Introduction to Data Science

Data Science Primer

First Edition

Data Science

First Edition

Arjun Anil Kumar
Trivandrum, India

Self Publishers Worldwide
Seattle San Francisco New York
London Paris Rome Beijing Barcelona

This book was typeset using LaTeX software with the help of Generative AI tools.

Preface

Table of Contents

Chapter 1

Introduction

1.1 Introduction to Data Science Framework

Data science is an interdisciplinary field that combines various domains to extract insights from data (Figure 1.1). The key components of data science are:

1. **Data Collection**
   The process of gathering raw data from various sources such as databases, sensors, web scraping, surveys, and APIs.
   - **Sources**: Structured databases, unstructured data (text, images), real-time data streams.
   - **Tools**: Web scraping libraries, APIs (RESTful), IoT devices.
2. **Data Processing & Cleaning**
   Ensures that the collected data is accurate, complete, and formatted for analysis by handling missing values, outliers, and inconsistencies.
   - **Techniques**: Handling missing data, data normalization, data transformation.
   - **Tools**: Pandas, NumPy, PySpark, ETL pipelines.
3. **Data Exploration & Visualization**
   Involves analyzing and visualizing data to understand patterns, trends, and relationships.
   - **Techniques**: Descriptive statistics, exploratory data analysis (EDA).
   - **Tools**: Matplotlib, Seaborn, Tableau, Power BI.
4. **Feature Engineering**
   Creating new variables or features from existing data to improve model performance.

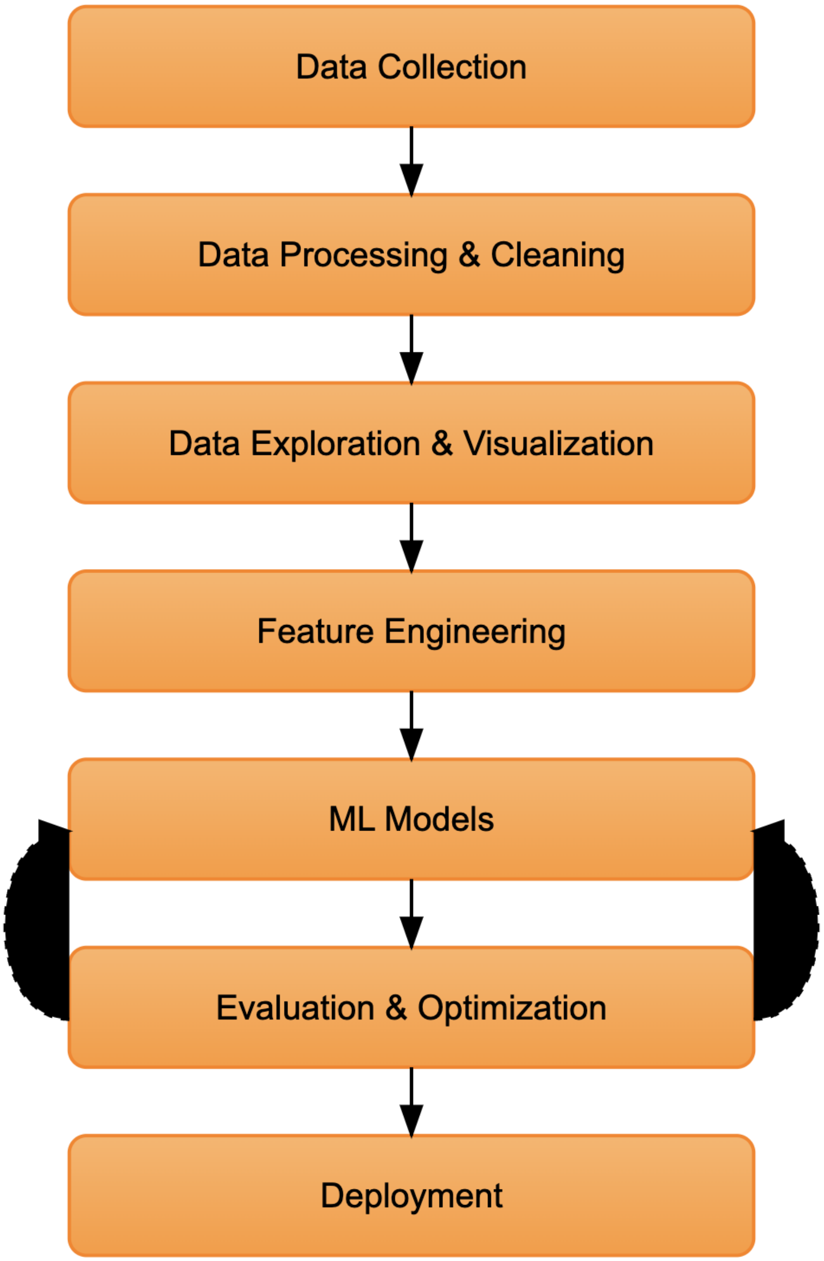

Data Collection
Data Processing & Cleaning
Data Exploration & Visualization
Feature Engineering
ML Models
Evaluation & Optimization
Deployment

Figure 1.1: Data Science

- **Techniques**: Scaling, encoding categorical variables, polynomial features, dimensionality reduction.
- **Tools**: Scikit-learn, FeatureTools.

5. **Modeling & Machine Learning**
   Building predictive models using machine learning algorithms and statistical methods.

   - **Techniques**: Regression, classification, clustering, neural networks, deep learning.
   - **Tools**: Scikit-learn, TensorFlow, PyTorch.

6. **Evaluation & Optimization**
   Assessing model performance using appropriate metrics and optimizing models for better accuracy and efficiency.

   - **Metrics**: Accuracy, precision, recall, F1 score, ROC-AUC, MSE, R^2.
   - **Techniques**: Hyperparameter tuning, cross-validation, model selection.
   - **Tools**: GridSearchCV, RandomizedSearchCV, Optuna.

7. **Deployment**
   Making models available for use in real-world applications. This includes creating APIs, integrating models into applications, and ensuring scalability.

   - **Tools**: Flask, FastAPI, Docker, Kubernetes, CI/CD pipelines, cloud services (AWS, Azure, GCP).

8. **Communication & Reporting**
   Presenting the findings and results of data science work to stakeholders through dashboards, reports, or presentations.

   - **Tools**: Jupyter Notebooks, PowerPoint, Tableau, Plotly.

9. **Data Science Ethics**
   Ensures ethical handling of data, addressing privacy concerns, bias in models, and transparency.

   - **Concepts**: Data privacy (GDPR, HIPAA), fairness, accountability.

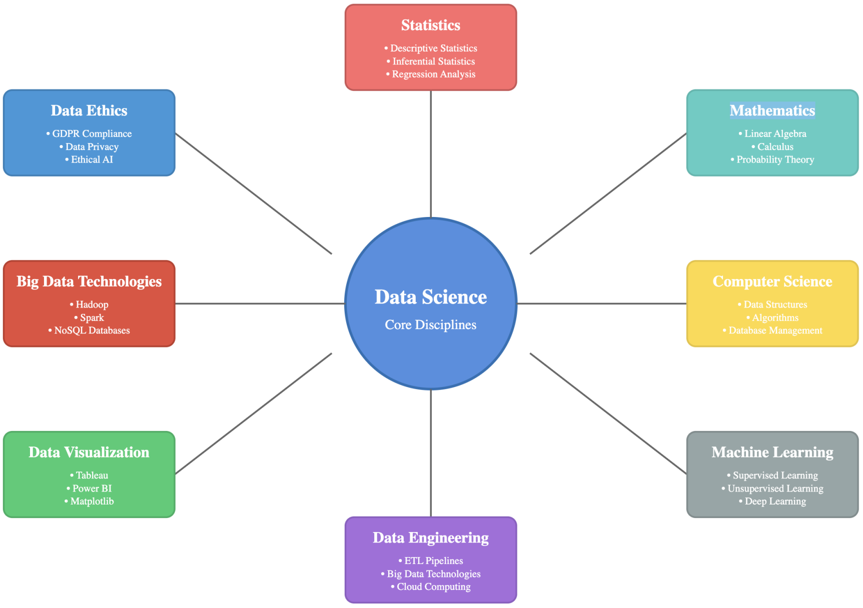


Figure 1.2: Data Science - Interdisciplinary Nature

1.2 Decision Making using Linear Regression

Problem Definition

- **Objective:** Predict a student's final exam score based on study hours.
- **Assumption:** There is a linear relationship between study hours and exam scores.

1. Data Collection
   - **Method:** Gather data from a sample of students, recording the number of hours each student studied and their corresponding exam scores.
2. Data Preparation
   - **Cleaning:** Ensure there are no missing or erroneous values in the dataset.
   - **Exploration:** Visualize the data to understand the relationship between study hours and exam scores.
   - **Splitting:** Divide the data into training and testing sets to evaluate the model's performance.

Student	**Study Hours (X)**	**Exam Score (Y)**
1	2	50
2	3	55
3	5	65
4	7	70
5	9	85

Table 1.1: Sample Data of Study Hours and Exam Scores

3. Exploratory Data Analysis (EDA)

   - **Visualization:** Create a scatter plot of study hours vs. exam scores to observe any linear trends.

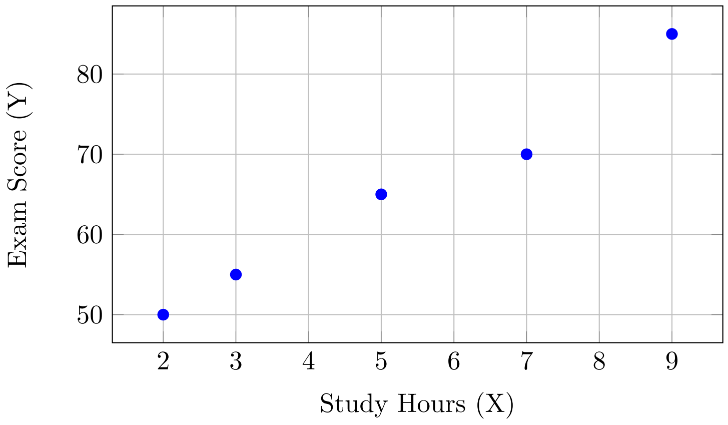


Figure 1.3: Scatter Plot of Study Hours vs. Exam Scores

4. Model Building

   - **Model Selection:** Choose simple linear regression as the modeling technique.
   - **Training:** Fit the linear regression model to the training data to learn the relationship between study hours and exam scores.

   The simple linear regression model is represented as:

$$Y = \beta_0 + \beta_1 X + \epsilon$$

   Where:

   - Y is the dependent variable (Exam Score).

- X is the independent variable (Study Hours).
- β_0 is the intercept.
- β_1 is the slope coefficient.
- ϵ is the error term.

5. Model Evaluation

   - **Testing:** Use the testing set to assess the model's predictive performance.
   - **Metrics:** Calculate evaluation metrics such as Mean Absolute Error (MAE), Mean Squared Error (MSE), and R-squared to determine the model's accuracy.

6. Model Deployment

   - **Application:** Implement the model in a real-world setting where it can predict exam scores based on new data of study hours.
   - **Monitoring:** Continuously monitor the model's performance and update it with new data as necessary.

1.3 Decision Making using Logistic Regression

Problem Definition

- **Objective:** Predict the likelihood of a customer making a purchase (Yes/No) based on their browsing behavior.
- **Assumption:** Customer behavior data can be used to model the probability of a purchase decision.

1. Data Collection

   - **Method:** Collect data on customers' browsing behavior, such as the number of pages viewed, time spent on the website, and previous purchase history.

2. Data Preparation

   - **Cleaning:** Handle missing values, remove duplicates, and correct any inconsistencies in the dataset.
   - **Feature Engineering:** Create relevant features, such as average time per page or engagement score, to enhance predictive power.
   - **Normalization:** Scale numerical features to ensure uniformity and improve model performance.

Customer ID	Pages Viewed	Time Spent (minutes)	Previous Purchases	Purchase (Yes=1, No=0)
1	5	10	0	0
2	15	25	1	1
3	7	12	0	0
4	20	30	2	1
5	3	5	0	0

Table 1.2: Sample Data of Customer Browsing Behavior and Purchase Decisions

- **Splitting:** Divide the data into training and testing sets to evaluate the model's performance.

3. Exploratory Data Analysis (EDA)

   - **Visualization:** Generate histograms, box plots, and scatter plots to understand the distribution of variables and relationships between them.
   - **Correlation Analysis:** Calculate correlation coefficients to identify significant predictors of the purchase decision.

4. Model Building

   - **Model Selection:** Choose logistic regression as the modeling technique due to its suitability for binary classification.
   - **Training:** Fit the logistic regression model to the training data to estimate the relationship between predictors and the probability of purchase.

   The logistic regression model is represented as:

$$\text{logit}(P) = \beta_0 + \beta_1 \times \text{Pages Viewed} + \beta_2 \times \text{Time Spent} + \beta_3 \times \text{Previous Purchases}$$

   Where:

   - P is the probability of purchase.
   - $\beta_0, \beta_1, \beta_2, \beta_3$ are the coefficients estimated by the model.

5. Model Evaluation

   - **Testing:** Apply the model to the testing set to predict purchase probabilities.
   - **Metrics:** Assess model performance using metrics such as accuracy, precision, recall, F1-score, and the area under the ROC curve (AUC-ROC).

- **Confusion Matrix:** Construct a confusion matrix to evaluate the model's classification performance.

6. Model Deployment

   - **Implementation:** Integrate the trained model into the company's website to provide real-time purchase probability predictions for new customers.
   - **Decision-Making:** Use the model's predictions to inform marketing strategies, such as offering discounts to customers with a high probability of purchase.
   - **Monitoring:** Continuously monitor the model's performance and retrain it with new data to maintain accuracy over time.

1.4 Decision Making using Decision Tree

Problem Definition

- **Objective:** Predict whether a customer will make a purchase (Yes/No) based on their browsing behavior.
- **Assumption:** Customer browsing patterns can provide insights into their purchasing decisions.

1. Data Collection

   - **Method:** Gather data on customers' browsing activities, including:
     - Number of pages viewed
     - Time spent on the website
     - Previous purchase history
     - Demographic information (age, gender, location)

Customer ID	**Pages Viewed**	**Time Spent (min)**	**Previous Purchases**	**Age**	**Gender**	**Purchase (1=Yes, 0=No)**
1	5	10	0	25	M	0
2	15	25	1	34	F	1
3	7	12	0	22	M	0
4	20	30	2	45	F	1
5	3	5	0	30	M	0

Table 1.3: Sample Data of Customer Browsing Behavior and Purchase Decisions

2. Data Preparation

- **Cleaning:** Address missing values, remove duplicates, and correct inconsistencies.
- **Feature Engineering:** Create new features, such as:
  - Average time per page
  - Engagement score
- **Encoding:** Convert categorical variables (e.g., gender) into numerical formats using techniques like one-hot encoding.
- **Splitting:** Divide the dataset into training and testing subsets to evaluate model performance.

3. Exploratory Data Analysis (EDA)

   - **Visualization:** Create histograms, box plots, and scatter plots to understand variable distributions and relationships.
   - **Correlation Analysis:** Compute correlation coefficients to identify significant predictors of purchase behavior.

4. Model Building

   - **Model Selection:** Choose a decision tree classifier for its interpretability and ability to handle both numerical and categorical data.
   - **Training:** Fit the decision tree model to the training data to learn decision rules that predict purchase behavior.

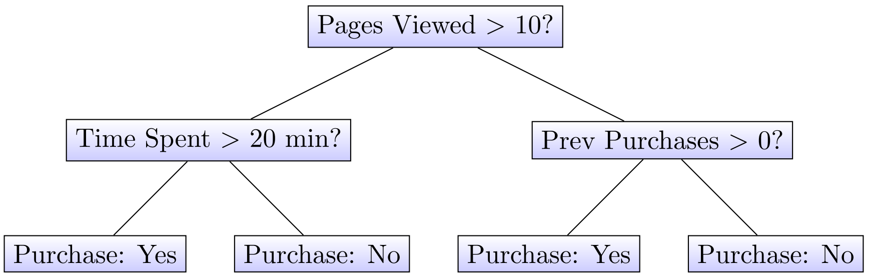


Figure 1.4: Simplified Decision Tree for Predicting Customer Purchases
Left direction indicates "yes" and Right direction indicates "no"

5. Model Evaluation

   - **Testing:** Apply the trained model to the testing set to predict purchase decisions.
   - **Metrics:** Assess performance using metrics such as accuracy, precision, recall, F1-score, and the area under the ROC curve (AUC-ROC).

- **Confusion Matrix:** Construct a confusion matrix to evaluate the model's classification performance.

6. Model Deployment

   - **Implementation:** Integrate the decision tree model into the company's website to provide real-time purchase predictions for new customers.
   - **Decision-Making:** Use model predictions to inform marketing strategies, such as personalized recommendations or targeted promotions.
   - **Monitoring:** Continuously monitor model performance and retrain with new data to maintain accuracy over time.

7. Decision Tree Visualization The decision tree is provided in Figure 1.4.

Chapter 2

Modelling using ML Algorithms

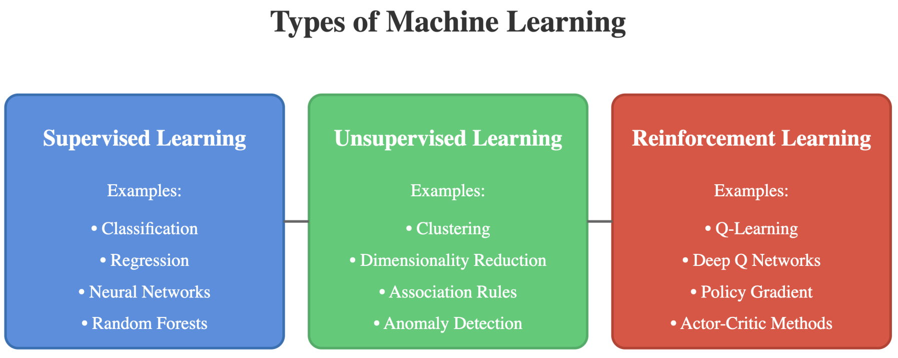


Figure 2.1: Machine Learning Models

Supervised learning involves training a model on a labeled dataset, meaning each training example is paired with an output label. The objective is for the model to learn the mapping from inputs to outputs, enabling it to make accurate predictions on new, unseen data. Common tasks include:

- **Classification:** Assigning inputs to discrete categories. For example, determining whether an email is 'spam' or 'not spam'.
- **Regression:** Predicting continuous numerical values. For instance, forecasting house prices based on features like size and location.

Unsupervised learning involves training models on datasets without explicit output labels. The goal is to uncover underlying patterns, structures, or relationships within the data. Key tasks include:

- **Clustering:** Grouping similar data points together. An example is segmenting customers into distinct groups based on purchasing behavior.
- **Dimensionality Reduction:** Reducing the number of features in a dataset while preserving significant information. Techniques like Principal Component Analysis (PCA) are used for this purpose.

The primary distinction between these methods lies in the presence of labeled data: supervised learning relies on labeled datasets to train models, while unsupervised learning works with unlabeled data to identify inherent structures.

Reinforcement learning is outside the scope of this book.

2.1 Decision Tree

2.1.1 Decision Tree Example Using Gini Impurity

In decision tree algorithms, **Gini Impurity** is a metric used to evaluate the quality of a split at each node. **It measures the likelihood of a randomly chosen element being incorrectly classified if it were assigned a class label based on the distribution of classes in that node**. A Gini Impurity of 0 indicates perfect purity (all elements belong to a single class), while a value closer to 0.5 suggests higher impurity. An example problem is given below.

Consider a dataset of 10 customers with the following attributes:

Customer	**Age**	**Income**	**Purchased Product**
1	25	$30k$	Yes
2	30	$40k$	No
3	35	$50k$	Yes
4	40	$60k$	No
5	45	$70k$	Yes
6	50	$80k$	No
7	55	$90k$	Yes
8	60	$100k$	No
9	65	$110k$	Yes
10	70	$120k$	No

1. Objective: Determine the best attribute to split the dataset to predict whether a customer will purchase the product.

2. Calculate Gini Impurity for the Entire Dataset

   The dataset has 10 customers, with 5 purchasing the product (*Yes*) and 5 not purchasing (*No*).

$$\text{Probability of 'Yes'}(p_1) = \frac{5}{10} = 0.5$$

$$\text{Probability of 'No'}(p_2) = \frac{5}{10} = 0.5$$

The Gini Impurity (G) is calculated as:

$G = 1 - (p_1^2 + p_2^2) = 1 - (0.5^2 + 0.5^2) = 1 - (0.25 + 0.25) = 1 - 0.5 = 0.5$

3. Evaluate Potential Splits based on 'Age' Let's consider splitting the dataset based on the *Age* attribute.

   Split 1: Age <= 50

Customer	**Age**	**Income**	**Purchased Product**
1	25	$30k$	Yes
2	30	$40k$	No
3	35	$50k$	Yes
4	40	$60k$	No
5	45	$70k$	Yes

   - *Yes* count = 3, *No* count = 2
   - $p_1 = \frac{3}{5} = 0.6$, $p_2 = \frac{2}{5} = 0.4$
   Gini Impurity for this split:

$$G_{\text{split 1}} = 1 - (0.6^2 + 0.4^2) = 1 - (0.36 + 0.16) = 1 - 0.52 = 0.48$$

   Split 2: Age > 50

Customer	**Age**	**Income**	**Purchased Product**
6	50	$80k$	No
7	55	$90k$	Yes
8	60	$100k$	No
9	65	$110k$	Yes
10	70	$120k$	No

   - *Yes* count = 2, *No* count = 3
   - $p_1 = \frac{2}{5} = 0.4$, $p_2 = \frac{3}{5} = 0.6$
   Gini Impurity for this split:

$$G_{\text{split 2}} = 1 - (0.4^2 + 0.6^2) = 1 - (0.16 + 0.36) = 1 - 0.52 = 0.48$$

Calculate Weighted Gini Impurity for the Splits

- Weighted Gini for Split 1:

$$\frac{5}{10} \times 0.48 = 0.24$$

- Weighted Gini for Split 2:

$$\frac{5}{10} \times 0.48 = 0.24$$

Total Gini Impurity after the split:

$$G_{\text{after split}} = 0.24 + 0.24 = 0.48$$

Compare with Original Gini Impurity

- Original Gini Impurity = 0.5 - Gini Impurity after split = 0.48

Since the Gini Impurity decreased from 0.5 to 0.48, the *Age* attribute is able to reduce the Gini Impurity.

Conclusion- By calculating the Gini Impurity for potential splits, we can determine that splitting the dataset based on the *Age* attribute (<= 50 vs. >50) results in a lower Gini Impurity, indicating a more effective split for predicting product purchases.

4. Evaluate Potential Splits Based on 'Income'

Let's consider splitting the dataset based on the 'Income' attribute at various thresholds.

Split 1: Income <= 60k

Customer	**Age**	**Income**	**Purchased Product**
1	25	$30k$	Yes
2	30	$40k$	No
3	35	$50k$	Yes
4	40	$60k$	No

- 'Yes' count = 2, 'No' count = 2 - $p_1 = \frac{2}{4} = 0.5$, $p_2 = \frac{2}{4} = 0.5$

Gini Impurity for this split:

$$G_{\text{split 1}} = 1 - (0.5^2 + 0.5^2) = 1 - (0.25 + 0.25) = 1 - 0.5 = 0.5$$

Split 2: Income > 60k

Customer	**Age**	**Income**	**Purchased Product**
5	45	$70k$	Yes
6	50	$80k$	No
7	55	$90k$	Yes
8	60	$100k$	No
9	65	$110k$	Yes
10	70	$120k$	No

- 'Yes' count = 3, 'No' count = 3 - $p_1 = \frac{3}{6} = 0.5$, $p_2 = \frac{3}{6} = 0.5$
Gini Impurity for this split:

$$G_{\text{split 2}} = 1 - (0.5^2 + 0.5^2) = 1 - (0.25 + 0.25) = 1 - 0.5 = 0.5$$

Calculate Weighted Gini Impurity for the Splits

- Weighted Gini for Split 1:

$$\frac{4}{10} \times 0.5 = 0.2$$

- Weighted Gini for Split 2:

$$\frac{6}{10} \times 0.5 = 0.3$$

Total Gini Impurity after the split:

$$G_{\text{after split}} = 0.2 + 0.3 = 0.5$$

Compare with Original Gini Impurity

- Original Gini Impurity = 0.5 - Gini Impurity after split = 0.5
Since the Gini Impurity remains the same after the split, the 'Income' attribute does not provide a better split than 'Age'.

Conclusion-In this example, splitting the dataset based on the 'Income' attribute does not reduce the Gini Impurity, indicating that 'Income' may not be a useful attribute for predicting product purchases in this dataset.

5. Concept of Gini Impurity - It quantifies the likelihood of a randomly chosen element being incorrectly classified if it were assigned a class label according to the distribution of classes in the dataset. Assume there are two classes of customers C_1 and C_2 based on the features age and income. Suppose we randomly pick a customer in our dataset, he has a probability of the customer belonging to C_1 is $p_1 = \frac{n_1}{n_1+n_2}$ and C_2 is

Event	Probability
C1 selected and C1 classified	.25
C1 selected and C2 classified	.25
C2 selected and C1 classified	.25
C2 selected and C2 classified	.25

Table 2.1: Event and Probability for randomly sampling from 2 Classes of Customers

$p_2 = \frac{n_2}{n_1+n_2}$, where n_1 and n_2 are the number of customers in classes C_1 and C_2.

Consider a dataset D containing samples from k classes. Let p_i denote the probability of a randomly selected sample belonging to class i. The Gini Impurity G is defined as:

$$G = 1 - \sum_{i=1}^{k} p_i^2$$

(a) Random Selection of a Sample The probability of selecting a sample from class i is p_i. The probability of selecting a sample from any class other than i is $1 - p_i$.

(b) Incorrect Classification Probability

If a sample is from class i, the probability of misclassifying it (i.e., assigning it to any class other than i) is $1 - p_i$. If a sample is from any class other than i, the probability of misclassifying it as class i is p_i.

(c) Total Probability of Misclassification

The total probability of misclassifying a randomly selected sample is the sum of the probabilities of misclassification for all classes:

$$\text{Total Misclassification Probability} = \sum_{i=1}^{k} p_i \times (1 - p_i)$$

(d) Gini Impurity Definition

The Gini Impurity is defined as the probability of misclassifying a randomly selected sample:

$$G = \sum_{i=1}^{k} p_i \times (1 - p_i)$$

Expanding the expression:

$$G = \sum_{i=1}^{k} p_i - \sum_{i=1}^{k} p_i^2$$

Since the sum of all probabilities equals 1 ($\sum_{i=1}^{k} p_i = 1$):

$$G = 1 - \sum_{i=1}^{k} p_i^2$$

This formula quantifies the Gini Impurity of a dataset, where a lower value indicates a purer node in a decision tree.

2.1.2 Binary Regression Decision Tree using Weighted Variance

We aim to construct a binary regression decision tree for predicting continuous values, specifically the price of a house, based on categorical features (location and type). We will split the data using binary splits based on the features and compute the weighted variance for each split.

The dataset consists of the following features:

$$\text{Location: A, B, C,} \quad \text{Type: Villa, Flat,} \quad \text{Price: continuous}$$

Location	Type	Price
A	*Villa*	200
A	*Flat*	100
A	*Villa*	220
A	*Flat*	120
B	*Villa*	800
B	*Flat*	700
B	*Villa*	880
B	*Flat*	780
C	*Villa*	1600
C	*Flat*	1500
C	*Villa*	1800
C	*Flat*	1600

We will compute the variance for each possible split: one based on *Location* and one based on *Type*.

Group the data by Location and compute the variance for each group.

Group 1: Location A + Location B

$$\text{Prices} = \{200, 100, 220, 120, 800, 700, 880, 780\}$$

$$\bar{y}_1 = \frac{200 + 100 + 220 + 120 + 800 + 700 + 880 + 780}{8} = 600$$

Variance for Group 1:

$$\text{Variance}_1 = \frac{1}{8}\Big[(200-600)^2 + (100-600)^2 + (220-600)^2 + (120-600)^2 + (800-600)^2 + (700-600)^2 + (880-600)^2 + (780-600)^2\Big]. \quad (2.1)$$

Group 2: Location C

$$\text{Prices} = \{1600, 1500, 1800, 1600\}$$

$$\bar{y}_2 = \frac{1600+1500+1800+1600}{4} = 1625$$

Variance for Group 2:

$$\text{Variance}_2 = \frac{1}{4}\Big[(1600-1625)^2 + (1500-1625)^2 + (1800-1625)^2 + (1600-1625)^2\Big]. \quad (2.2)$$

Split by Type Group the data by Type and compute the variance for each group.

Group 1: Villa

$$\text{Prices} = \{200, 220, 800, 880, 1600, 1800\}$$

$$\bar{y}_1 = \frac{200+220+800+880+1600+1800}{6} = 1000$$

Variance for Group 1:

$$\text{Variance}_1 = \frac{1}{6}\Big[(200-1000)^2 + (220-1000)^2 + (800-1000)^2 + (880-1000)^2 + (1600-1000)^2 + (1800-1000)^2\Big]. \quad (2.3)$$

Group 2: Flat

$$\text{Prices} = \{100, 120, 700, 780, 1500, 1600\}$$

$$\bar{y}_2 = \frac{100+120+700+780+1500+1600}{6} = 950$$

Variance for Group 2:

$$\text{Variance}_2 = \frac{1}{6}\Big[(100-950)^2 + (120-950)^2 + (700-950)^2 + (780-950)^2 + (1500-950)^2 + (1600-950)^2\Big]. \quad (2.4)$$

Step 3: Compute Weighted Variance - The weighted variance for a given split is calculated as:

$$\text{Weighted Variance} = \frac{N_1}{N} \times \text{Variance}_1 + \frac{N_2}{N} \times \text{Variance}_2$$

Where: - N_1 and N_2 are the number of samples in each group. - N is the total number of samples.

For the *Location* split, we compute the weighted variance as:

$$\text{Weighted Variance for Location Split} = \frac{8}{12} \times \text{Variance}_1 + \frac{4}{12} \times \text{Variance}_2$$

For the *Type* split, we compute the weighted variance as:

$$\text{Weighted Variance for Type Split} = \frac{6}{12} \times \text{Variance}_1 + \frac{6}{12} \times \text{Variance}_2$$

Step 4: Choose the Best Split Finally, the best split is the one that results in the lowest weighted variance. We compare the weighted variances from both splits and choose the one with the smallest value.

$$\begin{aligned}\text{Best Split} = \text{Location} \quad \text{if} \quad & \text{Weighted Variance for Location Split} \\ & < \text{Weighted Variance for Type Split}. \end{aligned} \tag{2.5}$$

$$\begin{aligned}\text{Best Split} = \text{Type} \quad \text{if} \quad & \text{Weighted Variance for Type Split} \\ & < \text{Weighted Variance for Location Split}. \end{aligned} \tag{2.6}$$

2.2 Neural Networks

A **Neural Network** is a computational model inspired by the way biological neural networks in the human brain work. It is a fundamental tool in machine learning and deep learning, widely used for tasks such as classification, regression, image recognition, natural language processing, and more.

Components of a Neural Network

- **Neurons (Nodes)**: These are the basic units in the network. Each neuron receives inputs, processes them, and passes the result to the next layer of neurons. For example, the neural network shown in Figure 2.2 is a 9 node neural network with 4 nodes in the input layer, 3 nodes in the hidden layer and 2 nodes in the output layer.

- **Layers**: Neural networks are composed of layers:

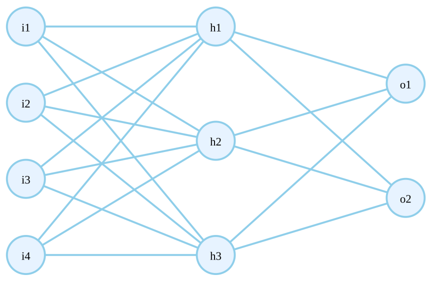


Figure 2.2: 9 Node Neural Network (4,3,2)

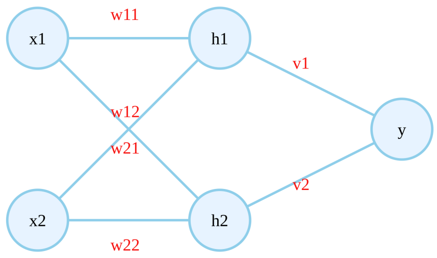


Figure 2.3: Weights in a 5 Node Neural Network

  - **Input Layer**: This is the first layer where the input data is fed into the network.
  - **Hidden Layers**: These layers perform computations and extract features from the data.
  - **Output Layer**: The final layer that provides the result of the neural network's computation (classification, regression output).

- **Weights**: Weights are the parameters that control the strength of the connections between neurons. They are adjusted during training to minimize the error. For example, Figure 2.3 shows a neural network with 5 nodes with 2 nodes in the input layer, 2 nodes in the hidden layer and 1 node in the output layer.

- **Bias**: Bias is added to the output of the neuron, helping the network make better predictions by shifting the activation function.

- **Activation Function**: The activation function determines whether a neuron should be activated or not. Common activation functions, shown in Figure 2.4, include:

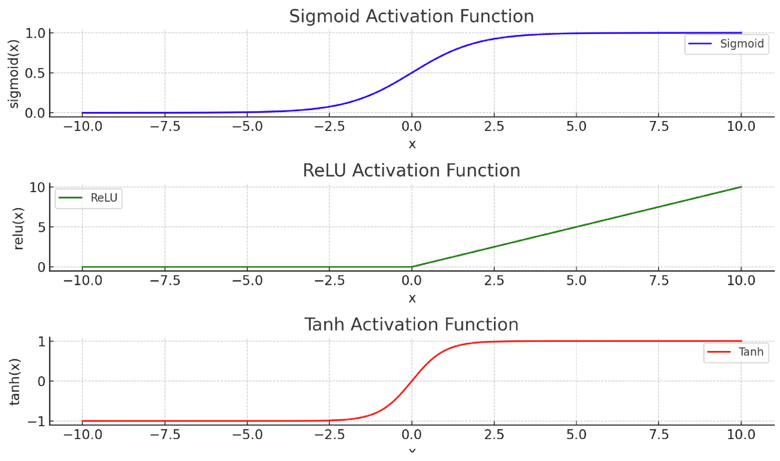


Figure 2.4: Activation Functions

- **Sigmoid**: Sigmoid Activation Function is a smooth, S-shaped symmetric curve that outputs values between 0 an 1, rendering them useful as output layer in binary classification problems.

$$f_{SIGMOID}(x) = \frac{1}{1+e^{-x}}$$

- **ReLU (Rectified Linear Unit)**: Outputs the input directly if positive, otherwise 0. The output ranges between 0 and ∞.

$$f_{RELU}(x) = \max(0, x)$$

  Note that RELU has the feature of SPARSE activation and turns on the neuron where the input is greater than zero.

- **Tanh**: Hyperbolic Tangent Function is a smooth, S-shaped symmetric curve that outputs values between -1 and 1.

$$f_{TANH}(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

  While Sigmoid and Tanh suffers from the vanishing gradient problem, RELU does not suffer from the vanishing gradient problem. The computation cost associated with RELU is also minimal when compared to the same of Sigmoid and Tanh.

Vanishing Gradient Problem : Consider a simple neural network with three layers: an input layer, one hidden layer, and an output layer. Let the activation function for the hidden layer be the sigmoid function:

$$f(x) = \frac{1}{1+e^{-x}}.$$

Forward Pass: For a single input x, the output at the hidden layer is:

$$h = f(w_1 x + b_1),$$

and the final output is:

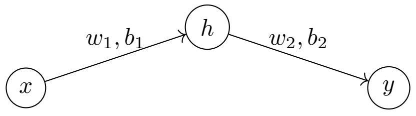
$$y = f(w_2 h + b_2).$$

Backward Pass: Using the chain rule, the gradient of the loss L with respect to the weights w_1 and w_2 involves derivatives of the sigmoid function:

$$\frac{df(x)}{dx} = f(x)(1 - f(x)).$$

When x is large or small, $f(x)$ approaches 0 or 1, making $f(x)(1 - f(x))$ very small. This causes gradients to vanish during backpropagation, slowing down or halting learning.

Illustration:Vanishing Gradient Problem in Sigmoid



In this network, the gradients of w_1 and w_2 shrink exponentially as they propagate backward, making learning inefficient. The ReLU (Rectified Linear Unit) activation function is defined as

$$f(x) = \max(0, x).$$

can solve the vanishing gradient problem as the derivative of the RELU function:

$$\frac{df(x)}{dx} = 1$$

$\forall x > 0$

Replacing sigmoid with ReLU in the hidden layer, the forward pass becomes:

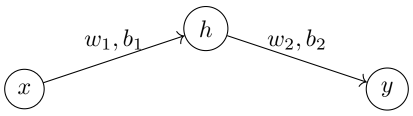
$$h = \max(0, w_1 x + b_1),$$

and the final output is:

$$y = \max(0, w_2 h + b_2).$$

The backward pass involves gradients that remain significant for $x > 0$.

Illustration: How RELU solves Vanishing Gradient Problem



- For large positive inputs ($x \gg 0$):

$$f(x) = x \quad \text{and} \quad f'(x) = 1,$$

  ensuring a constant gradient.

- For large negative inputs ($x \ll 0$):

$$f(x) = 0 \quad \text{and} \quad f'(x) = 0,$$

  which prevents the neuron from updating but does not interfere with other neurons' gradients.

- ReLU keeps gradients non-zero for active neurons (i.e., $x > 0$), preventing exponential decay of gradients in deep networks.

How Neural Networks Work

- **Forward Propagation**: The input data is passed through the network layer by layer. Each neuron processes the data, applies weights, adds bias, and passes the result through an activation function. The final output is computed in the output layer.

- **Loss Function**: The loss function computes the error between the predicted output and the actual output. The most common loss functions are Mean Squared Error (for regression) and Cross-Entropy Loss (for classification).

- **Backpropagation**: Once the error is calculated, the network uses **backpropagation** to adjust the weights and biases by propagating the error backward from the output layer to the input layer. The weights are updated using optimization algorithms like **Gradient Descent** to minimize the error.

Types of Neural Networks

- **Feedforward Neural Networks (FNN)**: The simplest type of neural network where data moves in one direction from input to output without looping.

- **Convolutional Neural Networks (CNN)**: Specialized for processing structured grid data like images. CNNs use convolutional layers to automatically detect features like edges, textures, and patterns.

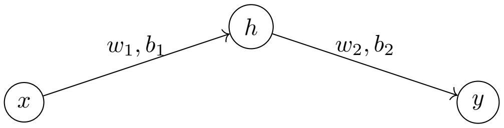
- **Recurrent Neural Networks (RNN)**: Designed for sequential data, such as time series or text. RNNs have connections that loop back, allowing the network to maintain memory of past inputs.
- **Generative Adversarial Networks (GANs)**: Consist of two networks: a generator that creates data and a discriminator that tries to distinguish between real and generated data. They are often used in image generation.
- **Deep Neural Networks (DNN)**: Networks with multiple hidden layers. DNNs are capable of learning complex patterns and are commonly used in deep learning tasks.

Training a Neural Network

- **Data Preparation**: The data must be normalized and preprocessed before feeding it into the neural network.
- **Forward Propagation**: The network makes predictions.
- **Loss Calculation**: The loss function computes the error.
- **Backpropagation**: The network updates the weights using the gradients of the loss with respect to the weights.
- **Optimization**: An optimization algorithm (e.g., Gradient Descent) adjusts the weights to minimize the loss.

Back Propogation: We will illustrate the concept of backpropagation in a simple neural network with:

- One input node
- One hidden layer with a single neuron
- One output node
- Sigmoid activation function



Forward Pass:

$$h = \sigma(w_1 x + b_1), \quad \text{where } \sigma(z) = \frac{1}{1+e^{-z}}$$
$$y = \sigma(w_2 h + b_2)$$

Loss Function The loss function is typically the Mean Squared Error (MSE):

$$L = \frac{1}{2}(\hat{y} - y)^2,$$

where $\hat{y}$ is the target value and y is the predicted output.

Backpropagation Steps

- Step 1: Compute Gradients at the Output Layer

$$\begin{aligned}\delta_{\text{output}} &= \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial(w_2 h + b_2)} \\ &= (y - \hat{y}) \cdot y \cdot (1 - y)\end{aligned}$$

- Step 2: Compute Gradients at the Hidden Layer

$$\begin{aligned}\delta_{\text{hidden}} &= \frac{\partial L}{\partial h} \cdot \frac{\partial h}{\partial(w_1 x + b_1)} \\ &= \delta_{\text{output}} \cdot w_2 \cdot h \cdot (1 - h)\end{aligned}$$

- Step 3: Update Weights and Biases Using gradient descent, update the weights and biases:

$$\begin{aligned}w_2 &\leftarrow w_2 - \eta \cdot \delta_{\text{output}} \cdot h \\ b_2 &\leftarrow b_2 - \eta \cdot \delta_{\text{output}} \\ w_1 &\leftarrow w_1 - \eta \cdot \delta_{\text{hidden}} \cdot x \\ b_1 &\leftarrow b_1 - \eta \cdot \delta_{\text{hidden}},\end{aligned}$$

  where η is the learning rate.

The following steps are performed repeatedly in Back Propogation.

- Compute the error at the output node.
- Propagate the error backward to the hidden layer.
- Update weights and biases using the computed gradients.
- Repeat until convergence.

Python Code for a Basic Neural Network (Using Keras and TensorFlow)

Here's an example of building a simple neural network for classification using **Keras** (a high-level API for TensorFlow):

Listing 2.1: Simple Neural Network in Keras

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import Adam

# Sample dataset (e.g., Iris dataset)
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

# Load the dataset
data = load_iris()
X = data.data
y = data.target

# Encode target labels
encoder = LabelEncoder()
y = encoder.fit_transform(y)

# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.2, random_state=42)

# Define the Neural Network model
model = Sequential([
    Dense(64, input_dim=4, activation='relu'),   # Input
        layer with 4 features
    Dense(64, activation='relu'),                # Hidden
        layer with 64 neurons
    Dense(3, activation='softmax')               # Output
        layer for 3 classes (softmax for classification)
])

# Compile the model
model.compile(optimizer=Adam(), loss='
    sparse_categorical_crossentropy', metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=100, batch_size=10,
    validation_split=0.2)

# Evaluate the model
accuracy = model.evaluate(X_test, y_test)
print(f"Test accuracy: {accuracy[1]:.2f}")
```

Explanation of the Code

1. **Loading the Data**: - The Iris dataset is loaded from `sklearn.datasets`. The target variable `y` is encoded into numerical labels using `LabelEncoder`.

2. **Building the Model**: - A `Sequential` model is used where layers are added sequentially. - The model consists of: - **Input Layer**: With 4 input features (from the Iris dataset). - **Hidden Layers**: Two hidden layers with 64 neurons and ReLU activation. - **Output Layer**: 3 neurons for 3 classes, using `softmax` for multi-class classification.

3. **Compiling the Model**: - The optimizer used is `Adam`, a popular variant of gradient descent. - The loss function is `sparse categorical crossentropy` because the target labels are integers (not one-hot encoded).

4. **Training the Model**: - The model is trained for 100 epochs, with a batch size of 10 and validation split of 0.2.

5. **Evaluating the Model**: - The test set accuracy is computed by evaluating the model against thr test dataset.

2.3 Clustering: Concepts, Mathematics, and Example

Clustering is a technique used in machine learning and statistics to group a set of objects based on their similarity. It is an *unsupervised learning* method since it doesn't use labels for the data points. The objective of clustering is to ensure that objects within the same group (or cluster) are more similar to each other than to objects in other groups.

2.3.1 Key Concepts of Clustering

- **Centroid**: Represents the center of a cluster.
- **Distance Measures**: Determines the similarity or dissimilarity between data points.
  - Euclidean distance: $d(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$
  - Manhattan distance: $d(x, y) = \sum_{i=1}^{n}|x_i - y_i|$
- **Objective**: Minimize intra-cluster variance while maximizing inter-cluster separation.
- **Applications**:
  - Customer segmentation
  - Image compression
  - Anomaly detection

2.3.2 Distance Metrics in Clustering

Clustering heavily depends on distance metrics to measure similarity or dissimilarity between data points. Choosing the appropriate metric can influence the shapes, sizes, and boundaries of clusters. Below are commonly used distance metrics along with their mathematical definitions and key concepts.

Euclidean Distance

The Euclidean distance is the straight-line distance between two points in space.

$$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Key Properties:

- Sensitive to the scale and magnitude of data.
- Works best for compact and circular (or hyperspherical) clusters.

Example: For $x = (2, 3)$ and $y = (5, 7)$:

$$d(x, y) = \sqrt{(5-2)^2 + (7-3)^2} = \sqrt{9+16} = \sqrt{25} = 5$$

Manhattan Distance (Taxicab Distance)

Manhattan distance measures the absolute distance between two points along each dimension.

$$d(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

Key Properties:

- Captures orthogonal movement rather than straight-line distance.
- Common in grid-like structures (e.g., city layouts or robotics).

Example: For $x = (2, 3)$ and $y = (5, 7)$:

$$d(x, y) = |5-2| + |7-3| = 3 + 4 = 7$$

Minkowski Distance

A generalization of both Euclidean and Manhattan distances, controlled by a parameter p.

$$d(x, y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p}$$

Key Properties:

- $p = 1$: Manhattan distance.
- $p = 2$: Euclidean distance.
- Provides flexibility to adjust the measure based on the value of p.

Cosine Similarity

Cosine similarity measures the cosine of the angle between two vectors, commonly used for high-dimensional data.

$$\text{similarity}(x, y) = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i^2}\sqrt{\sum_{i=1}^{n} y_i^2}}$$

Cosine distance is defined as:

$$\text{distance}(x, y) = 1 - \text{similarity}(x, y)$$

Key Properties:

- Measures directional similarity while ignoring magnitude.
- Widely used in text clustering and vectorized datasets.

Example: For $x = (1, 2, 3)$ and $y = (4, 5, 6)$:

$$\text{similarity} = \frac{1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6}{\sqrt{1^2 + 2^2 + 3^2} \cdot \sqrt{4^2 + 5^2 + 6^2}} = \frac{32}{\sqrt{14} \cdot \sqrt{77}}$$

Mahalanobis Distance

Mahalanobis distance measures the distance between a point and a distribution, accounting for feature correlations.

$$d(x, y) = \sqrt{(x - y)^T S^{-1} (x - y)}$$

Key Properties:

- Accounts for variance and correlation in the dataset.
- Especially useful for high-dimensional data.

Hamming Distance

Hamming distance counts the number of differing components between two vectors, often used for categorical or binary data.

$$d(x, y) = \sum_{i=1}^{n} \delta(x_i, y_i)$$

Where $\delta(x_i, y_i) = 1$ if $x_i \neq y_i$, otherwise 0.

Key Properties:

Distance	**Use Cases**	**Key Features**
Euclidean	Compact and circular clusters	Sensitive to magnitude
Manhattan	Grid-like structures	Orthogonal movement
Minkowski	Generalized distance	Parameterized flexibility (p)
CosineSimilarity	Text/document clustering	Directional similarity
Mahalanobis	High-dimensional correlated data	Accounts for variance and correlation
Hamming	Binary or categorical data	Count of mismatches

Table 2.2: Comparison of Common Distance Metrics in Clustering.

- Used in clustering strings, binary data, or text similarity.
- Suitable for comparing categorical data.

Example: For $x = (0, 1, 1, 0)$ and $y = (1, 1, 0, 0)$:

$$d(x, y) = 1 + 0 + 1 + 0 = 2$$

Comparison Table of Distance Metrics

Key Notes:

- The choice of distance metric greatly influences clustering results.
- Normalization or scaling of data may be necessary to ensure fair comparisons, especially for Euclidean and Manhattan distances.

2.3.3 Mathematics Behind Clustering

The most popular clustering algorithm is **k-means**, and its goal is to partition n observations into k clusters. Each cluster is represented by a centroid, which is the mean of the data points within the cluster.

Steps of k-Means Clustering

1. **Initialize**: Choose k centroids randomly.
2. **Assignment**: Assign each data point to the nearest centroid:
$$C_i = \{x_p : \|x_p - \mu_i\|^2 \leq \|x_p - \mu_j\|^2, \forall j = 1, 2, \ldots, k\}$$
where μ_i is the centroid of cluster i, and $\|\cdot\|^2$ represents the squared Euclidean distance.
3. **Update**: Compute new centroids by taking the mean of points in each cluster:
$$\mu_i = \frac{1}{|C_i|} \sum_{x_p \in C_i} x_p$$
4. **Iterate**: Repeat steps 2 and 3 until the centroids no longer change significantly or the maximum number of iterations is reached.

Objective Function

The algorithm minimizes the *within-cluster sum of squares (WCSS)*, defined as:

$$\text{WCSS} = \sum_{i=1}^{k} \sum_{x_p \in C_i} \|x_p - \mu_i\|^2$$

2.3.4 Illustration with a Dataset of Size 12

Dataset (2D points for simplicity):

$(2,3),(3,4),(5,6),(8,8),(9,10),(10,12),(12,3),(11,4),(15,10),(16,11),(18,9),(20,12)$

Let us cluster the dataset into $k = 2$ clusters.

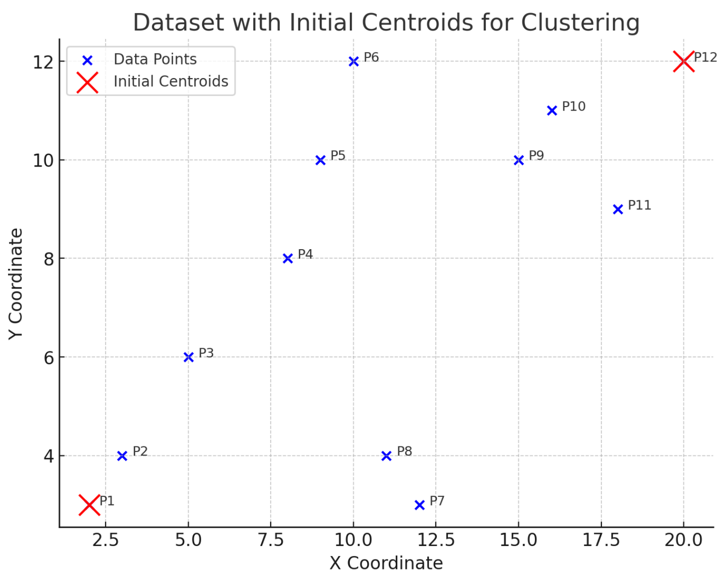
1. **Step 1: Initialize Centroids** Randomly pick two points as initial centroids: $(2,3)$, $(20,12)$
2. **Step 2: Assign Points to Clusters** Calculate distances and assign each point to the closest centroid:
   - *Cluster 1 (Closer to* $(2,3)$*)*: $(2,3),(3,4),(5,6),(8,8)$
   - *Cluster 2 (Closer to* $(20,12)$*)*: $(9,10),(10,12),(12,3),(11,4),(15,10),(16,11),$
3. **Step 3: Update Centroids** Compute the new centroids for each cluster:

   $$\text{Cluster 1: Mean} = (4.5, 5.25), \quad \text{Cluster 2: Mean} = (14.37, 8.87)$$

4. **Step 4: Repeat Steps 2 and 3** Reassign points to the nearest centroid and recalculate centroids. This process continues until convergence.

2.3.5 Visualization of Dataset and Initial Centroids

The dataset and initial centroids can be visualized in a scatter plot as follows:



Here, the blue dots represent the data points, and the red crosses represent the initial centroids.

After running the k-means algorithm for the given dataset, the algorithm converged with the following results:

- **Final Centroids:**

$$\text{Cluster 1 Centroid} : (7.14, 5.43), \quad \text{Cluster 2 Centroid} : (15.8, 10.8)$$

- **Cluster Assignments:**
  - *Cluster 1*: $(2,3), (3,4), (5,6), (8,8), (9,10), (12,3), (11,4)$
  - *Cluster 2*: $(10,12), (15,10), (16,11), (18,9), (20,12)$

The following figure shows the step-by-step iteration process during the k-means clustering:
Each subplot in Figure 2.5 represents:

- **Iteration 1**: Random initialization of centroids and initial cluster assignment.
- **Subsequent Iterations**: Reassignment of data points to clusters and updated centroid positions.
- **Final Iteration**: Converged state, where centroids remain unchanged.

The k-means algorithm proceeded through multiple iterations by minimizing the within-cluster sum of squares (WCSS). At each step:

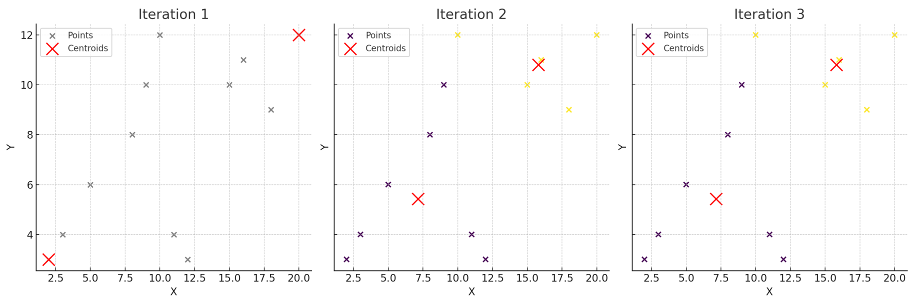


Figure 2.5: Visualization of k-means clustering iterations.

1. Points were reassigned to their nearest centroid.
2. Centroids were updated as the mean of the points assigned to their respective clusters.
3. The process repeated until centroids stabilized, signifying convergence.

2.3.6 Assumptions of K-Means Clustering

K-means clustering is a popular unsupervised learning algorithm for partitioning a dataset into distinct clusters. It is based on the following assumptions:

1. **Clusters Are Spherical or Convex**
   K-means assumes that clusters are approximately spherical (in Euclidean space). The algorithm works best when the clusters can be separated by circular (or convex) boundaries.
2. **Points in a Cluster Are Close to the Centroid**
   Data points within a cluster are expected to be close to their respective cluster center (centroid), minimizing the variance within each cluster.
3. **Clusters Have Equal Variance**
   The algorithm assumes that the clusters have roughly the same variance. Large differences in variance across clusters can lead to poor performance.
4. **Cluster Centroids Represent Cluster Characteristics**
   Each cluster is represented by its centroid, which is assumed to accurately reflect the central tendency of the cluster.
5. **Data is Numeric**
   K-means works with numerical data because it relies on measures like Euclidean distance to update centroids and assign cluster memberships.
6. **Clusters Are Independent and Non-Overlapping**
   K-means assumes that each point belongs to exactly one cluster, so clusters do not overlap.

7. **Number of Clusters (K) Is Predefined**
   The number of clusters (K) must be specified before running the algorithm, meaning the user assumes some prior knowledge of the data.

Limitations of These Assumptions

While these assumptions simplify the algorithm, they may not always hold true in real-world data, leading to potential challenges:

- K-means struggles with non-spherical or overlapping clusters.
- It is sensitive to outliers and noise.
- It can yield poor results when the clusters are not of equal sizes or variances.

For such cases, alternative clustering algorithms (like DBSCAN or Gaussian Mixture Models) may be more appropriate.

2.4 Market Basket Analysis

Let $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$ be a set of items, and $\mathcal{T} = \{t_1, t_2, \ldots, t_m\}$ be a set of transactions, where each $t_j \subseteq \mathcal{I}$.

- **Itemset:** A non-empty subset of $\mathcal{I}$.
- **Support:** The support of an itemset X is the proportion of transactions in $\mathcal{T}$ that contain X:
$$\text{Support}(X) = \frac{|\{t \in \mathcal{T} \mid X \subseteq t\}|}{|\mathcal{T}|}$$
- **Association Rule:** An implication of the form $X \Rightarrow Y$ where $X, Y \subseteq \mathcal{I}$ and $X \cap Y = \emptyset$.
- **Confidence:** The confidence of the rule $X \Rightarrow Y$ is the probability of finding Y in transactions that contain X:
$$\text{Confidence}(X \Rightarrow Y) = \frac{\text{Support}(X \cup Y)}{\text{Support}(X)}$$
- **Lift:** The lift of the rule indicates the strength of the association, defined as:
$$\text{Lift}(X \Rightarrow Y) = \frac{\text{Confidence}(X \Rightarrow Y)}{\text{Support}(Y)}$$

Apriori is a classic algorithm used to identify frequent itemsets. It works as follows:

1. Identify all itemsets with support above a user-defined threshold (minimum support).
2. Generate candidate rules from these frequent itemsets.
3. Retain rules with confidence above a minimum threshold.

Assume the following 5 transactions:

- t_1: {milk, bread, butter}
- t_2: {bread, butter}
- t_3: {milk, bread}
- t_4: {milk, bread, butter, jam}
- t_5: {bread, jam}

From this, we can compute support, confidence, and lift for rules such as:

$$\{milk\} \Rightarrow \{bread\}$$

Let:

- Support({milk}) = 3/5
- Support({milk, bread}) = 3/5
- Support({bread}) = 5/5

Then:

$$\text{Confidence}(\{milk\} \Rightarrow \{bread\}) = \frac{3/5}{3/5} = 1$$

$$\text{Lift} = \frac{1}{1} = 1$$

Chapter 3

Python

3.1 Introduction

Python is a **high-level**, interpreted, and general-purpose programming language. It is known for its simplicity, readability, and versatility, making it an ideal choice for both beginners and experienced developers.

A high-level language (HLL) is a type of programming language that allows developers to write code using human-readable syntax, making it easier to understand and develop programs. High-level languages provide a layer of abstraction that hides the complexities of hardware, such as memory management and processor instructions. Code is easier to read, write and maintain. Code written for one type of hardware can be used on another or portable.

Examples are Python, Java, C++ and JavaScript. High level language implementation for printing "Hello, World!".

```
print("Hello, World!")
```

Low-level languages are closely related to machine code (binary instructions that the CPU understands), making them more hardware-specific but offering greater control over system resources. These languages operate close to the hardware, with direct control over memory, CPU registers, and other hardware components. Code is harder to read, write, and maintain. Code written for one type of hardware cannot be directly used on another or unportable.

Examples include machine language and assembly language Machine Language is in binary form

```
10110000 01100001
```

High-level Language for 8086 Processor.

```
MOV AX, 5
ADD AX, 3
```

Python is considered an interpreted language because Python code is executed by an interpreter rather than being directly compiled into machine code. The interpreter reads the source code, translates it into an intermediate form, and executes it line by line at runtime. Python code is executed line by line, meaning each instruction is processed sequentially by the interpreter. Eg : Python, JavaScript, Ruby and PHP. Since the interpreter processes code line by line at runtime, interpreted languages are generally slower than compiled languages.

Python internal working is as follows.

- Source Code: You write Python code (.py file).
- Bytecode Compilation: The Python interpreter first converts the source code into bytecode (.pyc file). Bytecode is a lower-level representation that is still platform-independent.
- Execution by Python Virtual Machine (PVM): The Python Virtual Machine (PVM) reads the bytecode and executes it line by line.

A compiled language requires a compiler to translate the entire source code into machine code (or an intermediate form) before execution. The output of this compilation process is a binary executable file that can be run directly by the computer. Eg: C, C++ Since the code is already translated into machine code, it runs faster than interpreted code. The executable generated by a compiler is often platform-specific (e.g., a program compiled on Windows may not run on Linux.

3.2 Coding

3.2.1 Data Types

1. Numeric Types:

   - **int**: Integer numbers (whole numbers). Example:

     ```
     x = 10  # int
     y = -5
     ```

   - **float**: Floating-point numbers (numbers with a decimal point). Example:

     ```
     pi = 3.14  # float
     exp = 2.5e3  # 2500.0
     ```

   - **complex**: Complex numbers with real and imaginary parts. Example:

```
z = 2 + 3j   # complex number
```

2. Sequence Types:

   - **str**: A sequence of characters (string). Example:

     ```
     text = "Hello, World!"   # str
     ```

   - **list**: An ordered, mutable collection of items. Example:

     ```
     fruits = ["apple", "banana", "cherry"]   # list
     fruits[0] = "mango"   # lists are mutable
     ```

   - **tuple**: An ordered, immutable collection of items. Example:

     ```
     coordinates = (1, 2, 3)   # tuple
     # coordinates[0] = 10 will show an error
     ```

3. Set Types:

   - **set**: An unordered collection of unique items. Example:

     ```
     unique_numbers = {1, 2, 3, 3, 4}   # set -> {1,
         2, 3, 4}
     ```

   - **frozenset**: An immutable version of a set. Example:

     ```
     frozen = frozenset([1, 2, 3])
     ```

4. Mapping Type:

   - **dict**: A collection of key-value pairs. Example:

     ```
     person = {"name": "Alice", "age": 25}   # dict
     print(person["name"])   # Output: Alice
     ```

5. Boolean Type:

   - **bool**: Represents two values: `True` and `False`. Example:

     ```
     is_valid = True   # bool
     ```

6. Type Checking:

```
x = 42
print(type(x))  # Output: <class 'int'>
```

7. Python also allows type conversion using built-in functions:

   - `int()` – Converts to integer.
   - `float()` – Converts to float.
   - `str()` – Converts to string.
   - `list()` – Converts to list.
   - `tuple()` – Converts to tuple.
   - `set()` – Converts to set.
   - `dict()` – Converts to dictionary.

```
num = 10.5
int_num = int(num)  # Converts float to int -> 10
str_num = str(num)  # Converts float to string -> '10.5'
```

8. Array - Numpy Arrays are not built in data types.All elements in a NumPy array must be of the same data type (e.g., all integers, all floats). This homogeneity leads to faster processing and optimizations. Numpy Operations are applied element-wise directly to the entire array.

```
# NumPy array
import numpy as np
np_array = np.array([1, 2, 3, 4, 5])   # NumPy array

# element wise operations
result = np_array * 2
print(result)  # Output: [2 4 6 8]

np_2d_array = np.array([[1, 2], [3, 4], [5, 6]])
print(np_2d_array)

# dimension
import numpy as np

arr_2d = np.array([[1, 2, 3], [4, 5, 6]])
print(arr_2d.ndim)
print(arr_2d.shape)

# Output: 2 (indicating that the array is 2-dimensional)
```

```
arr_3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(arr_3d.ndim)
print(arr_3d.shape)

# Output: 3 (indicating that the array is 3-dimensional)


# Reshaping array


import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
print(arr.shape)
reshaped_arr = arr.reshape(3, 3)
print(reshaped_arr)
print(reshaped_arr.shape)

# Output:
# [[1 2 3]
#  [4 5 6]
#  [7 8 9]]

#Accessing Elements in an array

import numpy as np

arr_2d = np.array([[1, 2, 3], [4, 5, 6]])
print(arr_2d[0, 1])  # Access element at row 0, column 1
# Output: 2

# Slicing a 2D array
print(arr_2d[1, :])  # Access all columns of row 1
# Output: [4 5 6]
```

9. List - Python lists can store elements of different data types (heterogeneous). Lists are slower when performing operations on large datasets, especially when working with large numbers or complex mathematical operations.

```
# Python list
py_list = [1, 2, 3, 4, 5]

# Python list operation (using list comprehension)
# Run a loop for doing operations
py_list = [1, 2, 3, 4]
result = [x * 2 for x in py_list]
print(result)  # Output: [2, 4, 6, 8]
```

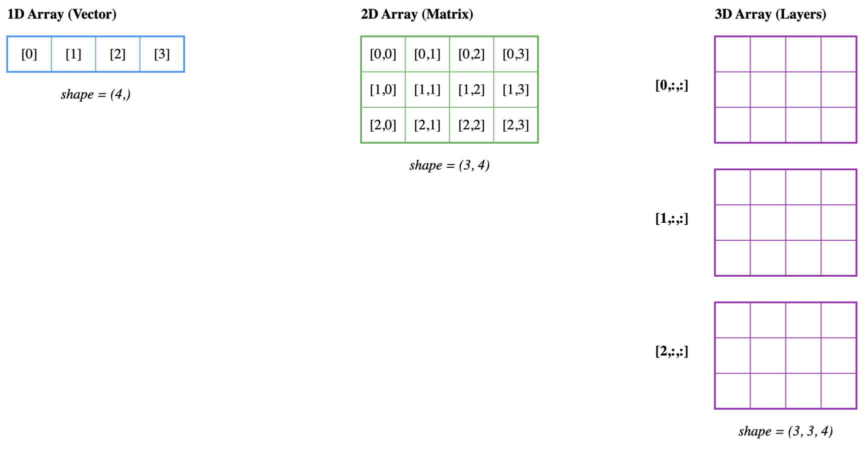


Figure 3.1: N-Dimensional Array

```
# list of lists for multi-dimensional arrays
# List of lists to simulate a 2D array
py_2d_list = [[1, 2], [3, 4], [5, 6]]
print(py_2d_list)
```

Lets compare the performance of arrays and list

```
    import time
import numpy as np

# NumPy performance test
start_time = time.time()
np_array1 = np.array([1] * 1000000)
np_array2 = np.array([1] * 1000000)
result = np_array1 + np_array2
print("NumPy execution time:", time.time() - start_time)

# Python list performance test
start_time = time.time()
py_list1 = [1] * 1000000
py_list2 = [1] * 1000000
result = [x + y for x, y in zip(py_list1, py_list2)]
print("Python list execution time:", time.time() -
    start_time)
```

3.2.2 Operators

1. Arithmetic Operators - Arithmetic operators are used to perform mathematical operations like addition, subtraction, multiplication, etc.

   - `+` : Addition

     ```
     result = 10 + 5  # result = 15
     ```

   - `-` : Subtraction

     ```
     result = 10 - 5  # result = 5
     ```

   - `*` : Multiplication

     ```
     result = 10 * 5  # result = 50
     ```

   - `/` : Division

     ```
     result = 10 / 5  # result = 2.0
     ```

   - `//` : Floor Division (quotient without the remainder)

     ```
     result = 10 // 3  # result = 3
     ```

   - `%` : Modulus (remainder of division)

     ```
     result = 10 % 3  # result = 1
     ```

   - `**` : Exponentiation (power)

     ```
     result = 2 ** 3  # result = 8
     ```

2. Comparison operators are used to compare two values and return a boolean value (`True` or `False`).

   - `==` : Equal to

     ```
     result = (10 == 5)  # result = False
     ```

   - `!=` : Not equal to

     ```
     result = (10 != 5)  # result = True
     ```

   - `<` : Less than

     ```
     result = (10 < 5)  # result = False
     ```

   - `>` : Greater than

```
result = (10 > 5)  # result = True
```

- `<=` : Less than or equal to

```
result = (10 <= 5)  # result = False
```

- `>=` : Greater than or equal to

```
result = (10 >= 5)  # result = True
```

3. Logical operators are used to perform logical operations, typically used in conditions and comparisons.

- `and` : Logical AND

```
result = (True and False)  # result = False
```

- `or` : Logical OR

```
result = (True or False)  # result = True
```

- `not` : Logical NOT

```
result = not True  # result = False
```

4. Assignment Operators Assignment operators are used to assign values to variables.

- `=` : Assigns a value

```
x = 10  # Assign 10 to x
```

- `+=` : Add and assign

```
x += 5  # x = x + 5
```

- `-=` : Subtract and assign

```
x -= 5  # x = x - 5
```

- `*=` : Multiply and assign

```
x *= 5  # x = x * 5
```

- `/=` : Divide and assign

```
x /= 5  # x = x / 5
```

   - `%=` : Modulus and assign

     ```
     x %= 5  # x = x % 5
     ```

   - `**=` : Exponentiation and assign

     ```
     x **= 2  # x = x ** 2
     ```

5. Bitwise operators are used to perform bit-level operations on binary numbers.

   - `&` : Bitwise AND

     ```
     result = 5 & 3  # result = 1
     ```

   - `|` : Bitwise OR

     ```
     result = 5 | 3  # result = 7
     ```

   - `^` : Bitwise XOR

     ```
     result = 5 ^ 3  # result = 6
     ```

   - : Bitwise NOT

     ```
     result = ~5  # result = -6
     ```

   - `<<` : Bitwise left shift

     ```
     result = 5 << 1  # result = 10
     ```

   - `>>` : Bitwise right shift

     ```
     result = 5 >> 1  # result = 2
     ```

6. Membership operators are used to test if a value is present in a sequence (like a list, tuple, or string).

   - `in` : Returns `True` if a value is found in the sequence

     ```
     result = 3 in [1, 2, 3, 4]  # result = True
     ```

   - `not in` : Returns `True` if a value is not found in the sequence

     ```
     result = 5 not in [1, 2, 3, 4]  # result = True
     ```

7. Identity operators are used to check if two objects share the same memory location.

- `is` : Returns `True` if two variables point to the same object

```
result = (x is y)  # True if x and y are the
    same object
```

- `is not` : Returns `True` if two variables point to different objects

```
result = (x is not y)  # True if x and y are
    different objects
```

3.2.3 Control Structures

Control structures in Python are constructs that allow you to dictate the flow of execution within your program. They enable decision-making, looping, and branching to control how and when specific blocks of code are executed.

1. Conditional Statements such as if, elif, else are used to execute code based on certain conditions.

```
x = 10
if x > 5:
    print("x is greater than 5")
elif x == 5:
    print("x is equal to 5")
else:
    print("x is less than 5")
```

2. Looping structures are used to repeat a block of code multiple times.

   - For loop iterates over a sequence (e.g., list, tuple, string).

```
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    print(num)
```

   - while Loop repeats as long as a condition is True.

```
count = 0
  while count < 5:
    print(count)
    count += 1
```

3. ”Control Flow Modifiers” are used to alter the flow of loops or conditionals.

   - Break exits the loop prematurely

```
for i in range(10):
  if i == 5:
    break
  print(i)
```

- "Continue" skips the current iteration and moves to the next.

```
for i in range(10):
  if i % 2 == 0:
    continue
  print(i)
```

- Functions provide another level of control by encapsulating logic. Return statement exits a function and optionally returns a value

```
# function 1
def greet(name):
  if name:
    return f"Hello, {name}!"
  return "Hello, World!"

def square(x):
  return x * x
```

- Comprehensions are shortened syntax for looping and conditionals to create collection.
  - List Comprehension
    ```
    squares = [x**2 for x in range(10) if x % 2
    ```
  - Dictionary Comprehension
    ```
    cubes = {x: x**3 for x in range(5)}
    ```

3.2.4 Data Structures

Python offers a variety of data structures that help organize and manage data efficiently.

1. Lists - Ordered, mutable, and allows duplicates.

   ```
   my_list = [1, 2, 3, 4, 5]
   my_list.append(6)
   print(my_list)  # Output: [1, 2, 3, 4, 5, 6]
   ```

2. Tuples - Ordered, immutable, and allows duplicates.

```
my_tuple = (1, 2, 3)
print(my_tuple[1])  # Output: 2
```

3. Dictionary - Key-value pairs, mutable, unordered (ordered from Python 3.7 onwards).

```
my_dict = {'name': 'Alice', 'age': 25}
my_dict['city'] = 'New York'
print(my_dict)  # Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

4. Sets - Unordered, mutable, no duplicates.

```
my_set = {1, 2, 3, 3}
print(my_set)  # Output: {1, 2, 3}
```

5. Frozen Sets (frozenset) - Immutable version of sets.

```
my_frozenset = frozenset([1, 2, 3, 3])
print(my_frozenset)  # Output: frozenset({1, 2, 3})
```

6. Stack - Implemented using list.

```
stack = []
stack.append(1)
stack.append(2)
print(stack.pop())  # Output: 2
```

7. Queues - Implemented using collections.deque

```
from collections import deque
queue = deque([1, 2, 3])
queue.append(4)
print(queue.popleft())  # Output: 1
```

3.2.5 Modules and Packages in Python

In Python, **modules** and **packages** are essential components for organizing code into manageable and reusable structures.

Modules

A *module* is a single Python file containing definitions and statements. Modules allow you to logically organize your code and reuse it across different programs.

Creating a Module To create a module, save the Python code in a file with a `.py` extension. For example:

```
# File: mymodule.py
def greet(name):
    return f"Hello, {name}!"
```

Using a Module Import the module into another Python file or interactive session:

```
import mymodule
print(mymodule.greet("Alice"))  # Output: Hello, Alice!
```

Packages

A *package* is a collection of related modules organized in a directory hierarchy. A package must include an `__init__.py` file in its directory to indicate that it is a package. This file can be empty or contain initialization code.

Creating a Package The structure of a package might look like this:

```
mypackage/
    __init__.py
    module1.py
    module2.py
```

Using a Package Import modules from the package as follows:

```
from mypackage import module1
print(module1.some_function())
```

Feature	Module
Definition	A file containing Python code (variables, functions, classes, etc.)
Purpose	Reuse across multiple scripts or projects.
Scope	Can contain functions, classes, and variables.
Usage	Imported into other scripts using the `import` keyword.
File Extension	Saved in a `.py` file.

Table 3.1: Module

Built-in and External Modules

Python includes many built-in modules, such as `math`, `os`, and `sys`. Additionally, external packages can be installed via `pip`, the Python package manager:

```
pip install numpy
```

Referencing and structuring code using modules and packages makes large-scale Python projects more maintainable and reusable.

Feature	**Function**
Definition	A block of reusable code that performs a specific task.
Purpose	Encapsulates logic to perform a task, reducing repetition in code.
Scope	A single unit of code that performs a task.
Usage	Defined and called directly within a program.
File Extension	Defined inside a module or script.

Table 3.2: Function

3.2.6 File : Read and Write

The `pandas` library in Python provides a powerful and easy-to-use method for reading and processing CSV files. The `pandas.read_csv()` function allows for reading CSV files into a DataFrame, which is a 2D table-like structure with labeled axes (rows and columns).

1. Reading a file - To read a CSV file, use the `read_csv()` function from the `pandas` library. This function returns a DataFrame, which provides various methods for data manipulation and analysis.

   Example:

   ```
   import pandas as pd

   # Read CSV into DataFrame
   df = pd.read_csv('data.csv')

   # Display the DataFrame
   print(df)
   ```

   This will read the `data.csv` file and print the content as a DataFrame, which looks like a table.

2. Reading a file with a delimiter - If the CSV file uses a delimiter other than a comma (e.g., semicolons), you can specify the delimiter using the `delimiter` parameter.

   Example:

   ```
   import pandas as pd

   # Read CSV with semicolon delimiter
   df = pd.read_csv('data_semicolon.csv', delimiter=';')

   # Display the DataFrame
   print(df)
   ```

3. Handling Missing Data - Pandas handles missing data automatically. However, you can also specify how to handle missing values using the `na_values` parameter.

   Example:

```
import pandas as pd

# Read CSV and treat 'NA' as a missing value
df = pd.read_csv('data_with_missing.csv', na_values='NA')

# Display the DataFrame
print(df)
```

4. Reading a Specific Column - To read specific columns from a CSV file, pass the column names as a list to the `usecols` parameter.

   Example:

```
import pandas as pd

# Read only specific columns from the CSV
df = pd.read_csv('data.csv', usecols=['Name', 'Age'])

# Display the DataFrame
print(df)
```

5. Reading CSV with a Header Row - If the CSV file contains a header row (i.e., column names are in the first row), `pandas.read_csv()` automatically treats the first row as the header. However, you can manually specify the row number containing the header using the `header` parameter.

   Example:

```
import pandas as pd

# Read CSV with specific header row (first row)
df = pd.read_csv('data_with_header.csv', header=0)

# Display the DataFrame
print(df)
```

3.3 Data Analysis

3.3.1 Plotting

- Basic Line Plot with Matplotlib

```
        import matplotlib.pyplot as plt

# Data
x = [1, 2, 3, 4, 5]
y = [2, 3, 5, 7, 11]

# Create a line plot
plt.plot(x, y, marker='o', linestyle='-', color='b')
plt.title('Line Plot Example')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.grid(True)

# Show plot
plt.show()
```

- Scatter Plot with Matplotlib

```
    import matplotlib.pyplot as plt

# Data
x = [1, 2, 3, 4, 5]
y = [2, 3, 5, 7, 11]

# Create a scatter plot
plt.scatter(x, y, color='r')
plt.title('Scatter Plot Example')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')

# Show plot
plt.show()
```

- Histogram with Matplotlib

```
    import matplotlib.pyplot as plt
import numpy as np

# Data
data = np.random.randn(1000)

# Create a histogram
plt.hist(data, bins=30, color='g', edgecolor='black',
    alpha=0.7)
```

```
plt.title('Histogram Example')
plt.xlabel('Value')
plt.ylabel('Frequency')

# Show plot
plt.show()
```

- Boxplot with Seaborn

```
import seaborn as sns
import matplotlib.pyplot as plt

# Data
data = sns.load_dataset('tips')

# Create a box plot
sns.boxplot(x='day', y='total_bill', data=data, palette=
    'coolwarm')
plt.title('Box Plot Example')

# Show plot
plt.show()
```

- Pairplot with Seaborn

```
import seaborn as sns

# Data
data = sns.load_dataset('iris')

# Create a pair plot
sns.pairplot(data, hue='species')
plt.title('Pair Plot Example')

# Show plot
plt.show()
```

- Heatmap with Seaborn

```
    import seaborn as sns
import matplotlib.pyplot as plt

# Data (Correlation matrix of a dataset)
data = sns.load_dataset('iris')
corr = data.corr()

# Create a heatmap
sns.heatmap(corr, annot=True, cmap='coolwarm',
    linewidths=0.5)
```

```
plt.title('Heatmap Example')

# Show plot
plt.show()
```

- 3-D plot with Matplotlib

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np

# Data
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
x, y = np.meshgrid(x, y)
z = np.sin(np.sqrt(x**2 + y**2))

# Create a 3D plot
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(x, y, z, cmap='viridis')

# Show plot
plt.show()
```

3.3.2 Basic Data Frame Operations

- **Creating a DataFrame:**
  - A DataFrame can be created from a dictionary, a list, or other data structures.

Listing 3.1: Creating a DataFrame

```
import pandas as pd

# Create a DataFrame from a dictionary
data = {
    'A': [1, 2, 3],
    'B': [4, 5, 6],
    'C': [7, 8, 9]
}

df = pd.DataFrame(data)
print(df)
```

- **Accessing Data:**

  - You can access columns, rows, or slices of the DataFrame.

Listing 3.2: Accessing Columns and Rows

```python
# Access a single column
print(df['A'])

# Access multiple columns
print(df[['A', 'B']])

# Access a row by index using iloc
print(df.iloc[0])  # First row

# Access a row by index label using loc
print(df.loc[0])  # First row
```

- **Basic Operations:**
  - You can perform arithmetic operations or apply functions on the columns.

Listing 3.3: Basic Operations

```python
# Adding a constant to a column
df['A'] = df['A'] + 10
print(df)

# Adding two columns
df['D'] = df['A'] + df['B']
print(df)
```

- **Filtering Data:**
  - Data can be filtered based on conditions, either using boolean indexing or the `query()` method.

Listing 3.4: Filtering Data

```python
# Filter rows where 'A' is greater than 10
filtered_df = df[df['A'] > 10]
print(filtered_df)

# Using query() method
filtered_df = df.query('A > 10')
print(filtered_df)
```

- **Sorting Data:**
  - You can sort the DataFrame based on one or more columns.

Listing 3.5: Sorting Data

```
# Sort by a single column
sorted_df = df.sort_values(by='A', ascending=False)
print(sorted_df)

# Sort by multiple columns
sorted_df = df.sort_values(by=['A', 'B'], ascending=[
    True, False])
print(sorted_df)
```

- **Handling Missing Data:**
  - Missing data can be detected using `isnull()`, filled with `fillna()`, or dropped with `dropna()`.

Listing 3.6: Handling Missing Data

```
# Check for missing values
print(df.isnull())

# Fill missing values with a constant
df.fillna(0, inplace=True)

# Drop rows with missing values
df.dropna(inplace=True)
```

- **Renaming Columns:**
  - Columns can be renamed using the `rename()` method.

Listing 3.7: Renaming Columns

```
# Rename columns
df.rename(columns={'A': 'Column1', 'B': 'Column2'},
    inplace=True)
print(df)
```

- **Adding and Removing Columns:**
  - Columns can be added or removed from the DataFrame.

Listing 3.8: Adding and Removing Columns

```
# Add a new column
df['D'] = df['A'] * 2
print(df)

# Drop a column
df.drop(columns=['D'], inplace=True)
print(df)
```

- **Aggregating Data:**
  - Data can be grouped using `groupby()` and aggregated using functions like `sum()` or `mean()`.

Listing 3.9: Aggregating Data

```
# Group by a column and sum
grouped = df.groupby('Category').sum()
print(grouped)
```

- **Merging DataFrames:**
  - DataFrames can be merged using a common column using `merge()`.

Listing 3.10: Merging DataFrames

```
# Create another DataFrame
data2 = {
    'Category': ['A', 'B'],
    'Description': ['Desc A', 'Desc B']
}
df2 = pd.DataFrame(data2)

# Merge DataFrames on 'Category'
merged_df = pd.merge(df, df2, on='Category')
print(merged_df)
```

- **Applying Functions to Columns or Rows:**
  - You can apply a custom function across columns or rows using `apply()`.

Listing 3.11: Applying Functions

```
# Apply a function to a column
df['A'] = df['A'].apply(lambda x: x + 5)
print(df)

# Apply a function to each row
df['Sum'] = df.apply(lambda row: row['A'] + row['B'],
    axis=1)
print(df)
```

- **Pivot Tables:**
  - Pivot tables can be created using the `pivot_table()` method.

Listing 3.12: Creating a Pivot Table

```
# Create a pivot table
pivot_df = df.pivot_table(values='Value', index='
    Category', aggfunc='sum')
print(pivot_df)
```

3.3.3 Descriptive Statistics

- **Reading the Data:** The function `pd.read_csv()` is used to read the CSV file and load the data into a `pandas DataFrame`. You need to replace `'your_data.csv'` with the actual file path to your dataset.

- **Displaying the First Few Rows:** The `df.head()` function is used to display the first few rows of the dataset. This allows us to inspect the structure and columns of the data, which helps in understanding how the data is organized.

- **Descriptive Statistics:** The function `df.describe()` generates descriptive statistics for the numeric columns in the dataset. It provides the following key statistics for each numeric column:
  - **count**: The number of non-null values in the column.
  - **mean**: The average value of the column.
  - **std**: The standard deviation of the column.
  - **min**: The minimum value in the column.
  - **25%**, **50%**, **75%**: The 25th, 50th (median), and 75th percentiles.
  - **max**: The maximum value in the column.

- **Including Non-Numeric Columns:** If the dataset contains non-numeric columns and you wish to include descriptive statistics for those as well, you can modify the `describe()` function call to include all data types:

  ```
  descriptive_stats = df.describe(include='all')
  ```

  This will display additional statistics such as frequency counts, unique values, and the most common value (top) for categorical data.

```
import pandas as pd

# Load the dataset (replace 'your_data.csv' with your file
    path)
df = pd.read_csv('your_data.csv')
```

```
# Print the first few rows of the dataset to understand its
    structure
print(df.head())

# Generate descriptive statistics
descriptive_stats = df.describe()

# Print the descriptive statistics
print(descriptive_stats)
```

3.3.4 Correlation Analysis

```
# Import necessary libraries
import pandas as pd
from statsmodels.stats.outliers_influence import
    variance_inflation_factor
import numpy as np

# Define the dataset
data = {
    "X1": [2.1, 3.4, 1.8, 2.9, 3.1, 1.5, 3.6, 2.3, 3.8, 1.7,
         2.5, 3.2, 1.9, 3.7, 2.6],
    "X2": [1.2, 2.5, 0.7, 1.8, 2.2, 0.6, 2.9, 1.5, 2.7, 0.8,
         1.4, 2.0, 0.9, 2.4, 1.6],
    "Y": [3.8, 6.9, 2.9, 5.4, 6.2, 2.5, 7.3, 4.2, 7.5, 3.0,
        4.5, 6.5, 3.2, 7.0, 4.9]
}

# Create a DataFrame
df = pd.DataFrame(data)

# 1. Correlation Analysis
print("Correlation Matrix:")
correlation_matrix = df.corr()
print(correlation_matrix)

# 2. Variance Inflation Factor (VIF)
# Add a constant term for VIF calculation
X = df[["X1", "X2"]]
X["Intercept"] = 1  # Add constant

# Compute VIF for each feature
vif_data = pd.DataFrame()
vif_data["Feature"] = X.columns
vif_data["VIF"] = [variance_inflation_factor(X.values, i)
    for i in range(X.shape[1])]

# Display VIF values
```

```
print("\nVariance Inflation Factor (VIF):")
print(vif_data)
```

- **Import Libraries:**
  - Necessary libraries such as `pandas`, `numpy`, and `statsmodels` are imported.
  - These libraries provide functions for data manipulation, mathematical operations, and VIF computation.
- **Dataset Definition:**
  - The dataset consists of three variables: two independent variables (X_1, X_2) and one dependent variable (Y).
  - It is stored as a dictionary and converted to a Pandas DataFrame for analysis.
- **Correlation Analysis:**
  - The correlation matrix is computed using the `df.corr()` method.
  - This calculates the Pearson correlation coefficients for all pairs of variables.
  - Values range from:
    - -1: Perfect negative correlation.
    - 0: No correlation.
    - 1: Perfect positive correlation.
- **Variance Inflation Factor (VIF):**
  - VIF quantifies multicollinearity among independent variables.
  - It is calculated as:

    $$\text{VIF}_j = \frac{1}{1 - R_j^2}$$

    Where R_j^2 is the coefficient of determination when X_j is regressed on the other independent variables.
  - High VIF values (> 5 or > 10) indicate strong multicollinearity, which can destabilize regression coefficients.
- **Adding a Constant Term:**
  - A constant term (`Intercept`) is added to the DataFrame using `X["Intercept"] = 1`.
  - This ensures accurate VIF calculation, as the intercept accounts for the base level of the regression equation.

- **Compute VIF:**
  - VIF values are computed for each variable using the `variance_inflation_fac` function from `statsmodels`.
  - The function iterates over all features and calculates their respective VIF values.
- **Output:**
  - The correlation matrix is displayed, showing the linear relationships between variables.
  - The VIF values for each feature are presented in a tabular format, identifying any multicollinearity issues.
- **Insights:**
  - The correlation matrix highlights strong correlations among variables.
  - High VIF values indicate multicollinearity, requiring further action like removing variables or using dimensionality reduction techniques (e.g., PCA).

3.3.5 Linear Regression using Python

```
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Define the dataset
data = {
    "X1": [2.1, 3.4, 1.8, 2.9, 3.1, 1.5, 3.6, 2.3, 3.8, 1.7,
        2.5, 3.2, 1.9, 3.7, 2.6],
    "X2": [1.2, 2.5, 0.7, 1.8, 2.2, 0.6, 2.9, 1.5, 2.7, 0.8,
        1.4, 2.0, 0.9, 2.4, 1.6],
    "Y": [3.8, 6.9, 2.9, 5.4, 6.2, 2.5, 7.3, 4.2, 7.5, 3.0,
        4.5, 6.5, 3.2, 7.0, 4.9]
}

# Create a DataFrame
df = pd.DataFrame(data)

# Define independent and dependent variables
X = df[["X1", "X2"]]
y = df["Y"]

# Initialize and fit the model
```

```
model = LinearRegression()
model.fit(X, y)

# Make predictions
y_pred = model.predict(X)

# Print model coefficients
print("Intercept:", model.intercept_)
print("Coefficients:", model.coef_)

# Evaluate the model
mse = mean_squared_error(y, y_pred)
r2 = r2_score(y, y_pred)

print("Mean Squared Error:", mse)
print("R-squared:", r2)

# Optional: Predict new values
new_data = np.array([[2.5, 1.8], [3.0, 2.0]])
predictions = model.predict(new_data)
print("Predictions for new data:", predictions)
```

- **Dataset:**
  - The dataset contains 15 samples with two independent variables (X_1, X_2) and one dependent variable (Y).
  - It is stored in a Python dictionary and converted to a Pandas DataFrame for easy handling and analysis.
- **Model Initialization:**
  - A linear regression model is initialized using the `LinearRegression()` class from `scikit-learn`.
  - The model is then trained using the `fit()` method, which takes X (independent variables) and y (dependent variable) as inputs.
- **Model Coefficients:**
  - The intercept (β_0) is accessed using `model.intercept_`.
  - The coefficients (β_1, β_2) for the independent variables are accessed using `model.coef_`.
  - The regression equation is of the form:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2$$

- **Model Evaluation:**
  - The performance of the model is assessed using:

  * **Mean Squared Error (MSE):** Measures the average squared difference between the actual (y) and predicted (y_{pred}) values:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - y_{\text{pred},i})^2$$

  * **R-squared (R^2):** Represents the proportion of variability in dependent variable (Y) explained by the model comprising of X_1 and X_2. It is calculated as:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - y_{\text{pred},i})^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

$R^2 = 1 - \frac{SSE}{SST} = \frac{SSR}{SST}$

- Values closer to $R^2 = 1$ indicate a good fit.

- **Predictions:**
  - Predictions for the dependent variable (Y) can be made using `model.predict`
  - The model can also predict Y for new data points by providing the values of X_1 and X_2.
- **Output:**
  - The intercept, coefficients, MSE, and R^2 values are printed as part of the model evaluation.
  - For new data, the predicted Y values are also displayed.

3.3.6 Decision Tree

```
# Import necessary libraries
import numpy as np
import pandas as pd
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score,
    classification_report

# Step 1: Generate a synthetic dataset
X, y = make_classification(n_samples=1000, n_features=20,
    n_informative=2, n_classes=2, random_state=42)

# Step 2: Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
    test_size=0.3, random_state=42)
```

```
# Step 3: Initialize the Decision Tree Classifier
clf = DecisionTreeClassifier(random_state=42)

# Step 4: Train the model on the training set
clf.fit(X_train, y_train)

# Step 5: Make predictions on the test set
y_pred = clf.predict(X_test)

# Step 6: Evaluate the model performance
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred)

# Print the results
print(f"Accuracy: {accuracy}")
print(f"Classification Report:\n{report}")
```

- **Step 1: Generate a Synthetic Dataset:** We use the `make_classification()` function from the `scikit-learn` library to generate a synthetic classification dataset. This function creates random datasets with specified numbers of features and classes. In our case, we generate a dataset with 1000 samples, 20 features, and 2 classes. The `random_state=42` parameter ensures that the dataset is reproducible across runs.

- **Step 2: Split the Dataset:** The dataset is split into training and testing sets using the `train_test_split()` function from `scikit-learn`. The data is divided into 70% for training and 30% for testing. This allows us to evaluate the model's performance on unseen data.

- **Step 3: Initialize the Decision Tree Classifier:** We initialize the decision tree classifier using the `DecisionTreeClassifier()` function from `scikit-learn`. The `random_state=42` parameter is used to ensure that the model's results are reproducible.

- **Step 4: Train the Model:** The decision tree model is trained using the `fit()` method, which takes the training data (`X_train`, `y_train`) as input. This step allows the model to learn patterns from the training data.

- **Step 5: Make Predictions:** After the model is trained, we use the `predict()` method to make predictions on the test set (`X_test`). This gives us the predicted labels for the test data.

- **Step 6: Evaluate the Model:** We evaluate the model's performance by calculating the accuracy using `accuracy_score()` and generating a classification report using `classification_report()`. The classification report provides additional metrics like precision, recall, and F1-score for each class, helping us assess how well the model performs in distinguishing between the two classes.

A confusion matrix is a table used to evaluate the performance of a classification model. In binary classification, the confusion matrix looks like the following:

	Predicted Positive	**Predicted Negative**
Actual Positive	TP	FN
Actual Negative	FP	TN

The performance metrics of a binary classifier is defined below.

- **Accuracy:** The proportion of correct predictions (both true positives and true negatives) out of the total number of predictions.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

- **Precision or Positive Prediction Value** The proportion of true positive predictions out of all positive predictions (i.e., how many selected items are relevant).

$$\text{Precision} = \frac{TP}{TP + FP}$$

- **Recall (Sensitivity or True Positive Rate):** The proportion of true positive predictions out of all actual positives (i.e., how many relevant items are selected).

$$\text{Recall} = \frac{TP}{TP + FN}$$

- **F1-Score:** The harmonic mean of precision and recall, providing a balance between the two. It is particularly useful when there is an uneven class distribution.

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

- **Specificity (True Negative Rate):** The proportion of true negative predictions out of all actual negatives (i.e., how many non-relevant items are correctly identified as non-relevant).

$$\text{Specificity} = \frac{TN}{TN + FP}$$

- **False Positive Rate (FPR):** The proportion of false positive predictions out of all actual negatives (i.e., how many non-relevant items are incorrectly classified as relevant).

$$\text{FPR} = \frac{FP}{FP + TN}$$

- **False Negative Rate (FNR):** The proportion of false negative predictions out of all actual positives (i.e., how many relevant items are incorrectly classified as non-relevant).

$$\text{FNR} = \frac{FN}{FN + TP}$$

Chapter 4

Database

Data and Database

Data refers to raw, unorganized facts, figures, or information such as numbers, text, images, videos, or any other form of information.

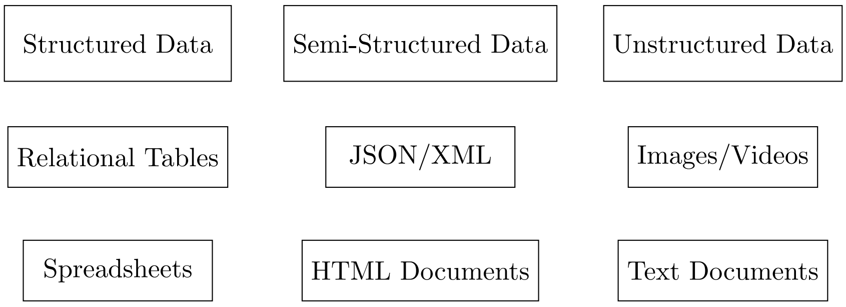


Figure 4.1: Classification of Data: Structured, Semi-Structured, and Unstructured

Database is a different term used for organized collection of data stored in a structured format, often in tables or schemas, to allow efficient retrieval, updating, and management. Examples of databases include **relational databases** like Oracle Database, Microsoft structured query language (SQL) server, MySQL, PostgreSQL, and non-relational (NoSQL) databases like MongoDB.

4.1 Types of Data

4.1.1 Structured, Semi-Structured and Unstructured

Structured data is highly organized and easily searchable. It typically follows a predefined schema, often consisting of tables with rows and columns. Structured data is stored in **relational databases** that use a **structured query language (SQL)** for data manipulation.

Characteristics of Structured Data:

- **Tabular format**: Data is stored in tables (relations) consisting of rows and columns.
- **Fixed schema**: The structure of the data (types, relationships) is predefined and rigid.
- **Easily searchable**: Due to its predictable format, it can be efficiently queried using SQL.
- **Examples**: Customer records, employee data, product catalogs.

Popular Structured Databases:

- MySQL
- PostgreSQL
- SQLite
- Oracle Database
- Microsoft SQL Server

Example of Structured Data:
Consider a simple **Employee** table in a structured database:

EmployeeID	Name	Age	Department	Salary
1	John Doe	28	Sales	50000
2	Jane Smith	34	HR	60000
3	Emma White	29	IT	70000

Each row represents a record, and each column stores a specific attribute (e.g., Name, Age, Salary). The schema (structure) of this table is predefined and consistent.

Semi-structured data is **partially organized** but does not follow a strict schema like structured data. It contains **tags, markers, or metadata**. The characteristics of semi-structured data is given below.

- Contains **both structured and unstructured elements**.
- Uses **flexible formats** (e.g., key-value pairs, nested data).

- Often stored in **NoSQL databases** (MongoDB, Cassandra).
- Used in **web applications, API[1]s, and big data systems**.

Examples of semi-structured data include JSON (JavaScript Object Notation),XML (Extensible Markup Language),HTML (HyperText Markup Language) and Emails (Headers + Text Content).

Example (JSON - Semi-Structured Data):

```
{
    "EmployeeID": 101,
    "Name": "Alice",
    "Age": 30,
    "Department": "HR",
    "Skills": ["Recruitment", "Payroll", "Compliance"]
}
```

Unstructured data does not follow a predefined schema and is often stored in formats that are not organized in a tabular form. This type of data can be more flexible, but accessing or processing it requires more advanced techniques, often involving **text analysis**, **machine learning**, or **NoSQL databases**.

Characteristics of Unstructured Data:

- **No fixed format**: Data is often stored in a variety of formats (e.g., text, images, video, audio, etc.).
- **Lack of schema**: There is no predefined structure or organization. The data is often raw and not directly usable in its native format.
- **Complex querying**: It requires special techniques, such as **full-text search**, **natural language processing (NLP)**, or machine learning for querying and analysis.
- **Examples**: Emails, social media posts, images, videos, audio recordings, documents.

Popular Unstructured Databases:

- MongoDB (a NoSQL database that allows flexible data storage)
- Cassandra (a distributed NoSQL database)
- Elasticsearch (used for searching large volumes of unstructured text data)
- Apache Hadoop (used for storing large-scale unstructured data in a distributed system)

[1]An API (Application Programming Interface) is like a waiter in a restaurant – it takes your request, fetches data from the kitchen (server), and brings back the response.

- Amazon S3 (storage service that handles unstructured data like images, videos, logs)

Example of Unstructured Data:
Consider an unstructured collection of data:

- **Text Documents**: A collection of company memos, reports, and presentations (these documents are in different formats like .txt, .pdf, .docx).
- **Images**: A collection of product photos, customer feedback images, or any multimedia files.
- **Social Media Posts**: Tweets, Facebook posts, or Instagram photos with captions, which don't follow a consistent structure.
- **Internet of Things (IoT) data**, such as ticker or sensor data from devices.
- **Healthcare data and imaging** such as MRIs, x-rays, and CT scans and other medical data like doctor's notes and prescriptions.

Example (Unstructured Data - Social Media Post):

```
"Feeling amazing after watching the latest sci-fi movie!
Can't wait to discuss it with friends!"
```

ature	**Structured Data**	**Semi-Structured Data**	**Unstructured Data**
rmat	Tabular (Rows/Cols)	Key-Value, Tags	Free Text, Media
hema	Fixed	Flexible	None
orage	SQL Databases	NoSQL Databases	File Systems, Cloud
mples	Excel, MySQL, ERP	JSON, XML, HTML	Images, Videos, Emails
cessing	SQL Queries	Parsing, Indexing	AI, NLP, Big Data

Table 4.1: Comparison of Structured, Semi-Structured, and Unstructured Data

<u>Application Programming Interface</u>
An API (Application Programming Interface) is like a waiter in a restaurant – it takes your request, fetches data from the kitchen (server), and brings back the response.

Imagine you want to order coffee using a coffee shop's mobile app. The process works like an API:

1. You (Client/App): Open the coffee app and select "Cappuccino."
2. API (Waiter): Sends your request to the coffee shop's system.
3. Coffee Machine (Server): Prepares your Cappuccino.

4. API (Waiter): Brings the Cappuccino back to you.

- APIs act as messengers that allow different software, platforms, or devices to interact without understanding each other's internal workings. For Example: A weather app fetches real-time weather updates from a remote server via an API.
- APIs help in Code Reusability and Efficiency. For Example: Instead of writing your own payment processing system, you can use PayPal or Stripe API.
- APIs provide a structured way for applications to exchange data using protocols like REST[2], SOAP[3].For example : A flight booking website can pull ticket prices from different airlines via APIs in a uniform way.
- APIs provide a structured way for applications to exchange data using protocols like REST, SOAP, or GraphQL. For example : A flight booking website can pull ticket prices from different airlines via APIs in a uniform way.
- APIs allow secure access to data without exposing the entire database. They use authentication mechanisms like OAuth, API keys, and tokens. For example : When you log into an app using Google Login, an API securely verifies your identity.
- APIs help automate tasks by integrating services. They also allow businesses to scale without additional effort. For Example: A hotel booking site automatically updates room availability by integrating with multiple hotel databases via APIs.
- APIs allow apps to work across different platforms (web, mobile, IoT, etc.). For Example: The Google Maps API is used by Uber, food delivery apps, and travel apps to display real-time locations.
- APIs let developers build on existing services and create new applications. For Example: Chatbots use the OpenAI API to provide intelligent responses without developing their own AI models.

4.1.2 Quantitative and Qualitative

Definition: Quantitative data represents numerical values that can be **measured and counted**. It answers questions like *"How many?"*, *"How much?"*, or *"How often?"*.

Examples:

[2]A REST API (Representational State Transfer API) is a web-based API that allows communication between clients (like web or mobile apps) and servers over HTTP.

[3]A SOAP (Simple Object Access Protocol) API is a protocol-based web service communication method that uses XML to send and receive messages over the internet. Unlike REST, SOAP is more structured, secure, and reliable, making it ideal for enterprise applications.

- A student's **exam score** (e.g., 85 out of 100)
- The **temperature** in a city (e.g., 30°C)
- The **number of products sold** in a store (e.g., 500 units)

Types of Quantitative Data:

- **Discrete Data** (Counts, Whole Numbers) Example: Number of employees in a company (e.g., 100, 200, 500)
- **Continuous Data** (Measurements, Decimals) Example: Weight of a fruit (e.g., 1.5 kg, 2.3 kg)

Definition: Qualitative data represents **descriptive, non-numerical** information. It answers questions like *"What type?"*, *"What category?"*, or *"What characteristics?"*.

Examples:

- **Customer feedback** (e.g., "Excellent," "Poor")
- **Eye color** (e.g., Blue, Brown, Green)
- **Types of music genres** (e.g., Rock, Jazz, Classical)

Types of Qualitative Data:

- **Nominal Data** (Categories with no order) Example: Types of pets (Dog, Cat, Bird)
- **Ordinal Data** (Ordered categories) Example: Satisfaction rating (Bad, Average, Good, Excellent)

'eature	**Quantitative Data**	**Qualitative Data**
efinition	Numerical, measurable	Descriptive, categorical
xamples	Age, height, salary	Eye color, product reviews
Types	Discrete, Continuous	Nominal, Ordinal
erations	Can do mathematical calculations	Cannot do calculations, but can categorize
se Cases	Sales reports, scientific research	Market research, customer feedback

Table 4.2: Comparison between Quantitative and Qualitative Data

Survey on Customer Satisfaction at a Restaurant

- **Quantitative Data:**
  - Number of customers visited: **200 per day**
  - Average bill amount: **$500 per table**
- **Qualitative Data:**
  - Customer reviews: **"Delicious food", "Slow service"**
  - Restaurant ambiance rating: **"Excellent", "Poor"**

4.2 Database

4.2.1 Primary Key and Foreign Key

Assume that there is a shoes company that launches many models of shoes. The models of shoes are stored in the ”PRODUCT TABLE” and the details customers of the shoes company are stored in the ”CUSTOMERS TABLE”. Whenever a sale gets recorded, a shoes is provided to a particular customer and this information regarding the model of the shoe and the customer is stored in the ”SALES TABLE”.

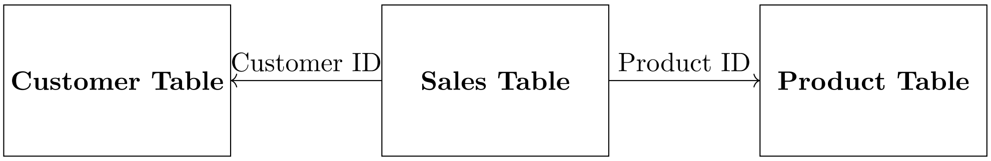


Figure 4.2: Sales Table interacting with Customer and Product Tables

A **Primary Key (PK)** is a unique identifier for each record in a table. It ensures that no two rows have the same value for this key and that it **cannot be NULL**. `Customer ID` is the **Primary Key** because it uniquely identifies

Customer ID (PK)	**Name**	**Email**
101	Alice	alice@email.com
102	Bob	bob@email.com

Table 4.3: Customer Table with Primary Key

each customer.

A **Foreign Key (FK)** is a column that **links to the Primary Key** of another table, creating a relationship between tables. `Customer ID` in the Sales

Sale ID (PK)	**Customer ID (FK)**	**ProductID (FK)**
1	101	R10101
2	102	R30303
3	101	R10101

Table 4.4: Sales Table with Foreign Key Relationship

Table is a **Foreign Key**, linking to `Customer ID` in the Customer Table 4.4. `Product ID` in the Sales Table is a **Foreign Key**, linking to `Product ID` in the Product Table 4.5.

The key differences between primary key and foreign key is given below.

Product No	**Product ID (PK)**	**Product**	**Amount**
1	R10101	Laptop	100000
2	R30303	Charger	5000

Table 4.5: Product Table with Foreign Key Relationship

Feature	**Primary Key (PK)**	**Foreign Key (FK)**
Uniqueness	Unique for each row	Can have duplicate values
NULL values	Cannot be NULL	Can have NULL values
Defines	Uniquely identifies a row	Establishes relationships between tables
Location	Defined in the same table	References a Primary Key in another table

Table 4.6: Comparison of Primary Key and Foreign Key

4.2.2 Database and Datawarehouse

The database consisting of tables that are related to each other is designed to store, retrieve, and manage current transactional data and primarily supports day-to-day operations (OLTP: Online Transaction Processing). However, if it is required to answer queries for which historical processing of data is required, then a ”data warehouse”, a special type of database is needed for reporting and decision making (OLAP: Online Analytical Processing). While a database is focussed on quick insert, update, and delete operations, a data warehouse is focussed on complex queries and analytical calculations. In our example, processing customer orders on an e-commerce site would be done by a database and a report on quarterly sales performance of shoes would be generated by a data warehouse. Examples of data warehouse includes Amazon Redshift, Google BigQuery, Snowflake, Microsoft Azure Synapse Analytics.

4.2.3 OLTP & OLAP

Transaction Processing (OLTP - Online Transaction Processing) systems handle real-time, day-to-day business transactions and have the following features.

- High volume, short transactions.
- Ensures **data integrity and consistency**.
- Supports **multiple concurrent users**.
- Uses **normalized databases** to reduce redundancy.

Examples:

- Banking transactions (withdrawals, deposits).
- Online booking systems (hotel, flights).

- Retail point-of-sale (POS) systems.

Analytical Processing (OLAP - Online Analytical Processing) systems are used for complex queries, analysis, and decision-making and have the following features.

- Focuses on **data aggregation and reporting**.
- Uses **historical data** for trends and insights.
- Optimized for **complex queries**, not frequent updates.
- Uses **denormalized databases** to improve query speed.

Examples:

- Business intelligence dashboards.
- Data warehousing for sales trends analysis.
- Customer behavior and market analytics.

Feature	**OLTP (Transaction Processing)**	**OLAP (Analytical Process**
Purpose	Handle real-time transactions	Perform complex analysis & rep
Data Type	Current operational data	Historical and aggregated da
Operations	Insert, Update, Delete, Read	Read-heavy, Aggregations
Database Type	Normalized (Relational DB)	Denormalized (Data Warehou
Speed	Fast transactions	Optimized for query speed
Users	Front-end users (e.g., cashiers)	Business analysts, executive

Table 4.7: Comparison of OLTP and OLAP

4.2.4 Normalized Schema and Denormalized Schema

The **normalized schema** and **denormalized schema** are two approaches to organizing data in a database. The choice between them depends on the use case: transactional processing (OLTP) or analytical processing (OLAP). Below is a detailed comparison:

Normalized Schema: A schema where data is divided into multiple related tables to minimize **redundancy** and ensure data **consistency**.

Characteristics:

- **Structure**: Data is organized into smaller tables with relationships between them.
- **Data Redundancy**: Minimal, as repeated data is eliminated.
- **Performance**: Optimized for insert, update, and delete operations.

- **Complexity**: Requires joins for queries, which can make retrieval slower for complex queries.
- **Use Case**: OLTP systems, where data **integrity** and efficiency in transactional operations are critical.

Example: Suppose you have customer and order data.

- **Customer Table**:

CustomerID	Name	Email
1	Alice	alice@mail.com
2	Bob	bob@mail.com

- **Order Table**:

OrderID	CustomerID	Amount
101	1	$50
102	2	$100
103	1	$500
104	2	$5000

Denormalized Schema: A schema where data is combined into fewer tables, often duplicating data to optimize query performance.
Characteristics:

- **Structure**: Fewer, larger tables with redundant data.
- **Data Redundancy**: High, as data may be repeated in multiple places.
- **Performance**: Optimized for read-heavy operations (fewer joins).
- **Complexity**: Simpler queries, as joins are minimized or unnecessary.
- **Use Case**: OLAP systems, where query speed and data aggregation are prioritized.

Example: Combine customer and order data into a single table:

CustomerID	Name	Email	OrderID	Amount
1	Alice	alice@mail.com	101	$50
2	Bob	bob@mail.com	102	$100
1	Alice	alice@mail.com	103	$500
2	Bob	bob@mail.com	104	$5000

Redundancy in Normalized & De-Normalized Schema
Tables 4.9, 4.10, 4.11 show customer table, product table and order table, representing a normalized schema. In the normalized schema, to minimize redundancy, the data is split into separate tables. The customer (customer name

Feature	**Normalized Schema**	**Denormalized Schema**
Structure	Data split across multiple tables	Data combined into fewer tables
Data Redundancy	Minimal	High
Query Speed	Slower for complex queries (many joins)	Faster for complex queries (fewer joins)
Maintenance	Easier (changes affect fewer places)	Harder (data duplication to update)
Use Case	OLTP (transactional systems)	OLAP (analytical systems)
Storage	Efficient (less storage required)	Larger (due to redundancy)

Table 4.8: Normalized and Denormalized Schema

Table 4.9: Customer Data (Normalized Schema)

CID (PK)	**Name**	**City**
101	Alice Smith	New York
102	Bob Johnson	Chicago
103	Charlie Brown	Seattle

Table 4.10: Product Data (Normalized Schema)

PID (PK)	**Product**	**Category**
P001	Laptop	Electronics
P002	Smartphone	Electronics
P003	Desk Chair	Furniture

Table 4.11: Order Table (Normalized Schema)

OID (PK)	**CID (FK)**	**PID (FK)**	**Amount**
1	101	P001	$1200
2	102	P002	$800
3	101	P003	$150
4	103	P001	$1200

and city) and product details (product and category) are stored in their respective tables and referenced via IDs. The normalized schemas requires complex queries for extracting information from the database.

Table 4.12 represents the sales table in denormalized schema where all information is combined into a single table, resulting in data redundancy as values are repeated multiple times. The same customer (e.g., Alice Smith) appears multiple times in the table, duplicating `CustomerName` and `CustomerCity`. The same product (e.g., Laptop with `ProductID` P001) appears multiple times, duplicating `ProductName` and `ProductCategory`. Repeated data increases storage requirements, such as `Electronics` being repeated for each product in that category. The denormalized schemas have simple queries for extracting information from the database.

Table 4.12: Sales Data (Denormalized Schema)

OID	CID	Name	City	PID	Product	Category	Amount
1	101	Alice Smith	New York	P001	Laptop	Electronics	$1200
2	102	Bob Johnson	Chicago	P002	Smartphone	Electronics	$800
3	101	Alice Smith	New York	P003	Desk Chair	Furniture	$150
4	103	Charlie Brown	Seattle	P001	Laptop	Electronics	$1200

Where Joins Make Retrieval Slower for Complex Queries?

In a relational database for an e-commerce system, multiple tables store different entities:

- **Customers Table** (Stores customer details)
- **Orders Table** (Stores each order placed by customers)
- **Products Table** (Stores product details)
- **OrderDetails Table** (Links orders and products with quantities and prices)

To fetch a customer's order history, including product names, quantities, and prices, we need to join multiple tables:

```
SELECT Customers.CustomerID, Customers.Name, Orders.OrderID,
       Products.ProductName, OrderDetails.Quantity, OrderDetails.Price
FROM Customers
JOIN Orders ON Customers.CustomerID = Orders.CustomerID
JOIN OrderDetails ON Orders.OrderID = OrderDetails.OrderID
JOIN Products ON OrderDetails.ProductID = Products.ProductID
WHERE Customers.CustomerID = 101;
```

The query is slow because

- **Multiple Joins:** Joins across four tables increase **query complexity**.
- **Indexes Needed:** Without proper indexing on `CustomerID`, `OrderID`, and `ProductID`, retrieval slows down.
- **Data Size:** As **millions of orders and products** grow, query execution time **increases** significantly.

4.2.5 Alternative Faster Approach

- **Denormalization:** Store a **Customer Purchase History Table** that pre-aggregates data.
- **NoSQL Solution:** Use a **document-based database (MongoDB)** where customer orders are **nested** inside customer documents.

4.3 Hands-On Relational Database - SQLite

SQLite is a free and open-source relational database engine written in the C programming language.

- Create an in-memory database - The statement import sqlite3 is used to import the SQLite module in Python, which allows you to interact with SQLite databases. SQLite is a lightweight, serverless, self-contained SQL database engine. It is built into Python, so you don't need to install any additional libraries. It stores data in a single file (e.g., database.db). It is widely used for small to medium applications, mobile apps, and local storage.

  Unlike traditional database systems like MySQL, PostgreSQL, or SQL Server, SQLite is serverless because it does not require a separate server process to manage the database. Instead, the SQLite engine is embedded directly into the application. Traditional databases require a separate server process running in the background to handle connections and queries. However, unlike traditional database systems, multi-user support is limited for sqlite databases.

  Why is there a server between the database and applications?

  1. Access Control: The database server acts as a gatekeeper, ensuring that only authorized users or applications can access the database. This allows for centralized authentication and management of user permissions.
  2. The database server is responsible for interpreting and executing SQL queries efficiently.

```
import sqlite3

# Create an in-memory database (for persistent storage,
    use 'example.db')

#creates an in-memory SQLite database, meaning the
    database exists only in RAM and disappears once the
    script ends.
```

```
conn = sqlite3.connect(":memory:")
#cursor = conn.cursor() is an object that allows you to
    execute SQL commands and fetch results from the
    database.

cursor = conn.cursor()
```

- Select Operation

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

  The SELECT statement in SQL is used to retrieve data from a database. It allows you to specify which columns (column1,column2) to fetch from a table and can include conditions to filter the results.

  – Retrieves all columns from the employees table.

```
SELECT * FROM employees;
```

  – Retrieves employees who work in the Sales department.

```
SELECT * FROM employees WHERE department_id = '101';
```

  – Retrieves employee id and names, sorted in descending order of department_id.

```
SELECT id, name FROM employees ORDER BY
    department_id DESC;
```

  – Returns the total number of employees.

```
SELECT COUNT(*) FROM employees;
```

  – Counts employees in each department.

```
SELECT COUNT(*) FROM employees GROUP BY
    department_id;
```

  – Retrieves only the first 5 rows.

```
SELECT * FROM employees LIMIT 5;
```

- Create two tables such as employees table and departments table.

```
# Create employees table
cursor.execute('''
    CREATE TABLE employees (
        id INTEGER PRIMARY KEY,
        name TEXT,
        department_id INTEGER
    )
''')

# Create departments table
cursor.execute('''
    CREATE TABLE departments (
        id INTEGER PRIMARY KEY,
        department_name TEXT
    )
''')

# Insert data into employees
cursor.executemany('''
    INSERT INTO employees (id, name, department_id)
        VALUES (?, ?, ?)
''', [
    (1, 'Alice', 101),
    (2, 'Bob', 102),
    (3, 'Charlie', 103),
    (4, 'David', None)
])

# Insert data into departments
cursor.executemany('''
    INSERT INTO departments (id, department_name) VALUES
         (?, ?)
''', [
    (101, 'HR'),
    (102, 'Engineering'),
    (104, 'Marketing')
])

# Commit the changes
# Save changes to the database
conn.commit()
```

- Printing the database employees

```
cursor.execute("SELECT * FROM employees")
rows = cursor.fetchall()

for row in rows:
    print(row)
```

The output is employees table.

```
(1, 'Alice', 101)
(2, 'Bob', 102)
(3, 'Charlie', 103)
(4, 'David', None)
```

- Printing the department table

```
cursor.execute("SELECT * FROM departments")
rows = cursor.fetchall()
for row in rows:
    print(row)
```

```
(101, 'HR')
(102, 'Engineering')
(104, 'Marketing')
```

- Perform join operations using the keys in tables. The JOIN operation in SQL is used to combine rows from two or more tables based on a related column.

  – INNER JOIN - The INNER JOIN keyword selects records that have matching values in both left (employees) and right tables (departments).

```
query = '''
    #This query retrieves name from table employees
        and department name from table departments
    SELECT employees.name, departments.
        department_name
    #Specifies that the primary table is employees
    FROM employees
    #Combines the employees and departments tables.
    #The join is performed where the department_id
    #column in employees matches the
    #id column in departments.
    INNER JOIN departments ON employees.
        department_id = departments.id
'''

for row in cursor.execute(query):
    print(row)
```

  The output of inner join is

```
('Alice', 'HR')
('Bob', 'Engineering')
```

Charlie and David are excluded because they have no matching department.

– LEFT JOIN (LEFT OUTER JOIN) - The LEFT JOIN keyword returns all records from the left table (employees), and the matching records from the right table (departments).

```
#left join
query = '''
    SELECT employees.name, departments.
       department_name
    FROM employees
    LEFT JOIN departments ON employees.department_id
        = departments.id
'''

for row in cursor.execute(query):
    print(row)
```

The output returns all employees, even if they don't belong to any department.

```
('Alice', 'HR')
('Bob', 'Engineering')
('Charlie', None)
('David', None)
```

– RIGHT JOIN - The RIGHT JOIN keyword returns all records from the right table (departments), and the matching records from the left table (employees).

```
#right join
query = '''
    SELECT employees.name, departments.
       department_name
    FROM departments
    LEFT JOIN employees ON employees.department_id =
        departments.id
'''

for row in cursor.execute(query):
    print(row)
```

The output returns all departments, even if none belong to that department.

```
('Alice', 'HR')
('Bob', 'Engineering')
(None, 'Marketing')
```

– FULL JOIN - The FULL OUTER JOIN keyword returns all records when there is a match in left (employees) or right (departments) table records.

```
#full join
query = '''
    SELECT employees.name, departments.
       department_name
    FROM employees
    LEFT JOIN departments ON employees.department_id
        = departments.id
    UNION
    SELECT employees.name, departments.
       department_name
    FROM departments
    LEFT JOIN employees ON employees.department_id =
        departments.id
'''

for row in cursor.execute(query):
    print(row)
```

The output is

```
(None, 'Marketing')
('Alice', 'HR')
('Bob', 'Engineering')
('Charlie', None)
('David', None)
```

(Charlie, David, and Marketing are included.)

– Display join results as a data frame, where query refers to one of the join operations.

```
import pandas as pd
df = pd.read_sql_query(query, conn)
df
```

The join operations are summarized using the Venn Diagrams as shown in Figure 4.3.

- conn.close() - In SQLite (or any SQL-based database), conn.close() is used to close the connection to the database.

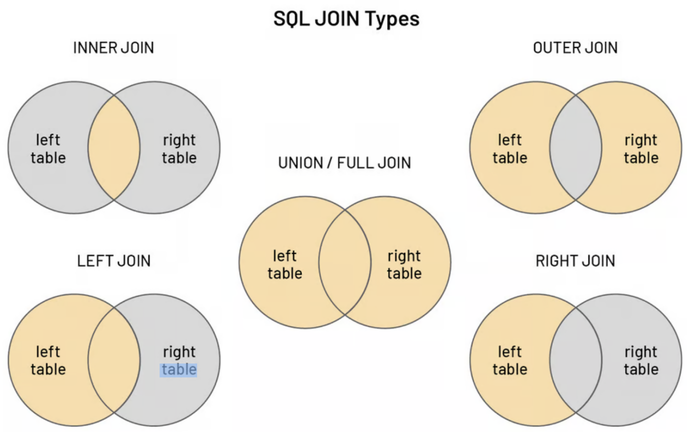


Figure 4.3: Join Operations in SQL

4.4 ACID Properties in SQLite: Case Study of a Bookstore Inventory

We will illustrate the ACID properties of a SQLite database used to manage a small bookstore's inventory and sales transactions.

Entities in the SQLite Database:

- **Books**: Contains details about each book in the store (e.g., title, author, price, stock quantity).
- **Sales**: Contains records of sales transactions, linking customers and books purchased.

The following subsections demonstrate how the ACID properties are applied in SQLite.

Atomicity ensures that a transaction is either fully completed or not executed at all. If any part of the transaction fails, the entire operation is rolled back.

Example Scenario: A customer purchases 2 copies of a book.

Explanation: If any part of the transaction fails, such as the stock update or the insertion of the sale record, the transaction is rolled back, ensuring that no partial updates are made to the database.

Consistency ensures that the database moves from one valid state to another while maintaining all integrity constraints.

Example Scenario: A customer tries to purchase 5 copies of a book, but only 3 are available.

Explanation: If the stock is insufficient for the requested purchase, the transaction is rolled back, ensuring that the database remains in a consistent state.

Isolation ensures that transactions are executed independently and their results are isolated from one another. Concurrent transactions do not interfere with each other.

Example Scenario: Two customers try to purchase the last available copy of the same book at the same time.

Explanation: With the EXCLUSIVE isolation level, Customer 2's transaction will be blocked until Customer 1's transaction is completed. This prevents both customers from purchasing the last available book at the same time.

Durability ensures that once a transaction is committed, the changes are permanent, even if the system crashes immediately afterward.

Example Scenario: A customer successfully places an order for a book.

Explanation: Even if the system crashes immediately after committing the transaction, the changes made to the stock and sales records will be permanent and recoverable from the database.

4.5 Caching

Caching is a technique used in computing to temporarily store copies of data in a storage location, known as a cache, to enable faster access upon future requests. This process enhances application and system performance by reducing the time and resources required to retrieve data from its original source.

How Caching works?

- Data Request: When an application or system requests data, it first checks the cache to see if the data is already stored—a scenario known as a "cache hit."
- Cache Hit: If the data is found in the cache, it is retrieved quickly, bypassing the need to access the slower primary storage.
- Cache Miss: If the data is not in the cache ("cache miss"), the system retrieves it from the primary storage, delivers it to the requester, and may also store a copy in the cache for future access.

Example : Web Page Loading - When you visit a website, your browser downloads various resources—such as HTML files, images, and stylesheets—from the web server to display the page.

- First Visit (Without Cache):Your browser sends a request to the web server for each resource. The server processes these requests and sends the resources back to your browser. This process involves network latency and server processing time, which can result in noticeable delays, especially if the server is distant or under heavy load.
- Subsequent Visits (With Cache):After the initial visit, your browser stores copies of these resources in its local cache. On subsequent visits to the same website, the browser checks its cache for the required resources.

If the resources are present and still valid (i.e., they haven't changed since the last download), the browser loads them directly from the cache. This bypasses the need to request the resources from the server again, significantly reducing load times and improving the user experience.

Common Caching Strategies?

- Write-Through Cache: Data is written to both the cache and the primary storage simultaneously, ensuring consistency but potentially increasing write latency.
- Write-Back (Lazy Write) Cache: Data is initially written only to the cache, with updates to primary storage deferred, which improves write performance but requires mechanisms to handle potential data loss in case of a cache failure.
- Least Recently Used (LRU): The cache evicts the least recently accessed items first when space is needed, operating on the principle that recently used data is more likely to be accessed again soon.

Content Delivery Network (CDN) Cache Performance - Caching enhances system performance by storing frequently accessed data in a readily accessible location, reducing the time required to retrieve it. Let's explore this concept through a quantitative example involving web content delivery.

The cache hit ratio indicates the percentage of user requests served directly from the cache versus those requiring retrieval from the origin server. It is calculated using the formula:

$$\text{Cache Hit Ratio (\%)} = \left(\frac{\text{Number of Cache Hits}}{\text{Total Number of Requests}} \right) \times 100$$

Where:

- **Cache Hits**: Number of requests successfully served from the cache.
- **Total Number of Requests**: Sum of cache hits and cache misses (requests not found in the cache).

Suppose over a specific period, the CDN records:

- **Cache Hits**: 950
- **Cache Misses**: 50

The **Total Number of Requests** is:

$$950 \text{ (hits)} + 50 \text{ (misses)} = 1000 \text{ requests}$$

Applying these values to the cache hit ratio formula:

$$\text{Cache Hit Ratio (\%)} = \left(\frac{950}{1000}\right) \times 100 = 95\%$$

Assume the following average response times:

- **Cache Hit Response Time**: 20 milliseconds (ms)
- **Cache Miss Response Time**: 200 ms

The **average response time** for all requests can be calculated as:

$$\text{Average Response Time} = \frac{(\text{Cache Hits} \times \text{Hit Response Time}) + (\text{Cache Misses} \times \text{Miss Response}}{\text{Total Number of Requests}}$$

Plugging in the values:

$$\text{Average Response Time} = \frac{(950 \times 20\text{ ms}) + (50 \times 200\text{ ms})}{1000}$$

$$\text{Average Response Time} = \frac{19000\text{ ms} + 10000\text{ ms}}{1000} = \frac{29000\text{ ms}}{1000} = 29\text{ ms}$$

In this scenario, caching reduces the average response time to 29 ms. Without caching (i.e., if every request resulted in a cache miss), the response time would consistently be 200 ms. Thus, effective caching improves performance by decreasing latency and enhancing user experience.

Cache : RAM or ROM? Cache memory is a type of high-speed volatile memory that stores frequently accessed data and instructions to speed up processing. It is typically implemented using static RAM (SRAM), which is faster but more expensive than the dynamic RAM (DRAM) used for main system memory. Unlike read-only memory (ROM), which is non-volatile and used to store permanent data that doesn't change during normal operation, both cache and RAM are volatile, meaning they lose their stored information when power is removed. Therefore, cache memory is more closely related to RAM in terms of volatility and functionality.

Cache & Personal Computer

- CPU Cache - Small, high-speed memory located within the CPU that stores frequently accessed data and instructions, reducing the time needed to retrieve information from the main memory.
  - L1 Cache: The smallest and fastest, located directly within the CPU core.
  - L2 Cache: Larger than L1, may be shared between cores or dedicated to a single core.

  - L3 Cache: Even larger and shared among all cores of the CPU.

- Disk Cache (Page Cache) - A portion of RAM allocated by the operating system to store frequently accessed data from storage devices, such as hard drives or SSDs, improving read and write speeds.

- Web Cache - Browsers like Chrome, Firefox, or Edge store copies of web pages, images, and other resources on your local drive, enabling faster loading times on subsequent visits. Web browser caches are primarily stored on your computer's hard drive or solid-state drive (SSD), not in RAM or ROM.

- Stores the IP addresses of recently accessed domain names, allowing your system to quickly resolve domain names without querying external DNS servers repeatedly.

Static Ram & Dynamic Ram - Static RAM (SRAM) and Dynamic RAM (DRAM) are two types of volatile memory used in computing, each with distinct characteristics and applications.

- Static RAM (SRAM):
  - Structure: Utilizes flip-flop circuits composed of multiple transistors (typically six) to store each bit of data.
  - Speed: Offers faster data access times due to its design, making it suitable for high-speed applications.
  - Cost and Density: More expensive and less dense because each bit requires multiple transistors, leading to higher costs and larger physical size per bit.
  - Usage: Commonly used for cache memory in CPUs and other applications where speed is critical.

- Dynamic RAM (DRAM):
  - Structure: Stores each bit of data in a cell consisting of a single transistor and a capacitor.
  - Speed: Slower than SRAM due to the need for periodic refreshing of data to maintain integrity.
  - Cost and Density: Less expensive and higher density, as each bit requires only one transistor and one capacitor, allowing for more bits per chip.
  - Usage: Typically used for main system memory (RAM) in computers, where larger capacity is beneficial.

4.6 No-SQL

NoSQL databases are designed to handle large volumes of unstructured or semi-structured data, offering flexibility and scalability beyond traditional relational databases. They are categorized based on their data models and use cases. The primary types include:

1. **Key-Value Stores:**
   These databases store data as key-value pairs, similar to a dictionary. Each key is unique, and its associated value can be a simple data type or a complex object. They are optimized for quick lookups and are ideal for caching and session storage. Examples include Redis and DynamoDB.

   Caching is a technique used in computing to temporarily store copies of data in a storage location, known as a cache, to enable faster access upon future requests.

2. **Document-Oriented Databases:** Data is stored in documents, typically in JSON or BSON format. Each document can have a different structure, allowing for flexible and hierarchical data representation. These databases are suitable for content management systems and applications requiring flexible schemas. MongoDB is a well-known example.

3. **Column-Oriented Databases:** Data is stored in columns rather than rows, which is efficient for read-heavy operations and analytical queries. They are ideal for time-series data and real-time analytics. Apache Cassandra and HBase are examples of column-family stores.

4. **Graph Databases:** These databases are designed to represent and traverse relationships between data points, making them ideal for applications like social networks and recommendation engines. Neo4j is a prominent example.

5. **Time-Series Databases:**
   Optimized for handling time-stamped data, these databases are ideal for applications like monitoring systems and IoT data storage. Examples include InfluxDB and TimescaleDB.

Each type of NoSQL database is tailored to specific use cases, offering advantages in scalability, flexibility, and performance for particular data models and access patterns.

4.6.1 Key-Value Database

A key-value database is a type of NoSQL database that stores data as pairs of keys and their corresponding values. This structure is akin to a dictionary or map in programming languages, where each key uniquely identifies a value. The simplicity of this model allows for efficient data retrieval and storage.

How It Works:

- Key: A unique identifier used to access a specific value.
- Value: The data associated with the key, which can be a string, number, JSON document, or even a binary object.
- Consider a key-value store used to manage user profiles:

```
Key: user123
Value: {"name": "John Doe", "email": "john.doe@example.
    com", "age": 30}
```

In this example, user123 is the unique identifier (key) for a user, and the associated value is a JSON object containing the user's details. Advantages
Advantages:

  – Simplicity: The straightforward key-value structure makes it easy to understand and implement.
  – Performance: Optimized for quick retrieval and storage operations, making it suitable for high-speed applications.
  – Scalability: Designed to handle large volumes of data across distributed systems efficiently.

Key-value databases are ideal for scenarios where data can be represented as simple key-value pairs, such as

  – Session Management: Storing user session information in web applications.
  – Caching: Temporarily storing data to reduce access times for frequently requested information.
  – User Preferences: Saving individual user settings and configurations

A JSON object is a fundamental data structure in JavaScript Object Notation (JSON), representing a collection of key-value pairs. It is enclosed within curly braces and consists of keys (also known as names) and their corresponding values, separated by colons. Each key-value pair is separated by a comma. The keys must be strings, and the values can be various data types, including strings, numbers, arrays, booleans, other objects, or null.

```
{
  "name": "John Doe",
  "age": 30,
  "isStudent": false,
  "courses": ["Math", "Science", "History"],
  "address": {
    "street": "123 Main St",
    "city": "Anytown",
    "zip": "12345"
  }
}
```

The python code for creating a document database is given below

```
import json
import os

class KeyValueNoSQL:
    def __init__(self, db_file='kv_store.json'):
        self.db_file = db_file
        self.data = {}
        self._load()

    def _load(self):
        """Load data from the JSON file."""
        if os.path.exists(self.db_file):
            with open(self.db_file, 'r') as f:
                self.data = json.load(f)

    def _save(self):
        """Save data to the JSON file."""
        with open(self.db_file, 'w') as f:
            json.dump(self.data, f, indent=4)

    def set(self, key, value):
        """Set a key-value pair."""
        self.data[key] = value
        self._save()

    def get(self, key):
        """Get the value for a given key."""
        return self.data.get(key, None)

    def delete(self, key):
        """Delete a key-value pair."""
        if key in self.data:
            del self.data[key]
            self._save()

    def keys(self):
        """Return all keys."""
        return list(self.data.keys())

    def values(self):
        """Return all values."""
        return list(self.data.values())
    def clear(self):
        """Clear the entire database."""
        self.data = {}
        self._save()

# Example Usage
if __name__ == "__main__":
    #Create an instance of KeyValueNoSQL
    db = KeyValueNoSQL()
    #Set a Key-Value Pair with key as name and value as
        Alice
    db.set("name", "Alice")
    print(db.get("name"))  # Output: Alice
    db.delete("name")
    print(db.get("name"))  # Output: None
    db.set("age", 30)
    #db.keys() returns ['age'], meaning the only
        existing key is "age"
    print(db.keys())  # Output: ['age']
    #Clear the Database and Check Keys Again
    db.clear()
    print(db.keys())  # Output: []

    db.set("address", {"street": "123 Main St", "city":
        "New York", "zip": "10001"})


    db.set("orders", [
        {"order_id": 101, "items": ["apple", "banana"],
            "total": 25.50},
        {"order_id": 102, "items": ["orange", "grape"],
            "total": 15.75}
    ])
    #print(db.get("address"))
    #print(db.get("orders"))
    print('data vbase data')
    print(db.data)
```

- A **document-oriented database** is a type of **NoSQL database** that stores data in a flexible, semi-structured format using **documents** (usually in `JSON`, `BSON`, or `XML` format). Unlike relational databases that use tables and rows, document-oriented databases store information as collections of documents.

  Key Features

  – **Schema-less**: Documents do not require a fixed structure, making them highly flexible.

- **Hierarchical Storage**: Supports nested documents (arrays, objects).
- **Scalability**: Designed for horizontal scaling (adding more servers instead of upgrading hardware).
- **Fast Reads/Writes**: Optimized for high-speed transactions.
- **No Joins Required**: Unlike relational databases, related data is often stored within the same document.

Structure of a Document Documents are stored as `JSON`, `BSON`, or `XML` objects.

Example JSON document for a user:

```
{
  "user_id": 1,
  "name": "Alice",
  "email": "alice@example.com",
  "address": {
    "street": "123 Main St",
    "city": "New York",
    "zip": "10001"
  },
  "orders": [
    { "order_id": 101, "amount": 50.00 },
    { "order_id": 102, "amount": 30.00 }
  ]
}
```

This structure eliminates the need for **JOIN operations** and allows **efficient querying**.

Database	Features
MongoDB	JSON-like BSON, highly scalable
CouchDB	JSON storage, HTTP API, MapReduce queries
Firebase Firestore	Cloud-based, real-time synchronization
RavenDB	ACID-compliant, optimized for transactions
Amazon DocumentDB	Managed AWS service, MongoDB-compatible

Table 4.13: Popular Document-Oriented Databases

Popular Document-Oriented Databases

Document-Oriented Database vs Relational Database

Feature	Document-Oriented DB	Relational DB
Schema	No fixed schema	Fixed schema
Data Storage	JSON/BSON/XML	Tables (rows/columns)
Scalability	Horizontal (add nodes)	Vertical (upgrade hardware)
Joins	Not required	Required
Performance	Optimized for fast reads/writes	Can be slower due to joins
Use Cases	Big data, IoT, real-time apps	Banking, ERP, structured data

Table 4.14: Comparison of Document-Oriented and Relational Databases

Use Cases of Document-Oriented Databases

– Content Management Systems (CMS)
– E-commerce product catalogs
– Real-time analytics (logs, user data, IoT)
– Mobile & web applications
– Gaming applications (user profiles, game progress)

Advantages

– **Flexible Schema**: Supports dynamic and evolving data structures.
– **High Performance**: Fast retrieval and storage of large volumes of data.
– **Scalability**: Easily scales horizontally across multiple servers.
– **No Joins Needed**: Reduces complexity and query processing time.

Disadvantages

– **Not Ideal for Complex Transactions**: Lacks strong ACID compliance (except in databases like RavenDB).
– **Data Duplication**: Since there are no joins, related data is often duplicated.
– **Limited Query Capabilities**: Compared to SQL databases, query capabilities are less powerful.

When to Use a Document-Oriented Database?

– When dealing with **semi-structured or unstructured data** (e.g., logs, social media data).
– When **frequent schema changes** are needed.
– When **high-speed reads and writes** are required (e.g., real-time applications).

  - When **scalability** is a priority.

- A **column-oriented database** is a type of database that stores data in columns rather than rows. This design optimizes read performance for analytical queries.

  Key Features

  - **Columnar Storage**: Data is stored by columns, enabling fast retrieval of specific attributes.
  - **Efficient Compression**: Similar data types in a column allow better compression ratios.
  - **Optimized for OLAP**: Best suited for analytical workloads, reporting, and data warehousing.
  - **Scalability**: Can scale horizontally across multiple nodes.

  Popular Column-Oriented Databases

  - Apache Cassandra
  - Apache HBase
  - Google Bigtable
  - Amazon Redshift
  - ClickHouse

  Advantages

  - Fast query performance for analytics.
  - Better compression reduces storage requirements.
  - Efficient for aggregate queries (SUM, AVG, COUNT).

  Disadvantages

  - Not ideal for transactional workloads.
  - Higher complexity in data updates compared to row-based databases.

  Use Cases

  - Business Intelligence (BI) applications.
  - Large-scale data analytics.
  - Data warehousing.

- A **graph database** is a type of NoSQL database designed to represent and store data as a network of nodes (entities) and edges (relationships). This structure is ideal for applications requiring complex relationships between data points.

Key Features

– **Nodes and Edges**: Nodes represent entities, and edges define relationships between them.

– **Highly Connected Data**: Optimized for storing and querying relationships.

– **Flexible Schema**: Can adapt dynamically to new relationships and properties.

– **Graph Query Languages**: Uses specialized query languages like Cypher (Neo4j) and Gremlin (Apache TinkerPop).

Popular Graph Databases

– Neo4j

– Amazon Neptune

– ArangoDB

– JanusGraph

– Microsoft Azure Cosmos DB (Graph API)

Advantages

– Optimized for relationship-based queries.

– Efficient for social networks, recommendation engines, and fraud detection.

– Flexible schema allows evolving data structures.

Disadvantages

– Not ideal for transactional workloads.

– Can be complex to design and maintain.

– Limited scalability compared to column-oriented databases.

Use Cases

– Social networks (e.g., LinkedIn, Facebook)

– Fraud detection in financial services

– Recommendation systems (e.g., Netflix, Amazon)

– Network and IT operations

– Knowledge graphs and semantic search

4.7 Horizontal and Vertical Scaling in Databases

Horizontal Scaling (Scaling Out): Adding more machines (nodes) to distribute the database load.

How it Works:

- Data is **sharded (partitioned)** across multiple servers.
- Each server handles a subset of data.
- Common in **NoSQL databases** like MongoDB, Cassandra.

Pros:

- Supports large-scale distributed systems.
- Improves availability and fault tolerance.

Cons:

- Increased complexity (data partitioning, consistency issues).
- Requires load balancing and replication.

Vertical Scaling (Scaling Up): Increasing the hardware resources (CPU, RAM, storage) of a single machine.

How it Works:

- Upgrading to a more powerful server.
- Common in **SQL databases** like MySQL, PostgreSQL.

Pros:

- Simpler to implement (no need for data partitioning).
- Lower software complexity.

Cons:

- Hardware limits exist.
- Downtime required for upgrades.

Key Differences of Horizontal and Vertical Scaling are

Feature	**Horizontal Scaling (Out)**	**Vertical Scaling (Up)**
Approach	Adds more machines	Increases machine power
Data Handling	Sharding across nodes	Centralized data storage
Best for	Large distributed systems	Single powerful database
Complexity	High (load balancing, partitions)	Low (simple upgrades)
Example DBs	MongoDB, Cassandra	MySQL, PostgreSQL

Table 4.15: Comparison of Horizontal and Vertical Scaling

Chapter 5

Problems

5.1 Matrix Factorization for User-Movie Rating Prediction

1. 1. Problem Setup: User-Movie Rating Matrix We define the user-movie rating matrix R as:

$$R = \begin{bmatrix} 5 & 3 & 0 & 1 \\ 4 & 0 & 0 & 1 \\ 1 & 1 & 0 & 5 \\ 0 & 1 & 5 & 4 \\ 0 & 0 & 5 & 4 \end{bmatrix}$$

   where:

   - Rows represent users.
   - Columns represent movies.
   - Entries are ratings (0 means missing ratings).

   The goal is to predict missing ratings.

   2. Matrix Factorization using SVD We approximate R using low-rank matrices:

$$R \approx U \Sigma V^T$$

   where:

   - U is the user-latent matrix.
   - Σ is the diagonal matrix with singular values.
   - V is the movie-latent matrix.

Step 1: Compute SVD Performing SVD on R:

$$U = \begin{bmatrix} -0.77 & -0.12 \\ -0.52 & -0.32 \\ -0.20 & -0.83 \\ -0.27 & 0.42 \\ -0.22 & 0.15 \end{bmatrix}$$

$$\Sigma = \begin{bmatrix} 9.76 & 0 \\ 0 & 5.47 \end{bmatrix}$$

$$V^T = \begin{bmatrix} -0.62 & -0.28 & -0.38 & -0.62 \\ -0.50 & -0.24 & 0.75 & 0.37 \end{bmatrix}$$

Step 2: Reduce Dimensionality Since Σ contains singular values sorted in decreasing order, we keep only the top 2 singular values and corresponding columns of U and V^T.

Step 3: Reconstruct Approximate R

$$\hat{R} = U_k \Sigma_k V_k^T$$

$$\hat{R} = \begin{bmatrix} 4.9 & 3.1 & 0.6 & 1.2 \\ 3.8 & 1.2 & 1.1 & 0.9 \\ 1.1 & 1.0 & 3.3 & 4.8 \\ 0.2 & 1.0 & 4.9 & 4.1 \\ 0.5 & 0.3 & 5.1 & 4.2 \end{bmatrix}$$

- Rows of U represent users in a latent space of 2 features.
- Rows of V^T represent movies in the same latent space.
- Singular values represent importance of latent features.

The missing ratings in R are now estimated in $\hat{R}$. For example, user 2's rating for movie 2 was missing in R and is now predicted as:

$$\hat{R}_{2,2} = 1.2$$

- Matrix factorization predicts missing ratings based on latent features.
- SVD decomposes the matrix into user and movie representations.
- Dimensionality reduction improves generalization.
- Reconstruction helps in recommendation systems.