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.

Copyright © 2025 Arjun Anil Kumar+ AI
License: Creative Commons Zero 1.0 Universal

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.

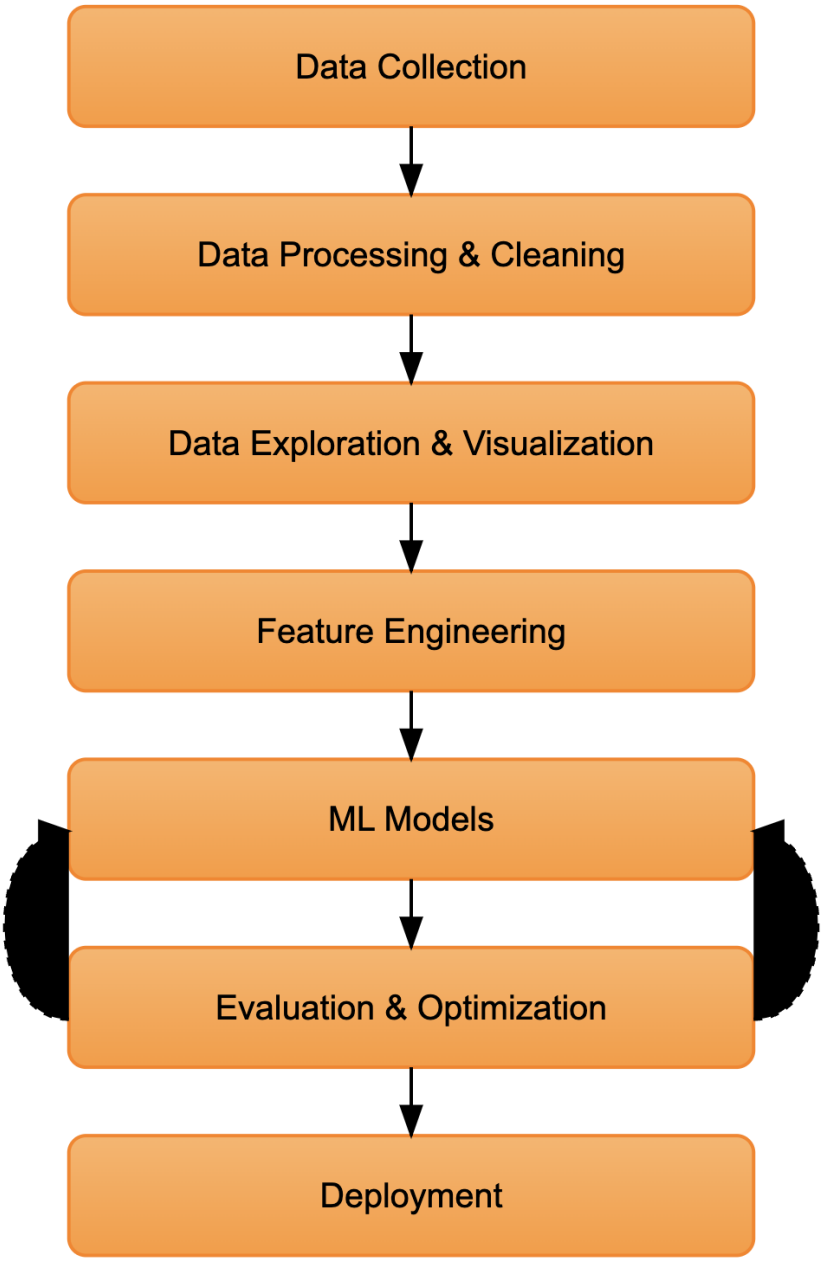

Figure 1.1: Data Science

- **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.

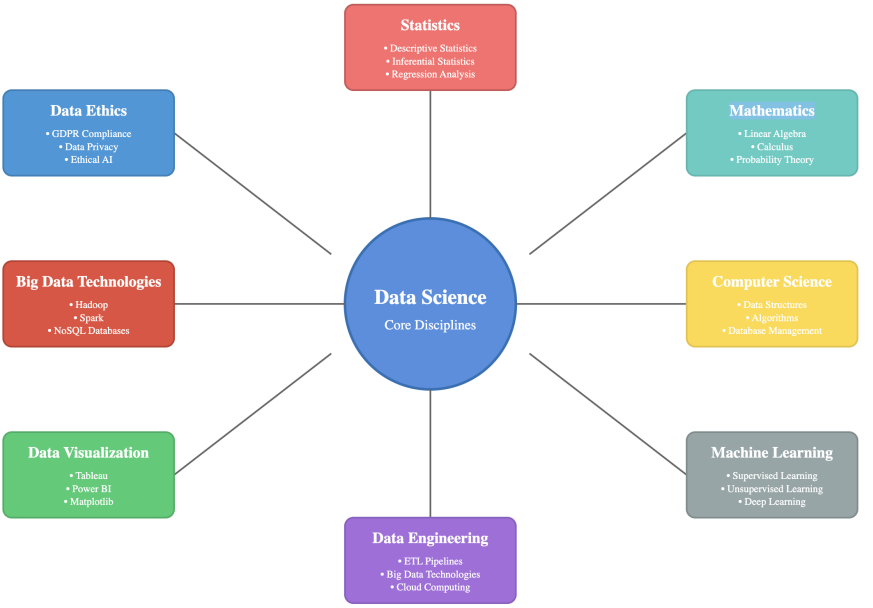

Figure 1.2: Data Science - Interdisciplinary Nature

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.

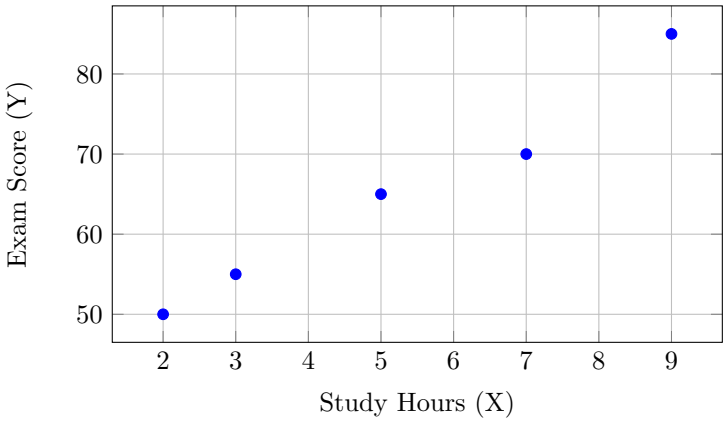

Figure 1.3: Scatter Plot of Study Hours vs. Exam Scores

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.

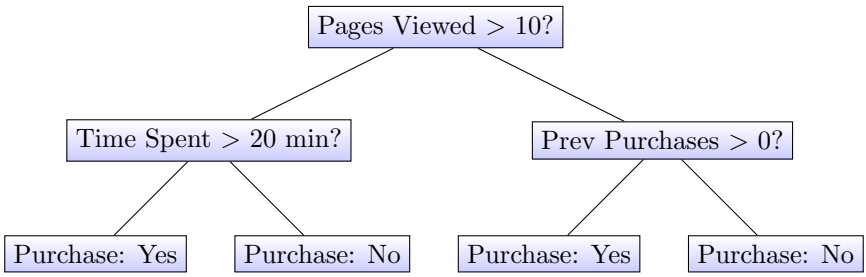

Figure 1.4: Simplified Decision Tree for Predicting Customer Purchases
Left direction indicates "yes" and Right direction indicates "no"

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

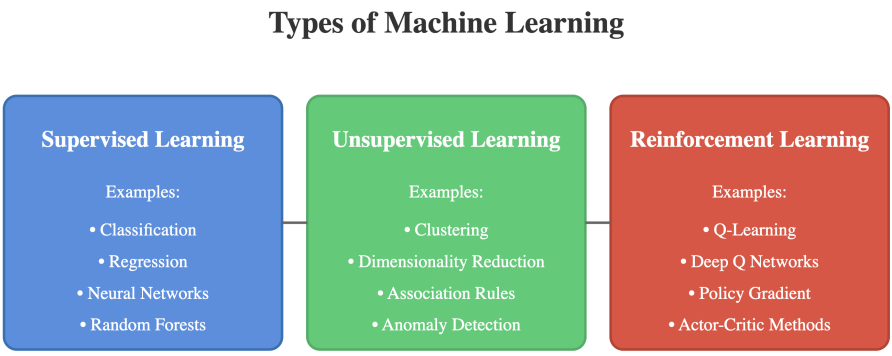

Figure 2.1: Machine Learning Models

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:

Location: A, B, C, Type: Villa, Flat, 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.

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

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

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]
```

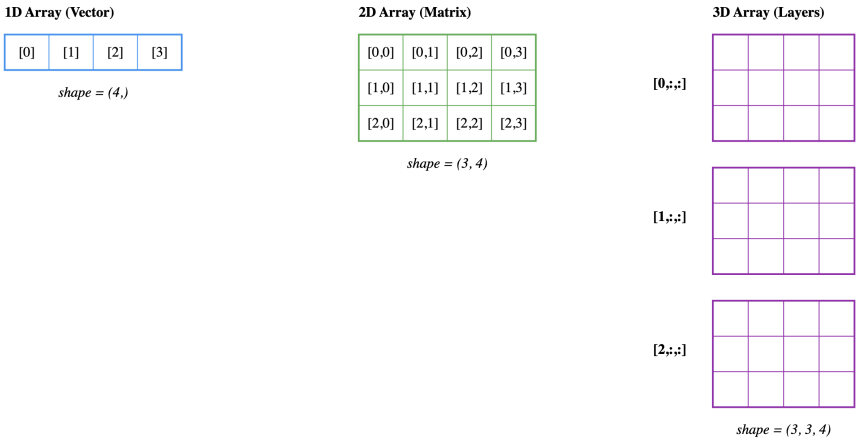

Figure 3.1: N-Dimensional Array

```
# 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.
le 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 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)
```

3.3.2 Modelling

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).

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