MALLA REDDY INSTITUTE OF TECHNOLOGY & SCIENCE

(SPONSORED BY MALLA REDDY EDUCATIONAL SOCIETY)

Permanently Affiliated to JNTUH & Approved by AICTE, New Delhi

NBA Accredited Institution, An ISO 9001:2015 Certified, Approved by UK Accreditation Centre

Granted Status of 2(f) & 12(b) under UGC Act. 1956, Govt. of India.

INTRODUCTION TO DATA SCIENCE

COURSE FILE

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING

(DATA SCIENCE)

(2022-2023)

Faculty In-Charge	**HOD-CSE**
R. Sony	**Dr. T. Srikanth**

INTRODUCTION TO DATA SCIENCE

B.Tech. III Year I Sem.

L	T	P	C
3	0	0	3

Course Objectives:

1. Learn concepts, techniques and tools they need to deal with various facets of data science practice, including data collection and integration
2. Understand the basic types of data and basic statistics
3. Identify the importance of data reduction and data visualization techniques

Course Outcomes: After completion of the course, the student should be able to

1. Understand basic terms what Statistical Inference means.
2. Identify probability distributions commonly used as foundations for statistical modelling. Fit a model to data
3. describe the data using various statistical measures
4. utilize R elements for data handling
5. perform data reduction and apply visualization techniques.

UNIT - I

Introduction: Definition of Data Science- Big Data and Data Science hype – and getting past the hype - Datafication - Current landscape of perspectives - Statistical Inference - Populations and samples - Statistical modeling, probability distributions, fitting a model – Over fitting. **Basics of R:** Introduction, R-Environment Setup, Programming with R, Basic Data Types.

UNIT - II

Data Types & Statistical Description

Types of Data: Attributes and Measurement, What is an Attribute? The Type of an Attribute, The Different Types of Attributes, Describing Attributes by the Number of Values, Asymmetric Attributes, Binary Attribute, Nominal Attributes, Ordinal Attributes, Numeric Attributes, Discrete versus Continuous Attributes. Basic Statistical Descriptions of Data: Measuring the Central Tendency: Mean, Median, and Mode, Measuring the Dispersion of Data: Range, Quartiles, Variance, Standard Deviation, and Inter-quartile Range, Graphic Displays of Basic Statistical Descriptions of Data.

UNIT - III

Vectors: Creating and Naming Vectors, Vector Arithmetic, Vector sub setting, **Matrices:** Creating and Naming Matrices, Matrix Sub setting, Arrays, Class. **Factors and Data Frames:** Introduction to Factors: Factor Levels, Summarizing a Factor, Ordered Factors, Comparing Ordered Factors, Introduction to Data Frame, subsetting of Data Frames, Extending Data Frames, Sorting Data Frames.
Lists: Introduction, creating a List: Creating a Named List, Accessing List Elements, Manipulating List Elements, Merging Lists, Converting Lists to Vectors

UNIT - IV

Conditionals and Control Flow: Relational Operators, Relational Operators and Vectors, Logical Operators, Logical Operators and Vectors, Conditional Statements. **Iterative Programming in R:** Introduction, While Loop, For Loop, Looping Over List. **Functions in R:** Introduction, writing a Function in R, Nested Functions, Function Scoping, Recursion, Loading an R Package, Mathematical Functions in R.

UNIT - V

Data Reduction: Overview of Data Reduction Strategies, Wavelet Transforms, Principal Components Analysis, Attribute Subset Selection, Regression and Log-Linear Models: Parametric Data Reduction, Histograms, Clustering, Sampling, Data Cube Aggregation. **Data Visualization:** Pixel-Oriented

Visualization Techniques, Geometric Projection Visualization Techniques, Icon-Based Visualization Techniques, Hierarchical Visualization Techniques, Visualizing Complex Data and Relations.

TEXT BOOKS:

1. Doing Data Science, Straight Talk from The Frontline. Cathy O'Neil and Rachel Schutt, O'Reilly, 2014
2. Jiawei Han, Micheline Kamber and Jian Pei. Data Mining: Concepts and Techniques, 3rd ed. The Morgan Kaufmann Series in Data Management Systems.
3. K G Srinivas, G M Siddesh, "Statistical programming in R", Oxford Publications.

REFERENCE BOOKS:

1. Introduction to Data Mining, Pang-Ning Tan, Vipin Kumar, Michael Steinbanch, Pearson Education.
2. Brain S. Everitt, "A Handbook of Statistical Analysis Using R", Second Edition, 4 LLC, 2014.
3. Dalgaard, Peter, "Introductory statistics with R", Springer Science & Business Media, 2008.
4. Paul Teetor, "R Cookbook", O'Reilly, 2011.

UNIT-1

Introduction: What Is 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. It's increasingly critical to businesses: The insights that data science generates help organizations increase operational efficiency, identify new business opportunities and improve marketing and sales programs, among other benefits. Ultimately, they can lead to competitive advantages over business rivals.

Big Data and Data Science Hype

Data Science	Big Data
Data Science is an area.	Big Data is a technique to collect, maintain and process huge information.
It is about the collection, processing, analyzing, and utilizing of data in various operations. It is more conceptual.	It is about extracting vital and valuable information from a huge amount of data.
It is a field of study just like Computer Science, Applied Statistics, or Applied Mathematics.	It is a technique for tracking and discovering trends in complex data sets.
The goal is to build data-dominant products for a venture.	The goal is to make data more vital and usable i.e. by extracting only important information from the huge data within existing traditional aspects.
Tools mainly used in Data Science include SAS, R, Python, etc	Tools mostly used in Big Data include Hadoop, Spark, Flink, etc.
It is a superset of Big Data as data science consists of Data scrapping, cleaning, visualization, statistics, and many more techniques.	It is a sub-set of Data Science as mining activities which is in a pipeline of Data science.
It is mainly used for scientific purposes.	It is mainly used for business purposes and customer satisfaction.

It broadly focuses on the science of the data.	It is more involved with the processes of handling voluminous data.

So, what is eyebrow-raising about Big Data and data science? Let's count the ways:

1. There's a lack of definitions around the most basic terminology. What is "Big Data" anyway? What does "data science" mean? What is the relationship between Big Data and data science? Is data science the science of Big Data? Is data science only the stuff going on in companies like Google and Facebook and tech companies? Why do many people refer to Big Data as crossing disciplines (astronomy, finance, tech, etc.) and to data science as only taking place in tech? Just how *big* is big? Or is it just a relative term? These terms are so ambiguous, they're well-nigh meaningless.
2. There's a distinct lack of respect for the researchers in academia and industry labs who have been working on this kind of stuff for years, and whose work is based on decades (in some cases, centuries) of work by statisticians, computer scientists, mathematicians, engineers, and scientists of all types. From the way the media describes it, machine learning algorithms were just invented last week and data was never "big" until Google came along. This is simply not the case. Many of the methods and techniques we're using—and the challenges we're facing now—are part of the evolution of everything that's come before. This doesn't mean that there's not new and exciting stuff going on, but we think it's important to show some basic respect for everything that came before.
3. The hype is crazy—people throw around tired phrases straight out of the height of the pre-financial crisis era like "Masters of the Universe" to describe data scientists, and that doesn't bode well. In general, hype masks reality and increases the noise-to-signal ratio. The longer the hype goes on, the more many of us will get turned off by it, and the harder it will be to see what's good underneath it all, if anything.
4. Statisticians already feel that they are studying and working on the "Science of Data." That's their bread and butter. Maybe you, dear reader, are not a statistician and don't care, but imagine that for the statistician, this feels a little bit like how identity theft might feel for you. Although we will make the case that data science is *not* just a rebranding of statistics or machine learning but rather a field unto itself, the media often describes data science in a way that makes it sound like as if it's simply statistics or machine learning in the context of the tech industry.

People have said to us, “Anything that has to call itself a science isn’t.” Although there might be truth in there, that doesn’t mean that the term “data science” *itself* represents nothing, but of course what it represents may not be science but more of a craft.

Getting Past the Hype

Rachel’s experience going from getting a PhD in statistics to working at Google is a great example to illustrate why we thought, in spite of the aforementioned reasons to be dubious, there might be some meat in the data science sandwich. In her words:

It was clear to me pretty quickly that the stuff I was working on at Google was different than anything I had learned at school when I got my PhD in statistics. This is not to say that my degree was useless; far from it—what I’d learned in school provided a framework and way of thinking that I relied on daily, and much of the actual content provided a solid theoretical and practical foundation necessary to do my work.

But there were also many skills I had to acquire on the job at Google that I hadn’t learned in school. Of course, my experience is specific to me in the sense that I had a statistics background and picked up more computation, coding, and visualization skills, as well as domain expertise while at Google. Another person coming in as a computer scientist or a social scientist or a physicist would have different gaps and would fill them in accordingly. But what is important here is that, as individuals, we each had different strengths and gaps, yet we were able to solve problems by putting ourselves together into a data team well-suited to solve the data problems that came our way.

Here’s a reasonable response you might have to this story. It’s a general truism that, whenever you go from school to a real job, you realize there’s a gap between what you learned in school and what you do on the job. In other words, you were simply facing the difference between academic statistics and industry statistics.

We have a couple replies to this:

- Sure, there’s is a difference between industry and academia. But does it really have to be that way? Why do many courses in school have to be so intrinsically out of touch with reality?

- Even so, the gap doesn't represent simply a difference between industry statistics and academic statistics. The general experience of data scientists is that, at their job, they have access to a *larger body of knowledge and methodology*, as well as a process, which we now define as the *data science process* (details in Chapter 2), that has foundations in both statistics and computer science.

Around all the hype, in other words, there is a ring of truth: this *is* something new. But at the same time, it's a fragile, nascent idea at real risk of being rejected prematurely. For one thing, it's being paraded around as a magic bullet, raising unrealistic expectations that will surely be disappointed.

Rachel gave herself the task of understanding the cultural phenomenon of data science and how others were experiencing it. She started meeting with people at Google, at startups and tech companies, and at universities, mostly from within statistics departments.

From those meetings she started to form a clearer picture of the new thing that's emerging. She ultimately decided to continue the investigation by giving a course at Columbia called "Introduction to Data Science," which Cathy covered on her blog.

Datafication

In the May/June 2013 issue of Foreign Affairs, Kenneth Neil Cukier and Viktor Mayer-Schoenberger wrote an article called "The Rise of Big Data". In it they discuss the concept of datafication, and their example is how we quantify friendships with "likes": it's the way everything we do, online or otherwise, ends up recorded for later examination in someone's data storage units. Or maybe multiple storage units, and maybe also for sale.

They define datafication as a process of "taking all aspects of life and turning them into data." As examples, they mention that "Google's augmented-reality glasses datafy the gaze. Twitter datafies stray thoughts. LinkedIn datafies professional networks."

Datafication is an interesting concept and led us to consider its importance with respect to people's intentions about sharing their own data. We are being datafied, or rather our actions are, and when we "like" someone or something online, we are intending to be datafied, or at least we should expect to be. But when we merely browse the Web, we are unintentionally, or at least passively, being datafied through cookies that we might or might not be aware of. And when we walk around in a store, or even on the street, we are being datafied in a completely unintentional way, via sensors, cameras, or Google glasses.

This spectrum of intentionality ranges from us gleefully taking part in a social media experiment we are proud of, to all-out surveillance and stalking. But it's all datafication. Our intentions may run the gamut, but the results don't.

They follow up their definition in the article with a line that speaks volumes about their perspective:

Once we datafy things, we can transform their purpose and turn the information into new forms of value.

Here's an important question that we will come back to throughout the book: who is "we" in that case? What kinds of value do they refer to? Mostly, given their examples, the "we" is the modelers and entrepreneurs making money from getting people to buy stuff, and the "value" translates into something like increased efficiency through automation.

The Current Landscape

So, what is data science? Is it new, or is it just statistics or analytics rebranded? Is it real, or is it pure hype? And if it's new and if it's real, what does that mean?

This is an ongoing discussion, but one way to understand what's going on in this industry is to look online and see what current discussions are taking place. This doesn't necessarily tell us what data science is, but it at least tells us what other people think it is, or how they're perceiving it. For example, on Quora there's a discussion from 2010 about "What is Data Science?" and here's Metamarket CEO Mike

Driscoll's answer:

Data science, as it's practiced, is a blend of Red-Bull-fueled hacking and espresso-inspired statistics. But data science is not merely hacking—because when hackers finish debugging their Bash one-liners and Pig scripts, few of them care about non-Euclidean distance metrics. And data science is not merely statistics, because when statisticians finish theorizing the perfect model, few could read a tab-delimited file into R if their job depended on it. Data science is the civil engineering of data. Its acolytes possess a practical knowledge of tools and materials, coupled with a theoretical understanding of what's possible.

Driscoll then refers to Drew Conway's Venn diagram of data science from 2010, shown in Figure 1-1.

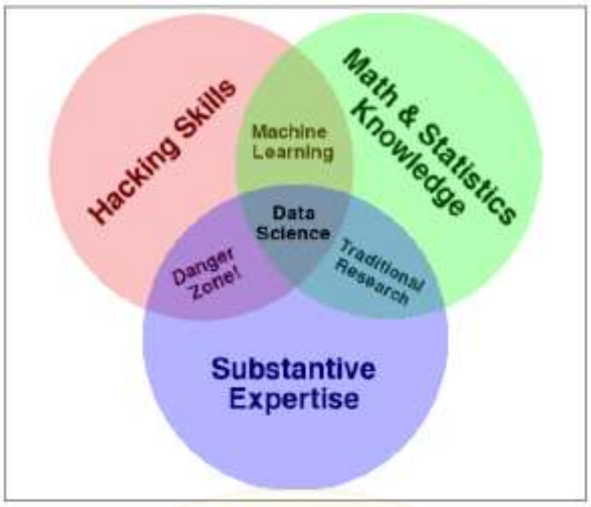


Figure 1-1. Drew Conway's Venn diagram of data science

He also mentions the sexy skills of data geeks from Nathan Yau's 2009 post, "Rise of the Data Scientist", which include:

• Statistics (traditional analysis you're used to thinking about)

• Data munging (parsing, scraping, and formatting data)

• Visualization (graphs, tools, etc.)

But wait, is data science just a bag of tricks? Or is it the logical extension of other fields like statistics and machine learning? For one argument, see Cosma Shalizi's posts here and here, and Cathy's posts here and here, which constitute an ongoing discussion of the difference between a statistician and a data scientist. Cosma basically argues that any statistics department worth its salt does all the stuff in the descriptions of data science that he sees, and therefore data science is just a rebranding and unwelcome takeover of statistics. For a slightly different perspective, see ASA President Nancy Geller's 2011 Amstat News article, "Don't shun the 'S' word", in which she defends statistics:

We need to tell people that Statisticians are the ones who make sense of the data deluge occurring in science, engineering, and medicine; that statistics provides methods for data analysis in all fields, from art history to zoology; that it is exciting to be a Statistician in the 21st century because of the many challenges brought about by the data explosion in all of these fields.

Though we get her point—the phrase "art history to zoology" is supposed to represent the concept of A to Z—she's kind of shooting herself in the foot with these examples because they don't correspond to the high-tech world where much of the data explosion is coming from. Much of the development of the field is happening in industry, not academia. That is, there are people with the job title data scientist in companies, but no professors of data science in academia. (Though this may be changing.)

Not long ago, DJ Patil described how he and Jeff Hammerbacher— then at LinkedIn and Facebook, respectively—coined the term "data scientist" in 2008. So that is when "data scientist" emerged as a job title. (Wikipedia finally gained an entry on data science in 2012.) It makes sense to us that once the skill set required to thrive at Google —working with a team on problems that required a hybrid skill set of stats and computer science paired with personal characteristics including curiosity and persistence—spread to other Silicon Valley tech companies, it required a new job title. Once it became a pattern, it deserved a name. And once it got a name, everyone and their mother wanted to be one. It got even worse when Harvard Business Review declared data scientist to be the "Sexiest Job of the 21st Century".

The Role of the Social Scientist in Data Science

Both LinkedIn and Facebook are social network companies. Oftentimes a description or definition of data scientist includes hybrid statistician, software engineer, and social scientist. This made sense in the context of companies where the product was a social product and still makes sense when we're dealing with human or user behavior. But if you think about Drew Conway's Venn diagram, data science problems cross disciplines—that's what the substantive expertise is referring to. In other words, it depends on the context of the problems you're trying to solve.

If they're social science-y problems like friend recommendations or people you know or user segmentation, then by all means, bring on the social scientist! Social scientists also do tend to be good question askers and have other good investigative qualities, so a social scientist who also has the quantitative and programming chops makes a great data scientist. But it's almost a "historical" (historical is in quotes because 2008 isn't that long ago) artifact to limit your conception of a data scientist to someone who works only with online user behavior data.

There's another emerging field out there called computational social sciences, which could be thought of as a subset of data science. But we can go back even further. In 2001, William Cleveland wrote a position paper about data science called "Data Science: An action plan to expand the field of statistics." So data science existed before data scientists? Is this semantics, or does it make sense? This all begs a few questions: can you define data science by what data scientists do? Who gets to define the field, anyway? There's lots of buzz and hype—does the media get to define it, or should we rely on the practitioners, the self-appointed data scientists? Or is there some actual authority? Let's leave these as open questions for now, though we will return to them throughout the book.

Data Science Jobs

Columbia just decided to start an Institute for Data Sciences and Engineering with Bloomberg's help. There are 465 job openings in New York City alone for data scientists last time we checked. That's a lot. So even if data science isn't a real field, it has *real* jobs. And here's one thing we noticed about most of the job descriptions: they ask data scientists to be experts in computer science, statistics, communication, data visualization, *and* to have extensive domain expertise. Nobody is an expert in everything, which is why it makes more sense to create teams of people who have different profiles and different expertise—together, as a team, they can specialize in all those things. We'll talk about this more after we look at the composite set of skills in demand for today's data scientists.

Statistical Inference

The world we live in is complex, random, and uncertain. At the same time, it's one big data-generating machine. As we commute to work on subways and in cars, as our blood moves through our bodies, as we're shopping, emailing, procrastinating at work by browsing the Internet and watching the stock market, as we're building things, eating things, talking to our friends and family about things, while factories are producing products, this all at least potentially produces data.

Imagine spending 24 hours looking out the window, and for every minute, counting and recording the number of people who pass by. Or gathering up everyone who lives within a mile of your house and making them tell you how many email messages they receive every day for the next year. Imagine heading over to your local hospital and rummaging around in the blood samples looking for patterns in the DNA. That all sounded creepy, but it wasn't supposed to. The point here is that the processes in our lives are actually data-generating processes.

We'd like ways to describe, understand, and make sense of these processes, in part because as scientists we just want to understand the world better, but many times, understanding these processes is part of the solution to problems we're trying to solve. Data represents the traces of the real-world processes, and exactly which traces we gather are decided by our data collection or sampling method. You, the data scientist, the observer, are turning the world into data, and this is an utterly subjective, not objective, process. After separating the process from the data collection, we can see clearly that there are two sources of randomness and uncertainty. Namely, the randomness and uncertainty underlying the process itself, and the uncertainty associated with your underlying data collection methods. Once you have all this data, you have somehow captured the world, or certain traces of the world. But you can't go walking around with a huge Excel spreadsheet or database of millions of transactions and look at it and, with a snap of a finger, understand the world and process that generated it. So you need a new idea, and that's to simplify those captured traces into something more comprehensible, to something that somehow captures it all in a much more concise way, and that something could be mathematical models or functions of the data, known as statistical estimators.

This overall process of going from the world to the data, and then from the data back to the world, is the field of statistical inference. More precisely, statistical inference is the discipline that concerns itself with the development of procedures, methods, and theorems that allow us to extract meaning and information from data that has been generated by stochastic (random) processes.

Populations and Samples

Let’s get some terminology and concepts in place to make sure we’re all talking about the same thing.In classical statistical literature, a distinction is made between the population and the sample. The word population immediately makes us think of the entire US population of 300 million people, or the entire world’s population of 7 billion people. But put that image out of your head, because in statistical inference population isn’t used to simply describe only people. It could be any set of objects or units, such as tweets or photographs or stars. If we could measure the characteristics or extract characteristics of all those objects, we’d have a complete set of observations, and the convention is to use N to represent the total number of observations in the population. Suppose your population was all emails sent last year by employees at a huge corporation, BigCorp. Then a single observation could be a list of things: the sender’s name, the list of recipients, date sent, text of email, number of characters in the email, number of sentences in the email, number of verbs in the email, and the length of time until first reply.

When we take a sample, we take a subset of the units of size n in order to examine the observations to draw conclusions and make inferences about the population. There are different ways you might go about getting this subset of data, and you want to be aware of this sampling mechanism because it can introduce biases into the data, and distort it, so that the subset is not a “mini-me” shrunk-down version of the population. Once that happens, any conclusions you draw will simply be wrong and distorted.

In the BigCorp email example, you could make a list of all the employees and select 1/10th of those people at random and take all the email they ever sent, and that would be your sample. Alternatively, you could sample 1/10th of all email sent each day at random, and that would be your sample. Both these methods are reasonable, and both methods yield the same sample size. But if you took them and counted how many email messages each person sent, and used that to estimate the underlying distribution of emails sent by all indiviuals at BigCorp, you might get entirely different answers. So if even getting a basic thing down like counting can get distorted when you’re using a reasonable-sounding sampling method, imagine what can happen to more complicated algorithms and models if you haven’t taken into account the process that got the data into your hands.

Modeling

In the next chapter, we’ll look at how we build models from the data we collect, but first we want to discuss what we even mean by this term. Rachel had a recent phone conversation with someone about a modelling workshop, and several minutes into it she realized the word “model” meant completely different things to them. He was using it to mean data models—the representation one is choosing to store one’s data, which is the realm of database managers—whereas she was talking about statistical models, which is what much of this book is about.

One of Andrew Gelman's blog posts on modeling was recently tweeted by people in the fashion industry, but that's a different issue. Even if you've used the terms statistical model or mathematical model for years, is it even clear to yourself and to the people you're talking to what you mean? What makes a model a model? Also, while we're asking fundamental questions like this, what's the difference between a statistical model and a machine learning algorithm? Before we dive deeply into that, let's add a bit of context with this deliberately provocative Wired magazine piece, "The End of Theory: The Data Deluge Makes the Scientific Method Obsolete," published in 2008 by Chris Anderson, then editor-in-chief. Anderson equates massive amounts of data to complete information and argues no models are necessary and "correlation is enough"; e.g., that in the context of massive amounts of data, "they [Google] don't have to settle for models at all."

Really? We don't think so, and we don't think you'll think so either by the end of the book. But the sentiment is similar to the Cukier and Mayer-Schoenberger article we just discussed about N=ALL, so you might already be getting a sense of the profound confusion we're witnessing all around us. To their credit, it's the press that's currently raising awareness of these questions and issues, and someone has to do it. Even so, it's hard to take when the opinion makers are people who don't actually work with data. Think critically about whether you buy what Anderson is saying; where you agree, disagree, or where you need more information to form an opinion. Given that this is how the popular press is currently describing and influencing public perception of data science and modeling, it's incumbent upon us as data scientists to be aware of it and to chime in with informed comments. With that context, then, what do we mean when we say models? And how do we use them as data scientists? To get at these questions, let's dive in.

What is a model?

Humans try to understand the world around them by representing it in different ways. Architects capture attributes of buildings through blueprints and three-dimensional, sca led-down versions. Molecular biologists capture protein structure with three-dimensional visualizations of the connections between amino acids. Statisticians and data scientists capture the uncertainty and randomness of data-generating processes with mathematical functions that express the shape and structure of the data itself. A model is our attempt to understand and represent the nature of reality through a particular lens, be it architectural, biological, or mathematical.

A model is an artificial construction where all extraneous detail has been removed or abstracted. Attention must always be paid to these abstracted details after a model has been analyzed to see what might have been overlooked. In the case of proteins, a model of the protein backbone with side chains by itself is removed from the laws of quantum mechanics that govern the behavior of the electrons, which ultimately dictate the structure and actions of proteins. In the case of a statistical model, we may have mistakenly excluded key variables, included irrelevant ones, or assumed a mathematical structure divorced from reality.

Statistical modelling

Before you get too involved with the data and start coding, it's useful to draw a picture of what you think the underlying process might be with your model. What comes first? What influences what? What causes what? What's a test of that? But different people think in different ways. Some prefer to express these kinds of relationships in terms of math. The mathematical expressions will be general enough that they have to include parameters, but the values of these parameters are not yet known. In mathematical expressions, the convention is to use Greek letters for parameters and Latin letters for data. So, for example, if you have two columns of data, x and y, and you think there's a linear relationship, you'd write down $y = \beta 0 + \beta 1x$. You don't know what $\beta 0$ and $\beta 1$ are in terms of actual numbers yet, so they're the parameters.

Other people prefer pictures and will first draw a diagram of data flow, possibly with arrows, showing how things affect other things or what happens over time. This gives them an abstract picture of the relationships before choosing equations to express them.

Probability distributions

Probability distributions are the foundation of statistical models. When we get to linear regression and Naive Bayes, you will see how this happens in practice. One can take multiple semesters of courses on probability theory, and so it's a tall challenge to condense it down for you in a small section.

Back in the day, before computers, scientists observed real-world phenomenon, took measurements, and noticed that certain mathematical shapes kept reappearing. The classical example is the height of humans, following a *normal* distribution—a bell-shaped curve, also called a Gaussian distribution, named after Gauss. Other common shapes have been named after their observers as well (e.g., the Poisson distribution and the Weibull distribution), while other shapes such as Gamma distributions or exponential distributions are named after associated mathematical objects.

Natural processes tend to generate measurements whose empirical shape could be approximated by mathematical functions with a few parameters that could be estimated from the data. Not *all* processes generate data that looks like a *named* distribution, but many do. We can use these functions as building blocks of our models. It's beyond the scope of the book to go into each of the distributions in detail, but we provide them in Figure 2-1 as an illustration of the various common shapes, and to remind you that they only have names because someone observed them enough times to think they deserved names. There is actually an infinite number of possible distributions. They are to be interpreted as assigning a *probability* to a subset of possible outcomes, and have corresponding functions. For example, the normal distribution is written as:

$$N(x|\mu,\sigma) \sim \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

The parameter μ is the mean and median and controls where the distribution is centered (because this is a symmetric distribution), and the parameter σ controls how spread out the distribution is. This is the general functional form, but for specific real-world phenomenon, these parameters have actual numbers as values, which we can estimate from the data.

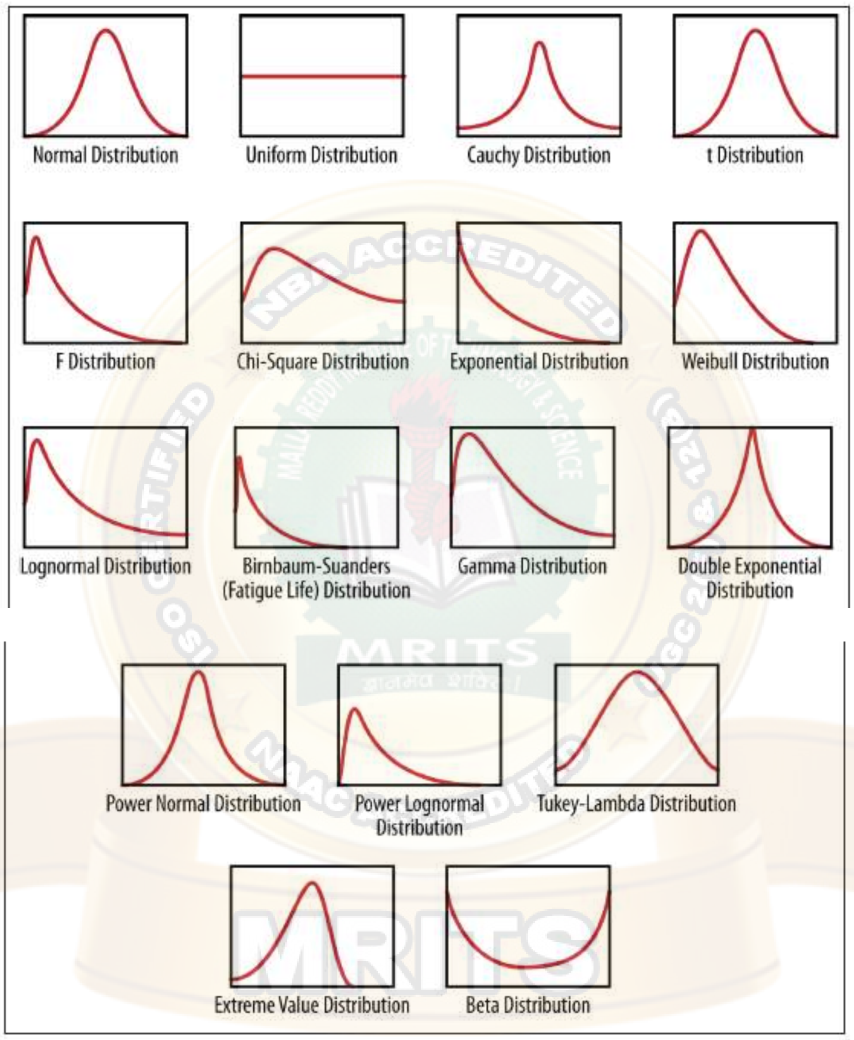


Figure 2-1. A bunch of continuous density functions (aka probability distributions)

A random variable denoted by x or y can be assumed to have a corresponding probability distribution, p(x), which maps x to a positive real number. In order to be a probability density function, were restricted to the set of functions such that if we integrate p(x) to get the area under the curve, it is 1, so it can be interpreted as probability.

In addition to denoting distributions of single random variables with functions of one variable, we use multivariate functions called joint distributions to do the same thing for more than one random variable. So in the case of two random variables, for example, we could denote our distribution by a function p(x, y), and it would take values in the plane and give us nonnegative values. In keeping with its interpretation as a probability, its (double) integral over the whole plane would be 1.

We also have what is called a conditional distribution, p(x|y) , which is to be interpreted as the density function of x given a particular value of y.

When we observe data points, i.e., (x1, y1), (x2, y2), . . ., (xn, yn), we are observing realizations of a pair of random variables. When we have an entire dataset with n rows and k columns, we are observing n realizations of the joint distribution of those k random variables.

Fitting a model

Fitting a model means that you estimate the parameters of the model using the observed data. You are using your data as evidence to help approximate the real-world mathematical process that generated the data. Fitting the model often involves optimization methods and algorithms, such as maximum likelihood estimation, to help get the parameters.

Fitting the model is when you start actually coding: your code will read n the data, and you'll specify the functional form that you wrote down on the piece of paper. Then R or Python will use built-in optimization methods to give you the most likely values of the parameters given the data. As you gain sophistication, or if this is one of your areas of expertise, you'll dig around in the optimization methods yourself. Initially you should have an understanding that optimization is taking place and how it works, but you don't have to code this part yourself—it underlies the R or Python functions.

Overfitting

Throughout the book you will be cautioned repeatedly about overfitting, possibly to the point you will have nightmares about it. Overfitting is the term used to mean that you used a dataset to estimate the parameters of your model, but your model isn't that good at capturing reality beyond your sampled data. You might know this because you have tried to use it to predict labels for another set of data that you didn't use to fit the model, and it doesn't do a good job, as measured by an evaluation metric such as accuracy.

Basics of R

Introduction

- R is a programming language and software environment for statistical analysis, graphics representation and reporting.
- R was created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and is currently developed by the R Development Core Team.
- R is freely available under the GNU General Public License, and pre-compiled binary versions are provided for various operating systems like Linux, Windows and Mac.

- This programming language was named **R**, based on the first letter of first name of the two R authors (Robert Gentleman and Ross Ihaka), and partly a play on the name of the Bell Labs Language **S**.
- The core of R is an interpreted computer language which allows branching and looping as well as modular programming using functions.
- R allows integration with the procedures written in the C, C++, .Net, Python or FORTRAN languages for efficiency.

Evolution of R

R was initially written by **Ross Ihaka** and **Robert Gentleman** at the Department of Statistics of the University of Auckland in Auckland, New Zealand. R made its first appearance in 1993.

- A large group of individuals has contributed to R by sending code and bug reports.
- Since mid-1997 there has been a core group (the "R Core Team") who can modify the R source code archive.

Features of R

As stated earlier, R is a programming language and software environment for statistical analysis, graphics representation and reporting. The following are the important features of R –

- R is a well-developed, simple and effective programming language which includes conditionals, loops, user defined recursive functions and input and output facilities.
- R has an effective data handling and storage facility,
- R provides a suite of operators for calculations on arrays, lists, vectors and matrices.
- R provides a large, coherent and integrated collection of tools for data analysis.
- R provides graphical facilities for data analysis and display either directly at the computer or printing at the papers.

R - Environment Setup

1. **Installation of R and RStudio in Windows.**

In Linux: (Through Terminal)

- Press Ctrl+Alt+T to open **Terminal**
- Then execute **sudo apt-get update**
- After that, **sudo apt-get install r-base**

In Windows:

<u>Install R on windows</u>

Step – 1: Go to CRAN R project website. (**Comprehensive R Archive Network**)

Step – 2: Click on the **Download R for Windows** link.

Step – 3: Click on the **base** subdirectory link or **install R for the first time link.**
Step – 4: Click **Download R X.X.X for Windows (**X.X.X stand for the latest version of R. eg: 3.6.1) and save the executable .exe file.
Step – 5: Run the **.exe** file and follow the installation instructions.

5.a. Select the desired language and then click **Next**.
5.b. Read the license agreement and click **Next**.
5.c. Select the components you wish to install (it is recommended to install all the components). Click **Next**.
5.d. Enter/browse the folder/path you wish to install R into and then confirm by clicking **Next**.
5.e. Select additional tasks like creating desktop shortcuts etc. then click **Next**.
5.f. Wait for the installation process to complete.
5.g. Click on **Finish** to complete the installation.

Install RStudio on Windows

Step – 1: With R-base installed, let's move on to installing RStudio. To begin, go to download RStudio and click on the download button for **RStudio desktop**.
Step – 2: Click on the link for the windows version of RStudio and save the .exe file.
Step – 3: Run the .exe and follow the installation instructions.

3.a. Click **Next** on the welcome window.
3.b. Enter/browse the path to the installation folder and click **Next** to proceed.
3.c. Select the folder for the start menu shortcut or click on do not create shortcuts and then click **Next**.
3.d. Wait for the installation process to complete.
3.e. Click **Finish** to end the installation.

Programming with R

R - Basic Syntax

To output text in R, use single or double quotes:
```
"Hello World!"
```

To output numbers, just type the number (without quotes):
```
5
10
25
```

To do simple calculations, add numbers together:
```
2+3
```

Output:5

R Print Output

Print

Unlike many other programming languages, you can output code in R without using a print function:

Example

```
"Hello!"
```

However, R does have a print() function available if you want to use it.

Example

```
print("Hello!")
```

And there are times you must use the print() function to output code, for example

```
x<-10
print(x)
```

R Comments

Comments

Comments can be used to explain R code, and to make it more readable. It can also be used to prevent execution when testing alternative code.

Comments starts with a #. When executing code, R will ignore anything that starts with #.

This example uses a comment before a line of code:

Example

```
# This is a comment
"Hello!"
```

This example uses a comment at the end of a line of code:

Example

```
"Hello World!" # This is a comment
```

Multiline Comments

Unlike other programming languages, such as Java, there are no syntax in R for multiline comments. However, we can just insert a # for each line to create multiline comments:

Example

```
# This is a comment
# written in
# more than just one line
"Hello!"
```

R Variables

Creating Variables in R

Variables are containers for storing data values.

R does not have a command for declaring a variable. A variable is created the moment you first assign a value to it. To assign a value to a variable, use the <- sign. To output (or print) the variable value, just type the variable name:

Example

```
name <- "John"
age <- 40

name   # output "John"
age    # output 40
```

In other programming language, it is common to use = as an assignment operator. In R, we can use both = and <- as assignment operators.

However, <- is preferred in most cases because the = operator can be forbidden in some context in R.

Print / Output Variables

Compared to many other programming languages, you do not have to use a function to print/output variables in R. You can just type the name of the variable:

Example

```
name <- "John"

name # auto-print the value of the name variable
```

Example

```
name <- "John"

print(name) # print the value of the name variable
```

Concatenate Elements

You can also concatenate, or join, two or more elements, by using the **paste()** function.

To combine both text and a variable, R uses comma (,):

Example

```
text <- "awesome"

paste("R is", text)
```

Example

```
text1 <- "R is"
text2 <- "awesome"

paste(text1, text2)
```

For numbers, the + character works as a mathematical operator:

Example

```
num1 <- 5
num2 <- 10

num1 + num2
```

If you try to combine a string (text) and a number, R will give you an error:

Example

```
num <- 5
text <- "Some text"

num + text
```

Result:

```
Error in num + text : non-numeric argument to binary operator
```

Multiple Variables

R allows you to assign the same value to multiple variables in one line:

Example

```
# Assign the same value to multiple variables in one line
var1 <- var2 <- var3 <- "Orange"

# Print variable values
var1
var2
var3
```

Variable Names

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for R variables are:

- A variable name must start with a letter and can be a combination of letters, digits, period(.) and underscore(_).
- If it starts with period(.), it cannot be followed by a digit.
- A variable name cannot start with a number or underscore (_)
- Variable names are case-sensitive (age, Age and AGE are three different variables)
- Reserved words cannot be used as variables (TRUE, FALSE, NULL, if...)

```
# Legal variable names:
myvar <- "John"
my_var <- "John"
myVar <- "John"
MYVAR <- "John"
myvar2 <- "John"
.myvar <- "John"

# Illegal variable names:
2myvar <- "John"
my-var <- "John"
my var <- "John"
_my_var <- "John"
my_v@ar <- "John"
TRUE <- "John"
```

R Data Types

Variables can store data of different types, and different types can do different things.

In R, variables do not need to be declared with any particular type, and can even change type after they have been set:

Example

```
my_var <- 30
my_var <- "raghul"
```

Basic Data Types

Basic data types in R can be divided into the following types:

- numeric - (10.5, 55, 787)
- integer - (1L, 55L, 100L, where the letter "L" declares this as an integer)
- complex - (9 + 3i, where "i" is the imaginary part)
- character (string) - ("k", "R is exciting", "FALSE", "11.5")
- logical (boolean) - (TRUE or FALSE)

We can use the class() function to check the data type of a variable:

Example

```
# numeric
x <- 10.5
class(x)

# integer
x <- 1000L
class(x)

# complex
x <- 9i + 3
class(x)

# character/string
x <- "R is exciting"
class(x)

# logical/boolean
x <- TRUE
class(x)
```

R Numbers

Numbers

There are three number types in R:

- numeric
- integer
- complex

Variables of number types are created when you assign a value to them:

Example

```
x <- 10.5   # numeric
y <- 10L    # integer
z <- 1i     # complex
```

Numeric

A numeric data type is the most common type in R, and contains any number with or without a decimal, like: 10.5, 55, 787:

Example

```
x <- 10.5
y <- 55

# Print values of x and y
x
y

# Print the class name of x and y
class(x)
class(y)
```

Integer

Integers are numeric data without decimals. This is used when you are certain that you will never create a variable that should contain decimals. To create an integer variable, you must use the letter L after the integer value:

Example

```
x <- 1000L
y <- 55L

# Print values of x and y
x
y

# Print the class name of x and y
class(x)
class(y)
```

Complex

A complex number is written with an "i" as the imaginary part:

Example

```
x <- 3+5i
y <- 5i

# Print values of x and y
x
y

# Print the class name of x and y
class(x)
class(y)
```

Type Conversion

You can convert from one type to another with the following functions:

- as.numeric()
- as.integer()
- as.complex()

Example

```
x <- 1L # integer
y <- 2 # numeric

# convert from integer to numeric:
a <- as.numeric(x)
```

```
# convert from numeric to integer:
b <- as.integer(y)

# print values of x and y
x
y

# print the class name of a and b
class(a)
class(b)
```

UNIT-2

Types of Data

A data set can often be viewed as a collection of data objects. Other names for a data object are record, point, vector, pattern, event, case, sample, observation, or entity. In turn, data objects are described by a number of attributes that capture the basic characteristics of an object, such as the mass of a physical object or the time at which an event occurred. Other names for an attribute are variable, characteristic, field, feature, or dimension.

Example (Student Information). Often, a data set is a file, in which the objects are records (or rows) in the file and each field (or column) corresponds to an attribute. For example, Table 2.1 shows a data set that consists of student information. Each row corresponds to a student and each column is an attribute that describes some aspect of a student, such as grade point average (GPA) or identification number (ID).

Table 2.1. A sample data set containing student information.

Student ID	Year	Grade Point Average (GPA)	...
	⋮		
1034262	Senior	3.24	...
1052663	Sophomore	3.51	...
1082246	Freshman	3.62	...
	⋮		

Although record-based data sets are common, either in flat files or relational database systems, there are other important types of data sets and systems for storing data.

Attributes and Measurement

What is an attribute?

We start with a more detailed definition of an attribute.

Definition 1. An attribute is a property or characteristic of an object that may vary; either from one object to another or from one time to another.

For example, eye color varies from person to person, while the temperature of an object varies over time. Note that eye color is a symbolic attribute with a small number of possible values {brown, black, blue, green, hazel, etc.}, while temperature is a numerical attribute with a potentially unlimited number of values.

At the most basic level, attributes are not about numbers or symbols. However, to discuss and more precisely analyze the characteristics of objects, we assign numbers or symbols to them. To do this in a well-defined way, we need a measurement scale.

Definition 2. A measurement scale is a rule (function) that associates a numerical or symbolic value with an attribute of an object. Formally, the process of measurement is the application of a measurement scale to associate a value with a particular attribute of a specific object. While this may seem a bit abstract, we engage in the process of measurement all the time.
For instance, we step on a bathroom scale to determine our weight, we classify someone as male or female, or we count the number of chairs in a room to see if there will be enough to seat all the people coming to a meeting. In all these cases) the "physical value" of an attribute of an object is mapped to a numerical or symbolic value. With this background, we can now discuss the type of an attribute, a concept that is important in determining if a particular data analysis technique is consistent with a specific type of attribute.

The Type of an Attribute

In other words, the values used to represent an attribute may have properties that are not properties of the attribute itself, and vice versa. This is illustrated with two examples.

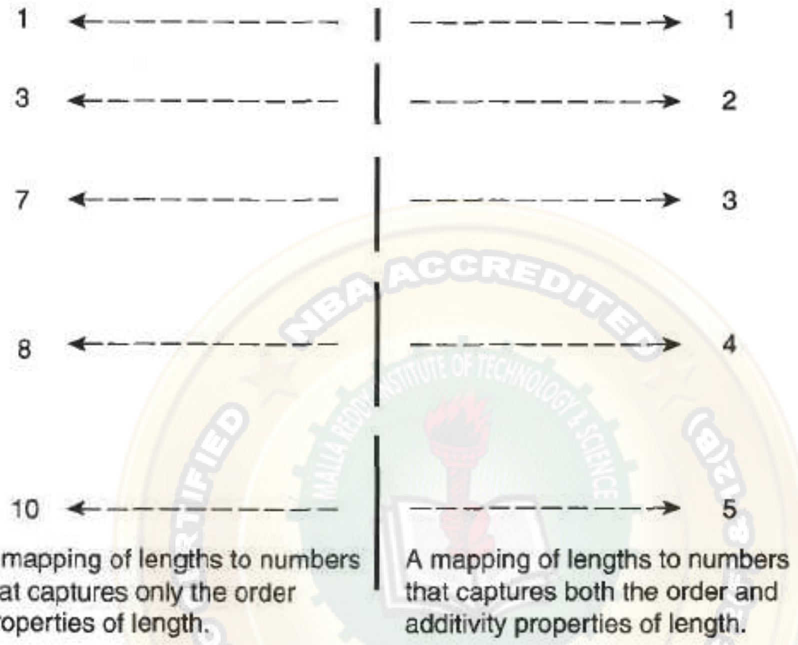
Example 1 (Employee Age and ID Number). Two attributes that might be associated with an employee are ID and age (in years). Both of these attributes can be represented as integers. However, while it is reasonable to talk about the average age of an employee, it makes no sense to talk about the average employee ID.

Indeed, the only aspect of employees that we want to capture with the ID attribute is that they are distinct. Consequently, the only valid operation for employee IDs is to test whether they are equal. There is no hint of this limitation, however, when integers are used to represent the employee ID attribute. For the age attribute, the properties of the integers used to represent age are very much the properties of the attribute. Even so, the correspondence is not complete since, for example, ages have a maximum' while integers do not.

Consider below Figure, which shows some objects-line segments and how the length attribute of these objects can be mapped to numbers in two different ways. Each successive line segment, going from the top to the bottom, is formed by appending the topmost line segment to itself. Thus, the second line segment from the top is formed by appending the topmost line segment to itself twice, the third line segment from the top is formed by appending the topmost line segment to itself three times, and so forth. In a very real (physical) sense, all the line segments are multiples of the first. This fact is captured by the measurements on the right-hand side of the figure, but not by those on the left hand-side.

More specifically, the measurement scale on the left-hand side captures only the ordering of the length attribute, while the scale on the right-hand side captures both the ordering and additivity properties. Thus, an attribute can be measured in a way that does not capture all the properties of the attribute. The type of an attribute should tell us what properties of the attribute are reflected in the values used to measure it. Knowing the type of an attribute is important because it tells us which properties of the measured values are consistent with the underlying properties of the attribute, and therefore, it allows us to avoid foolish actions, such as computing the average employee ID.

Note that it is common to refer to the type of an attribute as the type of a measurement scale.


1
3
7
8
10
1
2
3
4
5
A mapping of lengths to numbers that captures only the order properties of length.
A mapping of lengths to numbers that captures both the order and additivity properties of length.

The measurement of the length of line segments on two different scales of measurement.

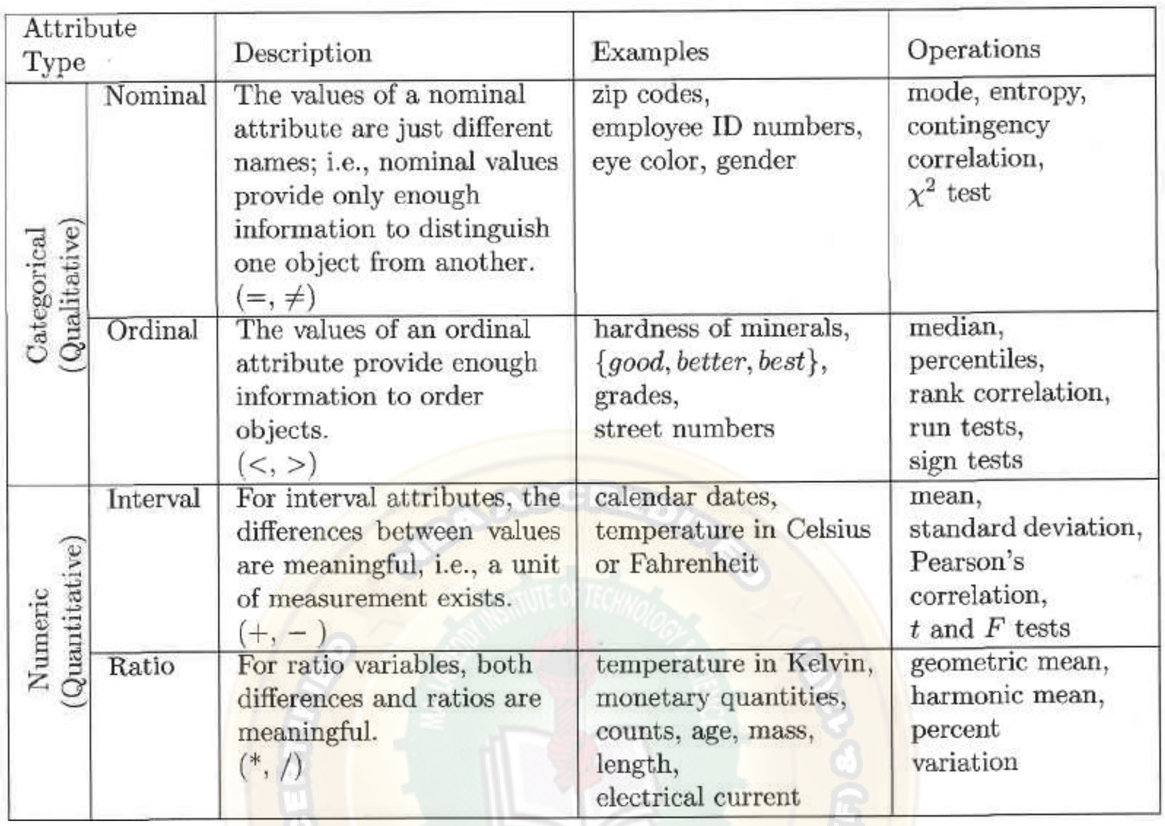
The Different Types of Attributes

A useful (and simple) way to specify the type of an attribute is to identify the properties of numbers that correspond to underlying properties of the attribute. For example, an attribute such as length has many of the properties of numbers. It makes sense to compare and order objects by length, as well as to talk about the differences and ratios of length. The following properties (operations) of numbers are typically used to describe attributes.

1. **Distinctness** $=$ and $\neq$
2. **Order** $<$, $\leq$, $>$, and $\geq$
3. **Addition** $+$ and $-$
4. **Multiplication** $*$ and $/$

Given these properties, we can define four types of attributes: nominal, ordinal, interval, and ratio. Table 2.2 gives the definitions of these types, along with information about the statistical operations that are valid for each type. Each attribute type possesses all of the properties and operations of the attribute types above it. Consequently, any property or operation that is valid for nominal, ordinal, and interval attributes is also valid for ratio attributes. In other words, the definition of the attribute types is cumulative. However, this does not mean that the operations appropriate for one attribute type are appropriate for the attribute types above it.

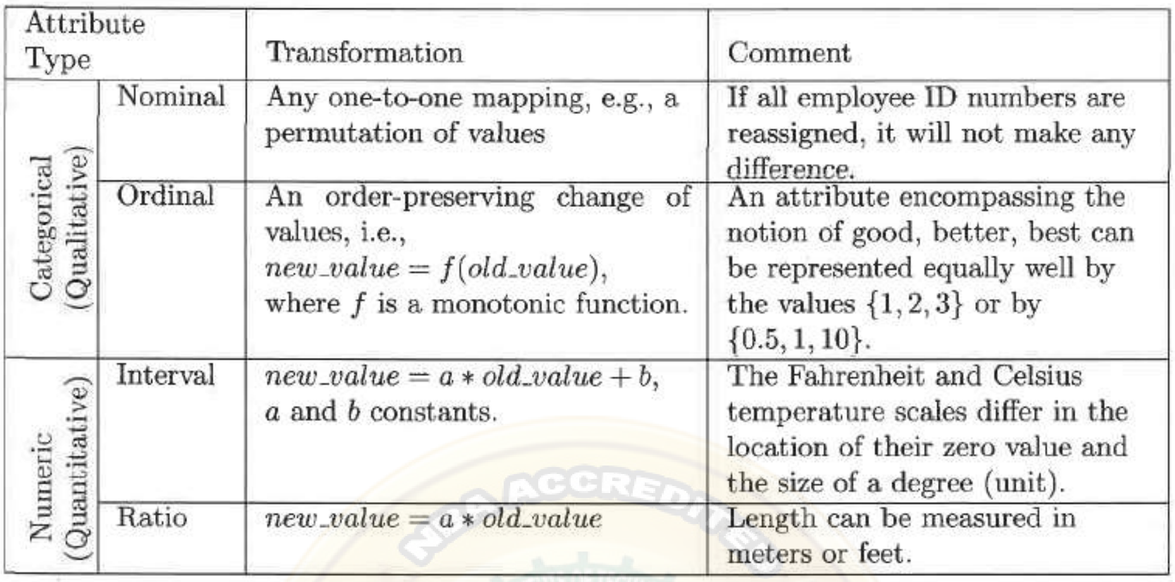
Table 2.2. Different attribute types.



Attribute Type		Description	Examples	Operations
Categorical (Qualitative)	Nominal	The values of a nominal attribute are just different names; i.e., nominal values provide only enough information to distinguish one object from another. ($=, \neq$)	zip codes, employee ID numbers, eye color, gender	mode, entropy, contingency correlation, χ^2 test
	Ordinal	The values of an ordinal attribute provide enough information to order objects. ($<, >$)	hardness of minerals, $\{good, better, best\}$, grades, street numbers	median, percentiles, rank correlation, run tests, sign tests
Numeric (Quantitative)	Interval	For interval attributes, the differences between values are meaningful, i.e., a unit of measurement exists. ($+, -$)	calendar dates, temperature in Celsius or Fahrenheit	mean, standard deviation, Pearson's correlation, t and F tests
	Ratio	For ratio variables, both differences and ratios are meaningful. ($*, /$)	temperature in Kelvin, monetary quantities, counts, age, mass, length, electrical current	geometric mean, harmonic mean, percent variation

Nominal and ordinal attributes are collectively referred to as categorical or qualitative attributes. As the name suggests, qualitative attributes, such as employee ID, lack most of the properties of numbers. Even if they are represented by numbers, i.e., integers, they should be treated more like symbols. The remaining two types of attributes, interval and ratio, are collectively referred to as quantitative or numeric attributes. Quantitative attributes are represented by numbers and have most of the properties of numbers. Note that quantitative attributes can be integer-valued or continuous. The types of attributes can also be described in terms of transformations that do not change the meaning of an attribute. Indeed, S. Smith Stevens, the psychologist who originally defined the types of attributes shown in Table 2.2, defined them in terms of these permissible transformations.

Table 2.3. Transformations that define attribute levels.



Attribute Type		Transformation	Comment
Categorical (Qualitative)	Nominal	Any one-to-one mapping, e.g., a permutation of values	If all employee ID numbers are reassigned, it will not make any difference.
	Ordinal	An order-preserving change of values, i.e., $new\_value = f(old\_value)$, where f is a monotonic function.	An attribute encompassing the notion of good, better, best can be represented equally well by the values $\{1, 2, 3\}$ or by $\{0.5, 1, 10\}$.
Numeric (Quantitative)	Interval	$new\_value = a * old\_value + b$, a and b constants.	The Fahrenheit and Celsius temperature scales differ in the location of their zero value and the size of a degree (unit).
	Ratio	$new\_value = a * old\_value$	Length can be measured in meters or feet.

For example, the meaning of a length attribute is unchanged if it is measured in meters instead of feet.

The statistical operations that make sense for a particular type of attribute are those that will yield the same results when the attribute is transformed using a transformation that preserves the attribute's meaning. To illustrate, the average length of a set of objects is different when measured in meters rather than in feet, but both averages represent the same length. Table 2.3 shows the permissible (meaning-preserving) transformations for the four attribute types of Table 2.2.

Example 2 (Temperature Scales). Temperature provides a good illustration of some of the concepts that have been described. First, temperature can be either an interval or a ratio attribute, depending on its measurement scale. When measured on the Kelvin scale, a temperature of 2^o is, in a physically meaningful way, twice that of a temperature of 1^o. This is not true when temperature is measured on either the Celsius or Fahrenheit scales, because, physically, a temperature of 1^o Fahrenheit (Celsius) is not much different than a temperature of 2" Fahrenheit (Celsius). The problem is that the zero points of the Fahrenheit and Celsius scales are, in a physical sense, arbitrary, and therefore, the ratio of two Celsius or Fahrenheit temperatures is not physically meaningful.

Describing Attributes by the Number of Values

An independent way of distinguishing between attributes is by the number of values they can take.

Discrete: A discrete attribute has a finite or countably infinite set of values. Such attributes can be categorical, such as zip codes or ID numbers, or numeric, such as counts. Discrete attributes are often represented using integer variables. Binary attributes are a special case of discrete attributes and assume only two values, e.g., true/false, yes/no, male/female, or 0 / 1. Binary attributes are often represented as Boolean variables, or as integer variables that only take the values 0 or 1.

Continuous: A continuous attribute is one whose values are real numbers. Examples include attributes such as temperature, height, or weight. Continuous attributes are typically represented as floating-point variables. Practically, real values can only be measured and represented with limited precision. In theory, any of the measurement scale types-nominal, ordinal, interval, and ratio could be combined with any of the types based on the number of attribute values-binary, discrete, and continuous. However, some combinations occur only infrequently or do not make much sense.
For instance, it is difficult to think of a realistic data set that contains a continuous binary attribute. Typically, nominal and ordinal attributes are binary or discrete, while interval and ratio attributes are continuous. However, count attributes, which are discrete, are also ratio attributes.

Asymmetric Attributes

For asymmetric attributes, only presence a non-zero attribute value-is regarded as important. Consider a data set where each object is a student and each attribute records whether or not a student took a particular course at a university. For a specific student, an attribute has a value of 1 if the student took the course associated with that attribute and a value of 0 otherwise. Because students take only a small fraction of all available courses, most of the values in such a data set would be 0. Therefore, it is more meaningful and more efficient to focus on the non-zero values.
To illustrate, if students are compared on the basis of the courses they don't take, then most students would seem very similar, at least if the number of courses is large. Binary attributes where only non-zero values are important are called asymmetric binary attributes. This type of attribute is particularly important for association analysis. It is also possible to have discrete or continuous asymmetric features. For instance, if the number of credits associated with each course is recorded, then the resulting data set will consist of asymmetric discrete or continuous attributes.

Binary Attributes

A **binary attribute** is a nominal attribute with only two categories or states: 0 or 1, where 0 typically means that the attribute is absent, and 1 means that it is present. Binary attributes are referred to as **Boolean** if the two states correspond to *true* and *false*.

Example 1 **Binary attributes.** Given the attribute *smoker* describing a *patient* object, 1 indicates that the patient smokes, while 0 indicates that the patient does not. Similarly, suppose the patient undergoes a medical test that has two possible outcomes. The attribute *medical test* is binary, where a value of 1 means the result of the test for the patient is positive, while 0 means the result is negative.

A binary attribute is **symmetric** if both of its states are equally valuable and carry the same weight; that is, there is no preference on which outcome should be coded as 0 or 1. One such example could be the attribute *gender* having the states *male* and *female*.

A binary attribute is **asymmetric** if the outcomes of the states are not equally important, such as the *positive* and *negative* outcomes of a medical test for HIV. By convention, we code the most important outcome, which is usually the rarest one, by 1 (e.g., *HIV positive*) and the other by 0 (e.g., *HIV negative*).

Nominal Attributes

Nominal means “relating to names.” The values of a **nominal attribute** are symbols or *names of things*. Each value represents some kind of category, code, or state, and so nominal attributes are also referred to as **categorical**. The values do not have any meaningful order. In computer science, the values are also known as *enumerations*.

Example 1 **Nominal attributes.** Suppose that *hair color* and *marital status* are two attributes describing *person* objects. In our application, possible values for *hair color* are *black, brown, blond, red, auburn, gray,* and *white*. The attribute *marital status* can take on the values *single, married, divorced,* and *widowed*. Both *hair color* and *marital status* are nominal attributes. Another example of a nominal attribute is *occupation,* with the values *teacher, dentist, programmer, farmer*, and so on.

Although we said that the values of a nominal attribute are symbols or “names of things,” it is possible to represent such symbols or “names” with numbers. With *hair color,* for instance, we can assign a code of 0 for *black,* 1 for *brown,* and so on.

Another example is *customor ID,* with possible values that are all numeric. However, in such cases, the numbers are not intended to be used quantitatively. That is, mathematical operations on values of nominal attributes are not meaningful. It makes no sense to subtract one customer ID number from another, unlike, say, subtracting an age value from another (where *age* is a numeric attribute). Even though a nominal attribute may have integers as values, it is not considered a numeric attribute because the integers are not meant to be used quantitatively.

Because nominal attribute values do not have any meaningful order about them and are not quantitative, it makes no sense to find the mean (average) value or median (middle) value for such an attribute, given a set of objects. One thing that is of interest, however, is the attribute’s most commonly occurring value. This value, known as the *mode,* is one of the measures of central tendency.

Ordinal Attributes

An **ordinal attribute** is an attribute with possible values that have a meaningful order or *ranking* among them, but the magnitude between successive values is not known.

Example 1 **Ordinal attributes.** Suppose that *drink size* corresponds to the size of drinks available at a fast-food restaurant. This nominal attribute has three possible values: *small, medium*, and *large*. The values have a meaningful sequence (which corresponds to increasing drink size); however, we cannot tell from the values *how much* bigger, say, a medium is than a large. Other examples of ordinal attributes include *grade* and *professional rank*. Professional ranks can be enumerated in a sequential order: for example, *assistant*, *associate*, and *full* for professors, and *private, private first class, specialist, corporal, and sergeant* for army ranks.

Ordinal attributes are useful for registering subjective assessments of qualities that cannot be measured objectively; thus ordinal attributes are often used in surveys for ratings. In one survey, participants were asked to rate how satisfied they were as customers. Customer satisfaction had the following ordinal categories: *0: very dissatisfied, 1: somewhat dissatisfied, 2: neutral, 3: satisfied*, and *4: very satisfied.*

Ordinal attributes may also be obtained from the discretization of numeric quantities by splitting the value range into a finite number of ordered categories as described in Chapter 3 on data reduction. The central tendency of an ordinal attribute can be represented by its mode and its median (the middle value in an ordered sequence), but the mean cannot be defined.

Note that nominal, binary, and ordinal attributes are *qualitative*. That is, they *describe* a feature of an object without giving an actual size or quantity. The values of such qualitative attributes are typically words representing categories. If integers are used, they represent computer codes for the categories, as opposed to measurable quantities (e.g., 0 for *small* drink size, 1 for *medium*, and 2 for *large*).

Numeric Attributes

A **numeric attribute** is *quantitative*; that is, it is a measurable quantity, represented in integer or real values. Numeric attributes can be *interval-scaled* or *ratio-scaled*.

Interval-Scaled Attributes

Interval-scaled attributes are measured on a scale of equal-size units. The values of interval-scaled attributes have order and can be positive, 0, or negative. Thus, in addition to providing a ranking of values, such attributes allow us to compare and quantify the *difference* between values.

Example 1 **Interval-scaled attributes.** A *temperature* attribute is interval-scaled. Suppose that we have the outdoor *temperature* value for a number of different days, where each day is an object. By ordering the values, we obtain a ranking of the objects with respect to *temperature*. In addition, we can quantify the difference between values. For example, a temperature of 20^oC is five degrees higher than a temperature of 15^oC. Calendar dates are another example. For instance, the years 2002 and 2010 are eight years apart.

Temperatures in Celsius and Fahrenheit do not have a true zero-point, that is, neither 0^oC nor 0” F indicates “no temperature.” (On the Celsius scale, for example, the unit of measurement is 1/100 of the difference between the melting temperature and the boiling temperature of water in atmospheric pressure.) Although we can compute the *difference* between temperature values, we cannot talk of one temperature value as being a *multiple* of another. Without a true zero, we cannot say, for instance, that 10^C is twice as warm as 5^C. That is, we cannot speak of the values in terms of ratios. Similarly, there is no true zero-point for calendar dates. (The year 0 does not correspond to the beginning of time.) This brings us to ratio-scaled attributes, for which a true zero-point exits.

Because interval-scaled attributes are numeric, we can compute their mean value, in addition to the median and mode measures of central tendency.

Ratio-Scaled Attributes

A **ratio-scaled attribute** is a numeric attribute with an inherent zero-point. That is, if a measurement is ratio-scaled, we can speak of a value as being a multiple (or ratio) of another value. In addition, the values are ordered, and we can also compute the difference between values, as well as the mean, median, and mode.

Example: **Ratio-scaled attributes.** Unlike temperatures in Celsius and Fahrenheit, the Kelvin (K) temperature scale has what is considered a true zero-point $(0°\mathrm{K} = -273.15°\mathrm{C})$ It is the point at which the particles that comprise matter have zero kinetic energy. Other examples of ratio-scaled attributes include *count* attributes such as *years of experience* (e.g., the objects are employees) and *number of words* (e.g., the objects are documents).

Additional examples include attributes to measure weight, height, latitude and longitude coordinates (e.g., when clustering houses), and monetary quantities (e.g., you are 100 times richer with $100 than with $1).

Discrete versus Continuous Attributes

The attributes are organized as nominal, binary, ordinal, and numeric types. The types are not mutually exclusive. Classification algorithms developed from the field of machine learning often talk of attributes as being either *discrete* or *continuous*. Each type may be processed differently.

A **discrete attribute** has a finite or countably infinite set of values, which may or may not be represented as integers. The attributes *hair color*, *smoker*, *medical test*, and *drink size* each have a finite number of values, and so are discrete. Note that discrete attributes may have numeric values, such as 0 and 1 for binary attributes or, the values 0 to 110 for the attribute *age*. An attribute is *countably infinite* if the set of possible values is infinite but the values can be put in a one-to-one correspondence with natural numbers. For example, the attribute *customer ID* is countably infinite. The number of customers can grow to infinity, but in reality, the actual set of values is countable (where the values can be put in one-to-one correspondence with the set of integers). Zip codes are another example.

If an attribute is not discrete, it is **continuous**. The terms *numeric attribute* and *continuous attribute* are often used interchangeably in the literature. (This can be confusing because, in the classic sense, continuous values are real numbers, whereas numeric values can be either integers or real numbers.) In practice, real values are represented using a finite number of digits. Continuous attributes are typically represented as floating-point variables.

Basic Statistical Descriptions of Data

For data pre-processing to be successful, it is essential to have an overall picture of your data. Basic statistical descriptions can be used to identify properties of the data and highlight which data values should be treated as noise or outliers.

This section discusses three areas of basic statistical descriptions. We start with *measures of central tendency*, which measure the location of the middle or center of a data distribution. Intuitively speaking, given an attribute, where do most of its values fall? In particular, we discuss the mean, median, mode, and midrange. In addition to assessing the central tendency of our data set, we also would like to have an idea of the *dispersion of the data*. That is, how are the data spread out? The most common data dispersion measures are the *range*, *quartiles*, and *interquartile range*; the *five-number summary* and *boxplots*; and the *variance* and *standard deviation* of the data. These measures are useful for identifying outliers.

Finally, we can use many graphic displays of basic statistical descriptions to visually inspect our data. Most statistical or graphical data presentation software packages include bar charts, pie charts, and line graphs. Other popular displays of data summaries and distributions include *quantile plots*, *quantile–quantile plots*, *histograms*, and *scatter plots*.

Measuring the Central Tendency: Mean, Median, and Mode

In this section, we look at various ways to measure the central tendency of data. Suppose that we have some attribute *X*, like *salary*, which has been recorded for a set of objects. Let *x*1, *x*2,… ,*xN* be the set of *N* observed values or *observations* for *X*. Here, these values may also be referred to as the data set (for *X*). If we were to plot the observations for *salary*, where would most of the values fall? This gives us an idea of the central tendency of the data. Measures of central tendency include the mean, median, mode, and midrange.

The most common and effective numeric measure of the "center" of a set of data is the *(arithmetic) mean*. Let *x*1,*x*2, … ,*xN* be a set of *N* values or *observations*, such as for some numeric attribute *X*, like *salary*. The **mean** of this set of values is

$$\bar{x} = \frac{\sum_{i=1}^{N} x_i}{N} = \frac{x_1 + x_2 + \cdots + x_N}{N}.$$

This corresponds to the built-in aggregate function, *average* (avg() in SQL), provided in relational database systems.

Example 1 **Mean.** Suppose we have the following values for *salary* (in thousands of dollars), shown in increasing order: 30, 36, 47, 50, 52, 52, 56, 60, 63, 70, 70, 110. Using **mean** Eqation, we have

$$\bar{x} = \frac{30+36+47+50+52+52+56+60+63+70+70+110}{12}$$
$$= \frac{696}{12} = 58.$$

Thus, the mean salary is $58,000.

Sometimes, each value *xi* in a set may be associated with a weight *wi* for *i* D 1,….,*N*. The weights reflect the significance, importance, or occurrence frequency attached to their respective values. In this case, we can compute

$$\bar{x} = \frac{\sum_{i=1}^{N} w_i x_i}{\sum_{i=1}^{N} w_i} = \frac{w_1 x_1 + w_2 x_2 + \cdots + w_N x_N}{w_1 + w_2 + \cdots + w_N}.$$

This is called the **weighted arithmetic mean** or the **weighted average**.

Although the mean is the single most useful quantity for describing a data set, it is not always the best way of measuring the center of the data. A major problem with the mean is its sensitivity to extreme (e.g., outlier) values. Even a small number of extreme values can corrupt the mean. For example, the mean salary at a company may be substantially pushed up by that of a few highly paid managers.

Similarly, the mean score of a class in an exam could be pulled down quite a bit by a few very low scores. To offset the effect caused by a small number of extreme values, we can instead use the **trimmed mean**, which is the mean obtained after chopping off values at the high and low extremes. For example, we can sort the values observed for *salary* and remove the top and bottom 2% before computing the mean. We should avoid trimming too large a portion (such as 20%) at both ends, as this can result in the loss of valuable information. For skewed (asymmetric) data, a better measure of the center of data is the **median**, which is the middle value in a set of ordered data values. It is the value that separates the higher half of a data set from the lower half.

In probability and statistics, the median generally applies to numeric data; however, we may extend the concept to ordinal data. Suppose that a given data set of *N* values for an attribute *X* is sorted in increasing order. If *N* is odd, then the median is the *middle value* of the ordered set. If *N* is even, then the median is not unique; it is the two middlemost values and any value in between. If *X* is a numeric attribute in this case, by convention, the median is taken as the average of the two middlemost values.

Example 2: **Median.** Let's find the median of the data from Example 1. The data are already sorted in increasing order. There is an even number of observations (i.e., 12); therefore, the median is not unique. It can be any value within the two middlemost values of 52 and 56 (that is, within the sixth and seventh values in the list). By convention, we assign the average of the two middlemost values as the median; that is, $\frac{52+56}{2} = \frac{108}{2} = 54.$ Thus, the median is $54,000. Suppose that we had only the first 11 values in the list. Given an odd number of values, the median is the middlemost value. This is the sixth value in this list, which has a value of $52,000.

The median is expensive to compute when we have a large number of observations. For numeric attributes, however, we can easily *approximate* the value. Assume that data are grouped in intervals according to their *xi* data values and that the frequency (i.e., number of data values) of each interval is known. For example, employees may be grouped according to their annual salary in intervals such as $10–20,000, $20–30,000, and so on. Let the interval that contains the median frequency be the *median interval*.

We can approximate the median of the entire data set (e.g., the median salary) by interpolation using the formula

$$median = L_1 + \left(\frac{N/2 - (\sum freq)_l}{freq_{median}}\right) width,$$

where *L*1 is the lower boundary of the median interval, *N* is the number of values in the entire data set, $(\sum freq)_l$ is the sum of the frequencies of all of the intervals that are lower than the median interval, $freq_{median}$ is the frequency of the median interval, and *width* is the width of the median interval.

The *mode* is another measure of central tendency. The **mode** for a set of data is the value that occurs most frequently in the set. Therefore, it can be determined for qualitative and quantitative attributes. It is possible for the greatest frequency to correspond to several different values, which results in more than one mode. Data sets with one, two, or three modes are respectively called **unimodal**, **bimodal**, and **trimodal**. In general, a data set with two or more modes is**multimodal**. At the other extreme, if each data value occurs only once, then there is no mode.

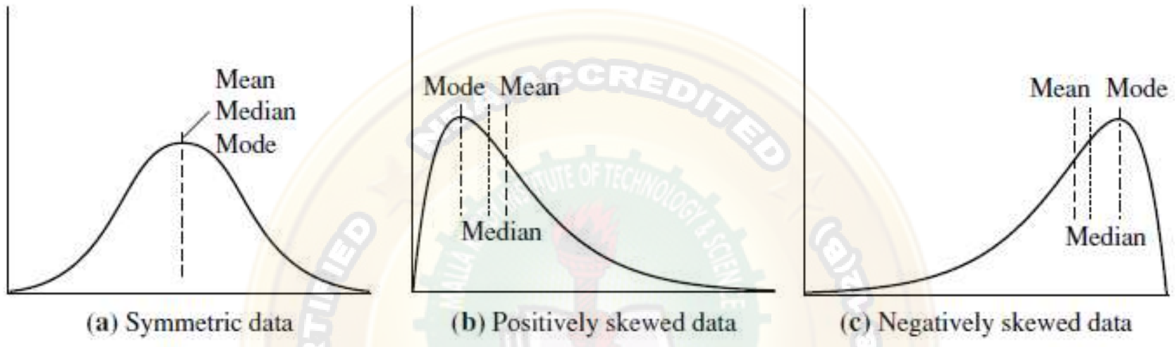
Example 3: **Mode.** The data from Example 1 are bimodal. The two modes are $52,000 and $70,000. For unimodal numeric data that are moderately skewed (asymmetrical), we have the following empirical relation:

$$mean - mode \approx 3 \times (mean - median).$$

This implies that the mode for unimodal frequency curves that are moderately skewed can easily be approximated if the mean and median values are known. The **midrange** can also be used to assess the central tendency of a numeric data set. It is the average of the largest and smallest values in the set. This measure is easy to compute using the SQL aggregate functions, max() and min().

Example 4: **Midrange.** The midrange of the data of Example 1 is $\frac{30{,}000+110{,}000}{2} = \$70{,}000.$

In a unimodal frequency curve with perfect **symmetric** data distribution, the mean, median, and mode are all at the same center value, as shown in Figure 2.1(a). Data in most real applications are not symmetric. They may instead be either **positively skewed**, where the mode occurs at a value that is smaller than the median (Figure 2.1b), or **negatively skewed**, where the mode occurs at a value greater than the median (Figure 2.1c).



Mean, median, and mode of symmetric versus positively and negatively skewed data.

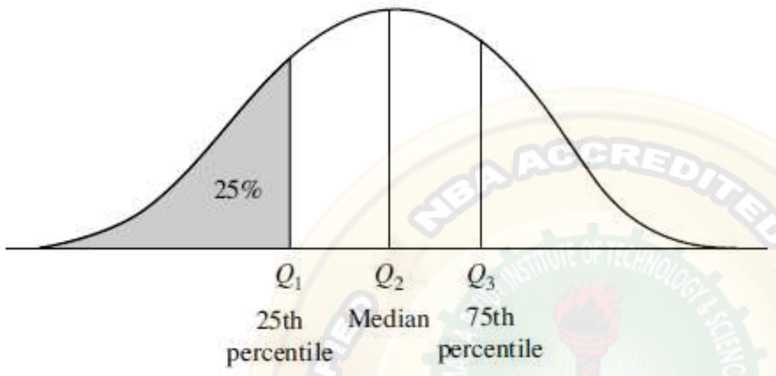
Measuring the Dispersion of Data: Range, Quartiles, Variance, Standard Deviation, and Interquartile Range

We now look at measures to assess the dispersion or spread of numeric data. The measures include range, quantiles, quartiles, percentiles, and the interquartile range. The five-number summary, which can be displayed as a boxplot, is useful in identifying outliers. Variance and standard deviation also indicate the spread of a data distribution.

Range, Quartiles, and Interquartile Range

To start off, let's study the *range*, *quantiles*, *quartiles*, *percentiles*, and the *interquartile range* as measures of data dispersion. Let $x1,x2, : : : ,xN$ be a set of observations for some numeric attribute, X. The **range** of the set is the difference between the largest (max()) and smallest (min()) values. Suppose that the data for attribute X are sorted in increasing numeric order. Imagine that we can pick certain data points so as to split the data distribution into equal-size consecutive sets, as in below Figure. These data points are called *quantiles*.

Quantiles are points taken at regular intervals of a data distribution, dividing it into essentially equal size consecutive sets. (We say "essentially" because there may not be data values of X that divide the data into exactly equal-sized subsets. For readability, we will refer to them as equal.) The k^{th} *q-quantile* for a given data distribution is the value x such that at most k=q of the data values are less than *x* and at most. (*q-k)=q* of the data values are more than *x*, where *k* is an integer such that $0 < k < q$. There are *q*-1 *q*-quantiles. The 2-quantile is the data point dividing the lower and upper halves of the data distribution. It corresponds to the median. The 4-quantiles are the three data points that split the data distribution into four equal parts; each part represents one-fourth of the data distribution. They are more commonly referred to as **quartiles**. The 100-quantiles are more commonly referred to as **percentiles**; they divide the data distribution into 100 equal-sized consecutive sets. The median, quartiles, and percentiles are the most widely used forms of quantiles.



A plot of the data distribution for some attribute *X*. The quantiles plotted are quartiles. The three quartiles divide the distribution into four equal-size consecutive subsets. The second quartile corresponds to the median.

The quartiles give an indication of a distribution's center, spread, and shape. The **first quartile**, denoted by *Q*1, is the 25th percentile. It cuts off the lowest 25% of the data. The **third quartile**, denoted by *Q*3, is the 75th percentile—it cuts off the lowest 75% (or highest 25%) of the data. The second quartile is the 50th percentile. As the median, it gives the center of the data distribution.

The distance between the first and third quartiles is a simple measure of spread that gives the range covered by the middle half of the data. This distance is called the **interquartile range** (**IQR**) and is defined as

$$IQR = Q_3 - Q_1.$$

Example 1: **Interquartile range.** The quartiles are the three values that split the sorted data set into four equal parts. The data contain 12 observations, already sorted in increasing order. Thus, the quartiles for this data are the third, sixth, and ninth values, respectively, in the sorted list. Therefore, *Q*1 = \$47,000 and *Q*3 is \$63,000. Thus, the interquartile range is *IQR* = 63 -47= \$16,000. (Note that the sixth value is a median, \$52,000, although this data set has two medians since the number of data values is even.)

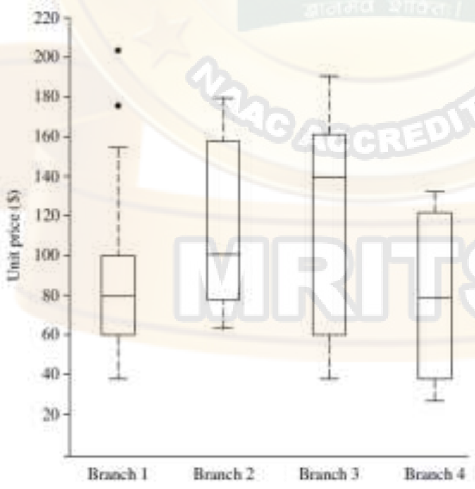
Five-Number Summary, Boxplots, and Outliers

No single numeric measure of spread (e.g., *IQR*) is very useful for describing skewed distributions. Have a look at the symmetric and skewed data distributions of below Figure. In the symmetric distribution, the median (and other measures of central tendency) splits the data into equal-size halves. This does not occur for skewed distributions. Therefore, it is more informative to also provide the two quartiles *Q*1 and *Q*3, along with the median. A common rule of thumb for identifying suspected **outliers** is to single out values falling at least 1.5X*IQR* above the third quartile or below the first quartile.

Because *Q*1, the median, and *Q*3 together contain no information about the endpoints (e.g., tails) of the data, a fuller summary of the shape of a distribution can be obtained by providing the lowest and highest data values as well. This is known as the *five-number summary*. The **five-number summary** of a distribution consists of the median (*Q*2), the quartiles *Q*1 and *Q*3, and the smallest and largest individual observations, written in the order of *Minimum*, *Q*1, *Median*, *Q*3, *Maximum*.

Boxplots are a popular way of visualizing a distribution. A boxplot incorporates the five-number summary as follows:

- Typically, the ends of the box are at the quartiles so that the box length is the interquartile range.
- The median is marked by a line within the box.
- Two lines (called *whiskers*) outside the box extend to the smallest (*Minimum*) and largest (*Maximum*) observations.



Boxplot for the unit price data for items sold at four branches of *AllElectronics* during a given time period.

When dealing with a moderate number of observations, it is worthwhile to plot potential outliers individually. To do this in a boxplot, the whiskers are extended to the extreme low and high observations *only if* these values are less than 1.5 X *IQR* beyond the quartiles. Otherwise, the whiskers terminate at the most extreme observations occurring within 1.5 X *IQR* of the quartiles. The remaining cases are plotted individually. Boxplots can be used in the comparisons of several sets of compatible data.

Example 2 **Boxplot.** Below Figure shows boxplots for unit price data for items sold at four branches of *All Electronics* during a given time period. For branch 1, we see that the median price of items sold is \$80, *Q*1 is \$60, and *Q*3 is \$100. Notice that two outlying observations for this branch were plotted individually, as their values of 175 and 202 are more than 1.5 times the IQR here of 40. Boxplots can be computed in *O*.(*n* log *n)* time. Approximate boxplots can be computed in linear or sublinear time depending on the quality guarantee required.

Variance and Standard Deviation

Variance and standard deviation are measures of data dispersion. They indicate how spread out a data distribution is. A low standard deviation means that the data observations tend to be very close to the mean, while a high standard deviation indicates that the data are spread out over a large range of values. The **variance** of *N* observations, *x*1,*x*2, … ,*xN*, for a numeric attribute *X* is

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2 = \left(\frac{1}{N}\sum_{i=1}^{N}x_i^2\right) - \bar{x}^2,$$

where *x'* is the mean value of the observations, as defined in Eq. (2.1). The **standard deviation**, σ, of the observations is the square root of the variance, σ^2.

Example 3: **Variance and standard deviation.** From previous example, we found *x'*=\$58,000 using mean equation. To determine the variance and standard deviation of the data from that example,

$$\sigma^2 = \frac{1}{12}(30^2 + 36^2 + 47^2 \ldots + 110^2) - 58^2$$
$$\approx 379.17$$
$$\sigma \approx \sqrt{379.17} \approx 19.47.$$

The basic properties of the standard deviation, σ, as a measure of spread are as follows:

σ measures spread about the mean and should be considered only when the mean is chosen as the measure of center. $\sigma = 0$ only when there is no spread, that is, when all observations have the same value. Otherwise, $\sigma > 0$.

Importantly, an observation is unlikely to be more than several standard deviations away from the mean. Mathematically, using Chebyshev's inequality, it can be shown that at least $\left(1 - \frac{1}{k^2}\right) \times 100\%$ of the observations are no more than *k* standard deviations from the mean.

Therefore, the standard deviation is a good indicator of the spread of a data set. The computation of the variance and standard deviation is scalable in large databases.

Graphic Displays of Basic Statistical Descriptions of Data

In this section, we study graphic displays of basic statistical descriptions. These include *quantile plots*, *quantile–quantile plots*, *histograms*, and *scatter plots*. Such graphs are helpful for the visual inspection of data, which is useful for data pre-processing. The first three of these show univariate distributions (i.e., data for one attribute), while scatter plots show bivariate distributions (i.e., involving two attributes).

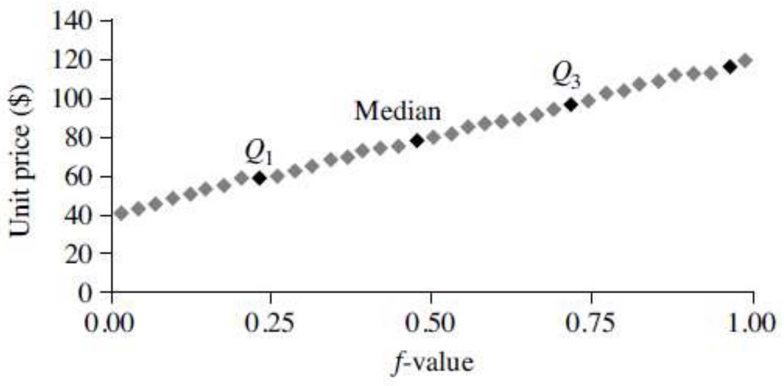
Quantile Plot

In this and the following subsections, we cover common graphic displays of data distributions. A **quantile plot** is a simple and effective way to have a first look at a univariate data distribution. First, it displays all of the data for the given attribute (allowing the user to assess both the overall behavior and unusual occurrences). Second, it plots quantile information. Let *xi* , for *i* = 1 to *N*, be the data sorted in increasing order so that *x*1 is the smallest observation and *xN* is the largest for some ordinal or numeric attribute *X*. Each observation, *xi* , is paired with a percentage, *fi* , which indicates that approximately *fi X* 100% of the data are below the value, *xi* .We say "approximately" because there may not be a value with exactly a fraction, *fi* , of the data below *xi* . Note that the 0.25 percentile corresponds to quartile *Q*1, the 0.50 percentile is the median, and the 0.75 percentile is *Q*3.

$$f_i = \frac{i - 0.5}{N}.$$

These numbers increase in equal steps of 1/*N*, ranging from $\frac{1}{2N}$ 1/2*N* (which is slightly above 0) to 1-$\frac{1}{2N}$ (which is slightly below 1). On a quantile plot, *xi* is graphed against *fi* . This allows us to compare different distributions based on their quantiles. For example, given the quantile plots of sales data for two different time periods, we can compare their *Q*1, median, *Q*3, and other *fi* values at a glance.

Example 2.13 Quantile plot. Figure 2.4 shows a quantile plot for the *unit price* data of Table 2.1.



A quantile plot for the unit price data of Table 2.1.

Quantile–Quantile Plot

A **quantile–quantile plot**, or **q-q plot**, graphs the quantiles of one univariate distribution against the corresponding quantiles of another. It is a powerful visualization tool in that it allows the user to view whether there is a shift in going from one distribution to another. Suppose that we have two sets of observations for the attribute or variable *unit price*, taken from two different branch locations. Let *x*1, … ,*xN* be the data from the first branch, and *y*1, …. , *yM* be the data from the second, where each data set is sorted in increasing order. If *M* = *N* (i.e., the number of points in each set is the same), then we simply plot *yi* against *xi* , where *yi* and *xi* are both .(*i* -0.5)/*N* quantiles of their respective data sets. If *M* < *N* (i.e., the second branch has fewer observations than the first), there can be only *M* points on the q-q plot. Here, *yi* is the .(*i* -0.5)=*M* quantile of the *y* data, which is plotted against the .(*i* -0.5)=*M* quantile of the *x* data. This computation typically involves interpolation.

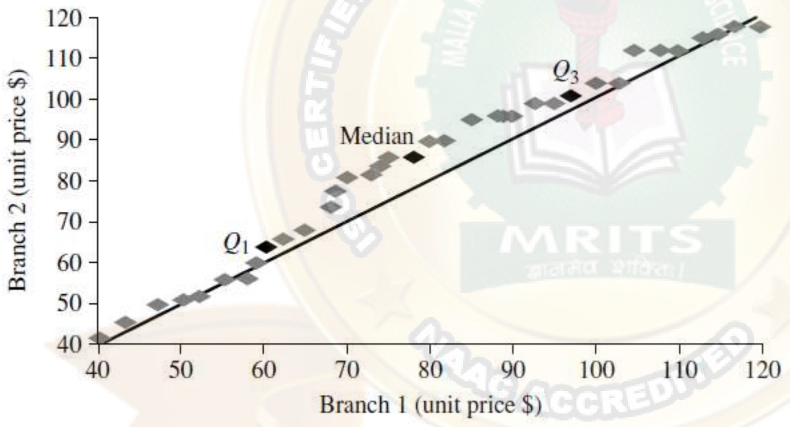
Example 1: **Quantile–quantile plot.** Below figure shows a quantile–quantile plot for *unit price* data of items sold at two branches of *AllElectronics* during a given time period. Each point corresponds to the same quantile for each data set and shows the unit price of items sold at branch 1 versus branch 2 for that quantile. (To aid in comparison, the straight line represents the case where, for each given quantile, the unit price at each branch is the same.

The darker points correspond to the data for *Q*1, the median, and *Q*3, respectively.) We see, for example, that at *Q*1, the unit price of items sold at branch 1 was slightly less than that at branch 2. In other words, 25% of items sold at branch 1 were less than or equal to $60, while 25% of items sold at branch 2 were less than or equal to $64. At the 50th percentile (marked by the median, which is also *Q*2), we see that 50% of items sold at branch 1 were less than $78, while 50% of items at branch 2 were less than $85.

In general, we note that there is a shift in the distribution of branch 1 with respect to branch 2 in that the unit prices of items sold at branch 1 tend to be lower than those at branch 2.

A Set of Unit Price Data for Items Sold at a Branch of *AllElectronics*

Unit price ($)	*Count of items sold*
40	275
43	300
47	250
—	—
74	360
75	515
78	540
—	—
115	320
117	270
120	350



A q-q plot for unit price data from two *AllElectronics* branches.

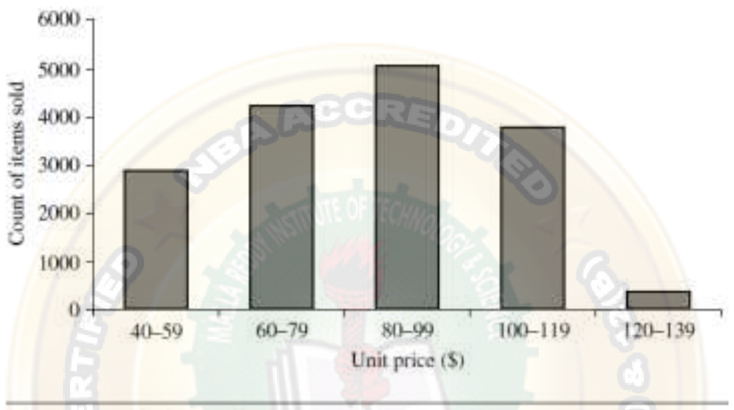
Histograms

Histograms (or **frequency histograms**) are at least a century old and are widely used. “Histos” means pole or mast, and “gram” means chart, so a histogram is a chart of poles. Plotting histograms is a graphical method for summarizing the distribution of a given attribute, *X*. If *X* is nominal, such as *automobile model* or *item type*, then a pole or vertical bar is drawn for each known value of *X*. The height of the bar indicates the frequency (i.e., count) of that *X* value. The resulting graph is more commonly known as a **bar chart**.

If *X* is numeric, the term *histogram* is preferred. The range of values for *X* is partitioned into disjoint consecutive subranges. The subranges, referred to as *buckets* or *bins*, are disjoint subsets of the data distribution for *X*. The range of a bucket is known as the **width**. Typically, the buckets are of equal width. For example, a *price* attribute with a value range of $1 to $200 (rounded up to the nearest dollar) can be partitioned into subranges 1 to 20, 21 to 40, 41 to 60, and so on. For each subrange, a bar is drawn with a height that represents the total count of items observed within the subrange. Histograms and partitioning rules are further discussed in Chapter 3 on data reduction.

Example 1: **Histogram.** Below figure shows a histogram for the data set of Table 2.1, where buckets (or bins) are defined by equal-width ranges representing $20 increments and the frequency is the count of items sold.

Although histograms are widely used, they may not be as effective as the quantile plot, q-q plot, and boxplot methods in comparing groups of univariate observations.

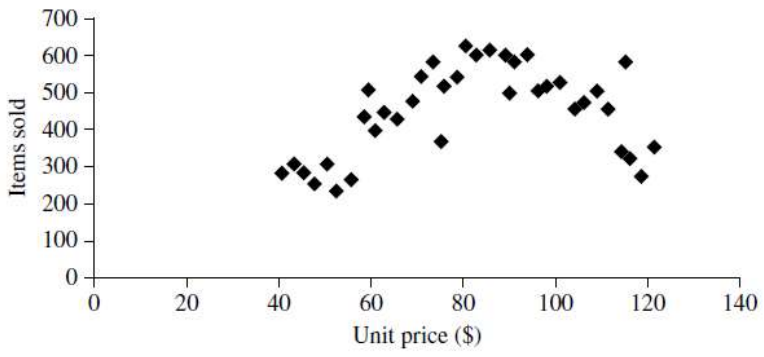


A histogram for the Table 2.1 data set.

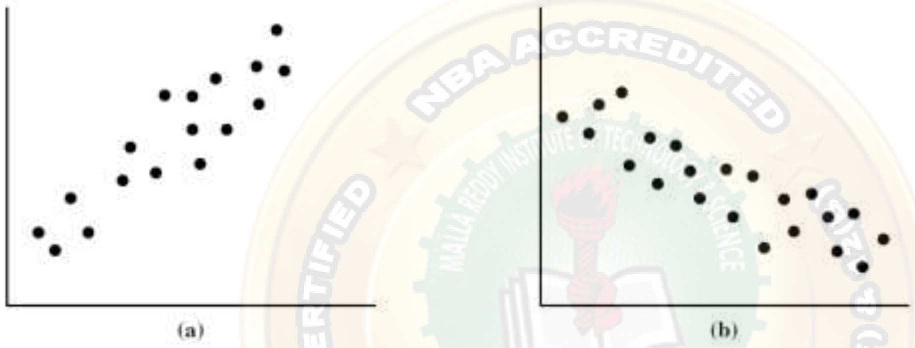
Scatter Plots and Data Correlation

A **scatter plot** is one of the most effective graphical methods for determining if there appears to be a relationship, pattern, or trend between two numeric attributes. To construct a scatter plot, each pair of values is treated as a pair of coordinates in an algebraic sense and plotted as points in the plane. Below figure shows a scatter plot for the set of data in Table 2.1.

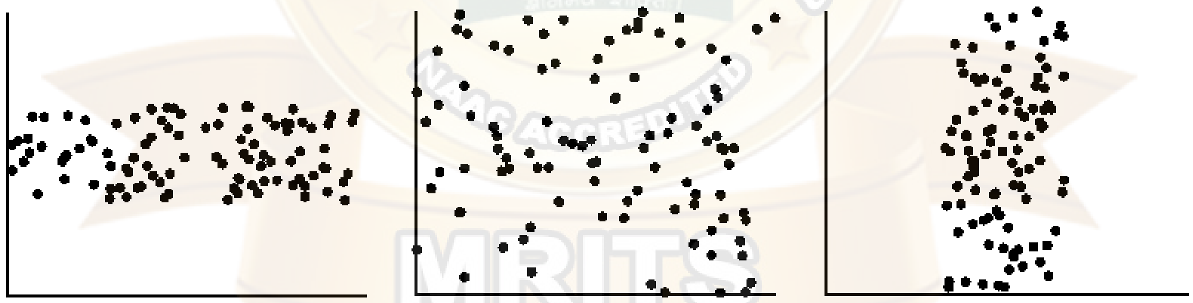
The scatter plot is a useful method for providing a first look at bivariate data to see clusters of points and outliers, or to explore the possibility of correlation relationships. Two attributes, *X*, and *Y*, are **correlated** if one attribute implies the other. Correlations can be positive, negative, or null (uncorrelated). Below figure shows examples of positive and negative correlations between two attributes.



A scatter plot for the Table 2.1 data set.



Scatter plots can be used to find (a) positive or (b) negative correlations between attributes.



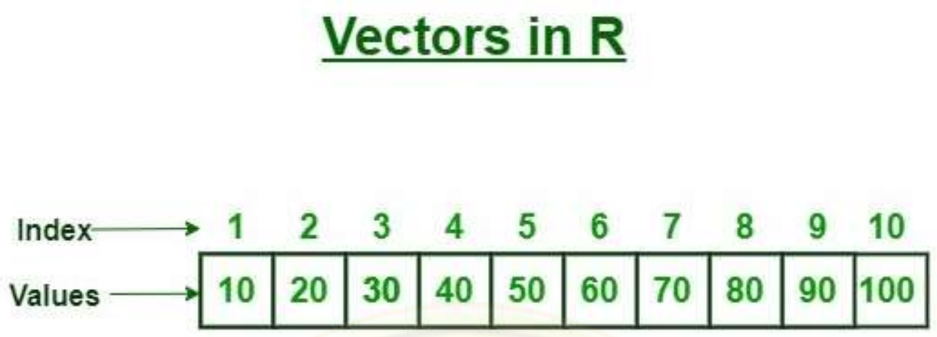
Three cases where there is no observed correlation between the two plotted attributes in each of the data sets.

If the plotted points pattern slopes from lower left to upper right, this means that the values of *X* increase as the values of *Y* increase, suggesting a *positive correlation*. If the pattern of plotted points slopes from upper left to lower right, the values of *X* increase as the values of *Y* decrease, suggesting a *negative correlation*. A line of best fit can be drawn to study the correlation between the variables. The above figure shows three cases for which there is no correlation relationship between the two attributes in each of the given data sets.

In conclusion, basic data descriptions (e.g., measures of central tendency and measures of dispersion) and graphic statistical displays (e.g., quantile plots, histograms, and scatter plots) provide valuable insight into the overall behavior of your data. By helping to identify noise and outliers, they are especially useful for data cleaning.

R - Vectors

Vectors in R are the same as the arrays in C language which are used to hold multiple data values of the same type. One major key point is that in R the indexing of the vector will start from ‘1’ and not from ‘0’. We can create numeric vectors and character vectors as well.



Vector Creation

Single Element Vector

Even when you write just one value in R, it becomes a vector of length 1 and belongs to one of the above vector types.

```
# Atomic vector of type character.

print("abc");

# Atomic vector of type double.

print(12.5)

# Atomic vector of type integer.

print(63L)

# Atomic vector of type logical.

print(TRUE)

# Atomic vector of type complex.

print(2+3i)

# Atomic vector of type raw.

print(charToRaw('hello'))
```

Multiple Elements Vector

Using colon operator with numeric data

```
# Creating a sequence from 5 to 13.

v <- 5:13

print(v)



# Creating a sequence from 6.6 to 12.6.

v <- 6.6:12.6

print(v)

# If the final element specified does not belong to the sequence then it is discarded.

v <- 3.8:11.4

print(v)
```

Using sequence (Seq.) operator

```
# Create vector with elements from 5 to 9 incrementing by 0.4.

print(seq(5, 9, by = 0.4))
```

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

```
[1] 5.0 5.4 5.8 6.2 6.6 7.0 7.4 7.8 8.2 8.6 9.0
```

Using the c() function

The non-character values are coerced to character type if one of the elements is a character.

```
# The logical and numeric values are converted to characters.

s <- c('apple','red',5,TRUE)

print(s)
```

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

```
[1] "apple" "red"   "5"     "TRUE"
```

Accessing Vector Elements

Elements of a Vector are accessed using indexing. The [ ] brackets are used for indexing. Indexing starts with position 1. Giving a negative value in the index drops that element from result.TRUE, FALSE or 0 and 1 can also be used for indexing.

```
# Accessing vector elements using position.

t <- c("Sun","Mon","Tue","Wed","Thurs","Fri","Sat")

u <- t[c(2,3,6)]

print(u)


# Accessing vector elements using logical indexing.

v <- t[c(TRUE,FALSE,FALSE,FALSE,FALSE,TRUE,FALSE)]

print(v)


# Accessing vector elements using negative indexing.

x <- t[c(-2,-5)]

print(x)


# Accessing vector elements using 0/1 indexing.

y <- t[c(0,0,0,0,0,0,1)]

print(y)
```

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

```
[1] "Mon" "Tue" "Fri"

[1] "Sun" "Fri"

[1] "Sun" "Tue" "Wed" "Fri" "Sat"

[1] "Sun"
```

Vector Manipulation

Vector arithmetic

Two vectors of same length can be added, subtracted, multiplied or divided giving the result as a vector output.

```
# Create two vectors.

v1 <- c(3,8,4,5,0,11)

v2 <- c(4,11,0,8,1,2)


# Vector addition.

add.result <- v1+v2

print(add.result)


# Vector subtraction.

sub.result <- v1-v2

print(sub.result)


# Vector multiplication.

multi.result <- v1*v2

print(multi.result)


# Vector division.

divi.result <- v1/v2

print(divi.result)
```

Vector Element Recycling

If we apply arithmetic operations to two vectors of unequal length, then the elements of the shorter vector are recycled to complete the operations.

```
v1 <- c(3,8,4,5,0,11)

v2 <- c(4,11)

# V2 becomes c(4,11,4,11,4,11)
```

```
add.result <- v1+v2
print(add.result)

sub.result <- v1-v2
print(sub.result)
```

Vector Element Sorting

Elements in a vector can be sorted using the sort() function.

```
v <- c(3,8,4,5,0,11, -9, 304)

# Sort the elements of the vector.
result <- sort(v)
print(result)

# Sort the elements in the reverse order.
result <- sort(v, decreasing = TRUE)
print(result)

# Sorting character vectors.
v <- c("Red","Blue","yellow","violet")
result <- sort(v)
print(result)

# Sorting character vectors in reverse order
result <- sort(v, decreasing = TRUE)
print(result)
```

Types of vectors

Vectors are of different types which are used in R. Following are some of the types of vectors:

- **Numeric vectors**
  Numeric vectors are those which contain numeric values such as integer, float, etc.

```
# R program to create numeric Vectors
  # creation of vectors using c() function.
v1 <- c(4, 5, 6, 7)
  # display type of vector
typeof(v1)




# by using 'L' we can specify that we want integer values.
v2 <- c(1L, 4L, 2L, 5L)
  # display type of vector
typeof(v2)
```

Output:

```
[1] "double"

[1] "integer"
```

- **Character vectors**
  Character vectors contain alphanumeric values and special characters.

```
# R program to create Character Vectors
  # by default numeric values
# are converted into characters
v1 <- c('geeks', '2', 'hello', 57)
  # Displaying type of vector
typeof(v1)
```

Output:

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

- **Logical vectors**
  Logical vectors contain boolean values such as TRUE, FALSE and NA for Null values.

```
# R program to create Logical Vectors

  # Creating logical vector

# using c() function

v1 <- c(TRUE, FALSE, TRUE, NA)
```

```
  # Displaying type of vector

typeof(v1)
```

- **Output:**
  [1] "logical"

Modifying a vector
Modification of a Vector is the process of applying some operation on an individual element of a vector to change its value in the vector. There are different ways through which we can modify a vector:

```
X <- c(2, 7, 9, 7, 8, 2)

  # modify a specific element

X[3] <- 1

X[2] <-9

cat('subscript operator', X, '\n')

 # Modify using different logics.

X[X>5] <- 0

cat('Logical indexing', X, '\n')

 # Modify by specifying

# the position or elements.

X <- X[c(3, 2, 1)]

cat('combine() function', X)
```

Output

```
subscript operator 2 9 1 7 8 2
```

Logical indexing 2 0 1 0 0 2

combine() function 1 0 2

Deleting a vector
Deletion of a Vector is the process of deleting all of the elements of the vector. This can be done by assigning it to a NULL value.

```
M <- c(8, 10, 2, 5)
# set NULL to the vector
M <- NULL
cat('Output vector', M)
```

Output:
```
Output vector NULL
```

Sorting elements of a Vector
sort() function is used with the help of which we can sort the values in ascending or descending order.

```
# R program to sort elements of a Vector



# Creation of Vector

X <- c(8, 2, 7, 1, 11, 2)



# Sort in ascending order

A <- sort(X)

cat('ascending order', A, '\n')

# sort in descending order

# by setting decreasing as TRUE

B <- sort(X, decreasing = TRUE)
```

```
  cat('descending order', B)
```

Output:

```
ascending order 1  2  2  7  8 11

descending order 11  8  7  2  2  1
```

Creating named vectors

Named vector can be created in several ways. With `c`:

```
xc <- c('a' = 5, 'b' = 6, 'c' = 7, 'd' = 8)
```

which results in:

```
> xc
a b c d
5 6 7 8
```

With the **setNames** function, two vectors of the same length can be used to create a named vector:

```
x <- 5:8
y <- letters[1:4]

xy <- setNames(x, y)
```

which results in a named integer vector:

```
> xy
a b c d
5 6 7 8
```

You may also use the **names** function to get the same result:

```
xy <- 5:8

names(xy) <- letters[1:4]

#With such a vector it is also possibly to select elements by name:

  xy["a"]
```

<u>**Vector sub-setting**</u>

In R Programming Language, subsetting allows the user to access elements from an object. It takes out a portion from the object based on the condition provided.

Method 1: Subsetting in R Using [ ] Operator

Using the '[ ]' operator, elements of vectors and observations from data frames can be accessed. To neglect some indexes, '-' is used to access all other indexes of vector or data frame.

```
x <- 1:15

# Print vector

cat("Original vector: ", x, "\n")

 # Subsetting vector

cat("First 5 values of vector: ", x[1:5], "\n")

 cat("Without values present at index 1, 2 and 3: ",  x[-c(1, 2, 3)], "\n")
```

Method 4: Subsetting in R Using subset() Function

subset() function in R programming is used to create a subset of vectors, matrices, or data frames based on the conditions provided in the parameters.

```
q <- subset(airquality, Temp < 65, select = c(Month))

print(q)
```

Matrices

Matrix is a rectangular arrangement of numbers in rows and columns. In a matrix, as we know rows are the ones that run horizontally and columns are the ones that run vertically.

Creating and Naming a Matrix

To create a matrix in R you need to use the function called **matrix()**. The arguments to this **matrix()** are the set of elements in the vector. You have to pass how many numbers of rows and how many numbers of columns you want to have in your matrix.

Note: *By default, matrices are in column-wise order.*

```
A = matrix(

  # Taking sequence of elements
  c(1, 2, 3, 4, 5, 6, 7, 8, 9),

  # No of rows
  nrow = 3,

  # No of columns
  ncol = 3,
  # By default matrices are in column-wise order
  # So this parameter decides how to arrange the matrix
  byrow = TRUE
)

# Naming rows
rownames(A) = c("r1", "r2", "r3")

# Naming columns
colnames(A) = c("c1", "c2", "c3")

cat("The 3x3 matrix:\n")
print(A)
```

<u>Creating special matrices</u>

R allows creation of various different types of matrices with the use of arguments passed to the matrix() function.

- **Matrix where all rows and columns are filled by a single constant 'k':**
  To create such a matrix the syntax is given below:

Syntax: *matrix(k, m, n)*
Parameters:
k: *the constant*
m: *no of rows*
n: *no of columns*

```
print(matrix(5, 3, 3))
```

Diagonal matrix:
A diagonal matrix is a matrix in which the entries outside the main diagonal are all zero. To create such a matrix the syntax is given below:

```
print(diag(c(5, 3, 3), 3, 3))
```

Identity matrix:
A square matrix in which all the elements of the principal diagonal are ones and all other elements are zeros. To create such a matrix the syntax is given below:

```
print(diag(1, 3, 3))
```

Matrix metrics

Matrix metrics mean once a matrix is created then

- How can you know the dimension of the matrix?
- How can you know how many rows are there in the matrix?
- How many columns are in the matrix?

- How many elements are there in the matrix? are the questions we generally wanted to answer.

```
A = matrix(
  c(1, 2, 3, 4, 5, 6, 7, 8, 9),
  nrow = 3,
  ncol = 3,
  byrow = TRUE
)
cat("The 3x3 matrix:\n")
print(A)

cat("Dimension of the matrix:\n")
print(dim(A))

cat("Number of rows:\n")
print(nrow(A))

cat("Number of columns:\n")
print(ncol(A))

cat("Number of elements:\n")
print(length(A))
# OR
print(prod(dim(A)))
```

Matrix subsetting

A matrix is subset with two arguments within single brackets, [], and separated by a comma. The first argument specifies the rows, and the second the columns.

```
M_new<-matrix(c(25,23,25,20,15,17,13,19,25,24,21,19,20,12,30,17),ncol=4)

#M_new<-matrix(1:16,4)
```

```
M_new
colnames(M_new)<-c("C1","C2","C3","C4")
rownames(M_new)<-c("R1","R2","R3","R4")

M_new[,1,drop=FALSE] # all rows with 1st column

M_new[1,,drop=FALSE] #1st row with all column

M_new[1,1,drop=FALSE] #display 1st row and 1st column, cell value

M_new[1:2,2:3]#display 1st ,2nd  rows and 2nd ,3rd  column

M_new[1:2,c(2,4)] #display 1st ,2nd  rows and 2nd ,4th  column
```

Arrays

Arrays are the R data objects which can store data in more than two dimensions. For example − If we create an array of dimension (2, 3, 4) then it creates 4 rectangular matrices each with 2 rows and 3 columns. Arrays can store only data type.

An array is created using the array() function. It takes vectors as input and uses the values in the dim parameter to create an array.

Example

The following example creates an array of two 3x3 matrices each with 3 rows and 3 columns.

```
# Create two vectors of different lengths.

vector1 <- c(5,9,3)

vector2 <- c(10,11,12,13,14,15)


# Take these vectors as input to the array.

result <- array(c(vector1,vector2),dim = c(3,3,2))

print(result)
```

Naming Columns and Rows

We can give names to the rows, columns and matrices in the array by using the dimnames parameter.

```
# Create two vectors of different lengths.
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)
column.names <- c("COL1","COL2","COL3")
row.names <- c("ROW1","ROW2","ROW3")
matrix.names <- c("Matrix1","Matrix2")

# Take these vectors as input to the array.
result <- array(c(vector1,vector2),dim = c(3,3,2),dimnames = list(row.names,column.names,
  matrix.names))
print(result)
```

Accessing arrays

The arrays can be accessed by using indices for different dimensions separated by commas. Different components can be specified by any combination of elements' names or positions.

Accessing Uni-Dimensional Array

The elements can be accessed by using indexes of the corresponding elements.

```
vec <- c(1:10)

# accessing entire vector
cat ("Vector is : ", vec)

# accessing elements
cat ("Third element of vector is : ", vec[3])
```

Accessing Array Elements

```
# Create two vectors of different lengths.
vector1 <- c(5,9,3)
vector2 <- c(10,11,12,13,14,15)
```

```
column.names <- c("COL1","COL2","COL3")

row.names <- c("ROW1","ROW2","ROW3")

matrix.names <- c("Matrix1","Matrix2")


# Take these vectors as input to the array.

result <- array(c(vector1,vector2),dim = c(3,3,2),dimnames = list(row.names,
   column.names, matrix.names))


# Print the third row of the second matrix of the array.

print(result[3,,2])


# Print the element in the 1st row and 3rd column of the 1st matrix.

print(result[1,3,1])


# Print the 2nd Matrix.

print(result[,,2])
```

Calculations across Array Elements

We can do calculations across the elements in an array using the **apply()** function.

Syntax

```
apply(x, margin, fun)
```

Following is the description of the parameters used −

- **x** is an array.
- **margin** is the name of the data set used.
- **fun** is the function to be applied across the elements of the array.

Example

We use the apply() function below to calculate the sum of the elements in the rows of an array across all the matrices.

```
# Create two vectors of different lengths.

vector1 <- c(5,9,3)

vector2 <- c(10,11,12,13,14,15)


# Take these vectors as input to the array.
```

```
new.array <- array(c(vector1,vector2),dim = c(3,3,2))
print(new.array)

# Use apply to calculate the sum of the rows across all the matrices.
result <- apply(new.array, c(1), sum)
print(result)
```

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

```
, , 1
[,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15
, , 2
[,1] [,2] [,3]
[1,]    5   10   13
[2,]    9   11   14
[3,]    3   12   15

[1] 56 68 60
```

Accessing subset of array elements

A smaller subset of the array elements can be accessed by defining a range of row or column limits.

```
row_names <- c("row1", "row2")
col_names <- c("col1", "col2", "col3", "col4")
mat_names <- c("Mat1", "Mat2")
arr = array(1:15, dim = c(2, 4, 2),
    dimnames = list(row_names, col_names, mat_names))

# print elements of both the rows and columns 2 and 3 of matrix 1
```

```
print (arr[, c(2, 3), 1])
```

Adding elements to array

Elements can be appended at the different positions in the array. The sequence of elements is retained in order of their addition to the array. The time complexity required to add new elements is O(n) where n is the length of the array. The length of the array increases by the number of element additions. There are various in-built functions available in R to add new values:

c(vector, values): c() function allows us to append values to the end of the array. Multiple values can also be added together.

append(vector, values): This method allows the values to be appended at any position in the vector. By default, this function adds the element at end.

append(vector, values, after=length(vector)) adds new values after specified length of the array specified in the last argument of the function.

Using the length function of the array:

Elements can be added at length+x indices where x>0.

```
# creating a uni-dimensional array

x <- c(1, 2, 3, 4, 5)


# addition of element using c() function

x <- c(x, 6)

print ("Array after 1st modification ")

print (x)


# addition of element using append function

x <- append(x, 7)

print ("Array after 2nd modification ")

print (x)


# adding elements after computing the length

len <- length(x)
```

```
x[len + 1] <- 8
print ("Array after 3rd modification ")
print (x)

# adding on length + 3 index
x[len + 3]<-9
print ("Array after 4th modification ")
print (x)

# append a vector of values to the array after length + 3 of array
print ("Array after 5th modification")
x <- append(x, c(10, 11, 12), after = length(x)+3)
print (x)

# adds new elements after 3rd index
print ("Array after 6th modification")
x <- append(x, c(-1, -1), after = 3)
print (x)
[1] "Array after 1st modification "
[1] 1 2 3 4 5 6
[1] "Array after 2nd modification "
[1] 1 2 3 4 5 6 7
[1] "Array after 3rd modification "
[1] 1 2 3 4 5 6 7 8
[1] "Array after 4th modification "
[1]  1  2  3  4  5  6  7  8 NA  9
[1] "Array after 5th modification"
[1]  1  2  3  4  5  6  7  8 NA  9 10 11 12
[1] "Array after 6th modification"
[1]  1  2  3 -1 -1  4  5  6  7  8 NA  9 10 11 12
```

Removing Elements from Array

Elements can be removed from arrays in R, either one at a time or multiple together. These elements are specified as indexes to the array, wherein the array values satisfying the conditions are retained and rest removed. The comparison for removal is based on array values. Multiple conditions can also be combined together to remove a range of elements. Another way to remove elements is by using %in% operator wherein the set of element values belonging to the TRUE values of the operator are displayed as result and the rest are removed.

```
# creating an array of length 9

m <- c(1, 2, 3, 4, 5, 6, 7, 8, 9)

print ("Original Array")

print (m)


# remove a single value element:3 from array

m <- m[m != 3]

print ("After 1st modification")

print (m)
```

Class in R

Class is the blueprint that helps to create an object and contains its member variable along with the attributes. As discussed earlier in the previous section, there are two classes of R, S3, and S4.

S3 Class

- S3 class is somewhat primitive in nature. It lacks a formal definition and object of this class can be created simply by adding a class attribute to it.
- This simplicity accounts for the fact that it is widely used in R programming language. In fact most of the R built-in classes are of this type.

Example 1: S3 class

```
 # create a list with required components

s <- list(name = "John", age = 21, GPA = 3.5)

# name the class appropriately

class(s) <- "student"
```

S4 Class

- S4 class are an improvement over the S3 class. They have a formally defined structure which helps in making object of the same class look more or less similar.
- Class components are properly defined using the setClass() function and objects are created using the new() function.

Example 2: S4 class

< setClass("student", slots=list(name="character", age="numeric", GPA="numeric"))

Reference Class

- Reference class were introduced later, compared to the other two. It is more similar to the object oriented programming we are used to seeing in other major programming languages.
- Reference classes are basically S4 classed with an environment added to it.

Example 3: Reference class

< setRefClass("student")

Factors

Introduction to Factors:

Factors in R Programming Language are data structures that are implemented to categorize the data or represent categorical data and store it on multiple levels.

They can be stored as integers with a corresponding label to every unique integer. Though factors may look similar to character vectors, they are integers and care must be taken while using them as strings. The factor accepts only a restricted number of distinct values. For example, a data field such as gender may contain values only from female, male.

Creating a Factor in R Programming Language

- The command used to create or modify a factor in R language is – factor() with a vector as input.

The two steps to creating a factor are:

- Creating a vector
- Converting the vector created into a factor using function factor()

Example:

```
# Create a vector as input.
data <- c("East","West","East","North","North","East","West","West","West","East","North")

print(data)
print(is.factor(data))

# Apply the factor function.
factor_data <- factor(data)

print(factor_data)
print(is.factor(factor_data))
```

Changing the Order of Levels

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

```
data <- c("East","West","East","North","North","East","West","West","West","East","North")
# Create the factors
factor_data <- factor(data)
print(factor_data)

# Apply the factor function with required order of the level.
new_order_data <- factor(factor_data,levels = c("East","West","North"))
print(new_order_data)
```

Generating Factor Levels

We can generate factor levels by using the gl() function. It takes two integers as input which indicates how many levels and how many times each level.

Syntax

gl(n, k, labels)

Following is the description of the parameters used −

- n is a integer giving the number of levels.
- k is a integer giving the number of replications.
- labels is a vector of labels for the resulting factor levels.

Example:

```
v <- gl(3, 4, labels = c("A", "B","C"))

print(v)
```

Accessing elements of a Factor in R

Like we access elements of a vector, the same way we access the elements of a factor. If gender is a factor then gender[i] would mean accessing ith element in the factor.

Example:

```
gender <- factor(c("female", "male", "male", "female"))
gender[3]
```

Summarizing a Factor

The **summary** function in R returns the results of basic statistical calculations (minimum, 1st quartile, median, mean, 3rd quartile, and maximum) for a numerical vector. The general way to write the R summary function is *summary*(x, na.rm=FALSE/TRUE*)*. Again, X refers to a numerical vector, while na.rm=FALSE/TRUE specifies whether to remove empty values from the calculation.

Example:

```
v <- gl(3, 4, labels = c("A", "B","C"))
print(v)
summary(v)
```

Level Ordering of Factors

Factors are data objects used to categorize data and store it as levels. They can store a string as well as an integer. They represent columns as they have a limited number of unique values. Factors in R can be created using factor() function. It takes a vector as input. c() function is used to create a vector with explicitly provided values.

Example:

```
x < - c("Pen", "Pencil", "Brush", "Pen",
       "Brush", "Brush", "Pencil", "Pencil")

print(x)
print(is.factor(x))

# Apply the factor function.
factor_x = factor(x)
levels(factor_x)
```

In the above code, x is a vector with 8 elements. To convert it to a factor the function factor() is used. Here there are 8 factors and 3 levels. Levels are the unique elements in the data. Can be found using levels() function.

Ordering Factor Levels

Ordered factors is an extension of factors. It arranges the levels in increasing order. We use two functions: factor() along with argument ordered().

Syntax: factor(data, levels =c(“”), ordered =TRUE)

Parameter:

data: input vector with explicitly defined values.

levels(): Mention the list of levels in c function.

ordered: It is set true for enabling ordering.

Example:

```
size = c("small", "large", "large", "small","medium", "large", "medium", "medium")
# converting to factor
size_factor <- factor(size)
print(size_factor)

# ordering the levels
ordered.size <- factor(size, levels = c("small", "medium", "large"), ordered = TRUE)
print(ordered.size)
```

In the above code, size vector is created using c function. Then it is converted to a factor. And for ordering factor() function is used along with the arguments described above. Thus the sizes arranged in order.

Data Frames

A data frame is a table or a two-dimensional array-like structure in which each column contains values of one variable and each row contains one set of values from each column. Data frames can also be interpreted as matrices where each column of a matrix can be of the different data types.

Following are the characteristics of a data frame.

- The column names should be non-empty.
- The row names should be unique.
- The data stored in a data frame can be of numeric, factor or character type.
- Each column should contain same number of data items.

Creating Data Frame

```
friend.data <- data.frame(
   friend_id = c(1:5),
   friend_name = c("Sachin", "Sourav",
                   "Dravid", "Sehwag",
                   "Dhoni"),
   )
# print the data frame
print(friend.data)
```

Output:

```
     friend_id friend_name
1        1      Sachin
2        2      Sourav
3        3      Dravid
4        4      Sehwag
5        5       Dhoni
```

Summary of Data in Data Frame

The statistical summary and nature of the data can be obtained by applying **summary()** function.

```
# Create the data frame.
emp.data <- data.frame(
  emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),

   start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11",
      "2015-03-27")),
   stringsAsFactors = FALSE
)
# Print the summary.
print(summary(emp.data))
```

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

```
     emp_id    emp_name             salary        start_date
 Min.   :1   Length:5          Min.   :515.2   Min.   :2012-01-01
 1st Qu.:2   Class :character   1st Qu.:611.0   1st Qu.:2013-09-23
 Median :3   Mode  :character   Median :623.3   Median :2014-05-11
 Mean   :3                     Mean   :664.4   Mean   :2014-01-14
 3rd Qu.:4                     3rd Qu.:729.0   3rd Qu.:2014-11-15
 Max.   :5                     Max.   :843.2   Max.   :2015-03-27
```

Extract Data from Data Frame

Extract specific column from a data frame using column name.

```
# Create the data frame.
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),

   start_date = as.Date(c("2012-01-01","2013-09-23","2014-11-15","2014-05-11","2015-03-27")),
)
# Extract Specific columns.
result <- data.frame(emp.data$emp_name,emp.data$salary)
print(result)
```

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

```
  emp.data.emp_name emp.data.salary
```

```
1            Rick         623.30
2             Dan         515.20
3        Michelle           611.00
4            Ryan          729.00
5            Gary          843.25
```

Extract the first two rows and then all columns

result <- emp.data[1:2,]

Extract 3rd and 5th row with 2nd and 4th column

result <- emp.data[c(3,5),c(2,4)]

Expand Data Frame / Extending Data Frame

A data frame can be expanded by adding columns and rows.

Add Column

Just add the column vector using a new column name.

```
# Create the data frame.
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),

   start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11", "2015-03-27")),
   stringsAsFactors = FALSE
)

# Add the "dept" coulmn.
emp.data$dept <- c("IT","Operations","IT","HR","Finance")
v <- emp.data
print(v)
```

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

```
  emp_id   emp_name    salary   start_date      dept
1     1    Rick        623.30    2012-01-01      IT
2     2    Dan         515.20    2013-09-23      Operations
3     3    Michelle    611.00    2014-11-15      IT
4     4    Ryan        729.00    2014-05-11      HR
5     5    Gary        843.25    2015-03-27      Finance
```

Add Row

To add more rows permanently to an existing data frame, we need to bring in the new rows in the same structure as the existing data frame and use the rbind() function.

In the example below we create a data frame with new rows and merge it with the existing data frame to create the final data frame.

```
# Create the first data frame.

emp.data <- data.frame(

   emp_id = c (1:5),

   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),

   salary = c(623.3,515.2,611.0,729.0,843.25),

   start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11", "2015-03-27")),

   dept = c("IT","Operations","IT","HR","Finance"),

 )


# Create the second data frame

emp.newdata <- data.frame(

   emp_id = c (6:8),

   emp_name = c("Rasmi","Pranab","Tusar"),

   salary = c(578.0,722.5,632.8),

   start_date = as.Date(c("2013-05-21","2013-07-30","2014-06-17")),

   dept = c("IT","Operations","Fianance"),

)


# Bind the two data frames.

emp.finaldata <- rbind(emp.data,emp.newdata)
```

```
print(emp.finaldata)
```

Remove Rows and Columns

Use the c() function to remove rows and columns in a Data Frame:

Example

```
Data_Frame <- data.frame (
  Training = c("Strength", "Stamina", "Other"),
  Pulse = c(100, 150, 120),
  Duration = c(60, 30, 45)
)

# Remove the first row and column
Data_Frame_New <- Data_Frame[-c(1), -c(1)]

# Print the new data frame
Data_Frame_New
```

```
Pulse Duration
2   150       30
3   120       45
```

Create Subsets of a Data frame

subset() function in R Programming Language is used to create subsets of a Data frame. This can also be used to drop columns from a data frame.

```
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),


   start_date = as.Date(c("2012-01-01", "2013-09-23", "2014-11-15", "2014-05-11", "2015-
03-27")),
   )
```

```
emp.data


subset(emp.data, emp_id == 3)


subset(emp.data, emp_id == c(1:3))
```

```
  emp_id emp_name salary start_date
1      1     Rick 623.30 2012-01-01
2      2      Dan 515.20 2013-09-23
3      3 Michelle 611.00 2014-11-15
4      4     Ryan 729.00 2014-05-11
5      5     Gary 843.25 2015-03-27

 emp_id emp_name salary start_date
3      3 Michelle    611 2014-11-15

 emp_id emp_name salary start_date
1      1     Rick  623.3 2012-01-01
2      2      Dan  515.2 2013-09-23
3      3 Michelle  611.0 2014-11-15
```

Sorting Data

To sort a data frame in R, use the order( ) function. By default, sorting is ASCENDING. Prepend the sorting variable by a minus sign to indicate DESCENDING order. Here are some examples.

```
data = data.frame(
  rollno = c(1, 5, 4, 2, 3),
  subjects = c("java", "python", "php", "sql", "c"))


print(data)
print("sort the data in decreasing order based on subjects ")
print(data[order(data$subjects, decreasing = TRUE), ]   )


print("sort the data in decreasing order based on rollno ")
print(data[order(data$rollno, decreasing = TRUE), ]   )
```

Output:

```
  rollno subjects
1      1     java
2      5   python
3      4      php
4      2      sql
5      3        c
[1] "sort the data in decreasing order based on subjects "
  rollno subjects
4      2      sql
2      5   python
3      4      php
1      1     java
5      3        c
[1] "sort the data in decreasing order based on rollno "
  rollno subjects
2      5   python
3      4      php
5      3        c
4      2      sql
1      1     java
```

Lists

Lists are one-dimensional, heterogeneous data structures. The list can be a list of vectors, a list of matrices, a list of characters and a list of functions, and so on.

A list is a vector but with heterogeneous data elements. A list in R is created with the use of list() function. R allows accessing elements of a list with the use of the index value. In R, the indexing of a list starts with 1 instead of 0 like other programming languages.

Creating a List

To create a List in R you need to use the function called "list()". In other words, a list is a generic vector containing other objects. To illustrate how a list looks, we take an example here. We want to build a list of employees with the details. So for this, we want attributes such as ID, employee name, and the number of employees.

```
empId = c(1, 2, 3, 4)

empName = c("Debi", "Sandeep", "Subham", "Shiba")

numberOfEmp = 4

empList = list(empId, empName, numberOfEmp)

print(empList)
```

or

```
list_data <- list("Red", "Green", c(21,32,11), TRUE, 51.23, 119.1)
print(list_data)
```

Accessing components of a list

We can access components of a list in two ways.

Access components by names: All the components of a list can be named and we can use those names to access the components of the list using the dollar command.

```
empId = c(1, 2, 3, 4)
empName = c("Debi", "Sandeep", "Subham", "Shiba")
numberOfEmp = 4
empList = list(
  "ID" = empId,
  "Names" = empName,
  "Total Staff" = numberOfEmp
  )
print(empList)

# Accessing components by names
cat("Accessing name components using $ command\n")
print(empList$Names)
```

<u>Access components by indices</u>: We can also access the components of the list using indices. To access the top-level components of a list we have to use a double slicing operator “[[ ]]” which is two square brackets and if we want to access the lower or inner level components of a list we have to use another square bracket “[ ]” along with the double slicing operator “[[ ]]“.

```
empId = c(1, 2, 3, 4)

empName = c("Debi", "Sandeep", "Subham", "Shiba")

numberOfEmp = 4

empList = list(

  "ID" = empId,

  "Names" = empName,

  "Total Staff" = numberOfEmp

  )

print(empList)


# Accessing a top level components by indices

cat("Accessing name components using indices\n")

print(empList[[2]])


# Accessing a inner level components by indices

cat("Accessing Sandeep from name using indices\n")

print(empList[[2]][2])


# Accessing another inner level components by indices

cat("Accessing 4 from ID using indices\n")

print(empList[[1]][4])
```

Modifying components of a list

A list can also be modified by accessing the components and replacing them with the ones which you want.

```
empId = c(1, 2, 3, 4)

empName = c("Debi", "Sandeep", "Subham", "Shiba")

numberOfEmp = 4
```

```
empList = list(
  "ID" = empId,
  "Names" = empName,
  "Total Staff" = numberOfEmp
)
cat("Before modifying the list\n")
print(empList)

# Modifying the top-level component
empList$`Total Staff` = 5

# Modifying inner level component
empList[[1]][5] = 5
empList[[2]][5] = "Kamala"

cat("After modified the list\n")
print(empList)
```

Merging list

We can merge the list by placing all the lists into a single list.

```
lst1 <- list(1,2,3)
lst2 <- list("Sun","Mon","Tue")

# Merge the two lists.
new_list <- c(lst1,lst2)

# Print the merged list.
print(new_list)
```

Deleting components of a list

To delete components of a list, first of all, we need to access those components and then insert a negative sign before those components. It indicates that we had to delete that component.

```
empId = c(1, 2, 3, 4)
empName = c("Debi", "Sandeep", "Subham", "Shiba")
numberOfEmp = 4
empList = list(
  "ID" = empId,
  "Names" = empName,
  "Total Staff" = numberOfEmp
)
cat("Before deletion the list is\n")
print(empList)

# Deleting a top level components
cat("After Deleting Total staff components\n")
print(empList[-3])

# Deleting a inner level components
cat("After Deleting sandeep from name\n")
print(empList[[2]][-2])
```

Converting List to Vector

Here we are going to convert the list to vector, for this we will create a list first and then unlist the list into the vector.

```
# Create lists.
lst <- list(1:5)
print(lst)

# Convert the lists to vectors.
vec <- unlist(lst)

print(vec)
```

Unit-4

Conditionals and control flow

R - Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. R language is rich in built-in operators and provides following types of operators.

Types of Operators

We have the following types of operators in R programming −

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators
- Miscellaneous Operators

Arithmetic Operators

Following table shows the arithmetic operators supported by R language. The operators act on each element of the vector.

Operator	Description	Example

+	Adds two vectors	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v+t)`
		it produces the following result −
		[1] 10.0 8.5 10.0

−	Subtracts second vector from the first	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v-t)`
		it produces the following result −
		[1] -6.0 2.5 2.0

*	Multiplies both vectors	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v*t)`
		it produces the following result −

		[1] 16.0 16.5 24.0

/	Divide the first vector with the second	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v/t)`
		When we execute the above code, it produces the following result −
		`[1] 0.250000 1.833333 1.500000`

%%	Give the remainder of the first vector with the second	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v%%t)`
		it produces the following result −
		`[1] 2.0 2.5 2.0`

%/%	The result of division of first vector with second (quotient)	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v%/%t)`
		it produces the following result −
		`[1] 0 1 1`

^	The first vector raised to the exponent of second vector	`v <- c( 2,5.5,6)`
		`t <- c(8, 3, 4)`
		`print(v^t)`
		it produces the following result −
		`[1] 256.000 166.375 1296.000`

Relational Operators

Following table shows the relational operators supported by R language. Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value.

Operator	Description	Example

>	Checks if each element of the first vector is greater than the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v>t)`
		it produces the following result −
		`[1] FALSE TRUE FALSE FALSE`

<	Checks if each element of the first vector is less than the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v < t)`
		it produces the following result –
		[1] TRUE FALSE TRUE FALSE

==	Checks if each element of the first vector is equal to the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v == t)`
		it produces the following result –
		[1] FALSE FALSE FALSE TRUE

<=	Checks if each element of the first vector is less than or equal to the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v<=t)`
		it produces the following result –
		[1] TRUE FALSE TRUE TRUE

>=	Checks if each element of the first vector is greater than or equal to the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v>=t)`
		it produces the following result –
		[1] FALSE TRUE FALSE TRUE

!=	Checks if each element of the first vector is unequal to the corresponding element of the second vector.	`v <- c(2,5.5,6,9)`
		`t <- c(8,2.5,14,9)`
		`print(v!=t)`
		it produces the following result –
		[1] TRUE TRUE TRUE FALSE

Logical Operators

Following table shows the logical operators supported by R language. It is applicable only to vectors of type logical, numeric or complex. All numbers greater than 1 are considered as logical value TRUE.

Each element of the first vector is compared with the corresponding element of the second vector. The result of comparison is a Boolean value.

Operator	Description	Example

&	It is called Element-wise Logical AND operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if both the elements are TRUE.	`v <- c(3,1,TRUE,2+3i)`
		`t <- c(4,1,FALSE,2+3i)`
		`print(v&t)`
		
		it produces the following result −
		
		[1] TRUE TRUE FALSE TRUE

|	It is called Element-wise Logical OR operator. It combines each element of the first vector with the corresponding element of the second vector and gives a output TRUE if one the elements is TRUE.	`v <- c(3,0,TRUE,2+2i)`
		`t <- c(4,0,FALSE,2+3i)`
		`print(v|t)`
		
		it produces the following result −
		
		[1] TRUE FALSE TRUE TRUE

!	It is called Logical NOT operator. Takes each element of the vector and gives the opposite logical value.	`v <- c(3,0,TRUE,2+2i)`
		`print(!v)`
		
		it produces the following result −
		
		[1] FALSE TRUE FALSE FALSE

The logical operator && and || considers only the first element of the vectors and give a vector of single element as output.

Operator	Description	Example

&&	Called Logical AND operator. Takes first element of both the vectors and gives the TRUE only if both are TRUE.	`v <- c(3,0,TRUE,2+2i)`
		`t <- c(1,3,TRUE,2+3i)`
		`print(v&&t)`
		
		it produces the following result −
		
		[1] TRUE

||	Called Logical OR operator. Takes first element of both the vectors and gives the TRUE if one of them is TRUE.	`v <- c(0,0,TRUE,2+2i)`
		`t <- c(0,3,TRUE,2+3i)`
		`print(v||t)`
		
		it produces the following result −
		
		[1] FALSE

Assignment Operators

These operators are used to assign values to vectors.

Operator	Description	Example

<−	Called Left Assignment	`v1 <- c(3,1,TRUE,2+3i)`
		`v2 <<- c(3,1,TRUE,2+3i)`

or
=
or
<<−
		`v3 = c(3,1,TRUE,2+3i)`
		`print(v1)`
		`print(v2)`
		`print(v3)`
		it produces the following result −
		`[1] 3+0i 1+0i 1+0i 2+3i`
		`[1] 3+0i 1+0i 1+0i 2+3i`
		`[1] 3+0i 1+0i 1+0i 2+3i`

->
or
->>
	Called Right Assignment	`c(3,1,TRUE,2+3i) -> v1`
		`c(3,1,TRUE,2+3i) ->> v2`
		`print(v1)`
		`print(v2)`
		it produces the following result −
		`[1] 3+0i 1+0i 1+0i 2+3i`
		`[1] 3+0i 1+0i 1+0i 2+3i`

Miscellaneous Operators

These operators are used to for specific purpose and not general mathematical or logical computation.

Operator	Description	Example

:	Colon operator. It creates the series of numbers in sequence for a vector.	`v <- 2:8`
		`print(v)`
		it produces the following result −
		`[1] 2 3 4 5 6 7 8`

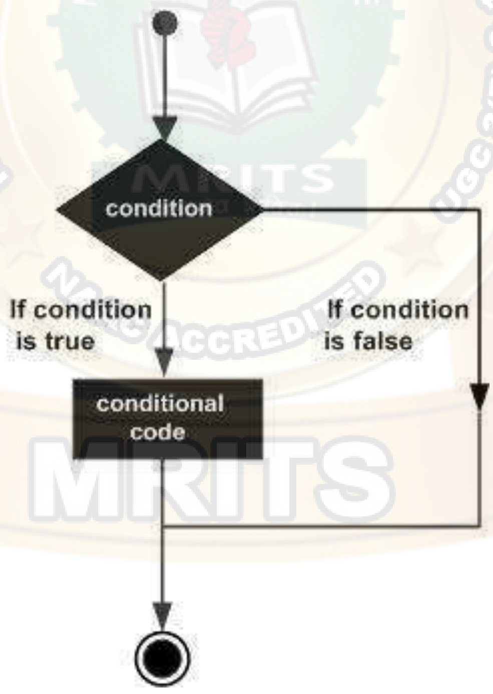
%in%	This operator is used to identify if an element belongs to a vector.	`v1 <- 8`
		`v2 <- 12`
		`t <- 1:10`
		`print(v1 %in% t)`
		`print(v2 %in% t)`
		it produces the following result −
		`[1] TRUE`
		`[1] FALSE`

%*%	This operator is used to multiply a matrix with its transpose.	`M = matrix( c(2,6,5,1,10,4), nrow = 2,ncol = 3,byrow = TRUE)`
		`t = M %*% t(M)`
		`print(t)`
		
		it produces the following result −
		
		[,1] [,2]
		[1,] 65 82
		[2,] 82 117

R - Decision making / Conditional statement

Decision making structures require the programmer to specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be **true**, and optionally, other statements to be executed if the condition is determined to be **false**.

Following is the general form of a typical decision making structure found in most of the programming languages −



R provides the following types of decision making statements. Click the following links to check their detail.

Sr.No.	Statement & Description
1	if statement An **if** statement consists of a Boolean expression followed by one or more statements.

2	if...else statement An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.
3	switch statement A **switch** statement allows a variable to be tested for equality against a list of values.

R - If Statement

An **if** statement consists of a Boolean expression followed by one or more statements.

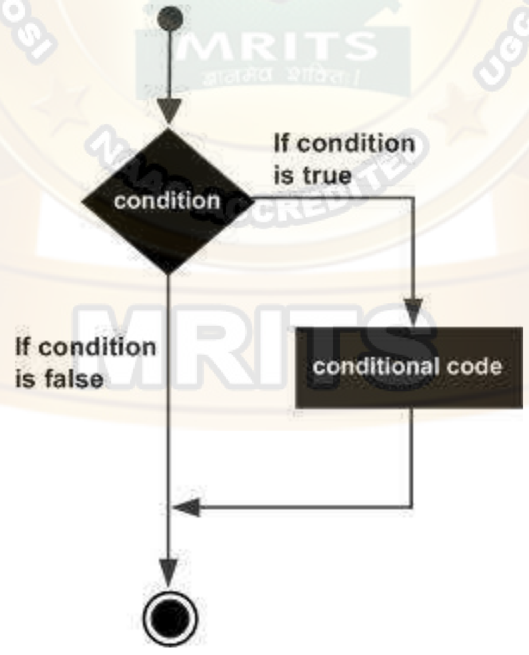
Syntax

The basic syntax for creating an **if** statement in R is −

```
if(boolean_expression) {
// statement(s) will execute if the boolean expression is true.
}
```

If the Boolean expression evaluates to be **true**, then the block of code inside the if statement will be executed. If Boolean expression evaluates to be **false**, then the first set of code after the end of the if statement (after the closing curly brace) will be executed.

Flow Diagram



Example

```
x <- 30L
if(is.integer(x)) {
  print("X is an Integer")
}
```

When the above code is compiled and executed, it produces the following result −

```
[1] "X is an Integer"
```

R - If...Else Statement

An **if** statement can be followed by an optional **else** statement which executes when the boolean expression is false.

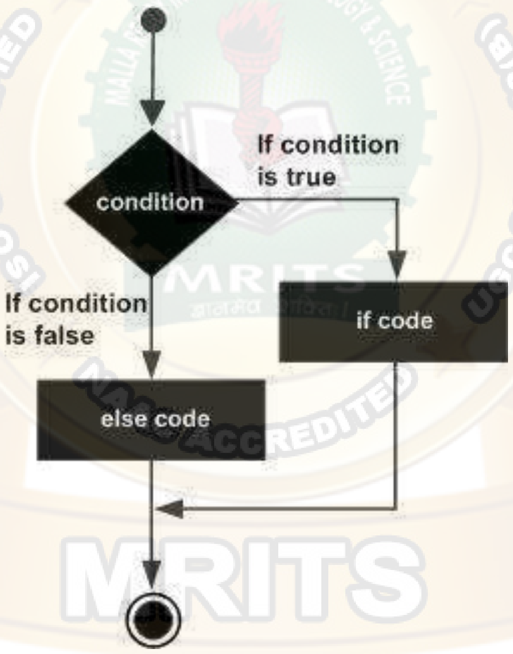
Syntax

The basic syntax for creating an **if...else** statement in R is −

```
if(boolean_expression) {
// statement(s) will execute if the boolean expression is true.
} else {
// statement(s) will execute if the boolean expression is false.
}
```

If the Boolean expression evaluates to be **true**, then the **if block** of code will be executed, otherwise **else block** of code will be executed.

Flow Diagram



Example

```
x <- c("what","is","truth")

if("Truth" %in% x) {
  print("Truth is found")
} else {
  print("Truth is not found")
}
```

When the above code is compiled and executed, it produces the following result −

```
[1] "Truth is not found"
```

Here "Truth" and "truth" are two different strings.

The if...else if...else Statement

An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement.

When using **if**, **else if**, **else** statements there are few points to keep in mind.

- An **if** can have zero or one **else** and it must come after any **else if**'s.
- An **if** can have zero to many **else if's** and they must come before the else.
- Once an **else if** succeeds, none of the remaining **else if**'s or **else**'s will be tested.

Syntax

The basic syntax for creating an **if...else if...else** statement in R is −

```
if(boolean_expression 1) {
// Executes when the boolean expression 1 is true.
} else if( boolean_expression 2) {
// Executes when the boolean expression 2 is true.
} else if( boolean_expression 3) {
// Executes when the boolean expression 3 is true.
} else {
// executes when none of the above condition is true.
}
```

Example

```
x <- c("what","is","truth")

if("Truth" %in% x) {
  print("Truth is found the first time")
} else if ("truth" %in% x) {
  print("truth is found the second time")
} else {
  print("No truth found")
}
```

When the above code is compiled and executed, it produces the following result −

```
[1] "truth is found the second time"
```

Nested If Statements

You can also have if statements inside if statements, this is called *nested* if statements.

Example

```
x <- 41

if (x > 10) {
 print("Above ten")
 if (x > 20) {
   print("and also above 20!")
 } else {
   print("but not above 20.")
 }
} else {
 print("below 10.")
}
```

AND

The & symbol (and) is a logical operator, and is used to combine conditional statements:

Example

Test if a is greater than b, AND if c is greater than a:

```
a <- 200
b <- 33
c <- 500

if (a > b & c > a){
  print("Both conditions are true")
}
```

OR

The | symbol (or) is a logical operator, and is used to combine conditional statements:

Example

Test if a is greater than b, or if c is greater than a:

```
a <- 200
b <- 33
c <- 500

if (a > b | a > c){
  print("At least one of the conditions is true")
}
```

R - Switch Statement

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax

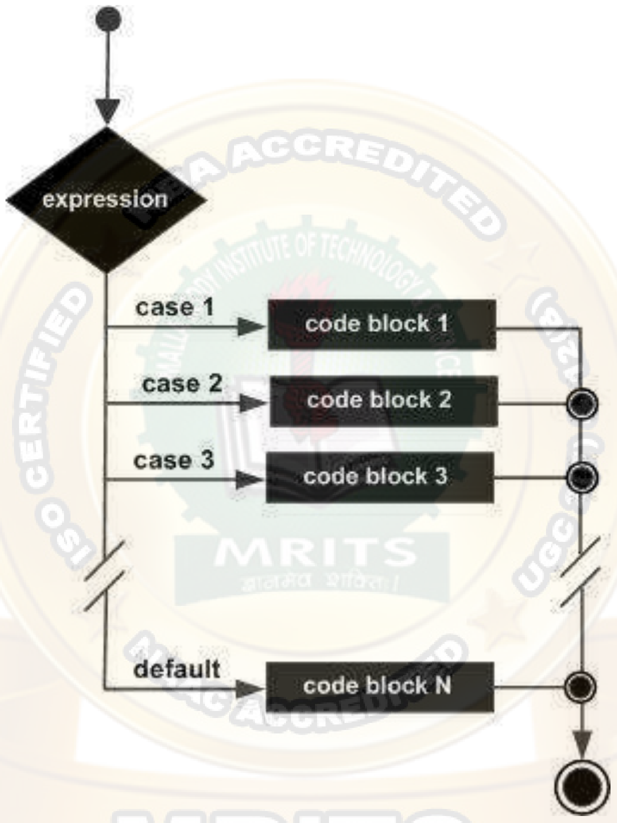
The basic syntax for creating a switch statement in R is −

```
switch(expression, case1, case2, case3....)
```

The following rules apply to a switch statement −

- If the value of expression is not a character string it is coerced to integer.
- You can have any number of case statements within a switch.
- If the value of the integer is between 1 and nargs()−1 (The max number of arguments)then the corresponding element of case condition is evaluated and the result returned.
- If expression evaluates to a character string then that string is matched (exactly) to the names of the elements.
- If there is more than one match, the first matching element is returned.
- No Default argument is available.
- In the case of no match, if there is a unnamed element of ... its value is returned.

Flow Diagram



Example

```
x <- switch(
  3,
  "first",
  "second",
  "third",
  "fourth"
)
print(x)
```

When the above code is compiled and executed, it produces the following result −

```
[1] "third"
```

<u>Example 2:</u>

# Following is val1 simple R program

```
# to demonstrate syntax of switch.

# Mathematical calculation

val1 = 6
val2 = 7
val3 = "s"
result = switch(
    val3,
    "a"= cat("Addition =", val1 + val2),
    "d"= cat("Subtraction =", val1 - val2),
    "r"= cat("Division = ", val1 / val2),
    "s"= cat("Multiplication =", val1 * val2),
    "m"= cat("Modulus =", val1 %% val2),
    "p"= cat("Power =", val1 ^ val2)
)

print(result)
```

Iterative Programming in R

R - Loops

Introduction:

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially. The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and the following is the general form of a loop statement in most of the programming languages −

R programming language provides the following kinds of loop to handle looping requirements. Click the following links to check their detail.

Sr.No.	Loop Type & Description
1	repeat loop Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
2	while loop

	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
3	for loop Like a while statement, except that it tests the condition at the end of the loop body.

R - For Loop

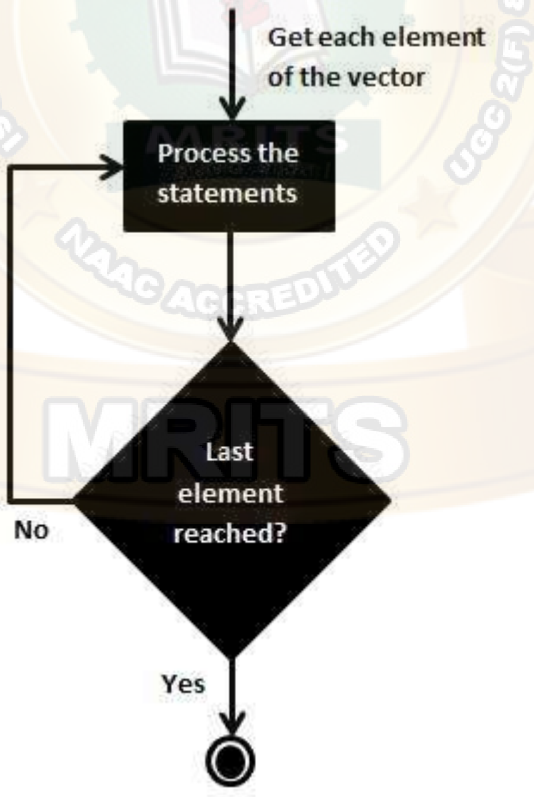
A **For loop** is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax

The basic syntax for creating a **for** loop statement in R is −

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

Flow Diagram



R’s for loops are particularly flexible in that they are not limited to integers, or even numbers in the input. We can pass character vectors, logical vectors, lists or expressions.

Example

```
v <- LETTERS[1:4]
for ( i in v) {
  print(i)
}
```

When the above code is compiled and executed, it produces the following result −

```
[1] "A"
[1] "B"
[1] "C"
[1] "D"
```

Example

```
for (x in 1:10) {
  print(x)
}
```

Example 2: Program to display days of a week.

```
week < - c('Sunday',

        'Monday',

        'Tuesday',

        'Wednesday',

        'Thursday',

        'Friday',

        'Saturday')

 for (day in week)

{    print(day)

}
```

R - While Loop

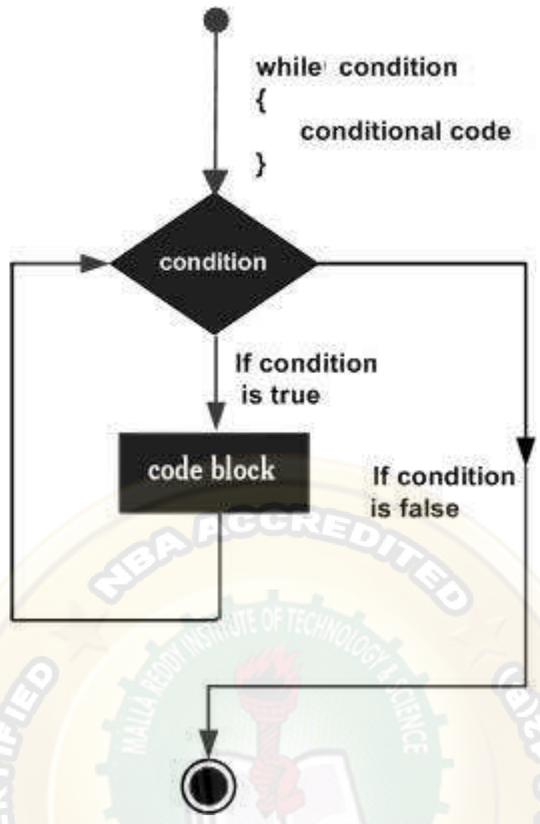
The While loop executes the same code again and again until a stop condition is met.

Syntax

The basic syntax for creating a while loop in R is −

```
while (test_expression) {
statement
}
```

Flow Diagram



Here key point of the **while** loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example1

```
val = 1

while (val <= 5)
{
   print(val)
   val = val + 1
}
```

Example2

```
n < - 5
factorial < - 1
i < - 1
 while (i <= n)
{
   factorial = factorial * i
   i = i + 1
}
 print(factorial)
```

R - Repeat Loop

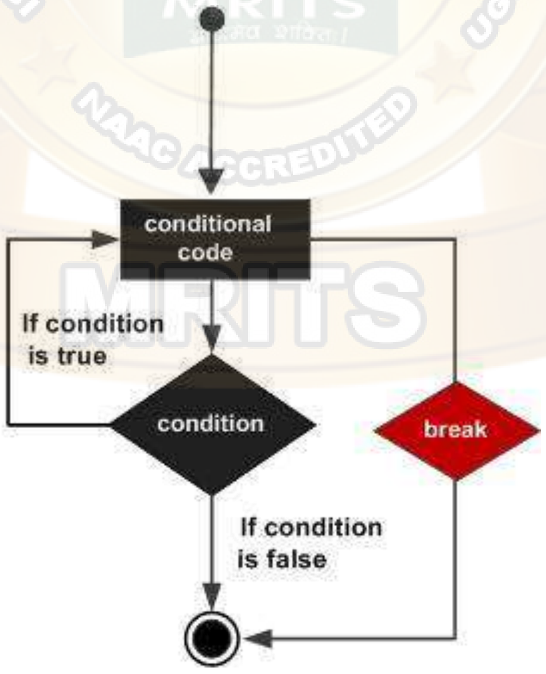
It is a simple loop that will run the same statement or a group of statements repeatedly until the stop condition has been encountered. Repeat loop does not have any condition to terminate the loop, a programmer must specifically place a condition within the loop's body and use the declaration of a break statement to terminate this loop. If no condition is present in the body of the repeat loop then it will iterate infinitely.

Syntax

The basic syntax for creating a repeat loop in R is −

```
repeat 
{
   statement
   
   if( condition ) 
   {
      break
   }
}
```

Flow Diagram



Example1

```
val = 1

repeat
{
    print(val)
    val = val + 1

    if(val > 5)
    {
        break
    }
}
```

Example 2:

```
i < - 0

repeat
{
   print("Geeks 4 geeks!")

   i = i + 1

   if (i == 5)
   {
     break
   }
}
```

Loop Control Statements/ Jump statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

R supports the following control statements. Click the following links to check their detail.

Sr.No.	Control Statement & Description
1	break statement Terminates the **loop** statement and transfers execution to the statement immediately following the loop.
2	Next statement The **next** statement simulates the behavior of R switch.

R - Break Statement

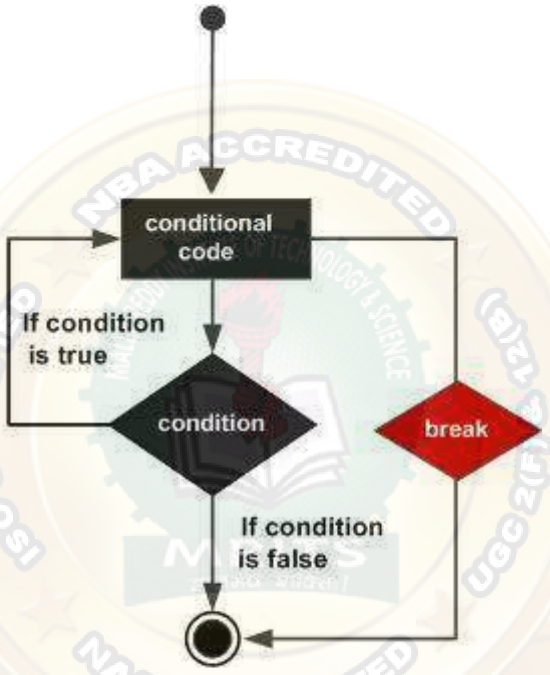
The break statement in R programming language has the following two usages −

- When the break statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
- It can be used to terminate a case in the switch statement

Syntax

The basic syntax for creating a break statement in R is −break

Flow Diagram



Example

```
for (val in 1: 5)
{
   # checking condition
   if (val == 3)
   {
      # using break keyword
      break
   }

   # displaying items in the sequence
   print(val)
}
```

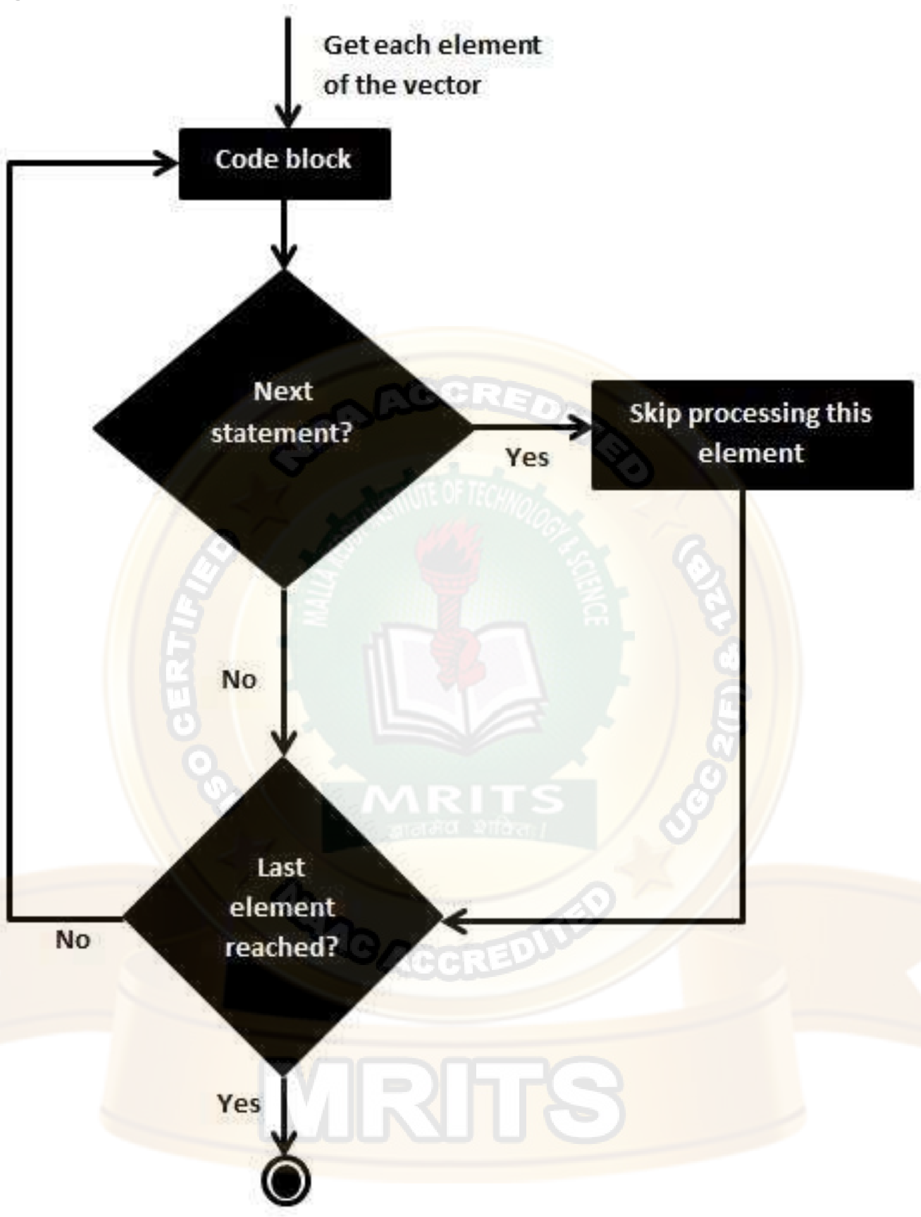
R - Next Statement

The **next** statement in R programming language is useful when we want to skip the current iteration of a loop without terminating it. On encountering next, the R parser skips further evaluation and starts next iteration of the loop.

Syntax

The basic syntax for creating a next statement in R is −next

Flow Diagram



Example

```
for (val in 1: 5)
{
   # checking condition
   if (val == 3)
   {
      # using next keyword
      next
   }

   # displaying items in the sequence
   print(val)
```

}

Loop over a list

A for loop is very valuable when we need to iterate over a list of elements or a range of numbers. Loop can be used to iterate over a list, data frame, vector, matrix or any other object. The braces and square bracket are compulsory.

For Loop in R Example 1: We iterate over all the elements of a vector and print the current value.

```
# Create fruit vector
fruit <- c('Apple', 'Orange', 'Passion fruit', 'Banana')
# Create the for statement
for ( i in fruit){
 print(i)
}
```

R - Functions

Functions are useful when you want to perform a certain task multiple times. A function accepts input arguments and produces the output by executing valid R commands that are inside the function. In R Programming Language when you are creating a function the function name and the file in which you are creating the function need not be the same and you can have one or more function definitions in a single R file.

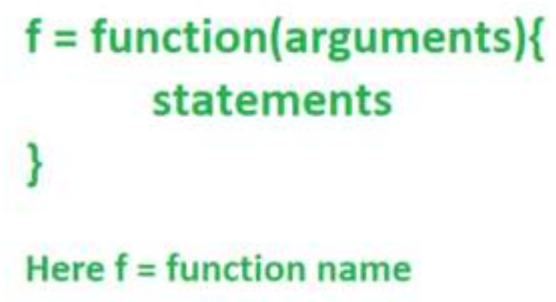
Types of function in R Language

Built-in Function: Built function R is sqrt(), mean(), max(), these function are directly call in the program by users.

User-defined Function: R language allow us to write our own function.

Functions in R Language

Functions are created in R by using the command function(). The general structure of the function file is as follows:



```
f = function(arguments){
        statements
}

Here f = function name
```

Built-in Function in R Programming Language

Here we will use built-in function like sum(), max() and min().

```
print(sum(4:6))

# Find max of numbers 4 and 6.
print(max(4:6))

# Find min of numbers 4 and 6.
print(min(4:6))
```

User-defined Functions in R Programming Language

R provides built-in functions like **print()**, **cat()**, etc. but we can also create our own functions. These functions are called user-defined functions.

```
evenOdd = function(x){
  if(x %% 2 == 0)
    return("even")
  else
    return("odd")
}

print(evenOdd(4))
print(evenOdd(3))
```

Single Input Single Output

Now create a function in R that will take a single input and gives us a single output.

```
areaOfCircle = function(radius){
  area = pi*radius^2
  return(area)
}

print(areaOfCircle(2))
```

Multiple Input Multiple Output

Now create a function in R Language that will take multiple inputs and gives us multiple outputs using a list.

The functions in R Language takes multiple input objects but returned only one object as output, this is, however, not a limitation because you can create lists of all the outputs which you want to create and once the list is created you can access them into the elements of the list and get the answers which you want.

```
Rectangle = function(length, width){
  area = length * width
  perimeter = 2 * (length + width)

  # create an object called result which is
  # a list of area and perimeter
  result = list("Area" = area, "Perimeter" = perimeter)
  return(result)
}

resultList = Rectangle(2, 3)
print(resultList["Area"])
print(resultList["Perimeter"])
```

Inline Functions in R Programming Language

Sometimes creating an R script file, loading it, executing it is a lot of work when you want to just create a very small function. So, what we can do in this kind of situation is an inline function.

To create an inline function you have to use the function command with the argument x and then the expression of the function.

```
f = function(x) x*100

print(f(4))
```

Passing arguments to Functions in R Programming Language

There are several ways you can pass the arguments to the function:

- **Case 1**: Generally in R, the arguments are passed to the function in the same order as in the function definition.
- **Case 2**: If you do not want to follow any order what you can do is you can pass the arguments using the names of the arguments in any order.
- **Case 3**: If the arguments are not passed the default values are used to execute the function.

```
Rectangle = function(length=5, width=4){
  area = length * width
  return(area)
}

# Case 1:
print(Rectangle(2, 3))
```

```
# Case 2:
print(Rectangle(width = 8, length = 4))

# Case 3:
print(Rectangle())
```

<u>Lazy evaluations of Functions in R Programming Language</u>

In R the functions are executed in a lazy fashion. When we say lazy what it means is if some arguments are missing the function is still executed as long as the execution does not involve those arguments.

<u>Example1:</u>

```
Cal= function(a,b,c){
  v = a*b
  return(v)
}

print(Cal(5, 10))
```

<u>Example2:</u>

```
Cal= function(a,b,c){
  v = a*b*c
  return(v)
}

print(Cal(5, 10))
```

Function Arguments in R Programming

Arguments are the parameters provided to a function to perform operations in a programming language. In R programming, we can use as many arguments as we want and are separated by a comma. There is no limit on the number of arguments in a function in R. In this article, we'll discuss different ways of adding arguments in a function in R programming.

<u>Adding Arguments in R</u>

We can pass an argument to a function while calling the function by simply giving the value as an argument inside the parenthesis. Below is an implementation of a function with a single argument.

```
divisbleBy5 <- function(n){
  if(n %% 5 == 0)
  {
    return("number is divisible by 5")
  }
  else
  {
    return("number is not divisible by 5")
  }
}
```

```
# Function call
divisbleBy5(100)
```

Adding Multiple Arguments in R

A function in R programming can have multiple arguments too. Below is an implementation of a function with multiple arguments.

```
divisible <- function(a, b){
  if(a %% b == 0)
  {
    return(paste(a, "is divisible by", b))
  }
  else
  {
    return(paste(a, "is not divisible by", b))
  }
}

# Function call
divisible(7, 3)
```

Adding Default Value in R

Default value in a function is a value that is not required to specify each time the function is called. If the value is passed by the user, then the user-defined value is used by the function otherwise, the default value is used. Below is an implementation of function with default value.

```
divisible <- function(a, b = 3){
  if(a %% b == 0)
  {
    return(paste(a, "is divisible by", b))
  }
  else
  {
    return(paste(a, "is not divisible by", b))
  }
}

# Function call
divisible(10, 5)
divisible(12)
```

Dots Argument

Dots argument (…) is also known as ellipsis which allows the function to take an undefined number of arguments. It allows the function to take an arbitrary number of arguments. Below is an example of a function with an arbitrary number of arguments.

```
fun <- function(n, ...){
  l <- list(n, ...)
  paste(l, collapse = " ")
}
```

```
# Function call
fun(5, 1L,  6i, 15.2, TRUE)
```

Recursive Functions in R Programming

The recursive function uses the concept of recursion to perform iterative tasks they call themselves, again and again, which acts as a loop. These kinds of functions need a stopping condition so that they can stop looping continuously.

Recursive functions call themselves. They break down the problem into smaller components. The function() calls itself within the original function() on each of the smaller components. After this, the results will be put together to solve the original problem.

Example1:

```
fac <- function(x){
   if(x==0 || x==1)
   {
      return(1)
   }
   else
   {
      return(x*fac(x-1))
   }
}

fac(3)
```

Nested Functions

There are two ways to create a nested function:

- Call a function within another function.
- Write a function within a function.

- Call a function within another function.

Example

Call a function within another function:

```
Nested_function <- function(x, y) {
  a <- x + y
  return(a)
}

Nested_function(Nested_function(2,2), Nested_function(3,3))
```

- Write a function within a function.

Example

Write a function within a function:

```
Outer_func <- function(x) {
  Inner_func <- function(y) {
    a <- x + y
    return(a)
  }
  return (Inner_func)
}
output <- Outer_func(3) # To call the Outer_func
output(5)
```

Loading an R package

Packages

Packages are collections of R functions, data, and compiled code in a well-defined format. The directory where packages are stored is called the library. R comes with a standard set of packages. Others are available for download and installation. Once installed, they have to be loaded into the session to be used.

```
.libPaths() # get library location
library()   # see all packages installed
search()    # see packages currently loaded
```

Adding Packages

You can expand the types of analyses you do be adding other packages. A complete list of contributed packages is available from CRAN.

Follow these steps:

1. Download and install a package (you only need to do this once).

2. To use the package, invoke the **library(*package*)** command to load it into the current session. (You need to do this once in each session, unless you customize your environment to automatically load it each time.)

   On **MS Windows**:

1. Choose **Install Packages** from the **Packages** menu.
2. Select a **CRAN Mirror.** (e.g. Norway)
3. Select a package. (e.g. boot)
4. Then use the **library(*package*)** function to load it for use. (e.g. library(boot))

Load an R Package

There are basically two extremely important functions when it comes down to R packages:

- install.packages(), which as you can expect, installs a given package.
- library() which loads packages, i.e. attaches them to the search list on your R workspace.

To install packages, you need administrator privileges. This means that install.packages() will thus not work in the DataCamp interface. However, almost all CRAN packages are installed on our servers. You can load them with library().

In this exercise, you'll be learning how to load the **ggplot2** package, a powerful package for data visualization. You'll use it to create a plot of two variables of the **mtcars** data frame. The data has already been prepared for you in the workspace.

Before starting, execute the following commands in the console:

- search(), to look at the currently attached packages and
- qplot(mtcars$wt, mtcars$hp), to build a plot of two variables of the mtcars data frame.

Mathematical Functions in R

R provides the various mathematical functions to perform the mathematical calculation. These mathematical functions are very helpful to find absolute value, square value and much more calculations. In R, there are the following functions which are used:

S. No	Function	Description	Example

1.	abs(x)	It returns the absolute value of input x.	x<- -4
			print(abs(x))
			Output
			[1] 4

2.	sqrt(x)	It returns the square root of input x.	x<- 4
			print(sqrt(x))
			Output
			[1] 2

3.	ceiling(x)	It returns the smallest integer which is larger than or equal to x.	x<- 4.5
			print(ceiling(x))
			Output
			[1] 5

4.	floor(x)	It returns the largest integer, which is smaller than or equal to x.	x<- 2.5
			print(floor(x))
			Output
			[1] 2

5.	trunc(x)	It returns the truncate value of input x.	x<- c(1.2,2.5,8.1)
			print(trunc(x))
			Output
			[1] 1 2 8

6.	round(x, digits=n)	It returns round value of input x.	x<- -4
			print(abs(x))
			Output
			4

7.	cos(x), sin(x), tan(x)	It returns cos(x), sin(x) value of input x.	x<- 4
			print(cos(x))
			print(sin(x))
			print(tan(x))
			Output
			[1] -06536436
			[2] -0.7568025
			[3] 1.157821

8.	log(x)	It returns natural logarithm of input x.	x<- 4
			print(log(x))
			Output
			[1] 1.386294

9.	log10(x)	It returns common logarithm of input x.	x<- 4
			print(log10(x))
			Output
			[1] 0.60206

10.	exp(x)	It returns exponent.	x<- 4
			print(exp(x))
			Output
			[1] 54.59815

Unit-5

Data Reduction

Imagine that you have selected data from the AllElectronics data warehouse for analysis. The data set will likely be huge! Complex data analysis and mining on huge amounts of data can take a long time, making such analysis impractical or infeasible.

Data reduction techniques can be applied to obtain a reduced representation of the data set that is much smaller in volume, yet closely maintains the integrity of the original data. That is, mining on the reduced data set should be more efficient yet produce the same (or almost the same) analytical results.

Overview of Data Reduction Strategies

Data reduction strategies include *dimensionality reduction*, *numerosity reduction*, and *data compression*.

Dimensionality reduction is the process of reducing the number of random variables or attributes under consideration.

- Dimensionality reduction methods include *wavelet transforms* and *principal components analysis*, which transform or project the original data onto a smaller space.
- *Attribute subset selection* is a method of dimensionality reduction in which irrelevant, weakly relevant, or redundant attributes or dimensions are detected and removed.

Numerosity reduction techniques replace the original data volume by alternative, smaller forms of data representation.

- These techniques may be parametric or nonparametric.
- For *parametric methods*, a model is used to estimate the data, so that typically only the data parameters need to be stored, instead of the actual data. (Outliers may also be stored.) Regression and log-linear models are examples.
- *Nonparametric methods* for storing reduced representations of the data include *histograms*, *clustering*, *sampling*, and *data cube aggregation*.

In **data compression**, transformations are applied so as to obtain a reduced or "compressed" representation of the original data. If the original data can be *reconstructed* from the compressed data without any information loss, the data reduction is called **lossless**.

- If, instead, we can reconstruct only an approximation of the original data, then the data reduction is called **lossy**.
- There are several lossless algorithms for string compression; however, they typically allow only limited data manipulation.
- Dimensionality reduction and numerosity reduction techniques can also be considered forms of data compression.
- There are many other ways of organizing methods of data reduction. The computational time spent on data reduction should not outweigh or "erase" the time saved by mining on a reduced data set size.

Wavelet Transforms

The **discrete wavelet transform (DWT)** is a linear signal processing technique that, when applied to a data vector ***X***, transforms it to a numerically different vector, ***X***', of **wavelet coefficients**.

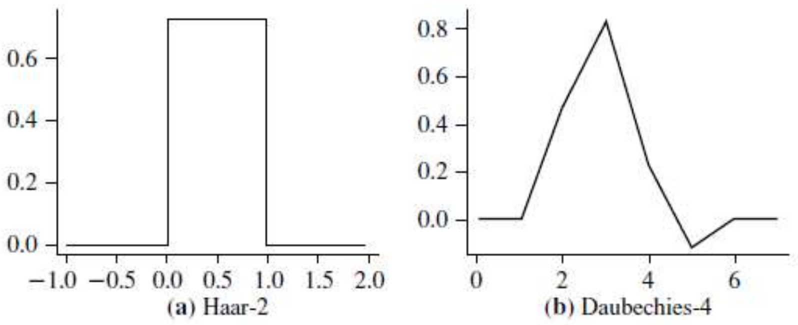
- The two vectors are of the same length. When applying this technique to data reduction, we consider each tuple as an *n*-dimensional data vector, that is, ***X* =(**.*x*1,*x*2, .,., ,*xn)*, depicting *n* measurements made on the tuple from *n* database attributes.
- *"How can this technique be useful for data reduction if the wavelet transformed data are of the same length as the original data?"* The usefulness lies in the fact that the wavelet transformed data can be truncated. A compressed approximation of the data can be retained by storing only a small fraction of the strongest of the wavelet coefficients.

- The technique also works to remove noise without smoothing out the main features of the data, making it effective for data cleaning as well. Given a set of coefficients, an approximation of the original data can be constructed by applying the *inverse* of the DWT used.

- The DWT is closely related to the *discrete Fourier transform (DFT)*, a signal processing technique involving sines and cosines. In general, however, the DWT achieves better lossy compression.
- Unlike the DFT, wavelets are quite localized in space, contributing to the conservation of local detail.

There is only one DFT, yet there are several families of DWTs. Figure 3.4 shows some wavelet families. Popular wavelet transforms include the Haar-2, Daubechies-4, and Daubechies-6. The general procedure for applying a discrete wavelet transform uses a hierarchical *pyramid algorithm* that halves the data at each iteration, resulting in fast computational speed.

The method is as follows:
1. The length, *L*, of the input data vector must be an integer power of 2. This condition can be met by padding the data vector with zeros as necessary (*L* >=*n*).
2. Each transform involves applying two functions. The first applies some data smoothing, such as a sum or weighted average. The second performs a weighted difference, which acts to bring out the detailed features of the data.
3. The two functions are applied to pairs of data points in ***X***, that is, to all pairs of measurements .*x*2*i* ,*x*2*i*+1. This results in two data sets of length *L*=2. In general, these represent a smoothed or low-frequency version of the input data and the high frequency content of it, respectively.
4. The two functions are recursively applied to the data sets obtained in the previous loop, until the resulting data sets obtained are of length 2. **5.** Selected values from the data sets obtained in the previous iterations are designated the wavelet coefficients of the transformed data.



Examples of wavelet families. The number next to a wavelet name is the number of *vanishing moments* of the wavelet. This is a set of mathematical relationships that the coefficients must satisfy and is related to the number of coefficients.

- Equivalently, a matrix multiplication can be applied to the input data in order to obtain the wavelet coefficients, where the matrix used depends on the given DWT.
- The matrix must be **orthonormal**, meaning that the columns are unit vectors and are mutually orthogonal, so that the matrix inverse is just its transpose. By factoring the matrix used into a product of a few sparse matrices, the resulting "fast DWT" algorithm has a complexity of *O(n)* for an input vector of length *n*.

- Wavelet transforms can be applied to multidimensional data such as a data cube. This is done by first applying the transform to the first dimension, then to the second, and so on.
- Lossy compression by wavelets is reportedly better than JPEG compression, the current commercial standard.
- Wavelet transforms have many real world applications, including the compression of fingerprint images, computer vision, analysis of time-series data, and data cleaning.

Principal Components Analysis

Principal components analysis (**PCA**; also called the K-L, method) searches for *k n*-dimensional orthogonal vectors that can best be used to represent the data, where $k <= n$.
The original data are thus projected onto a much smaller space, resulting in dimensionality reduction.

The basic procedure is as follows:
1. The input data are normalized, so that each attribute falls within the same range. This step helps ensure that attributes with large domains will not dominate attributes with smaller domains.
2. PCA computes *k* orthonormal vectors that provide a basis for the normalized input data.
3. The principal components are sorted in order of decreasing "significance" or strength.
4. Because the components are sorted in decreasing order of "significance," the data size can be reduced by eliminating the weaker components, that is, those with low variance. Using the strongest principal components, it should be possible to reconstruct a good approximation of the original data.

Attribute Subset Selection

Data sets for analysis may contain hundreds of attributes, many of which may be irrelevant to the mining task or redundant. For example, if the task is to classify customers based on whether or not they are likely to purchase a popular new CD at *AllElectronics* when notified of a sale, attributes such as the customer's telephone number are likely to be irrelevant, unlike attributes such as *age* or *music taste*.

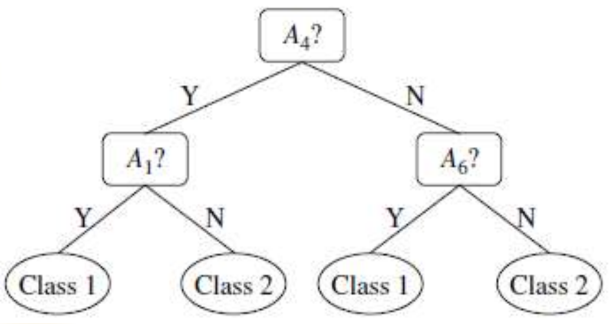
Although it may be possible for a domain expert to pick out some of the useful attributes, this can be a difficult and time consuming task, especially when the data's behavior is not well known. (Hence, a reason behind its analysis!) Leaving out relevant attributes or keeping irrelevant attributes may be detrimental, causing confusion for the mining algorithm employed. This can result in discovered patterns of poor quality. In addition, the added volume of irrelevant or redundant attributes can slow down the mining process.

Attribute subset selection reduces the data set size by removing irrelevant or redundant attributes (or dimensions). The goal of attribute subset selection is to find a minimum set of attributes such that the resulting probability distribution of the data classes is as close as possible to the original distribution obtained using all attributes. Mining on a reduced set of attributes has an additional benefit: It reduces the number of attributes appearing in the discovered patterns, helping to make the patterns easier to understand.

Therefore, heuristic methods that explore a reduced search space are commonly used for attribute subset selection. These methods are typically **greedy** in that, while searching through attribute space, they always make what looks to be the best choice at the time. Their strategy is to make a locally optimal choice in the hope that this will lead to a globally optimal solution. Such greedy methods are effective in practice and may come close to estimating an optimal solution.

The "best" (and "worst") attributes are typically determined using tests of statistical significance, which assume that the attributes are independent of one another. Many other attribute evaluation measures can be used such as the *information gain* measure used in building decision trees for classification.5

Basic heuristic methods of attribute subset selection include the techniques that follow, some of which are illustrated in Figure 3.6.

Forward selection	Backward elimination	Decision tree induction
Initial attribute set: $\{A_1, A_2, A_3, A_4, A_5, A_6\}$ Initial reduced set: $\{\}$ => $\{A_1\}$ => $\{A_1, A_4\}$ => Reduced attribute set: $\{A_1, A_4, A_6\}$	Initial attribute set: $\{A_1, A_2, A_3, A_4, A_5, A_6\}$ => $\{A_1, A_3, A_4, A_5, A_6\}$ => $\{A_1, A_4, A_5, A_6\}$ => Reduced attribute set: $\{A_1, A_4, A_6\}$	Initial attribute set: $\{A_1, A_2, A_3, A_4, A_5, A_6\}$  A_4? Y N A_1? A_6? Y N Y N Class 1 Class 2 Class 1 Class 2 => Reduced attribute set: $\{A_1, A_4, A_6\}$

Greedy (heuristic) methods for attribute subset selection.

1. Stepwise forward selection: The procedure starts with an empty set of attributes as the reduced set. The best of the original attributes is determined and added to the reduced set. At each subsequent iteration or step, the best of the remaining original attributes is added to the set.
2. Stepwise backward elimination: The procedure starts with the full set of attributes. At each step, it removes the worst attribute remaining in the set.
3. Combination of forward selection and backward elimination: The stepwise forward selection and backward elimination methods can be combined so that, at each step, the procedure selects the best attribute and removes the worst from among the remaining attributes.
4. Decision tree induction: Decision tree algorithms (e.g., ID3, C4.5, and CART) were originally intended for classification. Decision tree induction constructs a flowchart like structure where each internal (nonleaf) node denotes a test on an attribute, each branch corresponds to an outcome of the test, and each external (leaf) node denotes a class prediction. At each node, the algorithm chooses the "best" attribute to partition the data into individual classes.
When decision tree induction is used for attribute subset selection, a tree is constructed from the given data. All attributes that do not appear in the tree are assumed to be irrelevant. The set of attributes appearing in the tree form the reduced subset of attributes.

Regression and Log-Linear Models: Parametric Data Reduction

Regression and log-linear models can be used to approximate the given data. In (simple) **linear regression**, the data are modeled to fit a straight line. For example, a random variable, *y* (called a *response variable*), can be modeled as a linear function of another random variable, *x* (called a *predictor variable*), with the equation

$$y = wx + b, \qquad (3.7)$$

where the variance of *y* is assumed to be constant. In the context of data mining, *x* and *y* are numeric database attributes. The coefficients, *w* and *b* (called *regression coefficients*), specify the slope of the line and the *y*-intercept, respectively.

Multiple linear regression is an extension of (simple) linear regression, which allows a response variable, *y*, to be modeled as a linear function of two or more predictor variables.

Log-linear models approximate discrete multidimensional probability distributions.

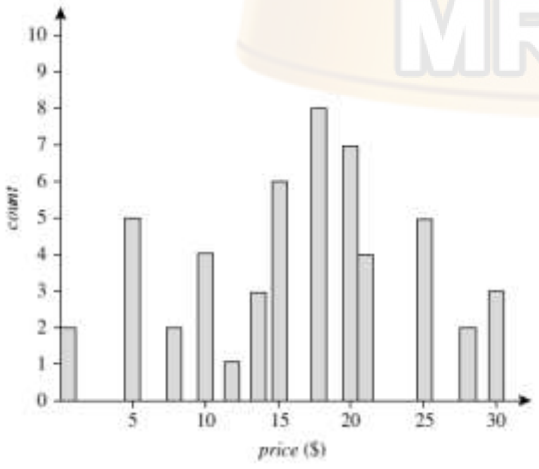
- Given a set of tuples in *n* dimensions (e.g., described by *n* attributes), we can consider each tuple as a point in an *n*-dimensional space.
- Log-linear models can be used to estimate the probability of each point in a multidimensional space for a set of discretized attributes, based on a smaller subset of dimensional combinations.
- This allows a higher-dimensional data space to be constructed from lower-dimensional spaces.

Histograms

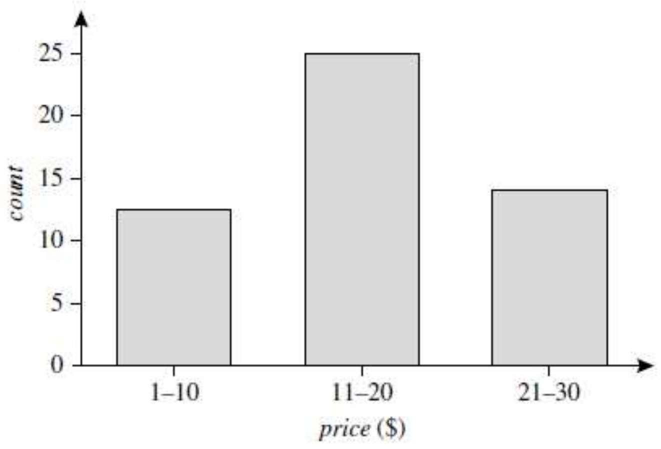
Histograms use discarding to approximate data distributions and are a popular form of data reduction. A **histogram** for an attribute, *A*, partitions the data distribution of *A* into disjoint subsets, referred to as *buckets or bins*. If each bucket represents only a single attribute–value/frequency pair, the buckets are called *singleton buckets*. Often, buckets instead represent continuous ranges for the given attribute.

Example 3.3 **Histograms.** The following data are a list of *AllElectronics* prices for commonly sold items (rounded to the nearest dollar). The numbers have been sorted: 1, 1, 5, 5, 5, 5, 5, 8, 8, 10, 10, 10, 10, 12, 14, 14, 14, 15, 15, 15, 15, 15, 15, 18, 18, 18, 18, 18, 18, 18, 18, 20, 20, 20, 20, 20, 20, 20, 21, 21, 21, 21, 25, 25, 25, 25, 25, 28, 28, 30, 30, 30.

Figure 3.7 shows a histogram for the data using singleton buckets. To further reduce the data, it is common to have each bucket denote a continuous value range for the given attribute. In Figure 3.8, each bucket represents a different \$10 range for *price*. ■



A histogram for *price* using singleton buckets—each bucket represents one price–value/frequency pair.



An equal-width histogram for *price*, where values are aggregated so that each bucket has a uniform width of $10.

Clustering

- Clustering techniques consider data tuples as objects.
- They partition the objects into groups, or *clusters*, so that objects within a cluster are "similar" to one another and "dissimilar" to objects in other clusters.
- Similarity is commonly defined in terms of how "close" the objects are in space, based on a distance function.
- The "quality" of a cluster may be represented by its *diameter*, the maximum distance between any two objects in the cluster.
- **Centroid distance** is an alternative measure of cluster quality and is defined as the average distance of each cluster object from the cluster centroid.
- In data reduction, the cluster representations of the data are used to replace the actual data.

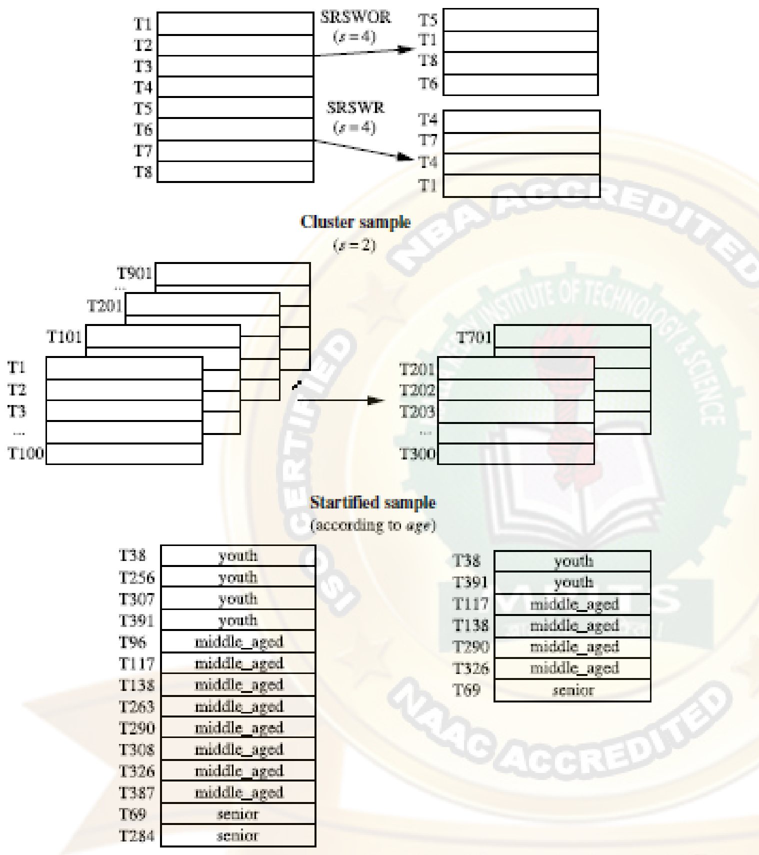
Sampling

Sampling can be used as a data reduction technique because it allows a large data set to be represented by a much smaller random data sample (or subset). Suppose that a large data set, *D*, contains *N* tuples. Let's look at the most common ways that we could sample *D* for data reduction, as illustrated in Figure 3.9.

Simple random sample without replacement (SRSWOR) of size *s*: This is created by drawing *s* of the *N* tuples from *D* ($s < N$), where the probability of drawing any tuple in *D* is 1=*N*, that is, all tuples are equally likely to be sampled.

Simple random sample with replacement (SRSWR) of size *s*: This is similar to SRSWOR, except that each time a tuple is drawn from *D*, it is recorded and then *replaced*. That is, after a tuple is drawn, it is placed back in *D* so that it may be drawn again.

Cluster sample: If the tuples in *D* are grouped into *M* mutually disjoint "clusters," then an SRS of *s* clusters can be obtained, where $s < M$. For example, tuples in a database are usually retrieved a page at a time, so that each page can be considered a cluster. A reduced data representation can be obtained by applying, say, SRSWOR to the pages, resulting in a cluster sample of the tuples. Other clustering criteria conveying rich semantics can also be explored. For example, in a spatial database, we may choose to define clusters geographically based on how closely different areas are located.



3.9 Sampling can be used for data reduction.

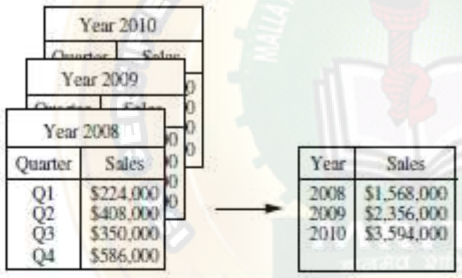
Stratified sample: If *D* is divided intomutually disjoint parts called *strata*, a stratified sample of *D* is generated by obtaining an SRS at each stratum. This helps ensure a representative sample, especially when the data are skewed. For example, a stratified sample may be obtained fromcustomer data, where a stratum is created for each customer age group. In this way, the age group having the smallest number of customers will be sure to be represented

An advantage of sampling for data reduction is that the cost of obtaining a sample *is proportional to the size of the sample*, *s*, as opposed to *N*, the data set size. Hence, sampling complexity is potentially *sublinear* to the size of the data. Other data reduction techniques can require at least one complete pass through *D*. For a fixed sample size, sampling complexity increases only linearly as the number of data dimensions, *n*, increases, whereas techniques using histograms, for example, increase exponentially in *n*.

When applied to data reduction, sampling is most commonly used to estimate the answer to an aggregate query. It is possible (using the central limit theorem) to determine a sufficient sample size for estimating a given function within a specified degree of error. This sample size, *s*, may be extremely small in comparison to *N*. Sampling is a natural choice for the progressive refinement of a reduced data set. Such a set can be further refined by simply increasing the sample size.

Data Cube Aggregation

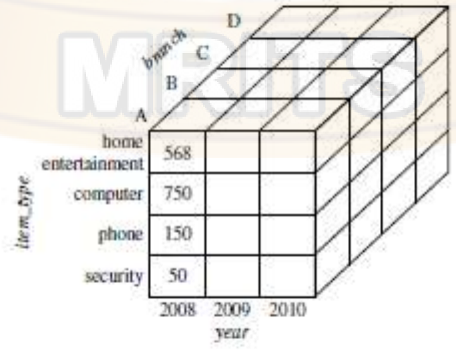
Imagine that you have collected the data for your analysis. These data consist of the *AllElectronics* sales per quarter, for the years 2008 to 2010. You are, however, interested in the annual sales (total per year), rather than the total per quarter. Thus, the data can be *aggregated* so that the resulting data summarize the total sales per year instead of per quarter. This aggregation is illustrated in Figure 3.10. The resulting data set is smaller in volume, without loss of information necessary for the analysis task. Data cubes are discussed in detail in Chapter 4 on data warehousing and Chapter 5 on data cube technology. We briefly introduce some concepts here. Data cubes store


Year 2010
Year 2009

Year 2008	
Quarter	Sales
Q1	$224,000
Q2	$408,000
Q3	$350,000
Q4	$586,000

Year	Sales
2008	$1,568,000
2009	$2,356,000
2010	$3,594,000

Sales data for a given branch of *AllElectronics* for the years 2008 through 2010. On the *left*, the sales are shown per quarter. On the *right*, the data are aggregated to provide the annual sales.



A data cube for sales at *AllElectronics*.

Multi-dimensional aggregated information. For example, the above Figure shows a data cube for multidimensional analysis of sales data with respect to annual sales per item type for each *AllElectronics* branch. Each cell holds an aggregate data value, corresponding to the data point in multidimensional space. (For readability, only some cell values are shown.) *Concept hierarchies* may exist for each attribute, allowing the analysis of data at multiple abstraction levels. For example, a hierarchy for *branch* could allow branches to be grouped into regions, based on their address. Data cubes provide fast access to precomputed, summarized data, thereby benefiting online analytical processing as well as data mining.

The cube created at the lowest abstraction level is referred to as the **base cuboid**. The base cuboid should correspond to an individual entity of interest such as *sales* or *customer*. In other words, the lowest level should be usable, or useful for the analysis. A cube at the highest level of abstraction is the **apex cuboid**. For the sales data in Figure 3.11, the apex cuboid would give one total—the total *sales* for all three years, for all item types, and for all branches. Data cubes created for varying levels of abstraction are often referred to as *cuboids*, so that a data cube may instead refer to a *lattice of cuboids*. Each higher abstraction level further reduces the resulting data size. When replying to data mining requests, the *smallest* available cuboid relevant to the given task should be used.

Data Visualization

Data visualization aims to communicate data clearly and effectively through graphical representation. Data visualization has been used extensively in many applications—for example, at work for reporting, managing business operations, and tracking progress of tasks. More popularly, we can take advantage of visualization techniques to discover data relationships that are otherwise not easily observable by looking at the raw data. Nowadays, people also use data visualization to create fun and interesting graphics.

This section start with multidimensional data such as those stored in relational databases. We discuss several representative approaches, including pixel-oriented techniques, geometric projection techniques, icon-based techniques, and hierarchical and graph-based techniques. We then discuss the visualization of complex data and relations.

Pixel-Oriented Visualization Techniques

A simple way to visualize the value of a dimension is to use a pixel where the color of the pixel reflects the dimension's value. For a data set of *m* dimensions, **pixel-oriented techniques** create *m* windows on the screen, one for each dimension. The *m* dimension values of a record are mapped to*m*pixels at the corresponding positions in the windows.

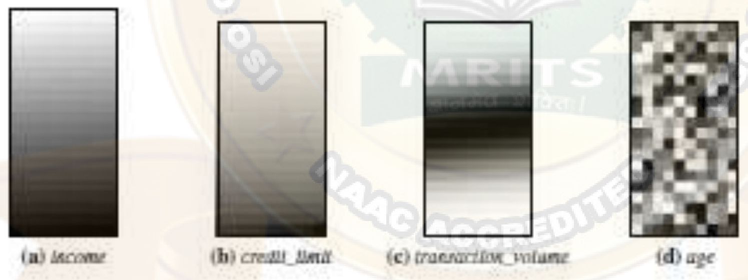
The colors of the pixels reflect the corresponding values. Inside a window, the data values are arranged in some global order shared by all windows. The global order may be obtained by sorting all data records in a way that's meaningful for the task at hand.

Pixel-oriented visualization. AllElectronics maintains a customer information table, which consists of four dimensions: income, credit limit, transaction volume, and age. Can we analyze the correlation between income and the other attributes by visualization? We can sort all customers in income-ascending order, and use this order to lay out the customer data in the four visualization windows, as shown in Figure 2.10. The pixel colors are chosen so that the smaller the value, the lighter the shading. Using pixelbased visualization, we can easily observe the following: credit limit increases as income increases; customers whose income is in the middle range are more likely to purchase more from AllElectronics; there is no clear correlation between income and age.

In pixel-oriented techniques, data records can also be ordered in a query-dependent way. For example, given a point query, we can sort all records in descending order of similarity to the point query.

Filling a window by laying out the data records in a linear way may not work well for a wide window. The first pixel in a row is far away fromthe last pixel in the previous row, though they are next to each other in the global order. Moreover, a pixel is next to the one above it in the window, even though the two are not next to each other in the global order. To solve this problem, we can lay out the data records in a space-filling curve to fill the windows. A space-filling curve is a curve with a range that covers the entire n-dimensional unit hypercube. Since the visualization windows are 2-D, we can use any 2-D space-filling curve.

Figure 2.11 shows some frequently used 2-D space-filling curves. Note that the windows do not have to be rectangular. For example, the circle segment technique uses windows in the shape of segments of a circle, as illustrated in Figure 2.12. This technique can ease the comparison of dimensions because the dimension windows are located side by side and form a circle.



0 Pixel-oriented visualization of four attributes by sorting all customers in *income* ascending order.

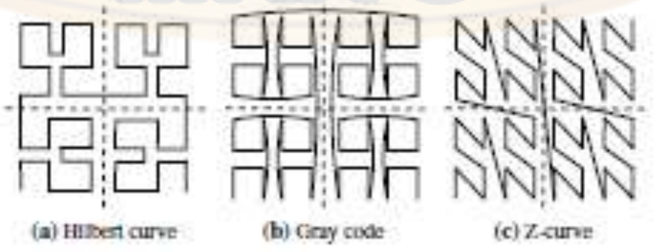


Figure 2.11 Some frequently used 2-D space-filling curves.

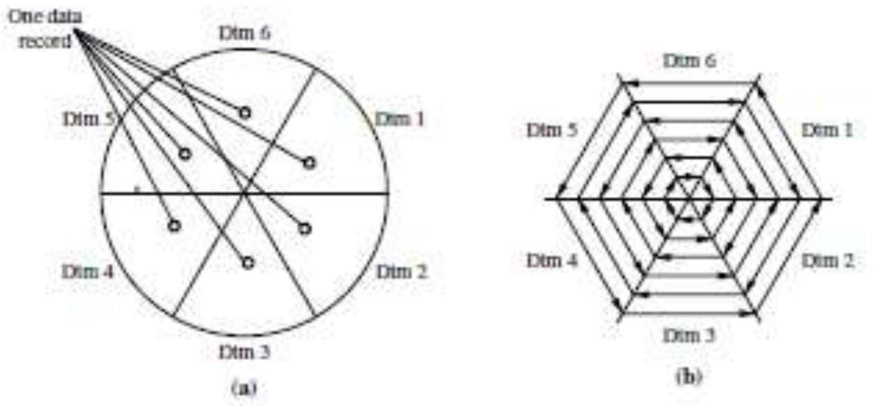


Figure 2.12 The circle segment technique. (a) Representing a data record in circle segments. (b) Laying out pixels in circle segments.

Geometric Projection Visualization Techniques

A drawback of pixel-oriented visualization techniques is that they cannot help us much in understanding the distribution of data in a multidimensional space. For example, they do not show whether there is a dense area in a multidimensional subspace. **Geometric projection techniques** help users find interesting projections of multidimensional data sets. The central challenge the geometric projection techniques try to address is how to visualize a high-dimensional space on a 2-D display.

A **scatter plot** displays 2-D data points using Cartesian coordinates. A third dimension can be added using different colors or shapes to represent different data points. Figure 2.13 shows an example, where *X* and *Y* are two spatial attributes and the third dimension is represented by different shapes. Through this visualization, we can see that points of types "+" and "_" tend to be colocated.

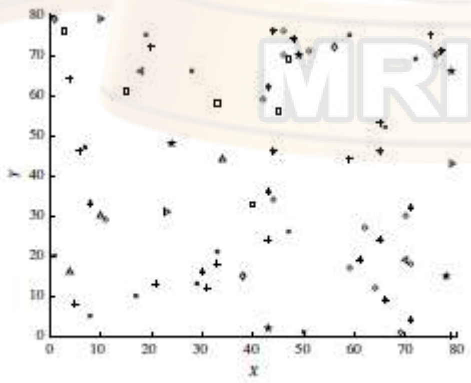


Figure 2.13 Visualization of a 2-D data set using a scatter plot. *Source: www.cs.sfu.ca/jpei/publications/rareevent-geoinformatica06.pdf.*

A 3-D scatter plot uses three axes in a Cartesian coordinate system. If it also uses color, it can display up to 4-D data points (Figure 2.14).

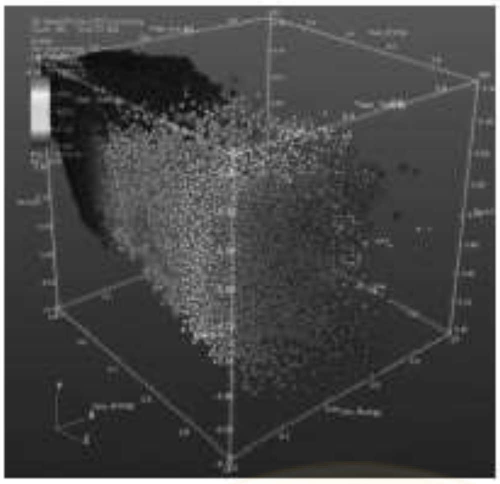


Figure 2.14 Visualization of a 3-D data set using a scatter plot. *Source: http://upload.wikimedia.org/wikipedia/commons/c/c4/Scatter_plot.jpg.*

For data sets with more than four dimensions, scatter plots are usually ineffective. The **scatter-plot matrix** technique is a useful extension to the scatter plot. For an *n* dimensional data set, a scatter-plot matrix is an *n_n* grid of 2-D scatter plots that provides a visualization of each dimension with every other dimension. Figure 2.15 shows an example, which visualizes the Iris data set. The data set consists of 450 samples from each of three species of Iris flowers. There are five dimensions in the data set: length and width of sepal and petal, and species. The scatter-plot matrix becomes less effective as the dimensionality increases. Another popular technique, called parallel coordinates, can handle higher dimensionality. To visualize *n*-dimensional data points, the **parallel coordinates** technique draws *n* equally spaced axes, one for each dimension, parallel to one of the display axes.

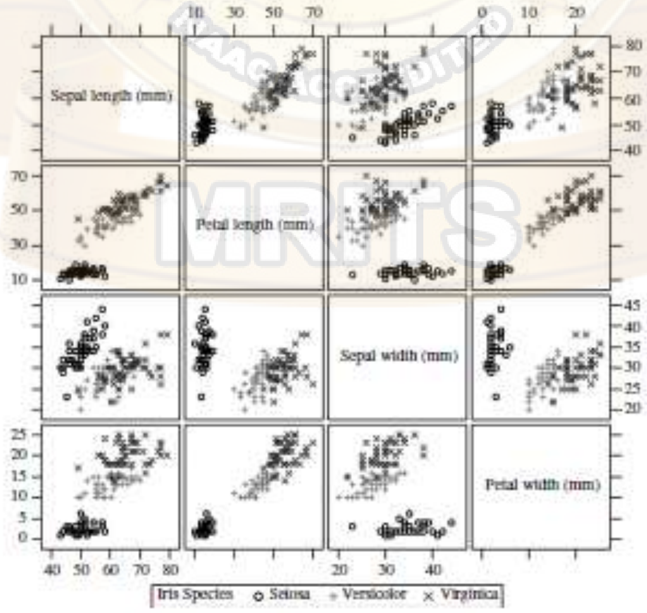


Figure 2.15 Visualization of the Iris data set using a scatter-plot matrix. *Source: http://support.sas.com/documentation/cdl/en/grstatproc/61948/HTML/default/images/gsgscmat.gif.*

A data record is represented by a polygonal line that intersects each axis at the point corresponding to the associated dimension value.

A major limitation of the parallel coordinates technique is that it cannot effectively show a data set of many records. Even for a data set of several thousand records, visual clutter and overlap often reduce the readability of the visualization and make the patterns hard to find.

Icon-Based Visualization Techniques

Icon-based visualization techniques use small icons to represent multidimensional data values. We look at two popular icon-based techniques: *Chernoff faces* and *stick figures*.

Chernoff faces were introduced in 1973 by statistician Herman Chernoff. They display multidimensional data of up to 18 variables (or dimensions) as a cartoon human face (Figure 2.17). Chernoff faces help reveal trends in the data. Components of the face, such as the eyes, ears, mouth, and nose, represent values of the dimensions by their shape, size, placement, and orientation.

For example, dimensions can be mapped to the following facial characteristics: eye size, eye spacing, nose length, nose width, mouth curvature, mouth width, mouth openness, pupil size, eyebrow slant, eye eccentricity, and head eccentricity. Chernoff faces make use of the ability of the human mind to recognize small differences in facial characteristics and to assimilate many facial characteristics at once.

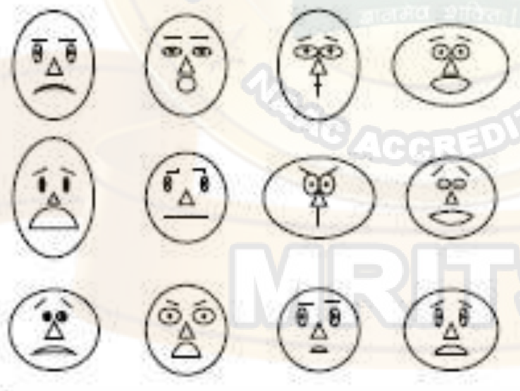
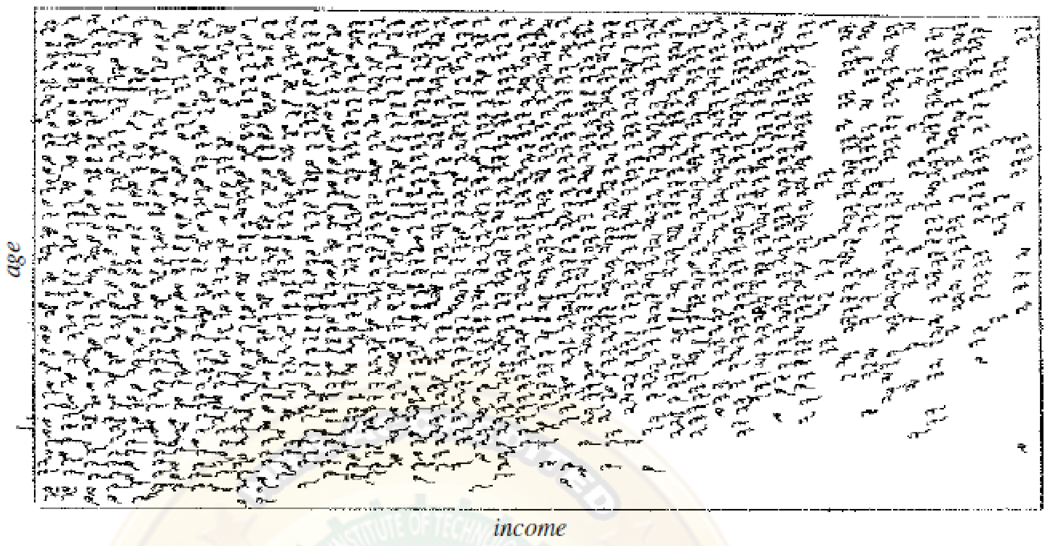


Figure 2.17 Chernoff faces. Each face represents an n-dimensional data point ($n \leq 18$).

Viewing large tables of data can be tedious. By condensing the data, Chernoff faces make the data easier for users to digest. In this way, they facilitate visualization of regularities and irregularities present in the data, although their power in relating multiple relationships is limited. Another limitation is that specific data values are not shown.

Furthermore, facial features vary in perceived importance. This means that the similarity of two faces (representing twomultidimensional data points) can vary depending on the order in which dimensions are assigned to facial characteristics. Therefore, this mapping should be carefully chosen. Eye size and eyebrow slant have been found to be important. *Asymmetrical Chernoff* faces were proposed as an extension to the original technique.



2.18 Census data represented using stick figures. *Source:* Professor G. Grinstein, Department of Computer Science, University of Massachusetts at Lowell.

Since a face has vertical symmetry (along the y-axis), the left and right side of a face are identical, which wastes space. Asymmetrical Chernoff faces double the number of facial characteristics, thus allowing up to 36 dimensions to be displayed. The stick figure visualization technique maps multidimensional data to five-piece stick figures, where each figure has four limbs and a body. Two dimensions are mapped to the display (x and y) axes and the remaining dimensions are mapped to the angle and/or length of the limbs. Figure 2.18 shows census data, where age and income are mapped to the display axes, and the remaining dimensions (gender, education, and so on) are mapped to stick figures. If the data items are relatively dense with respect to the two display dimensions, the resulting visualization shows texture patterns, reflecting data trends.

Hierarchical Visualization Techniques

The visualization techniques discussed so far focus on visualizing multiple dimensions simultaneously. However, for a large data set of high dimensionality, it would be difficult to visualize all dimensions at the same time. **Hierarchical visualization techniques** partition all dimensions into subsets (i.e., subspaces). The subspaces are visualized in a hierarchical manner. **“Worlds-within-Worlds,”** also known as *n*-Vision, is a representative hierarchical visualization method. Suppose we want to visualize a 6-D data set, where the dimensions are

F,*X*1, . , . , . , *X*5.We want to observe how dimension *F* changes with respect to the other dimensions.

We can first fix the values of dimensions *X*3,*X*4,*X*5 to some selected values, say, *c*3, *c*4, *c*5.We can then visualize *F*,*X*1,*X*2 using a 3-D plot, called a *world*, as shown in Figure 2.19. The position of the origin of the inner world is located at the point .*c*3, *c*4, *c*5/ in the outer world, which is another 3-D plot using dimensions *X*3,*X*4,*X*5. A user can interactively change, in the outer world, the location of the origin of the inner world. The user then views the resulting changes of the inner world. Moreover, a user can vary the dimensions used in the inner world and the outer world. Given more dimensions, more levels of worlds can be used, which is why the method is called "worlds-within worlds."

As another example of hierarchical visualization methods, **tree-maps** display hierarchical data as a set of nested rectangles. For example, Figure 2.20 shows a tree-map visualizing Google news stories. All news stories are organized into seven categories, each shown in a large rectangle of a unique color. Within each category (i.e., each rectangle at the top level), the news stories are further partitioned into smaller subcategories.

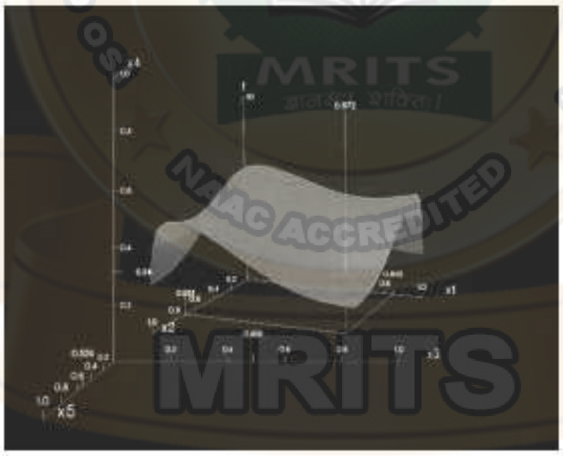


Figure 2.19 "Worlds-within-Worlds" (also known as *n*-Vision). *Source: http://graphics.cs.columbia.edu/projects/AutoVisual/images/1.dipstick.5.gif.*

Visualizing Complex Data and Relations

In early days, visualization techniques were mainly for numeric data. Recently, more and more non-numeric data, such as text and social networks, have become available. Visualizing and analyzing such data attracts a lot of interest. There are many new visualization techniques dedicated to these kinds of data.

For example, many people on theWeb tag various objects such as pictures, blog entries, and product reviews. A **tag cloud** is a visualization of statistics of user-generated tags. Often, in a tag cloud, tags are listed alphabetically or in a user-preferred order. The importance of a tag is indicated by font size or color. Figure 2.21 shows a tag cloud for visualizing the popular tags used in aWeb site. Tag clouds are often used in two ways. First, in a tag cloud for a single item, we can use the size of a tag to represent the number of times that the tag is applied to this item by different users.

Second, when visualizing the tag statistics on multiple items, we can use the size of a tag to represent the number of items that the tag has been applied to, that is, the popularity of the tag. In addition to complex data, complex relations among data entries also raise challenges for visualization. For example, Figure 2.22 uses a disease influence graph to visualize the correlations between diseases. The nodes in the graph are diseases, and the size of each node is proportional to the prevalence of the corresponding disease. Two nodes are linked by an edge if the corresponding diseases have a strong correlation.

The width of an edge is proportional to the strength of the correlation pattern of the two corresponding diseases.

In summary, visualization provides effective tools to explore data. We have introduced several popular methods and the essential ideas behind them. There are many existing tools and methods. Moreover, visualization can be used in data mining in various aspects. In addition to visualizing data, visualization can be used to represent the data mining process, the patterns obtained from a mining method, and user interaction with the data. Visual data mining is an important research and development direction.

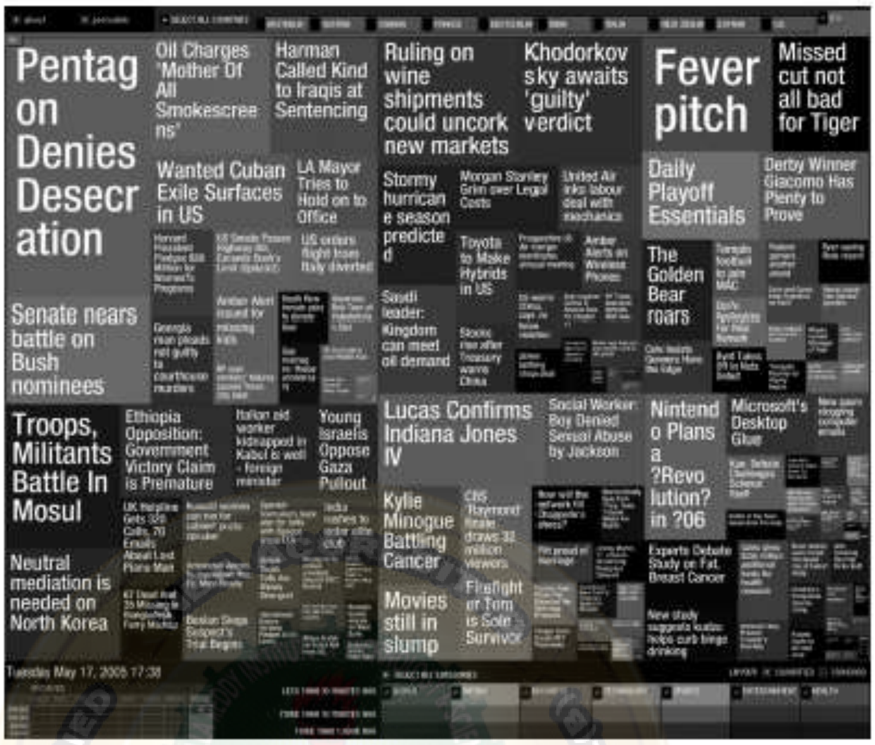

Pentagon Denies Desecration
Oil Charges 'Mother Of All Smokescreens'
Harman Called Kind to Iraqis at Sentencing
Ruling on wine shipments could uncork new markets
Khodorkov sky awaits 'guilty' verdict
Fever pitch
Missed cut not all bad for Tiger
Wanted Cuban Exile Surfaces in US
LA Mayor Tries to Hold on to Office
Stormy hurricane season predicted
Daily Playoff Essentials
Derby Winner Giacomo Has Plenty to Prove
Senate nears battle on Bush nominees
Saudi leader: Kingdom can meet oil demand
The Golden Bear roars
Troops, Militants Battle In Mosul
Ethiopia Opposition: Government Victory Claim is Premature
Italian aid worker kidnapped in Kabul is well - foreign minister
Young Israelis Oppose Gaza Pullout
Lucas Confirms Indiana Jones IV
Social Worker: Boy Denied Sexual Abuse by Jackson
Nintendo Plans a ?Revolution? in ?06
Microsoft's Desktop Glut
Kylie Minogue Battling Cancer
Neutral mediation is needed on North Korea
Movies still in slump
Firefight or Tom is Sole Survivor
Experts Debate Study on Fat, Breast Cancer
Tuesday May 17, 2005 17:38

Figure 2.20 Newsmap: Use of tree-maps to visualize Google news headline stories. *Source: www.cs.umd.edu/class/spring2005/cmsc838s/viz4all/ss/newsmap.png.*

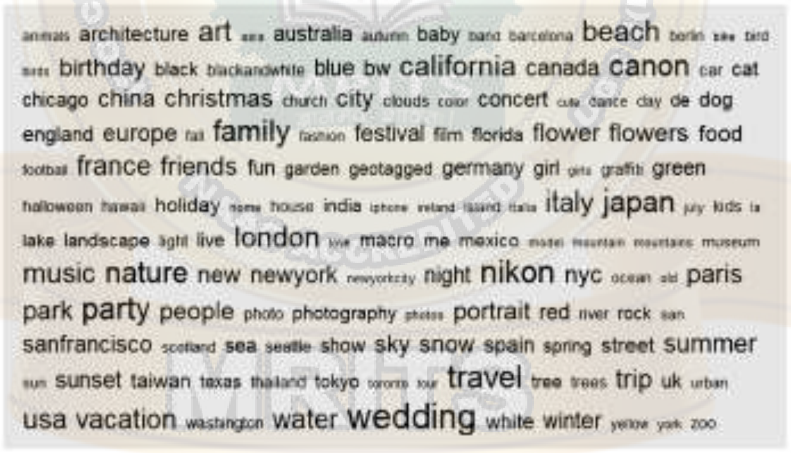

animals architecture art asia australia autumn baby band barcelona beach berlin bike bird birds birthday black blackandwhite blue bw california canada canon car cat chicago china christmas church city clouds color concert cute dance day de dog england europe fall family fashion festival film florida flower flowers food football france friends fun garden geotagged germany girl girls graffiti green halloween hawaii holiday home house india iphone ireland island italia italy japan july kids la lake landscape light live london love macro me mexico model mountain mountains museum music nature new newyork newyorkcity night nikon nyc ocean old paris park party people photo photography photos portrait red river rock san sanfrancisco scotland sea seattle show sky snow spain spring street summer sun sunset taiwan texas thailand tokyo toronto tour travel tree trees trip uk urban usa vacation washington water wedding white winter yellow york zoo

Figure 2.21 Using a tag cloud to visualize popular Web site tags. *Source:* A snapshot of *www.flickr.com/photos/tags/*, January 23, 2010.