Reading and Writing Data to and from R

Functions for Reading Data into R:

There are a few very useful functions for reading data into R.

1. **read.table()** and **read.csv()** are two popular functions used for reading tabular data into R.
2. **readLines()** is used for reading lines from a text file.

3. **source()** is a very useful function for reading in R code files from a another R program.
4. **dget()** function is also used for reading in R code files.
5. **load()** function is used for reading in saved workspaces
6. **unserialize()** function is used for reading single R objects in binary format.

Functions for Writing Data to Files:

There are similar functions for writing data to files

1. **write.table()** is used for writing tabular data to text files (i.e. CSV).
2. **writeLines()** function is useful for writing character data line-by-line to a file or connection.
3. **dump()** is a function for dumping a textual representation of multiple R objects.
4. **dput()** function is used for outputting a textual representation of an R object.
5. **save()** is useful for saving an arbitrary number of R objects in binary format to a file.
6. **serialize()** is used for converting an R object into a binary format for outputting to a connection (or
file).

Reading Data Files with read.table():

The read.table() function is one of the most commonly used functions for reading data in R. TO get the help file for read.table() just type **?read.table** in R console.

The read.table() function has a few important arguments:

- **file**, the name of a file, or a connection
- **header**, logical indicating if the file has a header line
- **sep**, a string indicating how the columns are separated
- **colClasses**, a character vector indicating the class of each column in the dataset
- **nrows**, the number of rows in the dataset. By default read.table() reads an entire file.
- **comment.char**, a character string indicating the comment character. This defalts to "#". If there are no commented lines in your file, it's worth setting this to be the empty string "".
- **skip**, the number of lines to skip from the beginning
- **stringsAsFactors**, should character variables be coded as factors? This defaults to TRUE because back in the old days, if you had data that were stored as strings, it was because those strings represented levels of a categorical variable. Now we

have lots of data that is text data and they don't always represent categorical variables. So you may want to set this to be FALSE in those cases. If you always want this to be FALSE, you can set a global option via options(stringsAsFactors = FALSE). I've never seen so much heat generated on discussion forums about an R function argument than the stringsAsFactors argument.

Check the following example how to work with read.table() in r. For this example a data set called **wine data set** will be used. You can download the data set by clicking **here**. The data set was originally taken from UCI Repository. You can get more details about the data set from **here.**

Download the Wine Data set

```
w<-read.table("https://makemeanalyst.com/wp-
content/uploads/2017/05/wine.txt",sep=",",header = TRUE)
head(w)
View(w)
```

Writing Data Files with write.table():

To write a R object into a file check the following code.

```
 write.table(w,"E:/MakeMeAnalyst/wine.txt")  #Give your own path
here.
```

readLines() and writeLines() function in R:

readLines() function is mainly used for reading lines from a text file and writeLines() function is useful for writing character data line-by-line to a file or connection. Check the following example to deal with readLines() and writeLines(). First, download the sample text from **here** and then read it into R.

Download the Sample Text

```
con <- file("https://makemeanalyst.com/wp-
content/uploads/2017/05/Sample.txt", "r")
w<-readLines(con)
close(con)
```

Output:

```
> w[1]
[1] "This is a sample text file."
> w[2]
[1] "Read this file using readLines() function."
> w[3]
[1] "And you can wrtie a file using writeLines() function."
```

dput() and dget() Function in R:

You can create a more descriptive representation of an R object by using the **dput()** or **dump()** functions. Unlike writing out a table or CSV file, dump() and dput() preserve the metadata, so that another user doesn't have to specify it all over again. For example, we can preserve the class of each column of a table or the levels of a factor variable.

```
# Create a data frame
x <- data.frame(Name = "Mr. A", Gender = "Male", Age=35)
#Print 'dput' output to your R console
dput(x)
#Write the 'dput' output to a file
dput(x, file = "F://w.R")
# Now read in 'dput' output from the file
y <- dget("F:/w.R")
y
```

dump() Function in R:

You can dump() R objects to a file by passing its names.

```
x<-1:10
d <- data.frame(Name = "Mr. A", Gender = "Male", Age=35)
dump(c("x", "d"), file = "F://dump_data.R")

rm(x, d) #After dumping just remove the variables from
environment.
```

source() Function in R:

The inverse of dump() is **source()** function. Now you can import that dump_data.R into R using following code.

```
source("F://dump_data.R")
x
d
str(d)
```

Output:

```
> x
[1] 1 2 3 4 5 6 7 8 9 10
> d
Name Gender Age
1 Mr. A Male 35
> str(d)
'data.frame': 1 obs. of 3 variables:
$ Name : Factor w/ 1 level "Mr. A": 1
$ Gender: Factor w/ 1 level "Male": 1
$ Mobile: num 35
```

➢ Using the readr Package

The readr package is recently developed by Hadley Wickham to deal with reading in large flat files quickly. The package provides replacements for functions like read.table() and read.csv(). The analogous functions in readr are read_table() and read_csv(). These functions are often *much* faster than their base R analogues and provide a few other nice features such as progress meters.

For the most part, you can read use read_table() and read_csv() pretty much anywhere you might use read.table() and read.csv(). In addition, if there are non-fatal problems that occur while reading in the data, you will get a warning and the returned data frame will have some information about which rows/observations triggered the warning. This can be very helpful for "debugging" problems with your data before you get neck deep in data analysis.

The importance of the read_csv function is perhaps better understood from an historical perspective. R's built in read.csv function similarly reads CSV files, but the read_csv function in readr builds on that by removing some of the quirks and "gotchas" of read.csv as well as dramatically optimizing the speed with which it can read data into R. The read_csv function also adds some nice user-oriented features like a progress meter and a compact method for specifying column types.

A typical call to read_csv will look as follows.

```
> library(readr)

> teams <- read_csv("data/team_standings.csv")

Rows: 32 Columns: 2

> teams

# A tibble: 32 × 2

  Standing Team

     <dbl> <chr>

 1        1 Spain

 2        2 Netherlands

 3        3 Germany

 4        4 Uruguay

 5        5 Argentina

 6        6 Brazil

 7        7 Ghana

 8        8 Paraguay

 9        9 Japan

10       10 Chile

# ... with 22 more rows
```

By default, read_csv will open a CSV file and read it in line-by-line. It will also (by default), read in the first few rows of the table in order to figure out the type of each column (i.e. integer, character, etc.). From the read_csv help page:

If 'NULL', all column types will be imputed from the first 1000 rows on the input. This is convenient (and fast), but not robust. If the imputation fails, you'll need to supply the correct types yourself.

You can specify the type of each column with the col_types argument.

In general, it's a good idea to specify the column types explicitly. This rules out any possible guessing errors on the part of read_csv. Also, specifying the column types

explicitly provides a useful safety check in case anything about the dataset should change without you knowing about it.

```
> teams <- read_csv("data/team_standings.csv", col_types = "cc")
```

Note that the `col_types` argument accepts a compact representation.
Here "cc" indicates that the first column is `character` and the second column is `character` (there are only two columns). Using the `col_types` argument is useful because often it is not easy to automatically figure out the type of a column by looking at a few rows (especially if a column has many missing values).

The `read_csv` function will also read compressed files automatically. There is no need to decompress the file first or use the `gzfile` connection function. The following call reads a gzip-compressed CSV file containing download logs from the RStudio CRAN mirror.

```
> logs <- read_csv("data/2016-07-19.csv.bz2", n_max = 10)

Rows: 10 Columns: 10
```

UNIT-III

➢ What is the Role of Machine Learning in Data Science?

Using modern techniques and tools, **Data science deals with a tremendous amount of data to find different and unseen patterns, derive information, and make business decisions.** Data science, to build models, uses complex machine learning algorithms.

Data science combines multiple fields such as scientific methods, statistics, data analysis, and artificial intelligence to extract the exact value from data. Data scientists and data engineers combine a range of skills to analyze and collect data from the web and other sources such as customers and smartphones to derive actionable insights

You are investing in ML like never before and hiring more data scientists and machine learning engineers. However, there is a lack of clarity on the role of machine learning and its place in the life cycle of a data science project. Here's an attempt to resolve this uncertainty.
Nowadays, many organizations and industries stress using data to improve their products and services. If we talk about just data science, then it is only data analysis using MLOps machine learning. **Both machine learning and data science have to go hand in hand.** Engineers have to use ML and data science prominently to make better and more appropriate decisions.

So, this article will introduce you to machine learning and data science, the role of ML in data science, and how they are different from each other yet work together.

➢ What is Machine Learning (ML)?

In simple words, you can explain **machine learning as a type of artificial intelligence (AI) or a subset of AI which allows any software applications or apps to be more precise and accurate for finding and predicting outcomes.**

Machine learning algorithms use historical data to predict new outcomes or output values. There are different use cases for machine learning like fraud detection, malware threat detection, recommendation engines, spam filtering, healthcare, and many others.

Machine Learning Importants

For any business, industry, and organization to run data as a primary record or lifeblood of it, and along with evolution, there is also a rise in demand and importance. This aspect is why data engineers and data scientists need machine learning.

With the help of this technology, you can analyze a large amount of data and calculate risk factors in no time. **Machine Learning has changed the way of data engineering in terms of data handling, extraction, and interpretation.**

➢ Data Science vs. Machine Learning

DATA SCIENCE	MACHINE LEARNING
It is a field that processes and extracts data from semi-structured data and structured data.	It is a field that offers systems the ability to learn without being programmed explicitly.
It needs an entire analytics universe.	It combines machine and data science.
The branch deals with data.	Machines utilize data science for learning data.
Data science operations include data gathering, manipulation, cleaning, etc.	There are three types of machine learning: unsupervised, supervised, and reinforcement.
It is a broad term that takes care of data processing and focuses on	ML only focuses on algorithm statistics.

DATA SCIENCE	MACHINE LEARNING
algorithms.	
Example: Netflix using data science is an example of this technology. With the advanced data and analytics obtained from applying data science, Netflix can provide users personalized recommendations on movies and shows. It can also predict the original content's popularity with trailers and thumbnail images.	**Example**: Facebook using machine learning is an example of this technology. Using machine learning, Facebook can produce the estimated action rate and the ad quality score which is used for the total equation. ML features such as facial recognition, textual analysis, targeted advertising, language translation and news feed are also used in many real-case scenarios.

➢ The Role of Machine Learning in Data Science

Data science is all about uncovering findings from raw data. This can be done by exploring data at a very granular level and understanding the complex behaviors and trends. This is where machine learning comes into play.

But, before analyzing data, you need to understand the business requirements clearly to apply machine learning.

machine learning

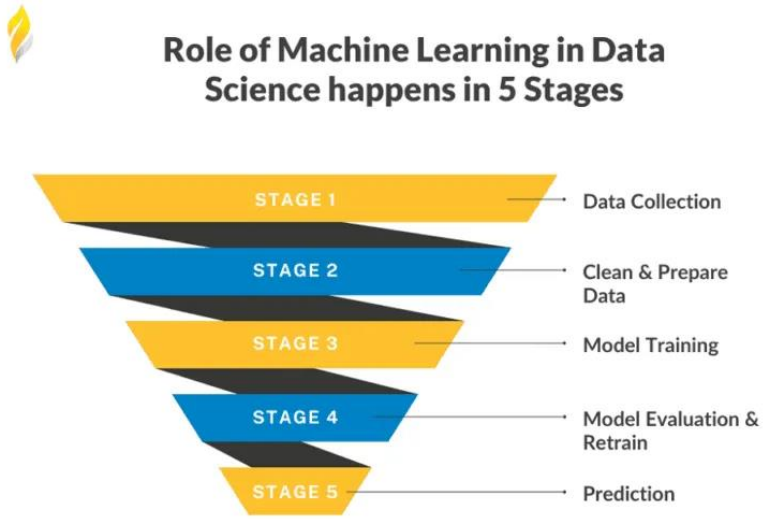
In simple terms, machine learning technology helps analyze and automate large chunks of data and make predictions in real-time without involving people.

We use machine learning algorithms in data science when we want to make accurate estimates about a given set of data—for instance, if we need to predict whether a patient has cancer-based on the results of their bloodwork. We can do this by feeding the algorithm a large set of examples: patients that

did or didn't have cancer and the lab results for each patient. The algorithm will learn from these examples until it can accurately predict whether a patient has cancer-based on their lab results.

That said, **the role of machine learning in data science happens in 5 stages**:

Watch this video from our data science expert, Sanjeeya Velayutham, to learn what exactly is machine learning and how it fits into the bigger picture of data science.



First, let's understand data collection.

Data collection is the first step of the machine learning process. As per the business problem, machine learning helps collect and analyze structured, unstructured, and semi-structured data from any database across systems. It can be a CSV file, pdf, document, image, or handwritten form.

The second step is data preparation and cleansing.

Machine learning technology helps analyze the data and prepare features related to the business problem in data preparation. ML systems, when clearly defined, understand the features and relationships between each other.

Note that features are the backbone of machine learning and any data science project.

Once data preparation is complete, we need to cleanse the data because data in the real world is quite dirty and corrupted with inconsistencies, noise, incomplete information, and missing values.

With the help of machine learning, we can find out the missing data and do data imputation, encode the categorical columns, remove the outliers, duplicate rows, and null values much faster in an automated fashion.

The next step is model training.

Model training depends on both the quality of the training data and the choice of the machine learning algorithm. An ML algorithm is selected based on end-user needs.

Additionally, you need to consider the model algorithm complexity, performance, interpretability, computer resource requirements, and speed for better model accuracy.

Once the right machine learning algorithm is selected, the training data set is divided into two parts for training and testing. This is done to determine the bias and variance of the ML model.

As a result of model training, you will achieve a working model that can be further validated, tested, and deployed.

The next step is evaluate your model

Once model training is completed, there are different metrics to evaluate your model. Remember, choosing a metric completely depends on the model type and implementation plan. Although the model has been trained and

assessed, this does not mean it is ready to solve your business problems. Any model can be fine-tuned further for better accuracy by further tuning the parameters.

The final and most crucial stage of a data science project is model prediction.

Whenever we discuss model prediction, it's vital to understand prediction errors (bias and variance).

Gaining a proper understanding of these errors would help you build accurate models and avoid the mistake of overfitting and underfitting the model.

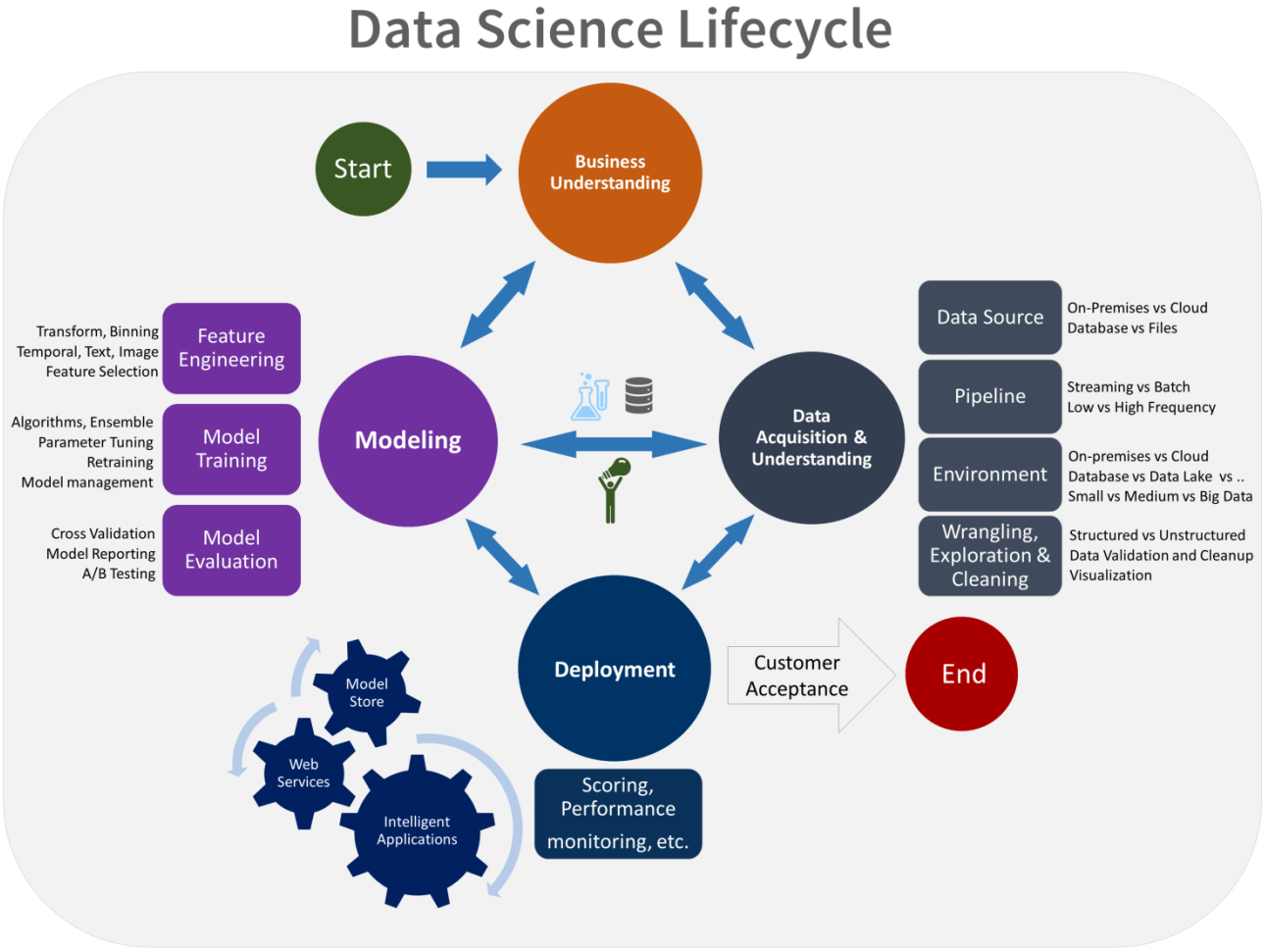
You can further minimize the prediction errors by finding a good balance between bias and variance for a successful data science project.

Overshadowing other data science aspects, machine learning (ML) and artificial intelligence (AI) have dominated the industry nowadays in the following ways:

1. Machine learning analyzes and examines large chunks of data automatically.
2. It automates the data analysis process and makes predictions in real-time without any human involvement.
3. You can further build and train the data model to make real-time predictions. This point is where you use machine learning algorithms in the data science lifecycle.

- **Major Steps of Machine Learning in Data Science Life Cycle**

(Where it is used in data science)



The diagram above is the pictorial representation of how you can train the data model and acquire data in making business decisions. Let us learn how to execute it:

Getting Data → Preparing Data → Training Model → Testing Data → Improve

1. **Data Collection**: It is known to be the foundation or primary step. It is essential to collect relevant and reliable data that impacts the outcomes.
2. **Data Preparation**: The overall first step of data preparation is data cleaning. It is an essential step for preparing the data. This step ensures that data is erroneous and corrupt data point-free.
3. **Model Training**: In this step, learning of data starts. You can use training to predict the output data value. You must repeat this training of the model step and do it, again and again, to improve and get more accurate predictions.

4. **Data Testing**: Once you complete the above steps, you can do the evaluation. The evaluation makes sure that the data set that we get will perform in real-life applications.
5. **Predictions**: Once you train and evaluate the model, it does not mean that the dataset is perfect and ready to be deployed. You have to further improve it by tuning. This stage is the final step of machine learning. Here the machine answers each of your questions by its learning.

➢ Applications of Machine Learning

These algorithms help in building intelligent systems that can learn from their past experiences and historical data to give accurate results. Many industries are thus applying ML solutions to their business problems, or to create new and better products and services. Healthcare, defense, financial services, marketing, and security services, among others, make use of ML.

1. Facial recognition/Image recognition

The most common application is Facial Recognition, and the simplest example of this application is the iPhone. There are a lot of use-cases of facial recognition, mostly for security purposes like identifying criminals, searching for missing individuals, aid forensic investigations, etc. Intelligent marketing, diagnose diseases, track attendance in schools, are some other uses.

2. Automatic Speech Recognition

Abbreviated as ASR, automatic speech recognition is used to convert speech into digital text. Its applications lie in authenticating users based on their voice and performing tasks based on the human voice inputs. Speech patterns and vocabulary are fed into the system to train the model. Presently ASR systems find a wide variety of applications in the following domains:

- Medical Assistance
- Industrial Robotics
- Forensic and Law enforcement
- Defense & Aviation

- Telecommunications Industry
- Home Automation and Security Access Control
- I.T. and Consumer Electronics

3. Financial Services

Machine learning has many use cases in Financial Services. Machine Learning algorithms prove to be excellent at detecting frauds by monitoring activities of each user and assess that if an attempted activity is typical of that user or not. Financial monitoring to detect money laundering activities is also a critical security use case.

It also helps in making better trading decisions with the help of algorithms that can analyze thousands of data sources simultaneously. Credit scoring and underwriting are some of the other applications. The most common application in our day to day activities is the virtual personal assistants like Siri and Alexa.

4. Marketing and Sales

It is improving lead scoring algorithms by including various parameters such as website visits, emails opened, downloads, and clicks to score each lead. It also helps businesses to improve their dynamic pricing models by using regression techniques to make predictions.

Sentiment Analysis is another essential application to gauge consumer response to a specific product or a marketing initiative. Machine Learning for Computer Vision helps brands identify their products in images and videos online. These brands also use computer vision to measure the mentions that miss out on any relevant text. Chatbots are also becoming more responsive and intelligent.

5. Healthcare

A vital application is in the diagnosis of diseases and ailments, which are otherwise difficult to diagnose. Radiotherapy is also becoming better.

Early-stage drug discovery is another crucial application which involves technologies such as precision medicine and next-generation sequencing. Clinical trials cost a lot of time and money to complete and deliver results. Applying ML based predictive analytics could improve on these factors and give better results.

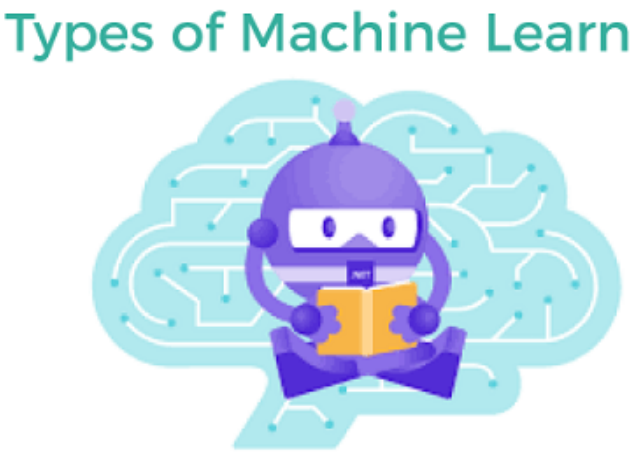
These technologies are also critical to make outbreak predictions. Scientists around the world are using ML technologies to predict epidemic outbreaks.

6. Recommendation Systems

Many businesses today use recommendation systems to effectively communicate with the users on their site. It can recommend relevant products, movies, web-series, songs, and much more. Most prominent use-cases of recommendation systems are e-commerce sites like Amazon, Flipkart, and many others, along with Spotify, Netflix, and other web-streaming channels.

➢ Types of Machine Learning

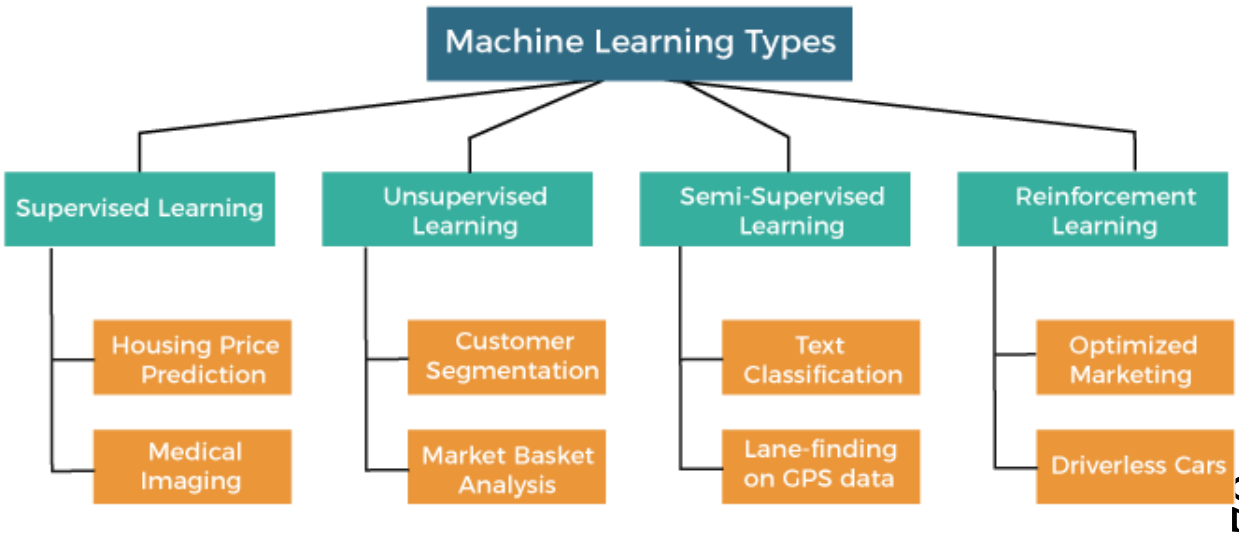
Machine learning is a subset of AI, which enables the machine to automatically learn from data, improve performance from past experiences, and make predictions. Machine learning contains a set of algorithms that work on a huge amount of data. Data is fed to these algorithms to train them, and on the basis of training, they build the model & perform a specific task.



These ML algorithms help to solve different business problems like Regression, Classification, Forecasting, Clustering, and Associations, etc.

Based on the methods and way of learning, machine learning is divided into mainly four types, which are:

1. Supervised Machine Learning
2. Unsupervised Machine Learning
3. Semi-Supervised Machine Learning
4. Reinforcement Learning



In this topic, we will provide a detailed description of the types of Machine Learning along with their respective algorithms:

Just have a look around you—we are using face detection algorithms to unlock phones and Youtube or Netflix recommender systems to suggest us content that's most likely to engage us (and make us binge-watch it).

But how do these systems work?

Well, I'm glad you asked because this article will help you understand key differences between two primary Machine Learning approaches that are the backbone of those systems: Supervised and Unsupervised Learning.

On the most basic level, the answer is simple—one of them uses labeled data to predict outcomes, while the other does not.

However—

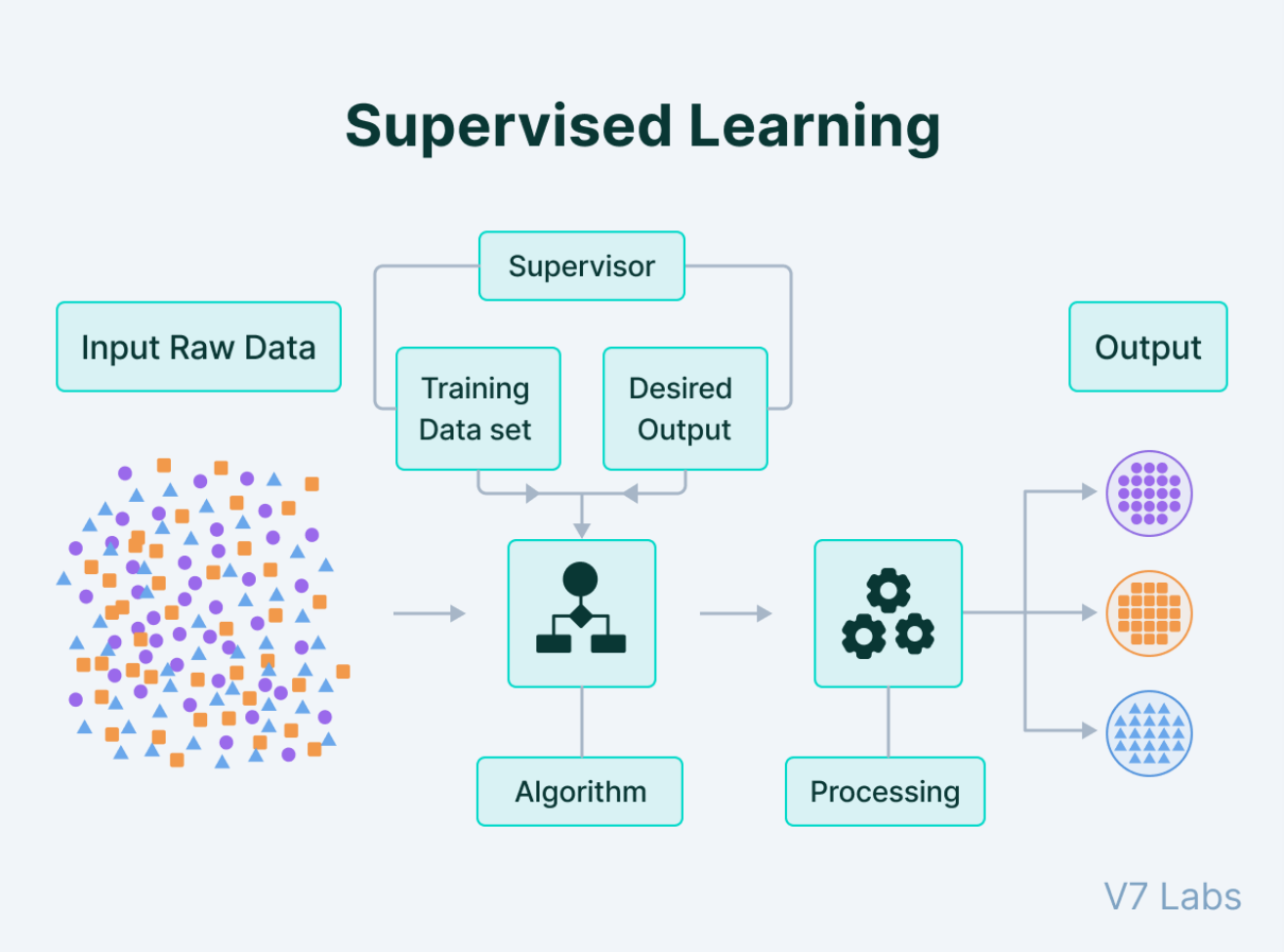
There's a bunch of nuances that you should know about because they determine which approach is more suitable for your use case.

What is Supervised Learning?

Supervised Learning is the machine learning approach defined by its use of labeled datasets to train algorithms to classify data and predict outcomes.

The labeled dataset has output tagged corresponding to input data for the machine to understand what to search for in the unseen data.

Here's how it looks in practice.



Supervised Learning process

Supervised Machine Learning Methods

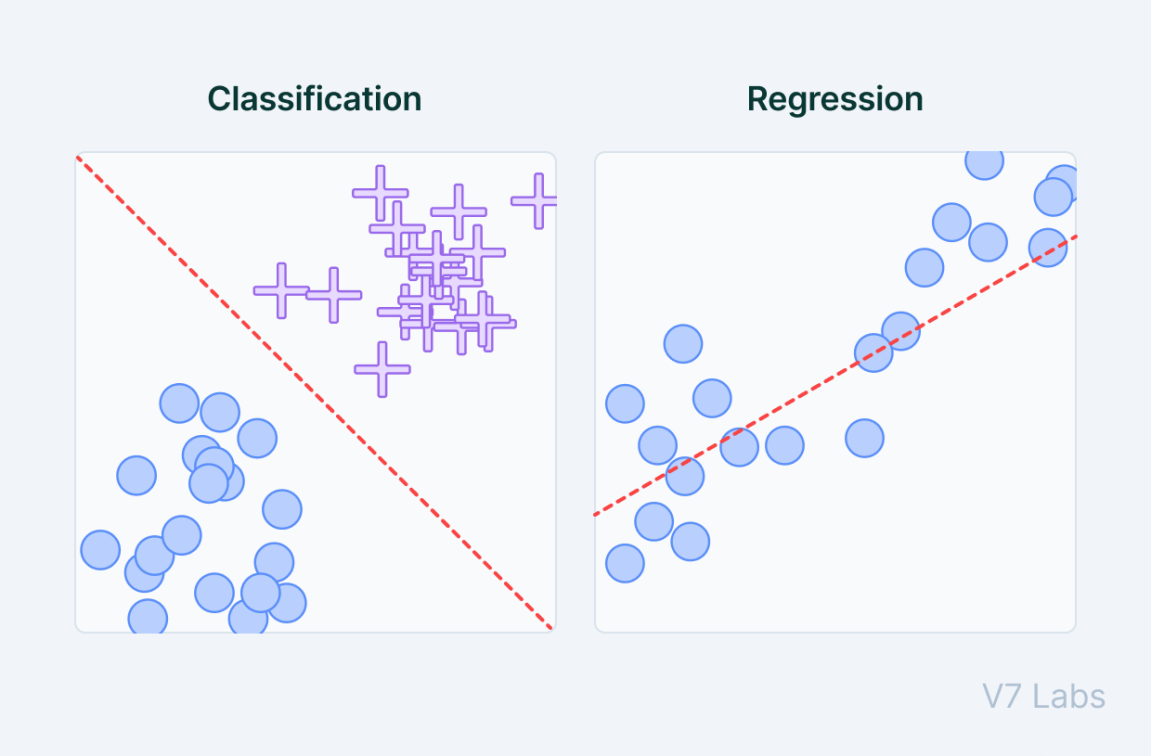
There are two main areas where supervised machine learning comes in handy: classification problems and regression problems.

Classification

Classification refers to taking an input value and mapping it to a discrete value. In classification problems, our output typically consists of classes or categories. This could be things like trying to predict what objects are present in an image (a cat/ a dog) or whether it is going to rain today or not.

Regression

Regression is related to continuous data (value functions). In Regression, the predicted output values are real numbers. It deals with problems such as predicting the price of a house or the trend in the stock price at a given time, etc.



Some of the most common algorithms in Supervised Learning include Support Vector Machines (SVM), Logistic Regression, Naive Bayes, Neural Networks, K-nearest neighbor (KNN), and Random Forest.

Supervised Machine Learning Applications

Now, let's have a look at some of the popular applications of Supervised Learning:

- Predictive analytics (house prices, stock exchange prices, etc.)
- Text recognition
- Spam detection
- Customer sentiment analysis
- Object detection (e.g. face detection)

Pro tip: **Refresh your knowledge by revisiting The Ultimate Guide to Object Detection.**

What is Unsupervised Learning?

Unsupervised Learning is a type of machine learning in which the algorithms are provided with data that does not contain any labels or explicit instructions on what to do with it. The goal is for the learning algorithm to find structure in the input data on its own.

To put it simply—Unsupervised Learning is a kind of self-learning where the algorithm can find previously hidden patterns in the unlabeled datasets and give the required output without any interference.

Identifying these hidden patterns helps in clustering, association, and detection of anomalies and errors in data.

Advantages and Disadvantages of Supervised Learning

Advantages:

- Since supervised learning work with the labelled dataset so we can have an exact idea about the classes of objects.
- These algorithms are helpful in predicting the output on the basis of prior experience.

Disadvantages:

- These algorithms are not able to solve complex tasks.
- It may predict the wrong output if the test data is different from the training data.
- It requires lots of computational time to train the algorithm.

Unsupervised Machine Learning Methods

Unsupervised Learning models can perform more complex tasks than Supervised Learning models, but they are also more unpredictable. Here are the main tasks that utilize this approach.

Clustering

Clustering is the type of Unsupervised Learning where we find hidden patterns in the data based on their similarities or differences. These patterns can relate to the shape, size, or color and are used to group data items or create clusters.

There are several types of clustering algorithms, such as exclusive, overlapping, hierarchical, and probabilistic.

Association

Association is the kind of Unsupervised Learning where we can find the relationship of one data item to another data item. We can then use those dependencies and map them in a way that benefits us—e.g., understanding consumers' habits regarding our products can help us develop better cross-selling strategies.

The association rule is used to find the probability of co-occurrence of items in a collection. These techniques are often utilized in customer behavior analysis in e-commerce websites and OTT platforms.

Dimensionality reduction

As the name suggests, the algorithm works to reduce the dimensions of the data. It is used for feature extraction.

Extracting the important features from the dataset is an essential aspect of machine learning algorithms. This helps reduce the number of random variables in the dataset by filtering irrelevant features.

Finally, here's a nice visual recap of everything we've covered so far (plus the Reinforcement Learning).

Advantages and Disadvantages of Unsupervised Learning Algorithm

Advantages:

- These algorithms can be used for complicated tasks compared to the supervised ones because these algorithms work on the unlabeled dataset.
- Unsupervised algorithms are preferable for various tasks as getting the unlabeled dataset is easier as compared to the labelled dataset.

Disadvantages:

- The output of an unsupervised algorithm can be less accurate as the dataset is not labelled, and algorithms are not trained with the exact output in prior.
- Working with Unsupervised learning is more difficult as it works with the unlabelled dataset that does not map with the output.

➢ Semi-Supervised Learning

Semi-Supervised learning is a type of Machine Learning algorithm that lies between Supervised and Unsupervised machine learning. It represents the intermediate ground between Supervised (With Labelled training data) and Unsupervised learning (with no labelled training data) algorithms and uses the combination of labelled and unlabeled datasets during the training period.

Although Semi-supervised learning is the middle ground between supervised and unsupervised learning and operates on the data that consists of a few labels, it mostly consists of unlabeled data. As labels are costly, but for corporate purposes, they may have few labels. It is completely different from supervised and unsupervised learning as they are based on the presence & absence of labels.

To overcome the drawbacks of supervised learning and unsupervised learning algorithms, the concept of Semi-supervised learning is introduced. The main aim of semi-supervised learning is to effectively use all the available data, rather than only labelled data like in supervised learning. Initially, similar data is clustered along with an unsupervised learning algorithm, and further, it helps to label the unlabeled data into labelled data. It is because labelled data is a comparatively more expensive acquisition than unlabeled data.

UNIT-IV

➢ 4.Handling large data on a single computer

This chapter covers

- Working with large data sets on a single computer
- Working with Python libraries suitable for larger data sets
- Understanding the importance of choosing correct algorithms and data structures
- Understanding how you can adapt algorithms to work inside databases

What if you had so much data that it seems to outgrow you, and your techniques no longer seem to suffice? What do you do, surrender or adapt?

Luckily you chose to adapt, because you're still reading. This chapter introduces you to techniques and tools to handle larger data sets that are still manageable by a single computer if you adopt the right techniques.

This chapter gives you the tools to perform the classifications and regressions when the data no longer fits into the RAM (random access memory) of your computer, whereas chapter 3 focused on in-memory data sets. Chapter 5 will go a step further and teach you how to deal with data sets that require multiple computers to be processed. When we refer to *large data* in this chapter we mean data that causes problems to work with in terms of memory or speed but can still be handled by a single computer.

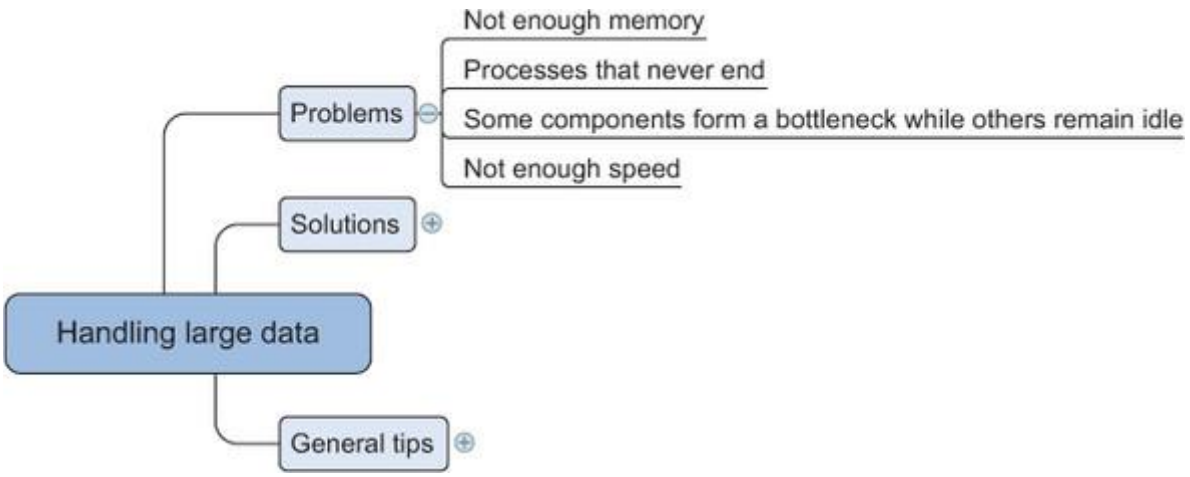
We start this chapter with an overview of the problems you face when handling large data sets. Then we offer three types of solutions to overcome these problems: adapt your algorithms, choose the right data structures, and pick the right tools. Data scientists aren't the only ones who have to deal with large data volumes, so you can apply general best practices to tackle the large data problem. Finally, we apply this knowledge to two case studies. The first case shows you how to detect malicious

URLs, and the second case demonstrates how to build a recommender engine inside a database.

➢ 4.1. The problems you face when handling large data

A large volume of data poses new challenges, such as overloaded memory and algorithms that never stop running. It forces you to adapt and expand your repertoire of techniques. But even when you can perform your analysis, you should take care of issues such as I/O (input/output) and CPU starvation, because these can cause speed issues. Figure 4.1 shows a mind map that will gradually unfold as we go through the steps: problems, solutions, and tips.

Figure 4.1. Overview of problems encountered when working with more data than can fit in memory



A computer only has a limited amount of RAM. When you try to squeeze more data into this memory than actually fits, the OS will start swapping out memory blocks to disks, which is far less efficient than having it all in memory. But only a few algorithms are designed to handle large data sets; most of them load the whole data set into memory at once, which causes the out-of-memory error. Other algorithms need to hold multiple copies of the data in memory or store intermediate results. All of these aggravate the problem.

Even when you cure the memory issues, you may need to deal with another limited resource: *time*. Although a computer may think you live for millions of years, in reality you won't (unless you go into cryostasis until your PC is done). Certain algorithms don't take time into account; they'll keep running forever. Other algorithms can't end in a reasonable amount of time when they need to process only a few megabytes of data.

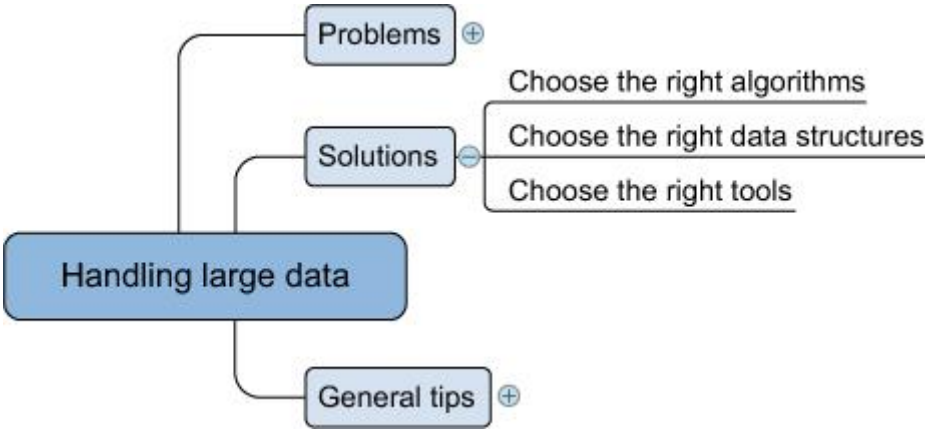
A third thing you'll observe when dealing with large data sets is that components of your computer can start to form a bottleneck while leaving other systems idle. Although this isn't as severe as a never-ending algorithm or out-of-memory errors, it still incurs a serious cost. Think of the cost savings in terms of person days and computing infrastructure for CPU starvation. Certain programs don't feed data fast enough to the processor because they have to read data from the hard drive, which is one of the slowest components on a computer. This has been addressed with the introduction of solid state drives (SSD), but SSDs are still much more expensive than the slower and more widespread hard disk drive (HDD) technology.

➢ 4.2 General techniques for handling large volumes of data

Never-ending algorithms, out-of-memory errors, and speed issues are the most common challenges you face when working with large data. In this section, we'll investigate solutions to overcome or alleviate these problems.

The solutions can be divided into three categories: using the correct algorithms, choosing the right data structure, and using the right tools (figure 4.2).

Figure 4.2. Overview of solutions for handling large data sets

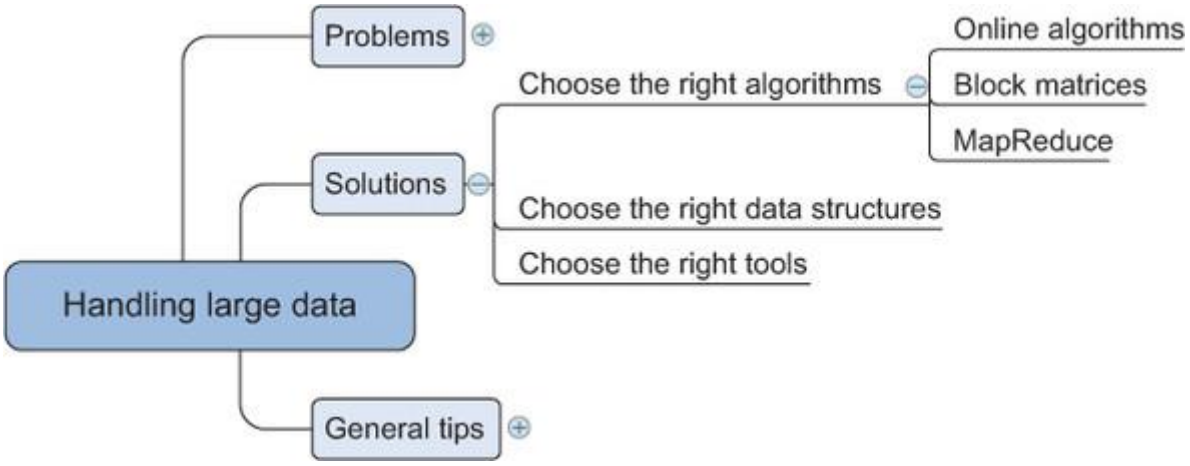


No clear one-to-one mapping exists between the problems and solutions because many solutions address both lack of memory and computational performance. For instance, data set compression will help you solve memory issues because the data set becomes smaller. But this also affects computation speed with a shift from the slow hard disk to the fast CPU. Contrary to RAM (random access memory), the hard disc will store everything even after the power goes down, but writing to disc costs more time than changing information in the fleeting RAM. When constantly changing the information, RAM is thus preferable over the (more durable) hard disc. With an unpacked data set, numerous read and write operations (I/O) are occurring, but the CPU remains largely idle, whereas with the compressed data set the CPU gets its fair share of the workload. Keep this in mind while we explore a few solutions.

➢ 4.2.1 Choosing the right algorithm

Choosing the right algorithm can solve more problems than adding more or better hardware. An algorithm that's well suited for handling large data doesn't need to load the entire data set into memory to make predictions. Ideally, the algorithm also supports parallelized calculations. In this section we'll dig into three types of algorithms that can do that: *online algorithms*, *block algorithms*, and *MapReduce algorithms*, as shown in figure 4.3.

Figure 4.3. Overview of techniques to adapt algorithms to large data sets



a) Online learning algorithms

Several, but not all, machine learning algorithms can be trained using one observation at a time instead of taking all the data into memory. Upon the arrival of a new data point, the model is trained and the observation can be forgotten; its effect is now incorporated into the model's parameters. For example, a model used to predict the weather can use different parameters (like atmospheric pressure or temperature) in different regions. When the data from one region is loaded into the algorithm, it forgets about this raw data and moves on to the next region. This "use and forget" way of working is the perfect solution for the memory problem as a single observation is unlikely to ever be big enough to fill up all the memory of a modern-day computer.

Most online algorithms can also handle mini-batches; this way, you can feed them batches of 10 to 1,000 observations at once while using a sliding window to go over your data. You have three options:

- ***Full batch learning (also called statistical learning)*** —Feed the algorithm all the data at once.
- ***Mini-batch learning*** —Feed the algorithm a spoonful (100, 1000, ..., depending on what your hardware can handle) of observations at a time.
- ***Online learning*** —Feed the algorithm one observation at a time.

Online learning techniques are related to *streaming algorithms*, where you see every data point only once. Think about incoming Twitter data: it gets loaded into the algorithms, and then the observation (tweet) is discarded because the sheer number of incoming tweets of data might soon overwhelm the hardware. Online learning algorithms differ from streaming algorithms in that they can see the same observations multiple times. True, the online learning algorithms and streaming algorithms can *both* learn from observations one by one. Where they differ is that *online algorithms* are also used on a static data source as well as on a streaming data source by presenting the data in small batches (as small as a single observation), which enables you to go over the data multiple times.

b) Block matrices and matrix formula of linear regression coefficient estimation

Certain algorithms can be translated into algorithms that use blocks of matrices instead of full matrices. When you partition a matrix into a block matrix, you divide the full matrix into parts and work with the smaller parts instead of the full matrix. In this case you can load smaller matrices into memory and perform calculations, thereby avoiding an out-of-memory error. Figure 4.4 shows how you can rewrite matrix addition A + B into submatrices.

Figure 4.4. Block matrices can be used to calculate the sum of the matrices A and B.

$$A + B = \begin{bmatrix} a_{1,1} & \cdots & a_{1,m} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,m} \end{bmatrix} + \begin{bmatrix} b_{1,1} & \cdots & b_{1,m} \\ \vdots & \ddots & \vdots \\ b_{n,1} & \cdots & b_{n,m} \end{bmatrix}$$

$$= \left[\begin{array}{ccc} a_{1,1} & \cdots & a_{1,m} \\ \vdots & \ddots & \vdots \\ a_{j,1} & \cdots & a_{j,m} \\ \hline a_{j+1,1} & \cdots & a_{j+1,m} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \cdots & a_{n,m} \end{array}\right] + \left[\begin{array}{ccc} b_{1,1} & \cdots & b_{1,m} \\ \vdots & \ddots & \vdots \\ b_{j,1} & \cdots & b_{j,m} \\ \hline b_{j+1,1} & \cdots & b_{j+1,m} \\ \vdots & \ddots & \vdots \\ b_{n,1} & \cdots & b_{n,m} \end{array}\right] = \begin{bmatrix} A_1 \\ \hline A_2 \end{bmatrix} + \begin{bmatrix} B_1 \\ \hline B_2 \end{bmatrix}$$

The formula in figure 4.4 shows that there's no difference between adding matrices A and B together in one step or first adding the upper half of the matrices and then adding the lower half.

All the common matrix and vector operations, such as multiplication, inversion, and singular value decomposition (a variable reduction technique like PCA), can be written in terms of block matrices.[1] Block matrix operations save memory by splitting the problem into smaller blocks and are easy to parallelize.

Although most numerical packages have highly optimized code, they work only with matrices that can fit into memory and will use block matrices in memory when advantageous. With out-of-memory matrices, they don't optimize this for you and it's up to you to partition the matrix into smaller matrices and to implement the block matrix version.

c) MapReduce

MapReduce algorithms are easy to understand with an analogy: Imagine that you were asked to count all the votes for the national elections. Your country has 25 parties, 1,500 voting offices, and 2 million people. You could choose to gather all the voting tickets from every office individually and count them centrally, or you could ask the local offices to count the votes for the 25 parties and hand over the results to you, and you could then aggregate them by party.

Map reducers follow a similar process to the second way of working. They first map values to a key and then do an aggregation on that key during the reduce phase. Have a look at the following listing's pseudo code to get a better feeling for this.

One of the advantages of MapReduce algorithms is that they're easy to parallelize and distribute. This explains their success in distributed environments such as Hadoop, but they can also be used on individual computers.

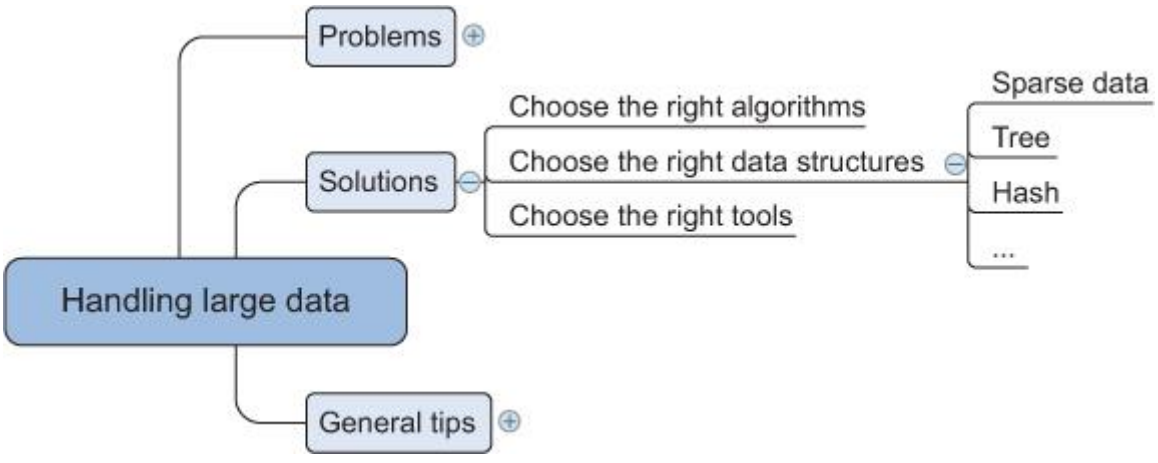
A number of libraries have done most of the work for you, such as Hadoopy, Octopy, Disco, or Dumbo.

➢ 4.2.2. Choosing the right data structure

Algorithms can make or break your program, but the way you store your data is of equal importance. Data structures have different storage requirements, but also influence the performance of *CRUD* (create, read, update, and delete) and other operations on the data set.

To shows you have many different data structures to choose from, three of which we'll discuss here: sparse data, tree data, and hash data. Let's first have a look at sparse data sets.

Figure 4.5. Overview of data structures often applied in data science when working with large data



a) Sparse data

A sparse data set contains relatively little information compared to its entries (observations). Look at figure 4.6: almost everything is "0" with just a single "1" present in the second observation on variable 9.

Figure 4.6. Example of a sparse matrix: almost everything is 0; other values are the exception in a sparse matrix

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Data like this might look ridiculous, but this is often what you get when converting textual data to binary data. Imagine a set of 100,000 completely unrelated Twitter tweets. Most of them probably have fewer than 30 words, but together they might have hundreds or thousands of distinct words. In the chapter on text mining we'll go through the process of cutting text documents into words and storing them as vectors. But for now imagine what you'd get if every word was converted to a binary variable, with "1" representing "present in this tweet," and "0" meaning "not present in this tweet." This would result in sparse data indeed. The resulting large matrix can cause memory problems even though it contains little information

Luckily, data like this can be stored compacted. In the case of figure 4.6 it could look like this:

```
data = [(2,9,1)]
```

Row 2, column 9 holds the value 1.

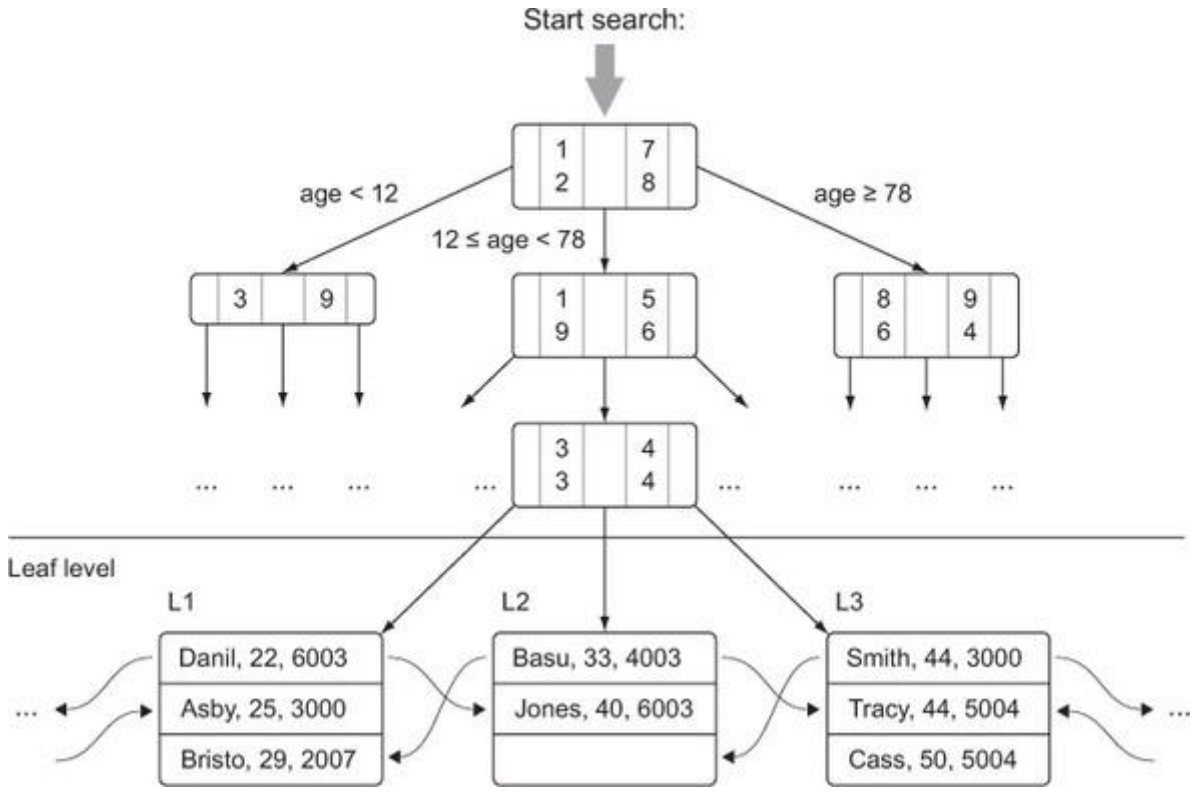
Support for working with sparse matrices is growing in Python. Many algorithms now support or return sparse matrices.

b) Tree structures

Trees are a class of data structure that allows you to retrieve information much faster than scanning through a table. A tree always has a root value and subtrees of children, each with its children, and so on. Simple examples would be your own family tree or a biological tree and the way it splits into branches, twigs, and leaves. Simple decision rules make it easy to find the child tree in which your data resides. Look at figure 4.7 to see how a tree structure enables you to get to the relevant information quickly.

.

Figure 4.7. Example of a tree data structure: decision rules such as age categories can be used to quickly locate a person in a family tree



In figure 4.7 you start your search at the top and first choose an age category, because apparently that's the factor that cuts away the most alternatives. This goes on and on until you get what you're looking for. For whoever isn't acquainted with the Akinator, we recommend visiting http://en.akinator.com/. The Akinator is a djinn

in a magical lamp that tries to guess a person in your mind by asking you a few questions about him or her. Try it out and be amazed . . . or see how this magic is a tree search.

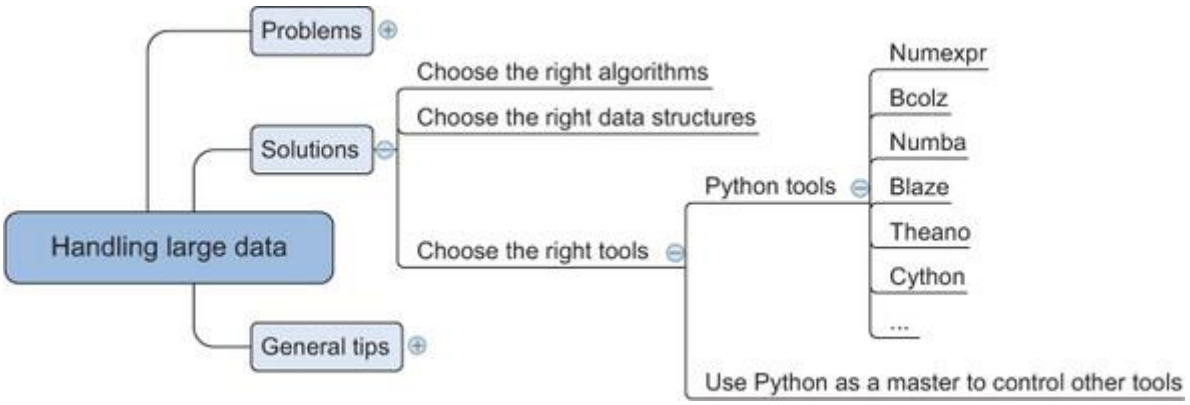
c) Hash tables

Hash tables are data structures that calculate a key for every value in your data and put the keys in a bucket. This way you can quickly retrieve the information by looking in the right bucket when you encounter the data. Dictionaries in Python are a hash table implementation, and they're a close relative of key-value stores. You'll encounter them in the last example of this chapter when you build a recommender system within a database. Hash tables are used extensively in databases as indices for fast information retrieval.

➢ 4.2.3. Selecting the right tools

With the right class of algorithms and data structures in place, it's time to choose the right tool for the job. The right tool can be a Python library or at least a tool that's controlled from Python, as shown figure 4.8. The number of helpful tools available is enormous, so we'll look at only a handful of them.

Figure 4.8. Overview of tools that can be used when working with large data



Python tools

Python has a number of libraries that can help you deal with large data. They range from smarter data structures over code optimizers to just-in-time

compilers. The following is a list of libraries we like to use when confronted with large data:

- ***Cython*** —The closer you get to the actual hardware of a computer, the more vital it is for the computer to know what types of data it has to process. For a computer, adding 1 + 1 is different from adding 1.00 + 1.00. The first example consists of integers and the second consists of floats, and these calculations are performed by different parts of the CPU. In Python you don't have to specify what data types you're using, so the Python compiler has to infer them. But inferring data types is a slow operation and is partially why Python isn't one of the fastest languages available. Cython, a superset of Python, solves this problem by forcing the programmer to specify the data type while developing the program. Once the compiler has this information, it runs programs much faster. See http://cython.org/ for more information on Cython.
- ***Numexpr*** —Numexpr is at the core of many of the big data packages, as is NumPy for in-memory packages. Numexpr is a numerical expression evaluator for NumPy but can be many times faster than the original NumPy. To achieve this, it rewrites your expression and uses an internal (just-in-time) compiler. See https://github.com/pydata/numexpr for details on Numexpr.
- See https://github.com/pydata/numexpr for details on Numexpr.
- ***Numba*** —Numba helps you to achieve greater speed by compiling your code right before you execute it, also known as *just-in-time compiling*. This gives you the advantage of writing high-level code but achieving speeds similar to those of C code. Using Numba is straightforward;
  see http://numba.pydata.org/.
- ***Bcolz*** —Bcolz helps you overcome the out-of-memory problem that can occur when using NumPy. It can store and work with arrays in an optimal compressed form. It not only slims down your data need but also uses Numexpr in the background to reduce the calculations needed when performing calculations with bcolz arrays. See http://bcolz.blosc.org/.

- ***Blaze*** —Blaze is ideal if you want to use the power of a database backend but like the "Pythonic way" of working with data. Blaze will translate your Python code into SQL but can handle many more data stores than relational databases such as CSV, Spark, and others. Blaze delivers a unified way of working with many databases and data libraries. Blaze is still in development, though, so many features aren't implemented yet.
  See http://blaze.readthedocs.org/en/latest/index.html.
- ***Theano*** —Theano enables you to work directly with the graphical processing unit (GPU) and do symbolical simplifications whenever possible, and it comes with an excellent just-in-time compiler. On top of that it's a great library for dealing with an advanced but useful mathematical concept: tensors.
  See http://deeplearning.net/software/theano/.
- ***Dask*** —Dask enables you to optimize your flow of calculations and execute them efficiently. It also enables you to distribute calculations.
  See http://dask.pydata.org/en/latest/.

These libraries are mostly about using Python itself for data processing (apart from Blaze, which also connects to databases). To achieve high-end performance, you can use Python to communicate with all sorts of databases or other software.

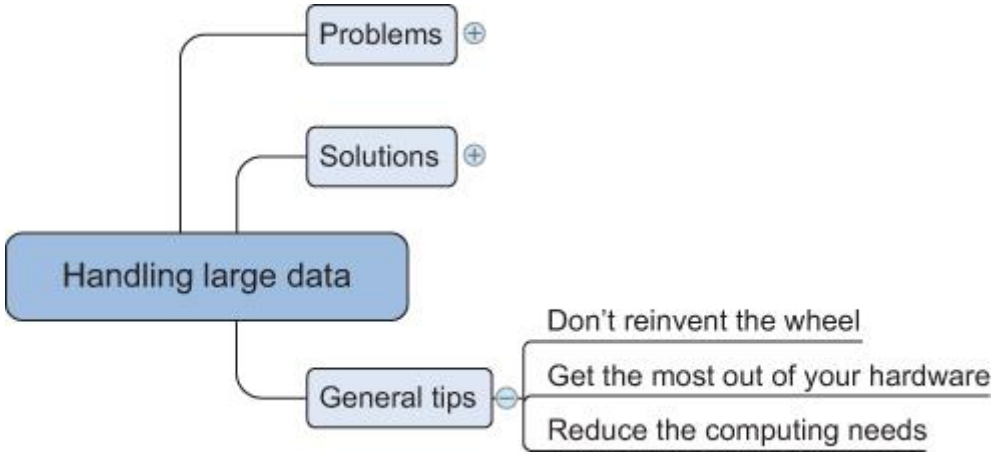
These libraries are mostly about using Python itself for data processing (apart from Blaze, which also connects to databases). To achieve high-end performance, you can use Python to communicate with all sorts of databases or other software.

➢ 4.3. General programming tips for dealing with large data sets

The tricks that work in a general programming context still apply for data science. Several might be worded slightly differently, but the principles are essentially the same for all programmers. This section recapitulates those tricks that are important in a data science *context*.

You can divide the general tricks into three parts, as shown in the figure 4.9 mind map:

Figure 4.9. Overview of general programming best practices when working with large data



- ***Don't reinvent the wheel.*** Use tools and libraries developed by others.
- ***Get the most out of your hardware.*** Your machine is never used to its full potential; with simple adaptions you can make it work harder.
- ***Reduce the computing need.*** Slim down your memory and processing needs as much as possible.

"Don't reinvent the wheel" is easier said than done when confronted with a specific problem, but your first thought should always be, 'Somebody else must have encountered this same problem before me.'

4.3.1. DON'T REINVENT THE WHEEL

"Don't repeat anyone" is probably even better than "don't repeat yourself." Add value with your actions: make sure that they matter. Solving a problem that has already been solved is a waste of time. As a data scientist, you have two large rules that can help you deal with large data and make you much more productive, to boot:

- ***Exploit the power of databases.*** The first reaction most data scientists have when working with large data sets is to prepare their analytical base tables inside a database. This method works well when the features you want to

prepare are fairly simple. When this preparation involves advanced modeling, find out if it's possible to employ user-defined functions and procedures. The last example of this chapter is on integrating a database into your workflow.

- ***Use optimized libraries.*** Creating libraries like Mahout, Weka, and other machine-learning algorithms requires time and knowledge. They are highly optimized and incorporate best practices and state-of-the art technologies. Spend your time on getting things done, not on reinventing and repeating others people's efforts, unless it's for the sake of understanding how things work.

Then you must consider your hardware limitation.

4.3.2. GET THE MOST OUT OF YOUR HARDWARE

Resources on a computer can be idle, whereas other resources are over-utilized. This slows down programs and can even make them fail. Sometimes it's possible (and necessary) to shift the workload from an overtaxed resource to an underutilized resource using the following techniques:

- ***Feed the CPU compressed data.*** A simple trick to avoid CPU starvation is to feed the CPU compressed data instead of the inflated (raw) data. This will shift more work from the hard disk to the CPU, which is exactly what you want to do, because a hard disk can't follow the CPU in most modern computer architectures.
- ***Make use of the GPU.*** Sometimes your CPU and not your memory is the bottleneck. If your computations are parallelizable, you can benefit from switching to the GPU. This has a much higher throughput for computations than a CPU. The GPU is enormously efficient in parallelizable jobs but has less cache than the CPU. But it's pointless to switch to the GPU when your hard disk is the problem. Several Python packages, such as Theano and NumbaPro, will use the GPU without much programming effort. If this doesn't suffice, you can use a CUDA (Compute Unified Device Architecture) package such as

PyCUDA. It's also a well-known trick in bitcoin mining, if you're interested in creating your own money.

- ***Use multiple threads.*** It's still possible to parallelize computations on your CPU. You can achieve this with normal Python threads.

4.3.3. REDUCE YOUR COMPUTING NEEDS

"Working smart + hard = achievement." This also applies to the programs you write. The best way to avoid having large data problems is by removing as much of the work as possible up front and letting the computer work only on the part that can't be skipped. The following list contains methods to help you achieve this:

- ***Profile your code and remediate slow pieces of code.*** Not every piece of your code needs to be optimized; use a profiler to detect slow parts inside your program and remediate these parts.
- ***Use compiled code whenever possible, certainly when loops are involved.*** Whenever possible use functions from packages that are optimized for numerical computations instead of implementing everything yourself. The code in these packages is often highly optimized and compiled.
- ***Otherwise, compile the code yourself.*** If you can't use an existing package, use either a just-in-time compiler or implement the slowest parts of your code in a lower-level language such as C or Fortran and integrate this with your codebase. If you make the step to *lower-level languages* (languages that are closer to the universal computer bytecode), learn to work with computational libraries such as LAPACK, BLAST, Intel MKL, and ATLAS. These are highly optimized, and it's difficult to achieve similar performance to them.
- ***Avoid pulling data into memory.*** When you work with data that doesn't fit in your memory, avoid pulling everything into memory. A simple way of doing this is by reading data in chunks and parsing the data on the fly. This won't work on every algorithm but enables calculations on extremely large data sets.

- ***Use generators to avoid intermediate data storage.*** Generators help you return data per observation instead of in batches. This way you avoid storing intermediate results.
- ***Use as little data as possible.*** If no large-scale algorithm is available and you aren't willing to implement such a technique yourself, then you can still train your data on only a sample of the original data.
- ***Use your math skills to simplify calculations as much as possible.*** Take the following equation, for example: $(a + b)^2 = a^2 + 2ab + b^2$. The left side will be computed much faster than the right side of the equation; even for this trivial example, it could make a difference when talking about big chunks of data.

➢ 4.4. Case study 1: Predicting malicious URLs

The internet is probably one of the greatest inventions of modern times. It has boosted humanity's development, but not everyone uses this great invention with honorable intentions. Many companies (Google, for one) try to protect us from fraud by detecting malicious websites for us. Doing so is no easy task, because the internet has billions of web pages to scan. In this case study we'll show how to work with a data set that no longer fits in memory.

What we'll use

- ***Data*** —The data in this case study was made available as part of a research project. The project contains data from 120 days, and each observation has approximately 3,200,000 features. The target variable contains 1 if it's a malicious website and -1 otherwise. For more information, please see "Beyond Blacklists: Learning to Detect Malicious Web Sites from Suspicious URLs"
- ***The Scikit-learn library*** —You should have this library installed in your Python environment at this point, because we used it in the previous chapter. As you can see, we won't be needing much for this case, so let's dive into it.

4.4.1. STEP 1: DEFINING THE RESEARCH GOAL

The goal of our project is to detect whether certain URLs can be trusted or not. Because the data is so large we aim to do this in a memory-friendly way. In the next step we'll first look at what happens if we don't concern ourselves with memory (RAM) issues.

4.4.2. STEP 2: ACQUIRING THE URL DATA

Start by downloading the data from http://sysnet.ucsd.edu/projects/url/#datasets and place it in a folder. Choose the data in SVMLight format. SVMLight is a text-based format with one observation per row. To save space, it leaves out the zeros.

surprise, we get an out-of-memory error. That is, unless you run this code on a huge machine. After a few tricks you'll no longer run into these memory problems and will detect 97% of the malicious sites.

Tools and techniques

We ran into a memory error while loading a single file—still 119 to go. Luckily, we have a few tricks up our sleeve. Let's try these techniques over the course of the case study:

- Use a sparse representation of data.
- Feed the algorithm compressed data instead of raw data.
- Use an online algorithm to make predictions.

4.4.3. STEP 3: DATA EXPLORATION

To see if we can even apply our first trick (sparse representation), we need to find out whether the data does indeed contain lots of zeros.

One of the file formats that implements this is SVMLight, and that's exactly why we downloaded the data in this format. We're not finished yet, though, because we need to get a feel of the dimensions within the data.

To get this information we already need to keep the data compressed while checking for the maximum number of observations and variables. We also need to *read in data file by file*. This way you consume even less memory. A second trick is to feed the CPU compressed files. In our example, it's already packed in the tar.gz format. You unpack a file only when you need it, without writing it to the hard disk (the slowest part of your computer).

4.4.4. STEP 4: MODEL BUILDING

Now that we're aware of the dimensions of our data, we can apply the same two tricks (sparse representation of compressed file) and add the third (using an online algorithm), in the following listing. Let's find those harmful websites!

The code in the previous listing looks fairly similar to what we did before, apart from the stochastic gradient descent classifier `SGDClassifier().`

Here, we trained the algorithm iteratively by presenting the observations in one file with the `partial_fit()` function.

➢ 4.6. .Handling large data Summary

This chapter discussed the following topics:

- The main *problems* you can run into when working with large data sets are these:
  - Not enough memory
  - Long-running programs

  - Resources that form bottlenecks and cause speed problems
- There are three main types of *solutions* to these problems:
  - Adapt your algorithms.
  - Use different data structures.
  - Rely on tools and libraries.
- Three main techniques can be used to *adapt an algorithm*:
  - Present algorithm data *one observation at a time* instead of loading the full data set at once.
  - *Divide matrices into smaller matrices* and use these to make your calculations.
  - Implement the *MapReduce* algorithm (using Python libraries such as Hadoopy, Octopy, Disco, or Dumbo).
- Three main *data structures* are used in data science. The first is a type of matrix that contains relatively little information, the *sparse matrix.* The second and third are data structures that enable you to retrieve information quickly in a large data set: the *hash function* and *tree structure.*
- Python has many *tools* that can help you deal with large data sets. Several tools will help you with the size of the volume, others will help you parallelize the computations, and still others overcome the relatively slow speed of Python itself. It's also easy to use Python as a tool to control other data science tools because Python is often chosen as a language in which to implement an API.
- The *best practices* from computer science are also valid in a data science context, so applying them can help you overcome the problems you face in a big data context.

UNIT-V

➢ Subsetting R Objects

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

1 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"
```

2 Subsetting a Matrix

Matrices can be subsetted in the usual way with (*i*,*j*) type indices. Here, we create simple 2×32×3 matrix with the `matrix` function.

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

3 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(hoo = 1:4, bar = 0.6)
> x
$hoo
[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(hoo = 1:4, bar = 0.6, baz = "hello")
> name <- "foo"
>
```

```
> ## computed index for "hoo"
> x[[name]]
[1] 1 2 3 4
> ## element "hoo" does exist
> x$hoo
[1] 1 2 3 4
```

4 Subsetting Nested Elements of a List

The [[operator can take an integer sequence if you want to extract a nested element of a list.

```
> x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
>
> ## Get the 3rd element of the 1st element
> x[[c(1, 3)]]
[1] 14
>
> ## Same as above
> x[[1]][[3]]
[1] 14
>
> ## 1st element of the 2nd element
> x[[c(2, 1)]]
[1] 3.14
```

5 Extracting Multiple Elements of a List

The [operator can be used to extract *multiple* elements from a list. For example, if you wanted to extract the first and third elements of a list, you would do the following

```
> x <- list(hoo = 1:4, bar = 0.6, baz = "hello")
> x[c(1, 3)]
$hoo
[1] 1 2 3 4


$baz
[1] "hello"
```

6 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
```

➢ Vectorized Operations

Many operations in R are *vectorized*, meaning that operations occur in parallel in certain R objects. This allows you to write code that is efficient, concise, and easier to read than in non-vectorized languages.

The simplest example is when adding two vectors together.

```
> x <- 1:4
> y <- 6:9
> z <- x + y
> z
[1]  7  9 11 13
```

Another operation you can do in a vectorized manner is logical comparisons. So suppose you wanted to know which elements of a vector were greater than 2. You could do he following.

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

Here are other vectorized logical operations.

```
> x >= 2
[1] FALSE  TRUE  TRUE  TRUE
> x < 3
[1]  TRUE  TRUE FALSE FALSE
> y == 8
[1] FALSE FALSE  TRUE FALSE
```

Notice that these logical operations return a logical vector of TRUE and FALSE.

Of course, subtraction, multiplication and division are also vectorized.

```
> x - y
[1] -5 -5 -5 -5
> x * y
[1]  6 14 24 36
> x / y
[1] 0.1666667 0.2857143 0.3750000 0.4444444
```

Vectorized Matrix Operations

Matrix operations are also vectorized, making for nicly compact notation. This way, we can do element-by-element operations on matrices without having to loop over every element.

```
> x <- matrix(1:4, 2, 2)
> y <- matrix(rep(10, 4), 2, 2)
>
> ## element-wise multiplication
> x * y
     [,1] [,2]
[1,]   10   30
[2,]   20   40
>
> ## element-wise division
> x / y
     [,1] [,2]
[1,]  0.1  0.3
[2,]  0.2  0.4
>
> ## true matrix multiplication
> x %*% y
     [,1] [,2]
[1,]   40   40
[2,]   60   60
```

➢ Managing Data Frames with the dplyr package

Data Frames

The *data frame* is a key data structure in statistics and in R. The basic structure of a data frame is that there is one observation per row and each column represents a variable, a measure, feature, or characteristic of that

observation. R has an internal implementation of data frames that is likely the one you will use most often

Installing the dplyr package

The dplyr package can be installed from CRAN or from GitHub using the devtools package and the install_github() function. The GitHub repository will usually contain the latest updates to the package and the development version.

To install from CRAN, just run

```
> install.packages("dplyr")
```

After installing the package it is important that you load it into your R session with the library() function.

```
> library(dplyr)


The following objects are masked


    intersect, setdiff, setequal, union
```

```
> install.packages("dplyr")
> library("dplyr")

> s <- data.frame(
+   name = c("sam", "jan", "ram", "sonu"),
+   age = c(16, NA, 14, 15),
+   school = c("s", "m", "a", "n"),
+   en = c(72, 84, 50, 65),
+   tl = c(76, 82, 58, 61),
+   ht = c(76, 62, 47, 67)
+ )
> s
```

```
  name age school en tl ht
1  sam  16      s 72 76 76
2  jan  NA      m 84 82 62
3  ram  14      a 50 58 47
4 sonu  15      n 65 61 67
```

.

1 select()

For the examples in this chapter we will be using a dataset containing air pollution and temperature data for the dataset

```
> select(s, starts_with("age"))
  age
1  16
2  NA
3  14
4  15
> select(s, starts_with("age"))
  age
1  16
2  NA
3  14
4  15
> select(s, -starts_with("age"))
  name school en tl ht
1  sam      s 72 76 76
2  jan      m 84 82 62
```

```
3  ram      a 50 58 47
4 sonu      n 65 61 67
> select(s, 1:2)
  name age
1  sam  16
2  jan  NA
3  ram  14
4 sonu  15
> select(s, contains("a"))
  name age
1  sam  16
2  jan  NA
3  ram  14
4 sonu  15
> select(s, matches("na"))
  name
1  sam
2  jan
3  ram
4 sonu
```

The select() function can be used to select columns of a data frame that you want to focus on. Often you'll have a large data frame containing "all" of the data, but any *given* analysis might only use a subset of variables or observations. The select() function allows you to get the few columns you might need.

2 filter()

The filter() function is used to extract subsets of rows from a data frame. This function is similar to the existing subset() function in R but is quite a bit faster in my experience.

```
> s %>% filter(is.na(age))
  name age school en tl ht
1  jan  NA      m 84 82 62
> s%>% filter(!is.na(age))
  name age school en tl ht
1  sam  16      s 72 76 76
2  ram  14      a 50 58 47
3 sonu  15      n 65 61 67
> s%>% filter(!is.na(age) & age==16)
  name age school en tl ht
1  sam  16      s 72 76 76
```

3 arrange()

The arrange() function is used to reorder rows of a data frame according to one of the variables/columns. Reordering rows of a data frame (while preserving corresponding order of other columns) is normally a pain to do in R. The arrange() function simplifies the process quite a bit.

Here we can order the rows of the data frame by date, so that the first row is the earliest (oldest) observation and the last row is the latest (most recent) observation.

```
> arrange(s, age)
  name age school en tl ht
1  ram  14      a 50 58 47
2 sonu  15      n 65 61 67
3  sam  16      s 72 76 76
```

```
4  jan  NA      m 84 82 62
```

4 mutate()

The mutate() function exists to compute transformations of variables in a data frame. Often, you want to *create new variables* that are derived from existing variables and mutate() provides a clean interface for doing that.

```
> mutate(s, total_marks = ht + tl+en)
  name age school en tl ht total_marks
1  sam  16      s 72 76 76         224
2  jan  NA      m 84 82 62         228
3  ram  14      a 50 58 47         155
4 sonu  15      n 65 61 67         193
> transmute(s, total = ht + tl+en)
  total
1   224
2   228
3   155
4   193
```

5 %>%

The pipeline operater %>% is very handy for stringing together multiple dplyr functions in a sequence of operations. Notice above that every time we wanted to apply more than one function, the sequence gets buried in a sequence of nested function calls that is difficult to read, i.e.

```
> s%>% filter(!is.na(age) & age==16)
  name age school en tl ht
1  sam  16      s 72 76 76
```

Notice in the data frame to the first call to mutate(), but then afterwards I do not have to pass the first argument to group_by() or summarize(). Once you travel down the pipeline with %>%, the first argument is taken to be the output of the previous element in the pipeline.

Another example might be computing the average pollutant level by month. This could be useful to see if there are any seasonal trends in the data.

```
> summarise(s, mean = mean(age))
  mean
1   NA
> summarise(s, mean = mean(ht))
  mean
1   63
> summarise(s, med = min(tl))
  med
1  58
```

Here we can see that o3 tends to be low in the winter months and high in the summer while no2 is higher in the winter and lower in the summer.

➢ Control Structures

Control structures in R allow you to control the flow of execution of a series of R expressions. Basically, control structures allow you to put some "logic" into your R code, rather than just always executing the same R code every time. Control structures allow you to respond to inputs or to features of the data and execute different R expressions accordingly.

Commonly used control structures are

- if and else: testing a condition and acting on it
- for: execute a loop a fixed number of times
- while: execute a loop *while* a condition is true

- repeat: execute an infinite loop (must break out of it to stop)
- break: break the execution of a loop
- next: skip an interation of a loop

1 if-else

The if-else combination is probably the most commonly used control structure in R (or perhaps any language). This structure allows you to test a condition and act on it depending on whether it's true or false.

For starters, you can just use the if statement.

```
if(<condition>) {
        ## do something
}
## Continue with rest of code
```

The above code does nothing if the condition is false. If you have an action you want to execute when the condition is false, then you need an else clause.

```
if(<condition>) {
        ## do something
}
else {
        ## do something else
}
```

You can have a series of tests by following the initial if with any number of else ifs.

```
if(<condition1>) {
        ## do something
} else if(<condition2>)  {
        ## do something different
```

```
} else {
        ## do something different
}
```

Here is an example of a valid if/else structure.

```
> x<-20
> y<-30
> if(x>y){
+   print("x is big")
+ }else{
+   print("y is big")
+ }
[1] "y is big"
```

The value of `y` is set depending on whether `x > 3` or not.

2 for Loops

For loops are pretty much the only looping construct that you will need in R.

In R, for loops take an iterator variable and assign it successive values from a sequence or vector. For loops are mos

t commonly used for iterating over the elements of an object (list, vector, etc.)

```
> for(i in 1:10) {
+         print(i)
+ }
[1] 1
[1] 2
[1] 3
[1] 4
```

```
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

This loop takes the i variable and in each iteration of the loop gives it values 1, 2, 3, ..., 10, executes the code within the curly braces, and then the loop exits.

The following three loops all have the same behavior.

```
> x <- c("a", "b", "c", "d")
>
> for(i in 1:4) {
+         ## Print out each element of 'x'
+         print(x[i])
+ }
[1] "a"
[1] "b"
[1] "c"
[1] "d"
```

The seq_along() function is commonly used in conjunction with for loops in order to generate an integer sequence based on the length of an object (in this case, the object x).

```
> ## Generate a sequence based on length of 'x'
> for(i in seq_along(x)) {
+         print(x[i])
+ }
[1] "a"
```

```
[1] "b"
[1] "c"
[1] "d"
```

It is not necessary to use an index-type variable.

3 `while` Loops

While loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed, the condition is tested again, and so forth, until the condition is false, after which the loop exits.

```
> count <- 0
> while(count < 10) {
+         print(count)
+         count <- count + 1
+ }
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
```

.

4 `next`, `break`

`next` is used to skip an iteration of a loop.

```
for(i in 1:100) {
        if(i <= 20) {
                ## Skip the first 20 iterations
                next
        }
        ## Do something here
}
```

break is used to exit a loop immediately, regardless of what iteration the loop may be on.

```
> for(i in 1:100) {
+   print(i)
+
+   if(i > 10) {
+     ## Stop loop after 21 iterations
+     break
+   }
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

```
[1] 11
```

➢ Functions

Writing functions is a core activity of an R programmer. It represents the key step of the transition from a mere "user" to a developer who creates new functionality for R. Functions are often used to encapsulate a sequence of expressions that need to be executed numerous times, perhaps under slightly different conditions. Functions are also often written when code must be shared with others or the public.

Functions in R

Functions in R are "first class objects", which means that they can be treated much like any other R object. Importantly,

- Functions can be passed as arguments to other functions. This is very handy for the various apply functions, like lapply() and sapply().
- Functions can be nested, so that you can define a function inside of another function

Your First Function

Functions are defined using the function() directive and are stored as R objects just like anything else. In particular, they are R objects of class "function".

Here's a simple function that takes no arguments and does nothing.

```
> f <- function() {
+       ## This is an empty function
+ }
> ## Functions have their own class
> class(f)
[1] "function"
> ## Execute this function
```

```
> f()
NULL
```

Not very interesting, but it's a start. The next thing we can do is create a function that actually has a non-trivial *function body*.

```
> f <- function() {
+       cat("Hello, world!\n")
+ }
> f()
Hello, world!
```

The last aspect of a basic function is the *function arguments*. These are the options that you can specify to the user that the user may explicity set. For this basic function, we can add an argument that determines how many times "Hello, world!" is printed to the console.

```
> f <- function(num) {
+       for(i in seq_len(num)) {
+             cat("Hello, world!\n")
+       }
+ }
> f(3)
Hello, world!
Hello, world!
Hello, world!
```

Lazy Evaluation

Arguments to functions are evaluated *lazily*, so they are evaluated only as needed in the body of the function.

In this example, the function `f()` has two arguments: `a` and `b`.

```
> f <- function(a, b) {
+       a^2
```

```
+ }
> f(2)
[1] 4
```

This function never actually uses the argument b, so calling f(2) will not produce an error because the 2 gets positionally matched to a. This behavior can be good or bad. It's common to write a function that doesn't use an argument and not notice it simply because R never throws an error.

This example also shows lazy evaluation at work, but does eventually result in an error.

```
> f <- function(a, b) {
+         print(a)
+         print(b)
+ }
> f(45)
[1] 45
Error in print(b): argument "b" is missing, with no default
```

Notice that "45" got printed first before the error was triggered. This is because b did not have to be evaluated until after print(a). Once the function tried to evaluate print(b) the function had to throw an error.

➢ 15 Scoping Rules of R

Scopes

The scope of a variable is nothing more than the place in the code where it is referenced and visible. There are two basic concepts of scoping, *lexical* scoping and is *dynamic* scoping. In R, there is a concept of *free variables*, which add some spice to the scoping. The values of such

variables are searched for in the environment in which the function was defined.

Let's look at an example of free variables.

```
f <- function(a, b) {

  (a * b) / z

}
```

In this function, you have two formal arguments, **a** and **b**. You have another symbol, **z**, in the body of the function, which is a free variable. The scoping rules of the language define how value is assigned to free variables. R uses lexical scoping, which says the value for **z** is searched for in the environment where the function was defined.

With dynamic scoping, the variable is bound to the most recent value assigned to that variable. Scoping also introduces another concept called *extent*. The extent is a specific interval of time during which references may occur throughout the execution. A fun fact: The origin of lexical scoping was in 1960 when John McCarthy first published his original paper on the LISP programming language.

R provides some escape routes to bypass the shortcomings of lexical scoping. The **<-** operator is called a variable assignment operator. Given the expression **a <- 3.14**, the value is assigned to the variable in the current

environment. If you already had an assignment for the variable before in the same environment, this one will overwrite it. Variable assignments only update in the current environment, and they never create a new scope. When R is looking for a value of a given variable, it will start searching from the bottom. This means the current environment is inspected first, then its enclosing environment. The search goes until either the value is found or the empty environment is reached.

Let's demonstrate lookup.

```
> a <- 3.14
> b = function(x,y){ x * y / a}
> b(10,11)
```

The output is the following:

```
[1] 35.03185
```

When the function is called, only the two arguments are passed. R tries to look up the **a** variable's value and first looks at the scope of the function. Since it cannot be found there, it look for the value in the enclosing scope, where it finally finds it. If you had not defined the **a** variable, it would give you the following error: **Error in b(10, 11) : object 'a' not found**, stating that the lookup has failed.

This brings us to the concept of environment. Environments in R are basically mappings from variables to values. Every function has a local environment and a reference to the enclosing environment. This helps scoping and lookup. You have the option to add, remove, or modify variable mappings and can even change the reference to the enclosing environment.

➢ 8 Coding Style Tips for R Programming

R is an open-source programming language that is widely used as a statistical software and data analysis tool. R generally comes with the Command-line interface. R is available across widely used platforms like Windows, Linux, and macOS. Also, the **R programming language** is the latest cutting-edge tool. Software engineering is not just all about learning a language and building some software. As a software engineer or software developer, you are expected to write **good software**.

If the code is easy to understand and easy to change then definitely it's good software and developers love to work on that. For a beginner R programmer, it is a good idea to acquire and start using good practices in coding. Google and R-guru Hadley Wickham have excellent tips on R coding style guide. The list contains things that what to do and not to do while programming in R. So in this article we are going to discuss six coding style tips that help you to become a better programmer in R language.

1. Commenting

It's a common thing that developers use comments to specify the purpose of a line in their code. It's true that comments are really helpful in explaining the code what it does but it also requires more maintenance of the code. Sometimes it is very important, So in R programming always start commenting a **line with the comment symbol # and one space**. Hadley Wickham suggests to use the remaining of commented lines with – **and =** to break up the file into easily readable chunks. Please refer to the below sample code snippet:

- R

```
# Read table ----------------------------------
# Read table ==================================
```

2. Assignment

R has an unusual assignment operator '<-' instead of '=' sign. So it's a good practice to **use the '<-' sign, instead of the '=' sign**. Please refer to the below sample code snippet:

Good Practice:

```
# Good Practice
x <- 10
```

Bad Practice:

```
# Bad Practice
x = 10
```

3. File Names

The name of the file should be meaningful and end with **'.R'**. Please refer to the below sample code snippet:

Good Practice:

```
# Good Practice
fit-models.R
linear-regression.R
```

Bad Practice:

```
# Bad Practice
models.R
stuff.R
```

4. Object Names

Variable and function names must be in **lowercase**. Use an underscore **'_'** to separate words within a name. Generally, variable names should be nouns,

and function names should be verbs. Please refer to the below sample code snippet:

Good Practice:

```
# Good Practice
number_of_students
get_price
```

Bad Practice:

```
# Bad Practice
GetPrice
getprice
```

5. Spacing

Put a place spaces around all infix operators (**=, +, -, <-, etc.**). The same rule implements when using = in function calls. Always put a space after a comma, and never before. Please refer to the below sample code snippet:

Good Practice:

```
# Good Practice
perimeter_of_rectangle = 2(length + width), na.rm = TRUE)
```

Bad Practice:

```
# Bad Practice
perimeter_of_rectangle=2(length+width),na.rm=TRUE)
```

There's a small exception to this rule e.g in case of **:, :: and :::** don't need spaces around them. Please refer to the below sample code snippet:

Good Practice:

```
# Good Practice
x <- 1:20
value::real
```

Bad Practice:

```
# Bad Practice
x <- 1 : 20
value :: real
```

Put a space before left parentheses, except in a function call. Please refer to the below sample code snippet:

Good Practice:

```
# Good Practice
if (yes) do(x)
run(x, y)
```

Bad Practice:

```
# Bad Practice
if(yes)do(x)
run(x, y)
```

Do not put spaces around code in parentheses or square brackets except there's a comma. Please refer to the below sample code snippet:

Good Practice:

```
# Good Practice
student[1, ]
```

Bad Practice:

```
# Bad Practice


# Needs a space after the comma
student[1,]


# Put space after comma not before
student[1 ,]
```

6. Curly Braces

An opening curly brace should never go on its own line and should always be followed by a new line. A closing curly brace should always go on its own line unless it's followed by else. Always indent the code inside curly braces. Please refer to the below sample code snippet:

Good Practice:

```
# Good Practice
if (x > 0 && foo) {
  cat("X is positive")
}


if (x == 0) {
  log(a)
} else {
  a ^ x
}
```

Bad Practice:

```
# Bad Practice


if (x > 0 && foo)
cat("X is positive")


if (x == 0) {
  log(a)
}
else {
  a ^ x
}
```

7. Line Length

Try to limit the code to 80 characters per line. This fits comfortably on a printed page with a reasonably sized font.

8. Indentation

When indenting your code, use **two spaces**. Never use tabs or mix tabs and spaces. The only exception is if a function definition runs over multiple lines. In that case, indent the second line to where the definition starts. Please refer to the below sample code snippet:

Good Practice:

```
# Good Practice
function_name <- function(a = "a long argument",
                          b = "another argument",
                          c = "another long argument") {
  # As usual code is indented by two spaces
```

```
}
```

➢ Loop Functions

1 Looping on the Command Line

Writing `for` and `while` loops is useful when programming but not particularly easy when working interactively on the command line. Multi-line expressions with curly braces are just not that easy to sort through when working on the command line. R has some functions which implement looping in a compact form to make your life easier.

- `lapply()`: Loop over a list and evaluate a function on each element
- `sapply()`: Same as `lapply` but try to simplify the result
- `apply()`: Apply a function over the margins of an array
- `tapply()`: Apply a function over subsets of a vector
- `mapply()`: Multivariate version of `lapply`

An auxiliary function `split` is also useful, particularly in conjunction with `lapply`.

2 `lapply()`

The `lapply()` function does the following simple series of operations:

1. it loops over a list, iterating over each element in that list
2. it applies a *function* to each element of the list (a function that you specify)

Here's an example of applying the `mean()` function to all elements of a list. If the original list has names, the the names will be preserved in the output.

```
> x <- list(a = 1:5, b = rnorm(10))
> lapply(x, mean)
$a
```

```
[1] 3


$b
[1] 0.1322028
```

Notice that here we are passing the mean() function as an argument to the lapply() function. Functions in R can be used this way and can be passed back and forth as arguments just like any other object. When you pass a function to another function, you do not need to include the open and closed parentheses () like you do when you are *calling* a function.

3 sapply()

The sapply() function behaves similarly to lapply(); the only real difference is in the return value. sapply() will try to simplify the result of lapply() if possible. Essentially, sapply() calls lapply() on its input and then applies the following algorithm:

- If the result is a list where every element is length 1, then a vector is returned
- If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- If it can't figure things out, a list is returned

Here's the result of calling lapply().

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))
> lapply(x, mean)
$a
[1] 2.5


$b
[1] -0.251483
$c
[1] 1.481246
```

```
$d
[1] 4.968715
```

Notice that lapply() returns a list (as usual), but that each element of the list has length 1.

Here's the result of calling sapply() on the same list.

```
> sapply(x, mean)
       a        b        c        d
 2.500000 -0.251483  1.481246  4.968715
```

Because the result of lapply() was a list where each element had length 1, sapply() collapsed the output into a numeric vector, which is often more useful than a list.

4 split()

The split() function takes a vector or other objects and splits it into groups determined by a factor or list of factors.

The arguments to split() are

```
> str(split)
function (x, f, drop = FALSE, ...)
```

where

- x is a vector (or list) or data frame
- f is a factor (or coerced to one) or a list of factors
- drop indicates whether empty factors levels should be dropped

The combination of split() and a function like lapply() or sapply() is a common paradigm in R. The basic idea is that you can take a data structure, split it into subsets defined by another variable, and apply a function over those subsets. The results of applying tha function over the subsets are then collated and returned as an object. This sequence of

```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> f <- gl(3, 10)
```

```
> split(x, f)
$`1`
 [1]  0.3981302 -0.4075286  1.3242586 -0.7012317 -0.5806143 -1.0010722
 [7] -0.6681786  0.9451850  0.4337021  1.0051592
```

5 tapply

tapply() is used to apply a function over subsets of a vector. It can be thought of as a combination of split() and sapply() for vectors only. I've been told that the "t" in tapply() refers to "table", but that is unconfirmed.

```
> str(tapply)
function (X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```

The arguments to tapply() are as follows:

- X is a vector
- INDEX is a factor or a list of factors (or else they are coerced to factors)
- FUN is a function to be applied
- ... contains other arguments to be passed FUN
- simplify, should we simplify the result?

Given a vector of numbers, one simple operation is to take group means.

```
> ## Simulate some data
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> ## Define some groups with a factor variable
> f <- gl(3, 10)
> f
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
Levels: 1 2 3
> tapply(x, f, mean)
```

```
      1       2       3
0.1896235 0.5336667 0.9568236
```

We can also take the group means without simplifying the result, which will give us a list. For functions that return a single value, usually, this is not what we want, but it can be done.

6 apply()

The apply() function is used to a evaluate a function (often an anonymous one) over the margins of an array. It is most often used to apply a function to the rows or columns of a matrix (which is just a 2-dimensional array). However, it can be used with general arrays, for example, to take the average of an array of matrices. Using apply() is not really faster than writing a loop, but it works in one line and is highly compact.

```
> str(apply)
function (X, MARGIN, FUN, ..., simplify = TRUE)
```

The arguments to apply() are

- X is an array
- MARGIN is an integer vector indicating which margins should be "retained".
- FUN is a function to be applied
- ... is for other arguments to be passed to FUN

Here I create a 20 by 10 matrix of Normal random numbers. I then compute the mean of each column.

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 2, mean)  ## Take the mean of each column
 [1]  0.02218266 -0.15932850  0.09021391  0.14723035 -0.22431309 -
0.49657847
 [7]  0.30095015  0.07703985 -0.20818099  0.06809774
```

➢ 18 Debugging

18.1 Something's Wrong!

R has a number of ways to indicate to you that something's not right. There are different levels of indication that can be used, ranging from mere notification to fatal error. Executing any function in R may result in the following *conditions*.

- message: A generic notification/diagnostic message produced by the message() function; execution of the function continues
- warning: An indication that something is wrong but not necessarily fatal; execution of the function continues. Warnings are generated by the warning() function
- error: An indication that a fatal problem has occurred and execution of the function stops. Errors are produced by the stop() function.
- condition: A generic concept for indicating that something unexpected has occurred; programmers can create their own custom conditions if they want.

Here is an example of a warning that you might receive in the course of using R.

```
> log(-1)
Warning in log(-1): NaNs produced
[1] NaN
```

This warning lets you know that taking the log of a negative number results in a NaN value because you can't take the log of negative numbers. Nevertheless, R doesn't give an error, because it has a useful value that it can return, the NaN value. The warning is just there to let you know that something unexpected happen. Depending on what you are programming, you may have intentionally taken the log of a negative number in order to move on to another section of code.

Here is another function that is designed to print a message to the console depending on the nature of its input.

```
> printmessage <- function(x) {
+       if(x > 0)
+             print("x is greater than zero")
```

```
+       else
+               print("x is less than or equal to zero")
+       invisible(x)
+ }
```

This function is simple—it prints a message telling you whether x is greater than zero or less than or equal to zero. It also returns its input *invisibly*, which is a common practice with "print" functions. Returning an object invisibly means that the return value does not get auto-printed when the function is called.

Take a hard look at the function above and see if you can identify any bugs or problems.

We can execute the function as follows.

```
> printmessage(1)
[1] "x is greater than zero"
```

The function seems to work fine at this point. No errors, warnings, or messages.

```
> printmessage(NA)
Error in if (x > 0) print("x is greater than zero") else print("x is less than or equal to zero"): missing value where TRUE/FALSE needed
```

What happened?

Well, the first thing the function does is test if x > 0. But you can't do that test if x is a NA or NaN value. R doesn't know what to do in this case so it stops with a fatal error.

We can fix this problem by anticipating the possibility of NA values and checking to see if the input is NA with the is.na() function.

```
> printmessage2 <- function(x) {
+       if(is.na(x))
+               print("x is a missing value!")
+       else if(x > 0)
+               print("x is greater than zero")
```

```
+       else
+               print("x is less than or equal to zero")
+       invisible(x)
+ }
```

Now we can run the following.

```
> printmessage2(NA)
[1] "x is a missing value!"
```

And all is fine.

Now what about the following situation.

```
> x <- log(c(-1, 2))
Warning in log(c(-1, 2)): NaNs produced
```

We expect some NaNs here because taking the log of a negative number doesn't make sense.

```
> printmessage2(x)
Error in if (is.na(x)) print("x is a missing value!") else if (x > 0) print("x is greater than zero") else print("x is less than or equal to zero"): the condition has length > 1
```

Now what?? Why are we getting this error?

The problem here is that I passed printmessage2() a vector x that was of length 2 rather then length 1. Inside the body of printmessage2() the expression is.na(x) returns a vector that is tested in the if statement. However, if cannot take vector arguments so you get an error (in previous versions of R you only got a warning). The fundamental problem here is that printmessage2() is not *vectorized*.

We can solve this problem two ways. One is by simply not allowing vector arguments. The other way is to vectorize the printmessage2() function to allow it to take vector arguments.

For the first way, we simply need to check the length of the input.

```
> printmessage3 <- function(x) {
```

```
+       if(length(x) > 1L)
+               stop("'x' has length > 1")
+       if(is.na(x))
+               print("x is a missing value!")
+       else if(x > 0)
+               print("x is greater than zero")
+       else
+               print("x is less than or equal to zero")
+       invisible(x)
+ }
```

Now when we pass printmessage3() a vector we should get an error.

```
> printmessage3(1:2)
Error in printmessage3(1:2): 'x' has length > 1
```

Vectorizing the function can be accomplished easily with the Vectorize() function.

```
> printmessage4 <- Vectorize(printmessage2)
> out <- printmessage4(c(-1, 2))
[1] "x is less than or equal to zero"
[1] "x is greater than zero"
```

You can see now that the correct messages are printed without any warning or error. Note that I stored the return value of printmessage4() in a separate R object called out. This is because when I use the Vectorize() function it no longer preserves the invisibility of the return value.

2 Figuring Out What's Wrong

The primary task of debugging any R code is correctly diagnosing what the problem is. When diagnosing a problem with your code (or somebody else's), it's important first understand what you were expecting to occur.

Then you need to idenfity what *did* occur and how did it deviate from your expectations. Some basic questions you need to ask are

- What was your input? How did you call the function?
- What were you expecting? Output, messages, other results?
- What did you get?
- How does what you get differ from what you were expecting?
- Were your expectations correct in the first place?
- Can you reproduce the problem (exactly)?

Being able to answer these questions is important not just for your own sake, but in situations where you may need to ask someone else for help with debugging the problem. Seasoned programmers will be asking you these exact questions.

18.3 Debugging Tools in R

R provides a number of tools to help you with debugging your code. The primary tools for debugging functions in R are

- traceback(): prints out the function call stack after an error occurs; does nothing if there's no error
- debug(): flags a function for "debug" mode which allows you to step through execution of a function one line at a time
- browser(): suspends the execution of a function wherever it is called and puts the function in debug mode
- trace(): allows you to insert debugging code into a function a specific places
- recover(): allows you to modify the error behavior so that you can browse the function call stack

These functions are interactive tools specifically designed to allow you to pick through a function. There's also the more blunt technique of inserting print() or cat() statements in the function.

18.4 Using traceback()

The traceback() function prints out the *function call stack* after an error has occurred. The function call stack is the sequence of functions that was called before the error occurred.

For example, you may have a function a() which subsequently calls function b() which calls c() and then d(). If an error occurs, it may not be immediately clear in which function the error occurred.
The traceback() function shows you how many levels deep you were when the error occurred.

```
> mean(x)

Error in mean(x) : object 'x' not found

> traceback()

1: mean(x)
```

Using debug()

The debug() function initiates an interactive debugger (also known as the "browser" in R) for a function. With the debugger, you can step through an R function one expression at a time to pinpoint exactly where an error occurs.

The debug() function takes a function as its first argument. Here is an example of debugging the lm() function.

```
> debug(lm)      ## Flag the 'lm()' function for interactive debugging

> lm(y ~ x)

debugging in: lm(y ~ x)

debug: {

   ret.x <- x

   ret.y <- y

   cl <- match.call()

   ...

   if (!qr)

      z$qr <- NULL

   z
```

```
}
```

➢ 20 Simulation

1 Generating Random Numbers

Simulation is an important (and big) topic for both statistics and for a variety of other areas where there is a need to introduce randomness. Sometimes you want to implement a statistical procedure that requires random number generation or sampling (i.e. Markov chain Monte Carlo, the bootstrap, random forests, bagging) and sometimes you want to simulate a system and random number generators can be used to model random inputs.

R comes with a set of pseudo-random number generators that allow you to simulate from well-known probability distributions like the Normal, Poisson, and binomial. Some example functions for probability distributions in R

- `rnorm`: generate random Normal variates with a given mean and standard deviation
- `dnorm`: evaluate the Normal probability density (with a given mean/SD) at a point (or vector of points)
- `pnorm`: evaluate the cumulative distribution function for a Normal distribution
- `rpois`: generate random Poisson variates with a given rate

For each probability distribution there are typically four functions available that start with a “r”, “d”, “p”, and “q”. The “r” function is the one that actually simulates randon numbers from that distribution. The other functions are prefixed with a

- `d` for density
- `r` for random number generation
- `p` for cumulative distribution
- `q` for quantile function (inverse cumulative distribution)

If you’re only interested in simulating random numbers, then you will likely only need the “r” functions and not the others. However, if you

intend to simulate from arbitrary probability distributions using something like rejection sampling, then you will need the other functions too.

Probably the most common probability distribution to work with the is the Normal distribution (also known as the Gaussian). Working with the Normal distributions requires using these four functions

```
dnorm(x, mean = 0, sd = 1, log = FALSE)

pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)

qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)

rnorm(n, mean = 0, sd = 1)
```

Here we simulate standard Normal random numbers with mean 0 and standard deviation 1.

```
> ## Simulate standard Normal random numbers

> x <- rnorm(10)

> x

 [1]  0.01874617 -0.18425254 -1.37133055 -0.59916772  0.29454513
0.38979430

 [7] -1.20807618 -0.36367602 -1.62667268 -0.25647839
```

We can modify the default parameters to simulate numbers with mean 20 and standard deviation 2.

```
> x <- rnorm(10, 20, 2)

> x

 [1] 22.20356 21.51156 19.52353 21.97489 21.48278 20.17869 18.09011
19.60970

 [9] 21.85104 20.96596

> summary(x)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.

  18.09   19.75   21.22   20.74   21.77   22.20
```

If you wanted to know what was the probability of a random Normal variable of being less than, say, 2, you could use the pnorm() function to do that calculation.

```
> pnorm(2)
[1] 0.9772499
```

You never know when that calculation will come in handy.

20.2 Setting the random number seed

When simulating any random numbers it is essential to set the *random number seed*. Setting the random number seed with set.seed() ensures reproducibility of the sequence of random numbers.

For example, I can generate 5 Normal random numbers with rnorm().

```
> set.seed(1)
> rnorm(5)
[1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078
```

Note that if I call rnorm() again I will of course get a different set of 5 random numbers.

```
> rnorm(5)
[1] -0.8204684  0.4874291  0.7383247  0.5757814 -0.3053884
```