**UNIT III:**

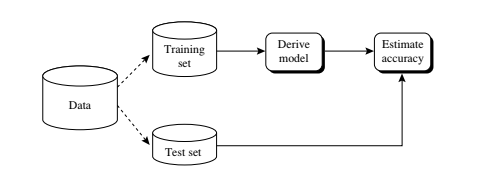
**Machine Learning: Understanding why data scientists use machine learning-What is machine learning and why we should care about, Applications of machine learning in data science, Where it is used in data science, The modeling process, Types of Machine Learning-Supervised and Unsupervised.**

**Q) Explain about validation strategies**

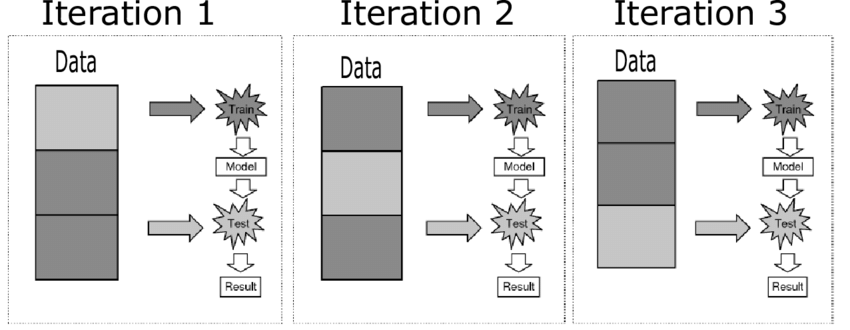
Validation strategies are ways to check if a model you've built in data science actually works well and gives good results on new data it hasn't seen before.

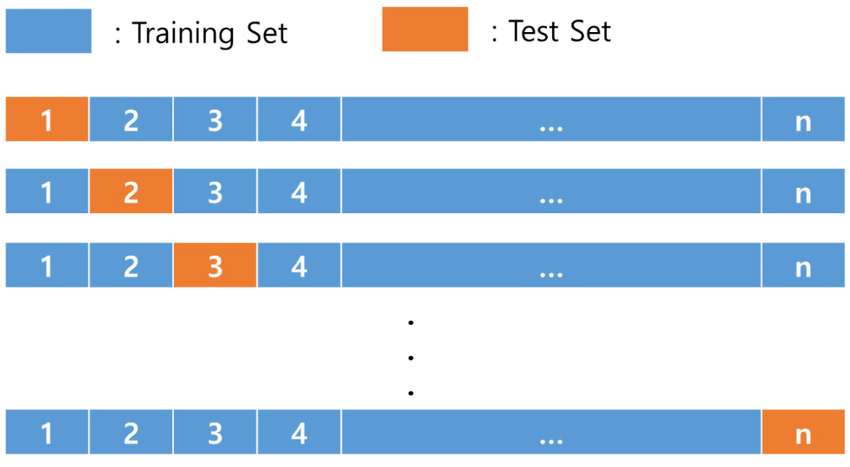
Here are some simple validation strategies

1. Holdout Method: In this method, the given data are randomly partitioned into two independent sets, a training set and a test set. Typically, two-thirds of the data are allocated to the training set, and the remaining one-third is allocated to the test set. The training set is used to derive the model. The model’s accuracy is then estimated with the test set as shown below



1. K- fold cross validation: You split the data into several folds or groups (let’s say 5 folds if k=5). You train the model using all groups except one and test it with the group you left out. You repeat this process until every group has been tested once. This way, you make full use of all the data and check the model multiple times. 3-fold cross validation is as shown in below figure



1. Leave one out: is a special case of k-fold cross-validation where k =1. You train the model using all the data except for one row, then test it on that single row. This process is repeated for every row as shown in below figure.

**Q) what is root cause analysis**

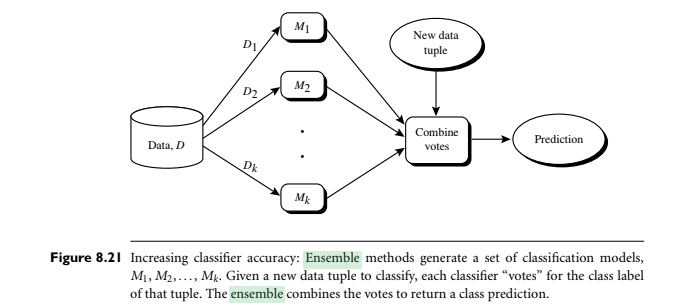
Root cause analysis in data science is used to understand the main reasons why something happened in the past. It concentrates on interpretation and understanding

Here are a few examples:

1. Understanding and improving a business process, like figuring out which products are most valuable to your company.
2. Discovering what causes diabetes
3. Determining the causes of traffic jams
4. find out the causes of species’ extinctions

**Q) What is ensemble learning**

Ensemble learning is a technique used in the model building process where you train multiple models seperately and then combine their results ss shown in below figure



**Q) What is availability bias?**

Availability bias occurs when you only consider the data that is easily accessible to you. Availability bias in model building happens when the features used in a model are mostly the ones that are easy to get. This means your model might only show a one-sided view of the truth because it's made with data that is easy to find, instead of data that fully explains the situation.

Because of this, models with availability bias often perform badly during testing. This is because testing shows that the model, made with incomplete or biased data, does not work well with new, unseen data. The model's narrow view, shaped by the easy-to-get data, means it's not a true representation of reality.

**Q) What is Regularization?**

Regularization is a technique in machine learning used to improve a model's performance by preventing overfitting**.**

Overfitting occurs when a model is overly complicated and focuses excessively on the training data. As a result, it struggles to perform well on new and unseen data.

Some types of regularization are:

**L1 Regularization (Lasso)** : Removes less important features by making their values zero, creating a simpler model. This helps in choosing only the useful features.

**L2 Regularization (Ridge)** : Reduces the impact of each feature without removing them, balancing their importance for better understanding.

**Q) Explain observable variables and latent variables   
observable variables are variables** that can be directly measured and seen. For instance, if you have data about wine, **observable variables** might be things like 'citric acid', 'residual sugar', 'density', 'pH', 'alcohol' content etc.. You can take measurements for each of these directly.

Latent variables cannot be directly measured but you think are influencing what you *can* see. They are hidden or underlying factors that affect the observable variables but aren't directly present in your data. You can only infer their existence and value. Principal Component Analysis (PCA) can be used to find a smaller number of latent variables that capture most of the information in the original data. These latent variables might represent underlying characteristics like 'persistent acidity' or 'volatile acidity' which are influenced by combinations of the original measurements.

Finding these **latent variables** can be useful because:

* They can represent several existing **observable variables**, simplifying the data.
* By reducing the number of variables, the data becomes easier to manage, algorithms run faster, and predictions might become more accurate.
* Because latent variables are designed or targeted toward the defined research goal, you lose little key information by using them. If we can reduce a data set from 14 observable variables per line to 5 or 6 latent variables, for example, we have a better chance of reaching our research goal because of the data set’s simplified structure.

**Q) What is machine learning and why should we care about it?**

Machine learning is about **teaching computers to learn from data**. “Machine learning is a field of study that gives computers the ability to learn without being explicitly programmed.”

Think of it like showing a child many examples of something so they can learn to recognise it themselves. Instead of giving the computer strict rules, we give it lots of information (data), and it figures out the patterns and connections.

Why should we care about this? Well, there are a few really good reasons:

* **It can tackle problems that are too tricky for normal computer programs.** For example, trying to write a program that can recognise your friends in any photo, in any light, from any angle, would be incredibly difficult. Machine learning lets the computer learn to do this by looking at lots of pictures.
* **Regression and classification are of primary importance to a data scientist. To achieve these goals, one of the main tools a data scientist uses is machine learning. The uses for regression and automatic classification are wide ranging, such as the following:**

■Finding oil fields, gold mines, etc..   
■ Finding place names or persons in text (classification)

■ Identifying people based on pictures or voice recordings (classification)

■ Recognizing birds based on their whistle (classification)

■ Identifying profitable customers (regression and classification)

■ Proactively identifying car parts that are likely to fail (regression)

■ Identifying tumors and diseases (classification)

■ Predicting the amount of money a person will spend on product X (regression)

■ Predicting the number of eruptions of a volcano in a period (regression)

■ Predicting your company’s yearly revenue (regression)

■ Predicting which team will win the Champions League in soccer (classification)

* **It powers many of the things we use every day.** Things like the suggestions you get on streaming services, the way your phone finishes your sentences, and the filters that keep unwanted emails out of your inbox often use machine learning behind the scenes.
* **It can find hidden insights in big piles of data.** Sometimes there are patterns in large amounts of information that humans can't easily spot. Machine learning can help us to find these and learn new things.
* **It can automate tasks and make things more efficient.** Once a computer has learned from data, it can often perform tasks automatically, and it can even get better at them over time as it gets more information.

Basically, machine learning gives computers the ability to learn and solve problems in a way that traditional programming can't always manage. It's becoming more and more important in lots of different areas of our lives.

**Q) Application of M.L in Data Science and Where it is used in data science**

Machine learning is very useful in data science. It helps computers to learn from data without being directly told what to do. This learning allows computers to solve many problems.

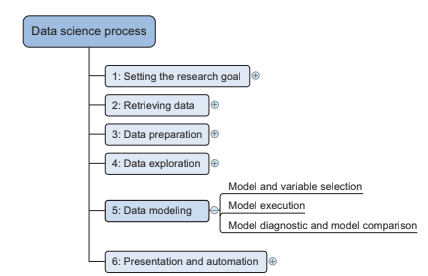
Here are some **applications of machine learning in data science**:

* **Making guesses or predictions (Regression)**: Machine learning can help guess a number. For example, it can predict how much money a person will spend on a product.
* **Putting things into groups or categories (Classification)**: Machine learning can help decide which group something belongs to. For example, it can identify if a picture is of a person or not.
* **Finding new places or things**: By learning from places where things were found before, machine learning can help find new places. For example, it can help find new oil fields.
* **Identifying people or objects**: Machine learning can learn what a person or object looks or sounds like and then identify them in new pictures or voice recordings.
* **Understanding customers**: Machine learning can help businesses understand which customers are likely to buy their products.
* **Predicting problems**: It can help predict when something might fail. For example, it can predict when a car part is likely to break down.
* **Helping with health**: Machine learning can be used to find tumours or identify diseases.
* **Understanding why things happen (Root Cause Analysis)**: Sometimes, the goal is not to predict but to understand. Machine learning can help find out the reasons behind something.

■ Understanding and optimizing a business process, such as determining which products add value to a product line

■ Discovering what causes diabetes

■ Determining the causes of traffic jams



**The above figure** shows where machine learning fits in the **data science process**. While machine learning is mostly used in the **data modeling** step, it can also be helpful in other steps like **data preparation** and **data exploration**.

In simple terms, machine learning gives data scientists powerful tools to analyse data, make predictions, and understand complex problems in many different areas.

Q) **The Modeling Process**

The process of building a machine learning model involves several steps. These steps help to create a model that can learn from data and make predictions or find patterns. Chapter 3 of the sources describes the modeling process in more detail.

Here are the main steps in the modeling process:

1. **Feature Engineering and Model Selection**: First, you need to decide what pieces of information (called **features** or **predictors**) will be used to build your model. You might also need to create new features from the data you have (**feature engineering**). Then, you choose the type of machine learning model that you think will work best for your problem.
2. **Training the Model**: Once you have your features and have chosen a model, you need to **train** the model. This means you feed the model with data so that it can learn the relationships between the features and what you want to predict or find.
3. **Model Validation and Selection**: After training the model, you need to check if it works well on new data it hasn't seen before (**validation**). You might try different models and compare how well they perform to choose the best one (**model selection**).
4. **Applying the Trained Model to Unseen Data**: If your model works well, you can use it to make predictions or find patterns in new data. This is called **model scoring**.

You will likely go back and forth between the first three steps until you are happy with your model. The last step is not always needed if your goal is only to understand the data rather than make predictions.

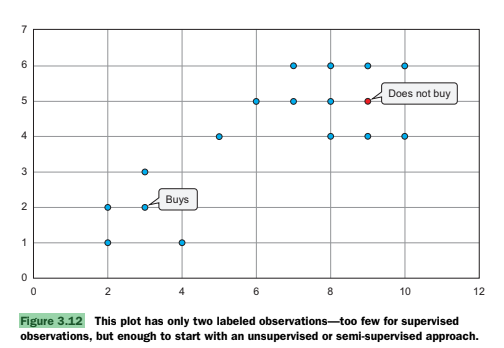
**Q) Types of Machine Learning**

Machine learning methods can be broadly divided into different types based on how they learn from data. Four important types are **supervised learning**, **unsupervised learning, semi supervised learning** and reinforcement learning.

**Supervised Learning**: The model is trained on labeled data to predict outcomes for new data. Algorithms include linear regression, neural networks, and decision trees. Applications: Fraud detection, price prediction.

**Unsupervised Learning**: The model identifies patterns in unlabeled data. Algorithms include k-means clustering, PCA, and Gaussian mixture models. Applications: Customer segmentation, dimensionality reduction.

**Semi-supervised Learning**: Combines a small amount of labeled data with a large amount of unlabeled data to improve learning efficiency. Techniques include label propagation. Applications: Speech recognition, bioinformatics.



**Reinforcement Learning**: The model learns by interacting with its environment, receiving rewards for correct actions and penalties for wrong ones. Algorithms include Q-learning and deep adversarial networks. Applications: Game playing, robotics.

**Q) Supervised Learning**:   
In **supervised learning**, the data you use to train the model has **labels**. A label is like an answer key that tells the model what the output should be for a given input. The model learns by trying to find patterns in the labelled data so that it can make correct predictions for new, unlabelled data.

Main two important supervised learning techniques are: 1. Classification 2. Regression

Classification is used to predict discrete values or put data into specific categories. For instance, it can classify if an email is spam or not.

**Example**: If you want to train a model to identify pictures of cats and dogs (**classification**), you would show it many pictures of cats and dogs, with each picture labelled as either "cat" or "dog". The model learns from these labels. When you show it a new picture, it can then predict whether it's a cat or a dog.

Regression is used to predict continuous numerical values, such as predicting house prices or temperature.

Another example is **regression**, where the label is a number. For instance, predicting the price of a house based on its size and location, where the actual selling price of other houses is the label.

Example: Naive Bayes is a machine learning algorithm used for classification problems. It is based on Bayes' theorem.

In a simple example, consider dataset below

S.No. Weather condition Road condition Accident

1 Rain bad yes

2 snow average yes

3 clear bad no

4 clear good no

5 snow good no

6 rain average yes

7 rain good no

8 snow bad yes

9 clear good no

10 clear bad yes

Here are the four steps to use Naive Bayes for classification:

Step 1: calculate probability of class attribute ( here it is accident column values yes, no…P( yes) = no. Of times yes appears in rows/ total number of rows)

P(yes) = 5/10

P(no) = 5/10

Step 2: Calculate class conditional probabilities:

We need to find whether accident is yes or no for x = { rain, bad} ( here x is new data to be classified)

P( x /yes) = P(rain/yes) \* P(bad/yes) = 1/5\* 3/5 = 3/25

P(x/no) = P(rain/no) \* P(bad/no) = 2/5\*1/5 = 2/25

we calculate the P(rain/yes) as follows :It is the number of rows in which rain and yes occurs / no. of rows yes occurs.

Step 3: Calculate likelihood:

Likelihood of yes = ly = P(x/yes)\*P(yes) = 3/25\*5/10 = 3/50

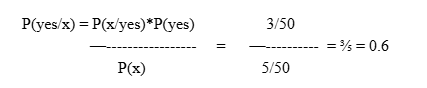
Likelihood of no = ln = P(x/no)\*P(no) = 2/25\*5/10 = 2/50

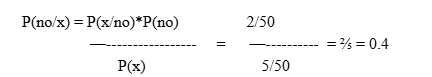
Total likelihood = P(x) = ly+ln = 3/50 + 2/50 = 5/50

Step 4: Calculate actual probability using Bayes theorem:

P(yes/x) = P(x/yes)\*P(yes) / P(x) = 3/50 / 5/50 = 3/5 = 0.6

Step 4: Calculate actual probability using Bayes' theorem:





Yes probability is more...accident is yes for x = (Rain, Bad)

Finally, we use Bayes' theorem to calculate the probability of each class given the features of the new email. The formula is as follows:

Bayes' theorem is a statistical theorem that states the following relationship between the probabilities of events A and B:

P(A|B) = P(B|A) \* P(A) / P(B)

where P(A|B) is the conditional probability of event A given that event B has occurred, P(B|A) is the conditional probability of event B given that event A has occurred, P(A) is the prior probability of event A, and P(B) is the prior probability of event B.

R program:

# Load necessary library

#install.packages("e1071") # Uncomment this line if you haven't installed the package

library(e1071)

# Create the dataset

data <- data.frame(

Weather = c("Rain", "Snow", "Clear", "Clear", "Snow", "Rain", "Rain", "Snow", "Clear", "Clear"),

RoadCondition = c("Bad", "Average", "Bad", "Good", "Good", "Average", "Good", "Bad", "Good", "Bad"),

Accident = c("Yes", "Yes", "No", "No", "No", "Yes", "No", "Yes", "No", "Yes")

)

# Convert categorical variables to factors

data$Weather <- as.factor(data$Weather)

data$RoadCondition <- as.factor(data$RoadCondition)

data$Accident <- as.factor(data$Accident)

# Train the Naive Bayes model

model <- naiveBayes(Accident ~ Weather + RoadCondition, data = data)

# Print the model

print(model)

# New data to classify

new\_data <- data.frame(

Weather = factor("Rain", levels = levels(data$Weather)),

RoadCondition = factor("Bad", levels = levels(data$RoadCondition))

)

# Predict the class for the new data

prediction <- predict(model, new\_data)

# Print the prediction

cat("The predicted class for the new data (Rain, Bad) is:", prediction, "\n")

Output:

Naive Bayes Classifier for Discrete Predictors

Call:

naiveBayes.default(x = X, y = Y, laplace = laplace)

A-priori probabilities:

Y

No Yes

0.5 0.5

Conditional probabilities:

Weather

Y Clear Rain Snow

No 0.6 0.2 0.2

Yes 0.2 0.4 0.4

RoadCondition

Y Average Bad Good

No 0.0 0.2 0.8

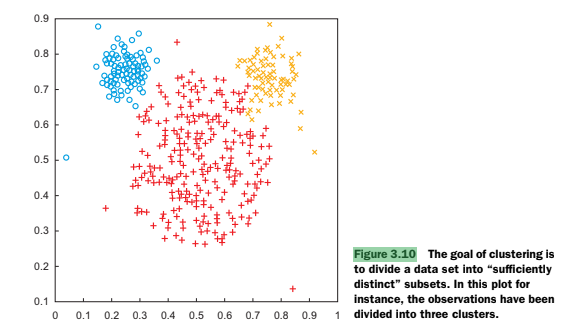
Yes 0.4 0.6 0.0

The predicted class for the new data (Rain, Bad) is: 2 (YES)

**Q) Unsupervised Learning**: In **unsupervised learning**, the data you use to train the model **does not have labels**. The model tries to find hidden patterns or structures in the data by itself, without any guidance from labels.

Example of unsupervised technique is called clustering. Clustering is a machine learning technique used to identify natural groupings within a dataset. The goal of clustering is to divide the data into subsets, known as clusters, such that observations within the same cluster are similar to each other, while observations in different clusters are dissimilar. This process helps in uncovering underlying patterns and structures in the data without prior knowledge of group labels, making it a key technique in unsupervised learning. **Example**: If you have a lot of data about customers' shopping habits but no labels telling you what types of customers there are, an unsupervised learning technique like clustering could try to group the customers into different segments based on similarities in their purchases The algorithm discovers these groups on its own.

Another example is **Principal Component Analysis (PCA)**, which can find important underlying patterns or reduce the number of variables in your data without using labels.



**Figure 3.12** shows an example where there is very little labelled data, which might lead to using unsupervised or **semi-supervised** learning (a mix of both). **Figure 3.10** gives a visual idea of **clustering** in unsupervised learning.

Example : k -means clustering

write k-means program in r to find 2 cluster members for the following data {2,3,4,10,11,12,20,25,30}. Initially the cluster means m1, m2 are m1=4 and m2=12

Program:

# Define the data

data <- c(2, 3, 4, 10, 11, 12, 20, 25, 30)

# Initial cluster means

m1 <- 4

m2 <- 12

# Function to perform K-Means clustering

kmeans\_custom <- function(data, m1, m2, max\_iter = 10) {

# Initialize cluster assignments

clusters <- numeric(length(data))

for (iter in 1:max\_iter) {

# Assign points to the nearest cluster

for (i in 1:length(data)) {

if (abs(data[i] - m1) < abs(data[i] - m2)) {

clusters[i] <- 1 # Assign to cluster 1

} else {

clusters[i] <- 2 # Assign to cluster 2

}

}

# Calculate new means

new\_m1 <- mean(data[clusters == 1])

new\_m2 <- mean(data[clusters == 2])

# Check for convergence (if means do not change)

if (new\_m1 == m1 && new\_m2 == m2) {

break

}

# Update means

m1 <- new\_m1

m2 <- new\_m2

}

return(list(clusters = clusters, means = c(m1, m2)))

}

# Run the K-Means clustering

result <- kmeans\_custom(data, m1, m2)

# Print the results

print("Cluster Assignments:")

print(result$clusters)

print("Final Cluster Means:")

print(result$means)

**Output:  
[1] "Cluster Assignments:"**

**[1] 1 1 1 1 1 1 2 2 2**

**[1] "Final Cluster Means:"**

**[1] 7 25**

**UNIT IV:**

**Handling large Data on a Single Computer: The problems we face when handling large data,**

**General Techniques for handling large volumes of data, Generating programming tips for dealing with large datasets.**

**Q) common problems you might encounter when working with large amounts of data:**

* **Not enough memory:** Large data requires a lot of memory to process. If there’s not enough RAM, programs either crash or become extremely slow. Many programs try to load the entire dataset into memory at once, leading to errors like "out of memory". Other algorithms need to hold multiple copies of the data in memory or store intermediate results. All of these increase the problem.  
  Example in R: If you have a file that is gigabytes in size, trying to read it all into a single R data frame might crash your R session if your computer doesn't have enough RAM.  
  Solution: Use techniques like streaming data in chunks, compression, or increasing RAM.
* **Processes That Never End**

Sometimes, a data processing task keeps running without finishing. This happens when the system doesn't have enough memory or when the method used to process the data isn't efficient.

Example: If a program is trying to sort billions of records without proper optimization, it may take hours or even days—or it might never complete.

Solution: Break data into smaller parts, use efficient algorithms, and apply parallel processing.

* **Not Enough Speed**

Large data often slows systems down because reading, writing, and analyzing it takes too much time.

Example: A business analyzing customer purchases might face delays because their database is too slow.

Solution: Use faster storage (like SSDs), optimize queries, and cache frequently used data.

* **Bottlenecks:** A bottleneck happens when one part of the system slows everything down. It could be slow processing, poor network speed, or inefficient data handling.For example, if your program is constantly reading data from a slow hard drive, the fast processor (CPU) might spend a lot of time waiting for the data. This makes the whole process slower than it needs to be.

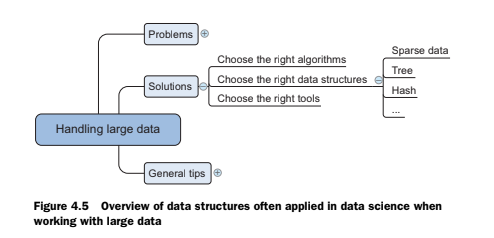
Example: If a computer has a powerful processor but a slow hard drive, reading data takes too long, and processing gets delayed.

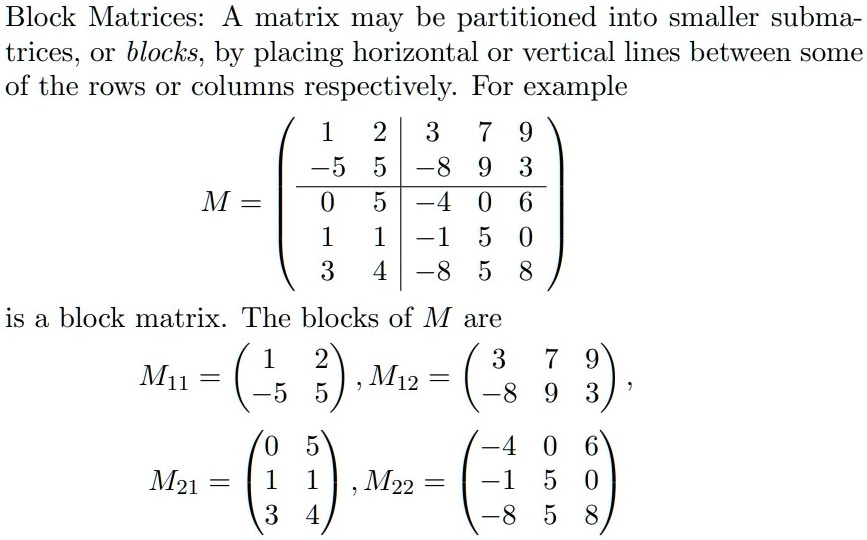
Solution: Identify where the slowdown occurs—optimize code, use better hardware, and distribute tasks across multiple systems.

Example 2:Imagine a factory assembly line where one worker is very slow. Even if all the other workers are fast, the entire production line will be held up by the slow worker. Similarly, if data input is slow, the processing power of the CPU can't be fully used.

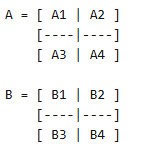
(figure 4.1) that illustrates these problems.

**Q) General Techniques for handling large volumes of data**When dealing with large amounts of data, there are generally three main ways to handle it effectively:

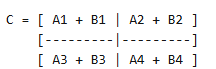


* **Choosing the Right Algorithms:** The program or set of instructions you use to process data can make a huge difference. Some algorithms are designed to work well even with very large datasets without needing to load everything into the computer's fast memory (RAM) all at once. Ideally, a good algorithm for large data can also do calculations in parallel, using multiple parts of the computer at the same time to speed things up.  
  + **Online Learning Algorithms:** Imagine learning about different types of animals by seeing one animal at a time. You learn its features and then can "forget" that specific animal because you've updated your understanding. Online learning algorithms work similarly. They process data one piece at a time (or in small chunks), update their understanding (the model), and then move on without needing to keep the entire massive dataset in memory. This is great for memory issues.  
     Consider an example of a 'perceptron' algorithm that learns one observation at a time. The idea here is that the algorithm adjusts its internal "weights" based on each new piece of data it sees.
  + **Block Algorithms:** Think of a giant puzzle. Instead of trying to solve the whole thing at once, you break it down into smaller sections (blocks). Block algorithms do this with large datasets, especially in calculations involving tables of numbers (matrices). They process these smaller blocks, which can fit into memory, and then combine the results.  
    

Imagine you have two large matrices, A and B. Instead of loading the entire A and the entire B into memory to add them, you divide each matrix into four smaller blocks:A1,A2,A3,A4,B1,B2,B3,B4



To calculate the sum C = A + B, you would add the corresponding blocks together:



By performing the addition block by block, you only need to have the smaller blocks (like A1 and B1) in memory at the same time.

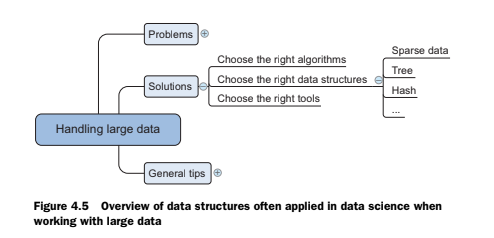
* **MapReduce:**MapReduce is a way to break down big tasks into smaller parts (mapping) and then combine the results (reducing). Imagine counting votes:

Mapping: Each local office counts the votes for each political party.  
Reducing: All the local results are combined to get the total votes for each party.

This approach is helpful because it works on smaller pieces of data independently, making it faster and easy to distribute across multiple computers. It also avoids overwhelming the system with too much data at once.

* **Choosing the Right Data Structures:** The way you organise your data can also significantly impact how efficiently you can work with large volumes. Different data structures have different strengths in terms of how much storage they need and how quickly you can do things like find, add, change, or remove data.  
  + **Sparse Data:**
  + **Tree Structures:**
  + **Hash Tables:**
* **Selecting the Right Tools:** There are specific software libraries and tools that are designed to help you work with large datasets efficiently. These tools often have built-in optimisations for speed and memory usage.  
  + R libraries that are helpful for large data, for speeding up , for faster numerical calculations, for just-in-time compilation, for out-of-memory arrays, for working with larger-than-memory datasets, for optimising and parallelising calculations.
  + use databases effectively, as they are often designed to handle large amounts of data and have built-in mechanisms for optimising queries.

**Q) How choosing a right data structure improve efficiency when handling large datasets**

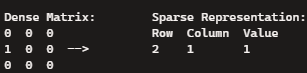
Choosing the right **data structure**, which is the way you organise your data, is very important for working efficiently with large amounts of information. It can significantly affect how quickly your computer can find, add, change, and remove data, as well as how much memory it uses. Think of it as organising a large library – a good system makes finding books much faster than just having them piled randomly on the floor.

Here are some examples of how different data structures can improve efficiency with large datasets, drawing on concepts from the sources:

* **Handling Data with Many Zeros (Sparse Data)**: Sparse data contains mostly zeros and very few non-zero values. For example, consider the data below:



A **sparse data structure** is a more efficient way to store this kind of data. Instead of storing every single cell, it only records the cells that have a value (and their location).

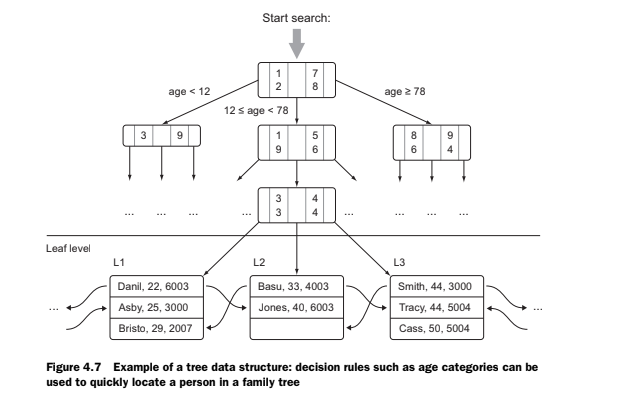


This can save a lot of memory and speed up calculations because the computer doesn't have to process all those zeros.

* **Quickly Finding Information (Tree Structures and Hash Tables)**: When you have a lot of data, finding a specific piece of information by looking through everything one by one can be very slow. **Tree structures** and **hash tables** are data structures designed for faster retrieval.

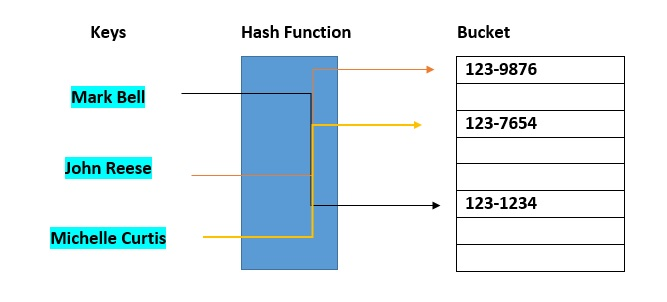
1. **Tree structure** Trees are hierarchical data structures useful for quick data retrieval. Starting from a root, you can quickly navigate to specific individuals by following branches based on certain characteristics (like age). Tree data structures allow for fast information retrieval by using decision rules to quickly locate the data you need instead of scanning through everything.

Tree is like a well-organised index where you can quickly narrow down where to look. For example, if you're searching for a name in a phone book, you don't start at the first page; you use the alphabetical order to jump to roughly the right section.



1. **hash table** uses a special function to assign a unique 'key' to each piece of data, which tells the computer exactly where to find it, much like having a specific address for each item. This makes looking up individual items very fast, even in large datasets. In R, you can think of lists with named elements as having some similarities to hash tables, where the name acts as a key.

Imagine a library where each book has a unique code that tells you exactly where to find it. Hash tables work similarly by calculating a unique 'key' for each piece of data and storing it in a 'bucket'. This allows you to quickly retrieve information by going directly to the right bucket instead of searching. Dictionaries in programming languages are often implemented as hash tables.



* **Organising Tabular Data Efficiently (Data Frames)**: A data frame organises data into rows (observations) and columns (variables). While base R data frames are generally stored in memory, packages like dplyr', provide highly optimised ways to manipulate these data frames efficiently, which is crucial when dealing with larger datasets that do fit in memory. Using the right functions from dplyr can speed up operations like filtering and sorting, making data management more efficient than using base R alone.

**Q) How does a high learning rate in machine learning contribute to overshooting?**

When the learning rate is too high, the weight updates can overshoot the optimal value.

### **For Example:**

* **Target weight**: 0.75 (the ideal value we are trying to reach).
* **Current weight**: 0.4 (our model’s current estimate).
* **Learning rate**: 0.5 (controls how much we adjust the weight in each step).
* **Error size**: 1 (difference between predicted value and actual value).

### **Why the Overshoot Happened**

The **adjustment calculation**:  
Adjustment = Learning Rate × Error × Input  
 = 0.5 × 1 × 1  
 = 0.5  
We add this **adjustment (0.5)** to the **current weight (0.4)**:  
New Weight = 0.4 + 0.5 = 0.9  
The new weight (0.9) **overshot** the target (0.75) instead of approaching it smoothly.

### **What Went Wrong?**

* The **learning rate** (0.5) is **too high**, causing a large adjustment.
* Instead of gradually moving towards **0.75**, we **jumped past it** to **0.9**.
* If this continues, the updates will fluctuate and make it harder to converge on the correct value.

### **How to Fix It**

* **Reduce the learning rate** (e.g., 0.1 instead of 0.5) to allow smaller, controlled updates.
* **Use adaptive learning rates** (e.g., **Adam optimizer**) to adjust dynamically.

**Q) Differentiate between online and Streaming Algorithms**

| **Feature** | **Streaming Algorithms** | **Online Learning Algorithms** |
| --- | --- | --- |
| **Type of data** | Works with real-time data (like live tweets or stock prices) | Works with real-time or stored data |
| **How data is handled** | Looks at each piece of data only once | Can look at the same data more than once |
| **Memory usage** | Uses very little memory, can't store all data | Uses more memory if needed, can store and revisit some data |
| **Main goal** | Quickly summarize or estimate things from huge data | Slowly improve a model or prediction over time |
| **Processing style** | Processes data instantly and moves on | Processes data in small steps or batches to keep learning |

**Q) Differentiate between full batch learning, mini-batch learning, and online learning**

1. Full Batch Learning: The model looks at all the data at once before updating itself.

Pros: Produces very accurate updates and helps find the best solution.

Cons: Requires a lot of memory and time, making it slow for huge datasets.

2. Mini-Batch Learning: The model looks at small groups (batches) of data before updating itself.

Pros: Balances speed and accuracy, making it suitable for large datasets.

Cons: Still requires computing power, and updates may not be as precise as full batch learning.

3. Online Learning: The model looks at one data point at a time before updating itself.

Pros: Works well for streaming data (like live predictions) and adjusts in real-time.

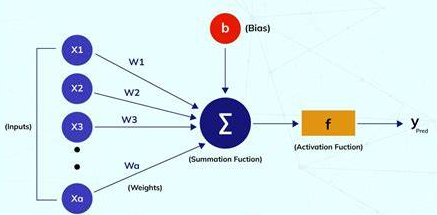
Cons: Can be unstable due to frequent updates, sometimes leading to less reliable results.

**Q) Explain perceptron in detail? How does it handle large data?**

Online learning algorithms process data one piece at a time (or in small chunks), update their understanding (the model), and then move on

Consider an example of a 'perceptron' algorithm that learns one observation at a time. The idea here is that the algorithm adjusts its internal "weights" based on each new piece of data it sees.

A Perceptron is one of the simplest types of artificial neural networks used for binary classification. The architecture diagram of perceptron is as shown below

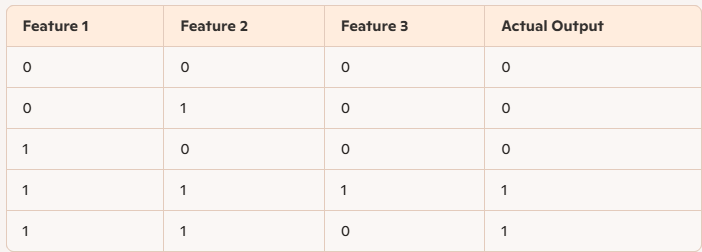


A perceptron consists of:

1️⃣ Inputs (Features) → Data points that influence the decision (e.g., age, income etc..).

2️⃣ Weights → These adjust the importance of each input.   
3️⃣ Summation & Activation → A calculation decides whether to classify an input as 0 or 1. 4️⃣ Output → The final decision (0 = negative, 1 = positive).

### **Problem Statement**

We are using a **Perceptron model** to predict whether a customer will make a purchase (1) or not (0) based on three features. The given dataset is:  


The perceptron will **learn weights** for these features and adjust them through training.

### **Step 1: Initialize Weights and Bias**

* Start with all **weights** as 0 → (W1 = 0, W2 = 0, W3 = 0).
* Bias term is also 0.

### **Step 2: Process Each Observation and Update Weights**

Using the perceptron rule:

Weighted Sum=W1×X1+W2×X2+W3×X3+Bias\text{Weighted Sum} = W1 \times X1 + W2 \times X2 + W3 \times X3 + \text{Bias}

If the weighted sum is **greater than or equal to 0**, predict 1, otherwise predict 0.

✔ If prediction is wrong, update the weights using:

New Weight=Old Weight+Learning Rate×Error×Feature Value\text{New Weight} = \text{Old Weight} + \text{Learning Rate} \times \text{Error} \times \text{Feature Value}

Let’s go observation by observation.

### **Iteration 1: Training on First Observation (0,0,0 → Output = 0)**

* **Weighted Sum** = 0 × 0 + 0 × 0 + 0 × 0 + 0 = 0
* **Prediction** = 1 (incorrect, should be 0).
* **Error** = 0 - 1 = -1
* **Update Weights**:
  + W1 = 0 + (0.1 × -1 × 0) = 0
  + W2 = 0 + (0.1 × -1 × 0) = 0
  + W3 = 0 + (0.1 × -1 × 0) = 0
  + **Bias** = 0 + (0.1 × -1) = -0.1

✔ **Updated Weights:** W1 = 0, W2 = 0, W3 = 0, Bias = -0.1

### **Iteration 2: Training on Second Observation (0,1,0 → Output = 0)**

* **Weighted Sum** = 0 × 0 + 0 × 1 + 0 × 0 - 0.1 = -0.1
* **Prediction** = 0 (correct, no update).

✔ **Weights remain the same.**

### **Iteration 3: Training on Third Observation (1,0,0 → Output = 0)**

* **Weighted Sum** = 0 × 1 + 0 × 0 + 0 × 0 - 0.1 = -0.1
* **Prediction** = 0 (correct, no update).

✔ **Weights remain the same.**

### **Iteration 4: Training on Fourth Observation (1,1,1 → Output = 1)**

* **Weighted Sum** = 0 × 1 + 0 × 1 + 0 × 1 - 0.1 = -0.1
* **Prediction** = 0 (incorrect, should be 1).
* **Error** = 1 - 0 = 1
* **Update Weights**:
  + W1 = 0 + (0.1 × 1 × 1) = 0.1
  + W2 = 0 + (0.1 × 1 × 1) = 0.1
  + W3 = 0 + (0.1 × 1 × 1) = 0.1
  + **Bias** = -0.1 + (0.1 × 1) = 0

✔ **Updated Weights:** W1 = 0.1, W2 = 0.1, W3 = 0.1, Bias = 0

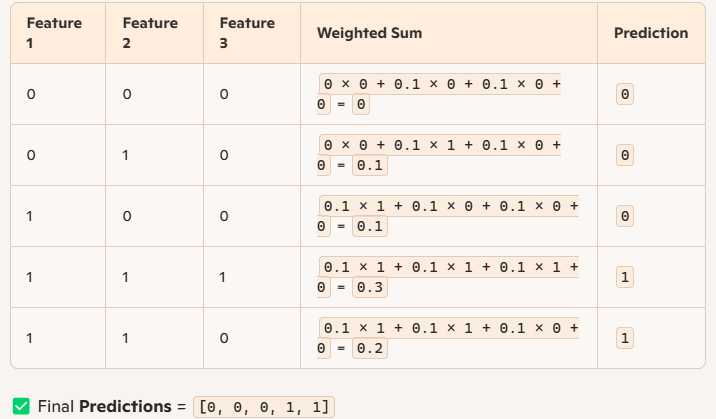
### **Iteration 5: Training on Fifth Observation (1,1,0 → Output = 1)**

* **Weighted Sum** = 0.1 × 1 + 0.1 × 1 + 0.1 × 0 + 0 = 0.2
* **Prediction** = 1 (correct, no update).

✔ **Weights remain the same.**

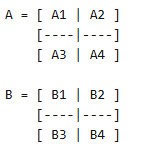
### **Step 3: Testing the Predictions**

Now that the perceptron is trained, let’s test it on the same data.

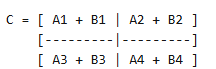


**Q) Handling large data using block matrices** Handling large data using **block matrices** involves dividing a large matrix into smaller submatrices, known as blocks, and then performing operations on these blocks instead of the entire matrix at once. This technique is particularly useful when the full matrix is too large to fit into your computer's memory (RAM). By processing data in smaller, manageable chunks, you can overcome memory limitations.

A simple example of adding two large matrices, A and B, using block matrices is as follows:  
Imagine you have two large matrices, A and B. Instead of loading the entire A and the entire B into memory to add them, you divide each matrix into four smaller blocks:



To calculate the sum C = A + B, you would add the corresponding blocks together:



By performing the addition block by block, you only need to have the smaller blocks (like A1 and B1) in memory at the same time.

Some R libraries for out-of-memory block matrix calculations are

1. bigmemory**:**
   * Handles large matrices/dataframes and stores them compactly.
   * Supports disk-based storage to handle out-of-memory data.
2. foreach**:**
   * Useful for iterating through data in chunks.
   * Enables processing tasks one block at a time.
3. doParallel**:**
   * Works with foreach to enable parallel computations, boosting efficiency.
4. ff**:**
   * Similar to bigmemory, helps manage large datasets by storing parts on disk.
5. data.table**:**
   * Efficiently handles and processes large tabular datasets in memory or disk.

Linear Regression with Large Matrices: In matrix-based regression, coefficients (weights) are calculated using formulas like:



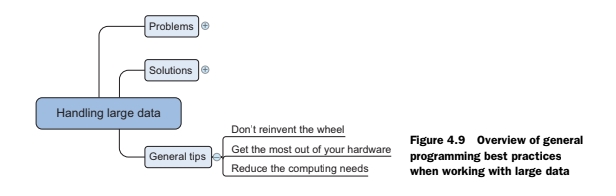
If the matrix X (predictors) is too big to fit into memory, it can be split into smaller blocks to:

1. Calculate intermediate results, such as XT X and XT y, block by block.
2. Combine the results from these smaller parts later to compute beta, without needing to handle the full matrix at once.

**Q) Generating programming tips for dealing with large datasets.**

Here are some general programming tips for handling large datasets:

* **Don't Reinvent the Wheel:** It's generally better to use tools and libraries that other people have already developed and optimised. These often incorporate best practices for handling large data and can save you a lot of time and effort.  
  + **Exploit the Power of Databases:** If your data is in a database, try to do as much of the initial processing and preparation there as possible. Databases are designed for managing and querying data efficiently. See if you can use database functions or procedures to filter, aggregate, or transform your data before bringing it into your program. The last example in the "Handling large data on a single computer" chapter (Case study 2) demonstrates integrating a database into a workflow.
  + **Use Optimised Libraries:** For tasks like machine learning or numerical calculations, use well-established libraries (like Scikit-learn in Python, as mentioned in the sources). These libraries are often highly optimised for performance with large datasets. Spend your time using these tools rather than trying to write everything from scratch.
* **Get the Most Out of Your Hardware:** Your computer likely has resources that aren't being fully used. By being mindful of how your program uses these resources, you can often speed things up.  
  + **Feed the CPU Compressed Data:** A simple trick to reduce the time your program spends waiting for data from the (relatively slow) hard drive is to work with compressed data. Your processor (CPU) is generally much faster at decompressing data than the hard drive is at reading large uncompressed files. This shifts more of the workload to the faster CPU.
  + **Make Use of the GPU:** If your computations can be done in parallel (meaning different parts of the calculation can be done at the same time), you might be able to leverage your graphics processing unit (GPU). GPUs have many more processing cores than CPUs and can be much faster for certain types of parallel tasks. The sources mention Python packages like Theano and NumbaPro that can help you use the GPU with minimal extra coding.
  + **Use Multiple Threads:** Even on a single CPU, you can often speed up processing by using multiple threads. This allows different parts of your program to run concurrently. Many programming languages, including Python, provide ways to use threads.
* **Reduce Your Computing Needs:** The best way to handle large data is often to minimise the amount of data you need to process in the first place and to make your calculations as efficient as possible.  
  + **Profile Your Code:** Identify the parts of your code that are running slowly. Use profiling tools to find these bottlenecks and focus your optimisation efforts there. Not all parts of your code need to be highly efficient.
  + **Use Compiled Code:** When possible, especially for parts of your code that involve loops or intensive calculations, use functions from optimised and compiled libraries. If you can't find an existing library, consider compiling parts of your code in a lower-level language like C or Fortran and then integrating it with your main program.
  + **Avoid Pulling All Data Into Memory:** If your dataset is too large to fit in your computer's RAM, avoid trying to load it all at once. Read data in smaller chunks or batches and process it piece by piece. The "Handling large data on a single computer" chapter (section 4.2.1) discusses online learning algorithms that process data one observation at a time.
  + **Use Generators:** Generators are a programming technique that allows you to process data one item at a time without storing intermediate results in memory. This can be very memory-efficient when working with large datasets.
  + **Use As Little Data As Possible:** If you don't need the entire dataset for your analysis or to build your model, consider working with a representative sample. This can significantly reduce processing time and memory usage, especially in the initial stages of exploration and model development.
  + **Simplify Calculations Mathematically:** Sometimes, you can rewrite equations or calculations in a more efficient way using your mathematical knowledge. Even seemingly small simplifications can make a big difference when applied to large amounts of data. The source provides a simple algebraic example: (a + b)^2 is often faster to compute than a^2 + 2ab + b^2.



**Q) Data chunking means** breaking down a very large dataset into smaller, more manageable pieces

It's important for processing large datasets because:

* It helps avoid memory problems . If the entire dataset is too big to fit in your computer's memory (RAM), processing it in chunks means only a small part needs to be in memory at any one time
* It can speed up processing . Some algorithms work better on smaller amounts of data, and chunking allows you to process data in steps . For example, online learning algorithms process data one piece at a time, and block algorithms work on smaller blocks of data.
* It's related to how some big data tools work. Frameworks like MapReduce also split data into smaller pieces for processing.

Think of it like eating a very large pizza. It's much easier to eat it slice by slice (chunks) rather than trying to eat the whole thing at once!

**Q) a bottleneck** when dealing with large data is when one part of your computer system can't keep up with the others, causing the whole process to slow down. It's like having a very fast car stuck in traffic – the car can go fast, but the traffic (the bottleneck) prevents it. For example, your fast processor (CPU) might be waiting for data from a slow hard drive, making the overall data processing take much longer

**UNIT V:**

**Sub setting R objects, Vectorised Operations, Managing Data Frames with the dplyr, Control structures, functions, Scoping rules of R, Coding Standards in R, Loop Functions, Debugging, Simulation. Case studies on preliminary data analysis.**

Here are some simple notes on subsetting R objects:

In R, you often need to select specific parts of your data. This is called subsetting. There are three main ways to do this using special operators: **[**, **[[**, and **$**.

**1. Subsetting Vectors (Single Dimension Data)**

Imagine a shopping list: shopping\_list <- c("apples", "bananas", "milk", "bread").

**Using [ and numbers:** You can pick items by their position.  
  
 shopping\_list # Gets the first item

This gives you: "apples"  
  
 shopping\_list[c(1, 3)] # Gets the first and third items

This gives you: "apples" "milk" You can also exclude items:  
  
 shopping\_list[-2] # Gets everything except the second item

* This gives you: "apples" "milk" "bread"

**Using [ and conditions:** You can select items based on whether they meet a condition (only works for certain types of vectors like logical or character vectors with comparisons).  
  
 numbers <- c(10, 5, 20, 15)

numbers[numbers > 10] # Gets numbers greater than 10

* This gives you: 20 15

**2. Subsetting Matrices (Two Dimensional Data - Rows and Columns)**

Think of a table of scores:

scores <- matrix(data = c(80, 90, 75, 85, 92, 78), nrow = 3, ncol = 2)

# [,1] [,2]

# 80 85

# 90 92

# 75 78

**Using [ and row and column numbers:** You select by [row\_number, column\_number].  
  
 scores # Gets the score in the first row and second column

This gives you: 85  
  
 scores # Gets all scores in the second row

This gives you: 90 92  
  
 scores[, 1] # Gets all scores in the first column

* This gives you: 80 90 75

**Using [ with names (if you have them):** You can use row and column names if you've added them to the matrix.  
  
 By default, if you select a single row or column, R might simplify the result to a vector. You can stop this by using drop = FALSE:  
  
 scores[1, , drop = FALSE] # Keeps the result as a row matrix

[,1] [,2]

80 85

**3. Subsetting Lists (Collections of Different Things)**

Lists can hold numbers, text, even other lists!

my\_list <- list(name = "Alice", age = 30, grades = c(A = 90, B = 85))

**Using [[ and a number:** This gets a single element by its position. The type of what you get back can be anything in the list.  
  
 my\_list[] # Gets the first element

* This gives you: "Alice"

**Using [[ and a name (in quotes):** This gets a single element by its name.  
  
 my\_list[["age"]] # Gets the element named "age"

* This gives you: 30

**Using $ and a name (no quotes):** This is a shortcut for getting an element by name.  
  
 my\_list$grades # Gets the element named "grades"

* This gives you: A B 90 85

**Using [ and numbers:** This gets a sub-list containing the selected element(s). The result is always a list.  
  
 my\_list[c(1, 3)] # Gets a list containing the first and third elements

This gives you a list:  
  
 $name

"Alice"

$grades

A B

90 85

**Subsetting Nested Lists:**

If a list contains another list, you can use [[ multiple times to get to the inner elements:

nested\_list <- list(outer = list(inner = "hello"))

nested\_list[][] # First element of outer, then first element of inner

This gives you: "hello" You can also combine names and numbers:

another\_nested <- list(info = list(age = 30))

another\_nested$info[] # Element named "info", then first element of that list

This gives you: 30

**Partial Matching:**

With [[ and $, you can sometimes use just the beginning of a name if it's clear which element you mean. **However, it's generally safer to use the full name in your code to avoid confusion**.

**Removing NA (Missing) Values:**

Sometimes your data has missing values represented by NA. You can use is.na() to find them and ! (not) to select everything else:

data\_with\_na <- c(1, 2, NA, 4, NA, 5)

data\_without\_na <- data\_with\_na[!is.na(data\_with\_na)]

This gives you: 1 2 4 5

You can also use complete.cases() if you have multiple objects and want to keep only the cases where there are no NAs in any of them.

Here are detailed notes on Vectorised Operations, Managing Data Frames with dplyr, and Control Structures in R:

### **Vectorised Operations**

**What are Vectorised Operations?**

In R, **vectorised operations** mean that you can perform operations on entire collections of numbers (vectors) at once, rather than having to loop through each number individually. This makes your code **simpler, faster, and easier to read**. R is designed to work this way, which is one of its strengths for data analysis.

**Simple Examples:**

Imagine you have two shopping lists of prices:

prices\_item1 <- c(1.50, 2.00, 1.00)

prices\_item2 <- c(0.75, 1.25, 0.50)

If you want to find the total price for each corresponding item, you can simply add the two vectors together:

total\_prices <- prices\_item1 + prices\_item2

print(total\_prices)

This will give you: 2.25 3.25 1.50

Behind the scenes, R adds the first element of prices\_item1 to the first element of prices\_item2, the second to the second, and so on. You didn't need to write a loop to do this!

**Other Vectorised Arithmetic Operations:**

The same principle applies to other arithmetic operations:

**Subtraction:** -  
  
 difference <- prices\_item1 - prices\_item2

print(difference) # 0.75 0.75 0.50

**Multiplication:** \*  
  
 doubled\_prices\_item1 <- prices\_item1 \* 2

print(doubled\_prices\_item1) # 3 4 2

**Division:** /  
  
 ratio <- prices\_item1 / prices\_item2

print(ratio) # 2.000000 1.600000 2.000000

**Vectorised Logical Operations:**

You can also perform logical comparisons on entire vectors:

temperatures <- c(20, 25, 18, 22)

above\_20 <- temperatures > 20

print(above\_20) # FALSE TRUE FALSE TRUE

This creates a new logical vector where each element is TRUE if the corresponding temperature is greater than 20, and FALSE otherwise. You can then use this logical vector for subsetting, as we discussed previously.

**Simple R Program:**

# Calculate the area of rectangles given lengths and widths

lengths <- c(5, 10, 7)

widths <- c(3, 2, 6)

areas <- lengths \* widths

print(paste("Areas:", areas))

# Check which areas are greater than 20

large\_areas <- areas > 20

print(paste("Large areas:", large\_areas))

**Simple Diagram Figure (Based on "R Programming for Data Science" Source):**

The source mentions "Vectorized Matrix Operations" and shows simple element-wise multiplication and division of matrices. Imagine two small 2x2 matrices, A and B. A diagram for element-wise multiplication would show that the element in the first row and first column of A is multiplied by the element in the first row and first column of B, and the result goes into the first row and first column of a new matrix C (the result). This happens for every corresponding element in the matrices. This visualises the operation occurring in parallel across the elements.

### **Managing Data Frames with the dplyr package**

**What is dplyr?**

**dplyr** is a very popular and powerful R package for **managing and manipulating data frames**. It provides a set of intuitive "verbs" that make common data frame operations much easier and more efficient than using base R functions. dplyr is designed to work in a way that is easy to understand and often mimics how you might think about data manipulation steps.

**Key dplyr Verbs:**

Here are some of the most important dplyr functions:

**select()**: **Chooses specific columns** from a data frame.  
  
 # Assuming you have a data frame called 'my\_data' with columns 'name', 'age', 'city'

library(dplyr)

selected\_data <- select(my\_data, name, age) # Selects the 'name' and 'age' columns

head(selected\_data)

You can also select ranges of columns or exclude columns using -:  
  
 select(my\_data, name:city) # Selects all columns from 'name' to 'city'

select(my\_data, -city) # Selects all columns except 'city'

**filter()**: **Chooses specific rows** from a data frame based on conditions.  
  
 filtered\_data <- filter(my\_data, age > 25) # Selects rows where 'age' is greater than 25

head(filtered\_data)

filtered\_data\_city <- filter(my\_data, city == "London") # Selects rows where 'city' is 'London'

head(filtered\_data\_city)

You can use multiple conditions with & (and) or | (or):  
  
 filter(my\_data, age > 20 & city == "Paris")

**arrange()**: **Reorders the rows** of a data frame based on the values of one or more columns.  
  
 arranged\_data\_age <- arrange(my\_data, age) # Arranges rows by 'age' in ascending order

head(arranged\_data\_age)

arranged\_data\_age\_desc <- arrange(my\_data, desc(age)) # Arranges by 'age' in descending order

head(arranged\_data\_age\_desc)

arranged\_data\_city\_age <- arrange(my\_data, city, age) # Arranges by 'city', then by 'age'

head(arranged\_data\_city\_age)

**rename()**: **Changes the names of columns**.  
  
 renamed\_data <- rename(my\_data, full\_name = name, years = age) # Renames 'name' to 'full\_name', 'age' to 'years'

head(renamed\_data)

**mutate()**: **Adds new columns** or modifies existing ones.  
  
 mutated\_data <- mutate(my\_data, age\_next\_year = age + 1) # Adds a new column 'age\_next\_year'

head(mutated\_data)

mutated\_data\_combined <- mutate(my\_data, city\_age = paste(city, age, sep = "-")) # Creates a combined column

head(mutated\_data\_combined)

**summarise() (or summarize()):** **Calculates summary statistics** for a data frame, often after grouping.  
  
 summarised\_data <- summarise(my\_data, average\_age = mean(age, na.rm = TRUE)) # Calculates the average age

print(summarised\_data)

**group\_by()**: **Groups the rows** of a data frame based on the values of one or more columns, for use with summarise().  
  
 grouped\_data <- group\_by(my\_data, city) # Groups data by 'city'

summarise(grouped\_data, average\_age = mean(age, na.rm = TRUE)) # Calculates average age per city

**%>% (Pipe Operator):** **Chains multiple dplyr operations together** in a readable sequence. It takes the output of the previous operation and passes it as the first argument to the next.  
  
 # Without pipe:

result <- summarise(group\_by(filter(select(my\_data, city, age), age > 20), city), average\_age = mean(age, na.rm = TRUE))

# With pipe:

result\_pipe <- my\_data %>%

select(city, age) %>%

filter(age > 20) %>%

group\_by(city) %>%

summarise(average\_age = mean(age, na.rm = TRUE))

print(result\_pipe)

**Simple R Program using dplyr:**

# Create a sample data frame

people <- data.frame(

name = c("Alice", "Bob", "Charlie", "David", "Eve"),

age = c(25, 30, 25, 35, 30),

city = c("London", "Paris", "London", "New York", "Paris"),

income = c(30000, 45000, 32000, 50000, 48000)

)

library(dplyr)

# Find the average income per city for people older than 28

average\_income\_per\_city <- people %>%

filter(age > 28) %>%

group\_by(city) %>%

summarise(mean\_income = mean(income))

print(average\_income\_per\_city)

**Simple Diagram Figure (Based on "R Programming for Data Science" Source):**

The source mentions the %>% operator and shows a sequence of dplyr operations. Imagine a data frame flowing through a pipe. The pipe has different "stations" along it. At the first station, the select() verb takes only certain columns. The data frame then flows to the next station where the filter() verb removes certain rows. It continues flowing to a group\_by() station where rows are grouped. Finally, at a summarise() station, a new summary table is created. This visualises how the data frame is transformed step-by-step through the dplyr verbs.

### **Control Structures**

**What are Control Structures?**

**Control structures** in R allow you to **control the order in which your code is executed**. They let you make decisions, repeat actions, and create more complex programs that don't just run line by line. The most common control structures are if-else, for loops, and while loops.

**1. if-else Statements:**

The if-else structure allows your program to **make decisions** based on whether a condition is true or false.

**if statement only:** Executes code only if the condition is TRUE.  
  
 temperature <- 15

if (temperature < 20) {

print("It's a bit chilly.")

}

# Output: "It's a bit chilly."

**if-else statement:** Executes one block of code if the condition is TRUE and another block if it's FALSE.  
  
 temperature <- 25

if (temperature < 20) {

print("It's a bit chilly.")

} else {

print("It's nice and warm.")

}

# Output: "It's nice and warm."

**if-else if-else statement:** Allows for multiple conditions to be checked in sequence.  
  
 score <- 75

if (score >= 80) {

grade <- "A"

} else if (score >= 70) {

grade <- "B"

} else if (score >= 60) {

grade <- "C"

} else {

grade <- "Fail"

}

print(paste("Grade:", grade))

# Output: "Grade: B"

**2. for Loops:**

A for loop allows you to **repeat a block of code a fixed number of times**. It's often used to iterate over elements of a vector or list.

# Loop through numbers 1 to 5

for (i in 1:5) {

print(paste("The current number is:", i))

}

# Output:

# "The current number is: 1"

# "The current number is: 2"

# "The current number is: 3"

# "The current number is: 4"

# "The current number is: 5"

# Loop through elements of a vector

fruits <- c("apple", "banana", "cherry")

for (fruit in fruits) {

print(paste("I like", fruit))

}

# Output:

# "I like apple"

# "I like banana"

# "I like cherry"

**3. while Loops:**

A while loop **repeats a block of code as long as a specified condition is TRUE**. You need to make sure that the condition eventually becomes FALSE to avoid an infinite loop.

count <- 0

while (count < 3) {

print(paste("Count is:", count))

count <- count + 1

}

# Output:

# "Count is: 0"

# "Count is: 1"

# "Count is: 2"

**Simple R Program demonstrating Control Structures:**

# Check if a number is positive, negative, or zero

number <- -5

if (number > 0) {

print("The number is positive.")

} else if (number < 0) {

print("The number is negative.")

} else {

print("The number is zero.")

}

# Print the first 5 even numbers using a for loop

cat("First 5 even numbers:\n")

even\_count <- 0

current\_number <- 2

while (even\_count < 5) {

print(current\_number)

current\_number <- current\_number + 2

even\_count <- even\_count + 1

}

**Simple Diagram Figure (Based on "R Programming for Data Science" Source):**

The source shows a basic if-else structure as a flowchart. A diamond shape represents the condition being tested. If the condition is TRUE, an arrow leads to a box containing the code to be executed. If the condition is FALSE, another arrow leads to a different box with the code for the else part. After either block of code is executed, the flowchart typically shows the program continuing to the next step. This visually demonstrates the decision-making flow of the if-else statement. For loops and while loops would be represented with shapes showing a repeated execution of a block of code until a certain condition is met.

Here are detailed notes on Functions, Scoping Rules of R, Coding Standards in R, and Loop Functions:

### **Functions**

**What are Functions?**

In R, **functions are blocks of reusable code** that perform a specific task. They allow you to organise your code, avoid repetition, and make your programs easier to understand and maintain. Think of a function as a mini-program within your main program.

**Defining a Function:**

You define a function in R using the function() directive, followed by the function's arguments (if any) in parentheses (), and the code that the function will execute within curly braces {}. You then assign this function to a name, just like you would assign a value to a variable.

# Define a function called 'add\_two' that takes one argument 'x'

add\_two <- function(x) {

result <- x + 2

return(result) # This line explicitly returns the value

}

# Define a function called 'greet' that takes no arguments

greet <- function() {

cat("Hello!\n") # 'cat' is used to print to the console

}

**Calling a Function:**

To use a function, you call its name followed by parentheses (). If the function has arguments, you provide values for those arguments inside the parentheses.

# Call the 'add\_two' function with the argument 5

output <- add\_two(5)

print(output) # Output: 7

# Call the 'greet' function

greet() # Output: Hello!

**Function Arguments:**

Functions can take **arguments** as input, which are variables that the function uses in its calculations or actions. You specify the names of the arguments in the function definition.

# Function with two arguments

multiply <- function(a, b) {

return(a \* b)

}

product <- multiply(3, 4)

print(product) # Output: 12

**Default Argument Values:**

You can also set **default values** for function arguments. If a user calls the function without providing a value for an argument with a default, the default value will be used.

# Function with a default value for 'exponent'

power <- function(base, exponent = 2) {

return(base ^ exponent)

}

# Call 'power' with only the 'base' argument (exponent defaults to 2)

squared <- power(5)

print(squared) # Output: 25

# Call 'power' with both arguments

cubed <- power(5, 3)

print(cubed) # Output: 125

**Returning Values:**

Functions often **return a value** as their result. You can explicitly return a value using the return() function. If you don't use return(), the function will automatically return the value of the **last expression** that was evaluated within the function body.

# Function that returns a list

create\_info <- function(name, age) {

info <- list(person\_name = name, person\_age = age)

return(info)

}

person\_details <- create\_info("Charlie", 30)

print(person\_details)

# Output:

# $person\_name

# "Charlie"

#

# $person\_age

# 30

**Simple R Program using Functions:**

# Function to calculate the area of a rectangle

calculate\_rectangle\_area <- function(length, width) {

area <- length \* width

return(area)

}

# Get the area of a rectangle with length 8 and width 5

rectangle\_area <- calculate\_rectangle\_area(8, 5)

print(paste("The area of the rectangle is:", rectangle\_area))

### **Scoping Rules of R**

**What are Scoping Rules?**

**Scoping rules** in R determine **how R looks up the value of a symbol** (like a variable name) when that symbol is used within a function. This is particularly important when you have variables with the same name in different parts of your code (e.g., inside a function and in your main workspace). R uses **lexical scoping** (also known as static scoping).

**Lexical Scoping:**

**Lexical scoping** means that **the value of a free variable within a function is looked for in the environment where the function was *defined***, not where the function was *called*.

Think of it like this: when R encounters a variable inside a function that wasn't passed as an argument and wasn't created within the function, it goes back to where the function was originally written down to find that variable.

**Environments:**

An **environment** in R is like a **collection of names (symbols) and the values they are associated with**. Your global workspace is an environment. When you load a package, it creates its own environment. Every function also has an associated environment – the environment in which it was created. Environments can have parent environments, forming a hierarchy.

**The Search Process:**

When R needs to find the value of a symbol, it follows these steps:

1. **Inside the current function:** First, it checks if the symbol is a formal argument of the function or if it was created as a local variable within the function's body.
2. **The defining environment:** If the symbol isn't found in the function itself, R looks in the environment where the function was *defined*.
3. **Parent environments:** If the symbol isn't in the defining environment, R continues to search in the parent environment of the defining environment, and so on, up the chain of environments.
4. **The global environment (workspace):** Eventually, R will reach your global workspace (where you type commands in the R console).
5. **The search list:** If still not found, R will search through the environments of the packages that you have loaded (in the order they appear on your search list, which you can see with search()).
6. **The empty environment:** The search ends at the empty environment. If the symbol is still not found, R will throw an error (e.g., "object 'y' not found").

**Function Closures:**

A **function closure** is a **function combined with the environment in which it was defined**. This means that even after the environment in which a function was created has ceased to exist (e.g., the function that created it has finished running), the function "remembers" the values of variables from that defining environment.

make\_multiplier <- function(factor) {

# The 'factor' variable is from the enclosing environment

function(x) {

return(x \* factor)

}

}

doubler <- make\_multiplier(2) # 'doubler' "remembers" factor = 2

tripler <- make\_multiplier(3) # 'tripler' "remembers" factor = 3

print(doubler(5)) # Output: 10 (5 \* 2)

print(tripler(5)) # Output: 15 (5 \* 3)

**Lexical vs. Dynamic Scoping (Conceptual):**

* **Lexical Scoping (R):** Looks at where the function was *written* to find variable values.
* **Dynamic Scoping (Less Common):** Would look at where the function was *called* to find variable values. This can lead to less predictable behaviour.

**Why Lexical Scoping Matters:**

Lexical scoping makes R code more **predictable and easier to reason about**. When you use a function, you can be more confident about where its free variables will get their values, based on how the function was defined.

### **Coding Standards in R**

**What are Coding Standards?**

**Coding standards** are a set of **guidelines and best practices** for writing code in a consistent and readable style. Following coding standards makes your code easier to understand, both for yourself in the future and for other people who might need to work with your code. While specific standards can vary, some general principles are widely accepted in the R community.

**Key Coding Standards (Based on Source):**

* **Always use text files and a text editor:** Write your R code in plain text files using a dedicated text editor or an Integrated Development Environment (IDE) like RStudio that has a built-in text editor with features like syntax highlighting. Avoid using word processors.

**Indent your code:** Use consistent **indentation** to visually represent the structure of your code (e.g., code within loops, if statements, and functions should be indented). The source suggests a minimum of 4 spaces and ideally 8 spaces per level of indentation. Consistent indentation significantly improves readability.  
  
 if (some\_condition) {

# Code inside the if block (indented)

result <- calculate\_something()

print(result)

} else {

# Code inside the else block (indented)

alternative\_result <- calculate\_alternative()

print(alternative\_result)

}

* **Limit the width of your code:** Keep lines of code to a reasonable length (e.g., 80 characters) to avoid horizontal scrolling and make your code easier to read on different screen sizes and when printed. This, combined with indentation, encourages writing modular and well-structured code.
* **Limit the length of individual functions:** Keep your functions focused and concise. A function should ideally perform one specific task. If a function becomes too long (e.g., more than one page of your editor), consider breaking it down into smaller, more manageable functions. This improves readability, testability, and maintainability.

**Other Common Coding Standards in R (Not Explicitly Detailed in Source but Good Practice):**

* **Use descriptive names for variables and functions:** Choose names that clearly indicate the purpose or content of the variable or the action of the function.
* **Add comments to your code:** Use comments (#) to explain complex logic, clarify the purpose of code sections, and document your work. Comments help others (and your future self) understand your code.
* **Be consistent with naming conventions:** Adopt a consistent style for naming variables and functions (e.g., snake\_case, camelCase).
* **Keep code simple and avoid unnecessary complexity:** Aim for clear and straightforward solutions.
* **Test your code:** Write and run tests to ensure your functions and programs work correctly.

### **Loop Functions**

**What are Loop Functions?**

**Loop functions** in R provide a way to **perform operations on elements of data structures (like lists, vectors, and arrays) in a more concise and often more efficient way than using explicit for or while loops**, especially in interactive work. They abstract away the explicit looping mechanism. The main loop functions are lapply(), sapply(), apply(), tapply(), and mapply().

**1. lapply() (List Apply):**

* **Purpose:** Applies a function to **each element of a list** and **returns a list** containing the results.
* **Input:** A list X, a function FUN, and optional arguments to FUN (...).
* **Output:** A list where each element is the result of applying FUN to the corresponding element of X.

my\_list <- list(a = 1:3, b = 4:6, c = 7:9)

# Calculate the sum of each vector in the list

sums <- lapply(my\_list, sum)

print(sums)

# Output: $a 6

# $b 15

# $c 24

# Calculate the mean of each vector in the list

means <- lapply(my\_list, mean)

print(means)

# Output: $a 2

# $b 5

# $c 8

**2. sapply() (Simplified Apply):**

* **Purpose:** Similar to lapply(), it applies a function to each element of a list or vector, but it **tries to simplify the output** to a vector or matrix if possible, instead of always returning a list.
* **Input:** Same as lapply().
* **Output:** A vector, matrix, or list, depending on the nature of the results.

my\_list <- list(a = 1:3, b = 4:6, c = 7:9)

# Calculate the sum of each vector (sapply simplifies to a named vector)

sums\_simple <- sapply(my\_list, sum)

print(sums\_simple)

# Output: a b c

# 6 15 24

# Calculate the mean of each vector

means\_simple <- sapply(my\_list, mean)

print(means\_simple)

# Output: a b c

# 2 5 8

**3. apply() (Apply Over Array Margins):**

* **Purpose:** Applies a function to the **rows or columns (or other margins) of a matrix or array**.
* **Input:** An array X, the margins to apply the function over MARGIN (1 for rows, 2 for columns, etc.), a function FUN, and optional arguments to FUN (...).
* **Output:** A vector, matrix, or array, depending on the result of FUN.

my\_matrix <- matrix(1:9, nrow = 3, byrow = TRUE)

print(my\_matrix)

# Output:

# [,1] [,2] [,3]

# 1 2 3

# 4 5 6

# 7 8 9

# Calculate the sum of each row (MARGIN = 1)

row\_sums <- apply(my\_matrix, 1, sum)

print(row\_sums) # Output: 6 15 24

# Calculate the mean of each column (MARGIN = 2)

col\_means <- apply(my\_matrix, 2, mean)

print(col\_means) # Output: 4 5 6

**4. tapply() (Table Apply):**

* **Purpose:** Applies a function to **subsets of a vector** defined by one or more factor variables.
* **Input:** A vector X, an index INDEX (a factor or list of factors), a function FUN, and optional arguments (...).
* **Output:** An array or list of values, structured according to the levels of the factors.

ages <- c(25, 30, 22, 35, 28, 32)

groups <- factor(c("A", "B", "A", "B", "A", "B"))

# Calculate the mean age for each group

mean\_ages <- tapply(ages, groups, mean)

print(mean\_ages)

# Output:

# A B

# 25.0 32.33333

**5. mapply() (Multivariate Apply):**

* **Purpose:** Applies a function to **corresponding elements of multiple vectors or lists** in parallel.
* **Input:** A function FUN, and one or more vectors or lists (...), along with optional MoreArgs and SIMPLIFY arguments.
* **Output:** A list or a simplified vector or matrix.

bases <- c(2, 3, 4)

exponents <- c(3, 2, 1)

# Calculate base raised to the power of exponent for each pair

results <- mapply(function(b, e) b ^ e, bases, exponents)

print(results) # Output: 8 9 4

# Simulate different numbers of random normals with different means

random\_data <- mapply(rnorm, n = 1:3, mean = 5:7)

print(random\_data)

# Output (will vary due to randomness):

# []

# 3.586563

#

# []

# 4.742398 6.834037

#

# []

# 5.511991 7.839925 5.269954

**Simple R Program using Loop Functions:**

# List of vectors

data\_list <- list(v1 = 1:5, v2 = 10:12, v3 = 20:22)

# Use lapply to square each element in each vector

squared\_list <- lapply(data\_list, function(x) x^2)

print(squared\_list)

# Use sapply to calculate the sum of each vector

sums\_vector <- sapply(data\_list, sum)

print(sums\_vector)

# Matrix

data\_matrix <- matrix(1:6, nrow = 2)

print(data\_matrix)

# Use apply to find the mean of each column

column\_means\_matrix <- apply(data\_matrix, 2, mean)

print(column\_means\_matrix)

Here are some notes on Debugging, Simulation, and Case studies on preliminary data analysis, drawing from the source "R Programming for Data Science":

### **Debugging**

Debugging is the process of finding and fixing errors in your code. When you run R code, things can go wrong, and R will give you different signals about these problems.

**Types of Issues:**

* **Message:** This is just a general piece of information that R gives you, often to tell you what it's doing. Your code usually continues to run after a message.

**Warning:** This indicates that something might be wrong, but your code doesn't necessarily stop. It's a sign that you should probably check what's happening, as it could lead to unexpected results later. For example, trying to take the logarithm of a negative number will give you a warning:  
  
 log(-1)

* This will produce a warning and give you NaN (Not a Number) as the result.

**Error:** This means that R has encountered a serious problem and cannot continue running your code. When an error occurs, the execution of the current function stops. For example, trying to use an object that hasn't been created will give you an error:  
  
 mean(x)

* This will result in an error if x doesn't exist.

**Basic Debugging Tools in R:**

R provides several tools to help you figure out what's going wrong in your code, especially within functions.

**traceback():** If your code produces an error, immediately after the error, you can run traceback(). This will show you the sequence of functions that were called before the error occurred. It helps you see which function the problem originated from. For example:  
  
 lm(y ~ x) # This will give an error if 'y' doesn't exist

traceback()

* The traceback() output will show you that the error happened within the lm() function.

**debug():** If you want to step through a function line by line to see what's happening at each step, you can use debug(your\_function). The next time you run your\_function, R will enter a special "debug mode". You can then use commands like n (for next line), c (to continue until the end or the next breakpoint), and Q (to quit debugging) to navigate the function's execution. To stop debugging a function, use undebug(your\_function).  
  
 my\_function <- function(a, b) {

result <- a + b

if (result > 10) {

print("Result is large")

} else {

print("Result is small")

}

return(result)

}

debug(my\_function)

my\_function(5, 6) # This will enter debug mode

**browser():** You can insert the browser() function directly into your code at a point where you want to pause execution and inspect the current state of your variables. When R reaches browser(), it will enter debug mode, similar to using debug().  
  
 another\_function <- function(x) {

y <- x \* 2

browser() # Execution will pause here

z <- y - 1

return(z)

}

another\_function(5)

**recover():** You can change R's default error behaviour so that when an error occurs, instead of just stopping, it allows you to examine the sequence of function calls (similar to traceback()) and even jump into the environment of any of those functions to inspect variables. You can set this option by running options(error = recover).  
  
 options(error = recover)

read.csv("nosuchfile") # This will trigger the recover mode after the error

### **Simulation**

Simulation in R involves creating artificial data that resembles real-world data. This is often done using random number generators that follow specific statistical distributions.

**Generating Random Numbers:**

R has functions for generating random numbers from many common probability distributions. These functions typically start with the letter r. Some examples include:

**rnorm(n, mean = 0, sd = 1):** Generates n random numbers from a Normal (Gaussian) distribution with a specified mean and standard deviation (sd). The default is a standard Normal distribution (mean 0, SD 1).  
  
 rnorm(5) # Generate 5 standard Normal random numbers

rnorm(10, mean = 20, sd = 2) # Generate 10 Normal random numbers with mean 20 and SD 2

**rpois(n, lambda):** Generates n random numbers from a Poisson distribution with a specified rate parameter (lambda). The Poisson distribution is often used for count data.  
  
 rpois(10, lambda = 1) # Generate 10 Poisson random numbers with rate 1

rpois(5, lambda = 5) # Generate 5 Poisson random numbers with rate 5

**rbinom(n, size, prob):** Generates n random numbers from a Binomial distribution, where size is the number of trials and prob is the probability of success on each trial.  
  
 rbinom(100, size = 1, prob = 0.5) # Simulate 100 coin flips (0 or 1)

rbinom(50, size = 10, prob = 0.3) # Simulate 50 sets of 10 trials with probability of success 0.3

**Setting the Random Number Seed:**

It's very important to set the random number seed using set.seed(your\_seed\_number) before generating random numbers if you want your results to be reproducible. The your\_seed\_number should be an integer. If you set the same seed, you will get the same sequence of random numbers each time you run your code.

set.seed(1)

rnorm(5)

# -0.6264538 0.1836433 -0.8356286 1.5952808 0.3295078

set.seed(1)

rnorm(5)

# -0.6264538 0.1836433 -0.8356286 1.5952808 0.3295078

**Simulating a Linear Model:**

You can simulate data from a linear model by first generating values for the predictor variable(s) and the error term, and then combining them according to the model equation.

# Define the model parameters

beta0 <- 0.5 # Intercept

beta1 <- 2 # Slope

sd\_error <- 2 # Standard deviation of the error term

n\_samples <- 100 # Number of data points

# Simulate the predictor variable 'x' from a Normal distribution

set.seed(20)

x <- rnorm(n\_samples)

# Simulate the error term 'e' from a Normal distribution with mean 0 and sd\_error

e <- rnorm(n\_samples, mean = 0, sd = sd\_error)

# Compute the outcome 'y' according to the linear model: y = beta0 + beta1\*x + e

y <- beta0 + beta1 \* x + e

# You can then plot the simulated data

plot(x, y)

### **Data Analysis Case Study: Changes in Fine Particle Air Pollution in the U.S.**

The source provides a case study on analyzing changes in fine particle air pollution (PM2.5) in the U.S. between 1999 and 2012. This case study demonstrates preliminary data analysis steps.

**Synopsis:**

The goal was to see if PM2.5 levels had decreased across the U.S. between 1999 and 2012, based on the Clean Air Act. Data from the Environmental Protection Agency (EPA) were used for these two years. The analysis found an overall decrease in PM2.5 levels on average across the U.S..

**Preliminary Data Analysis Steps:**

**Loading the Raw Data:** The 1999 and 2012 data were read into R using read.table(). The data files were pipe-delimited (sep = "|") with missing values as blank fields (na.strings = ""). Comment lines starting with # were skipped (comment.char = "#"). The header argument was initially set to FALSE to read the data without assuming a header.  
  
 pm99 <- read.table("pm25\_data/RD\_501\_88101\_1999-0.txt", comment.char = "#", header = FALSE, sep = "|", na.strings = "")

pm12 <- read.table("pm25\_data/RD\_501\_88101\_2012-0.txt", comment.char = "#", header = FALSE, sep = "|", na.strings = "")

**Adding Column Names:** Column names were read from the first line of one of the data files and then assigned to the datasets using names() and strsplit(). make.names() was used to ensure the names were valid R names.  
  
 cnames <- readLines("pm25\_data/RD\_501\_88101\_1999-0.txt", 1)

cnames <- strsplit(cnames, "|", fixed = TRUE)

names(pm99) <- make.names(cnames[])

names(pm12) <- make.names(cnames[])

**Extracting the Variable of Interest:** The Sample.Value column, containing the PM2.5 measurements, was extracted from both datasets.  
  
 pm99\_values <- pm99$Sample.Value

pm12\_values <- pm12$Sample.Value

**Checking for Missing Values:** The proportion of missing values (NA) in the Sample.Value column was checked using mean(is.na(x)).  
  
 mean(is.na(pm99\_values))

mean(is.na(pm12\_values))

**Summarizing the Data:** Basic descriptive statistics for the PM2.5 values in both years were obtained using the summary() function. This gave information about the minimum, maximum, quartiles, and mean.  
  
 summary(pm99\_values)

summary(pm12\_values)

**Initial Visualization:** Boxplots of the logarithm (base 2) of the PM2.5 values for 1999 and 2012 were created using boxplot(log2(x0), log2(x1)) to visually compare the distributions. The logarithm transformation can help in visualizing skewed data.  
  
 boxplot(log2(pm99\_values), log2(pm12\_values))

**Simple Diagram Figures:**

The source includes simple diagram figures related to this case study:

* The code produces a boxplot comparing the distributions of log2(PM2.5) in 1999 and 2012 [152, plot of chunk unnamed-chunk-5]. This is a simple visual representation of the change in PM2.5 levels over time across the entire U.S.

This case study demonstrates the initial steps in analyzing a real-world dataset: reading the data, ensuring it's in a usable format, checking for basic issues like missing values, obtaining summary statistics, and creating simple visualizations to get a first look at the data and potential trends. These are all crucial aspects of preliminary data analysis.