



Python

Workbook for Artificial Intelligence

Timea Bezdan

Belgrade, 2025.



Python Workbook for Artificial Intelligence

Timea Bezdan

Singidunum University
Belgrade, 2025

Python Workbook for Artificial Intelligence

Author:

Timea Bezdan

Reviewers:

Nebojša Bačanin Džakula

Miodrag Živković

Goran Kvašček

Publisher:

SINGIDUNUM UNIVERSITY

32 Danijelova Street, Belgrade

www.singidunum.ac.rs

For the publisher:

Milovan Stanišić

Prepress:

Timea Bezdan

Design:

Aleksandar Mihajlović

Year:

2025

Circulation:

300

Printed by:

Birograf, Belgrade

ISBN: 978-86-7912-851-5

Copyright © 2025 Singidunum University, Belgrade

All rights reserved. No part of this work covered by the copyright herein may be reproduced, transmitted, stored or used in any form or by any means graphic, electronic, or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, Web distribution, information networks, or information storage and retrieval systems, without the prior written permission of the publisher.

Contents

1	Introduction	1
2	Python Basics	3
2.1	Variables and Data Types	3
2.2	Basic Data Types	3
2.3	Built-in Data Types	4
2.3.1	Lists	4
2.3.2	Tuples	6
2.3.3	Sets	6
2.3.4	Dictionaries	7
2.4	Built-in Functions for Operations with Data Types	8
2.4.1	Built-in Functions for Operations with Strings	8
2.4.2	Built-in Functions for Operations with Lists	9
2.4.3	Built-in Functions for Operations with Tuples	9
2.4.4	Built-in Functions for Operations with Sets	9
2.4.5	Built-in Functions for Operations with Dictionaries	11
2.5	Fundamental Built-in Functions in Python	11
2.6	Loops	14
2.7	For Loop	14
2.8	While Loop	16
2.9	Functions	17
2.10	If Statement	17
2.11	List Comprehension	18
2.12	Lambda Function	19
2.13	Exercise Set	21
2.13.1	Exercise 1: Working with Lists and Functions	21
2.13.2	Exercise 2: Working with Strings and Functions	21
2.13.3	Exercise 3: Working with Dictionaries and Functions	22
2.13.4	Exercise 4: Working with Sets and Functions	23
2.13.5	Exercise 5: Working with Numbers and Functions	23
2.13.6	Exercise 6: Working with zip() and for loop	24
2.13.7	Exercise 7: List Comprehension	26
3	NumPy	28
3.1	NumPy Arrays	28

3.1.1	Array Information Functions	31
3.1.2	Array Reshaping and Manipulation Functions	32
3.2	Random sampling	36
3.3	Slicing and Indexing	39
3.4	Mathematical Operations	42
3.5	Statistical Functions	43
3.6	Aggregation Functions	43
3.7	Other Useful Functions	44
3.8	Boolean Arrays	45
3.9	Exercise Set	47
3.9.1	Exercise 1: Analyzing Student Scores	47
3.9.2	Exercise 2: Analyzing Sales Data	50
4	Pandas	54
4.1	Introduction to Pandas	54
4.2	Data Cleaning	66
4.3	Basic Data Operations	69
4.4	Data Integration	75
4.5	Data Aggregation	77
4.6	Time Series Data	81
4.7	Data Visualization	86
4.8	Data Input/Output	89
4.9	Exercise Set	90
4.9.1	Exercise 1: Basic Data Operations with Pandas	90
4.9.2	Exercise 2: Data Integration, Aggregation, and Visualization	92
4.9.3	Exercise 3: Analyzing Time Series Data with Pandas	95
5	Exploratory Data Analysis	101
5.1	Part 1: Data Overview and Summary Statistics	101
5.2	Part 2: Missing Values	103
5.3	Part 3: Data Distribution Visualization	103
5.4	Part 4: Relationship Between Features	106
6	Basic Concepts in Machine Learning	108
7	Linear Regression	110
8	Logistic Regression	119
9	Decision Trees	132

10 Deep Learning and AI Applications**143**

Chapter 1

Introduction

Artificial Intelligence in Context

Artificial Intelligence (AI) is one of the fastest growing areas in computer science, mathematics, and engineering. Artificial Intelligence is a discipline that studies how to design systems capable of reasoning, learning, and problem solving [1].

AI applications are now embedded in everyday technologies such as search engines, social media, recommender systems, medical diagnostics, and autonomous vehicles. Because of its broad influence, AI has become an essential subject of study for students in computing, engineering, and applied sciences.

Why Python for AI?

In this book, Python is used as the primary programming language for exploring AI concepts and methods. Python is widely used in artificial intelligence and data science because of its simplicity and the availability of powerful libraries such as NumPy [2], Pandas [3], and Matplotlib [4]. It also provides machine learning and deep learning frameworks such as scikit-learn [5], TensorFlow [6], and PyTorch [7].

These advantages make Python an excellent choice for both beginners and advanced learners in AI.

Structure of the Workbook

This workbook is designed to be **hands-on** and **exercise-driven**. Rather than focusing only on theoretical aspects, it guides the student through practical tasks that gradually build the skills needed to implement AI methods. Each chapter contains:

- **Conceptual explanation:** a short introduction to the idea or method.
- **Code examples:** illustrated in Python, using widely used libraries.
- **Exercises:** tasks that allow students to practice and apply the concepts immediately.

The workbook begins with Python programming fundamentals, then covers data handling, statistical and machine learning methods, and finally introduces deep

learning. The order is intentional: starting with the basics ensures that students without a strong programming background can follow the material, while advanced topics connect directly to modern AI applications.

It is recommended to solve the exercises while coding in a Python environment such as Jupyter Notebook, Google Colab, or a local IDE (PyCharm, VS Code, etc.). Hands-on practice will significantly improve understanding and retention.

Chapter 2

Python Basics

2.1 Variables and Data Types

In Python, variables are used to store data. Data types determine the type of data that can be stored in a variable.

2.2 Basic Data Types

Python has several basic data types that are commonly used to represent different kinds of data, such as:

- string
- integer
- float
- boolean

Example - Initializing variables

```
name = "Mary"           # string
age = 25                 # integer
height = 5.8             # float
is_student = True        # boolean
```

In this example, *name*, *age*, *height*, and *is_student* are variables that store different types of data.

The data types in this example are string (*name*), integer (*age*), float (*height*), and boolean (*is_student*).

Example - Printing variables

```
print("Name:", name)
print("Age:", age)
print("Height:", height)
print("Is Student:", is_student)
```

Output:

Name: Mary
Age: 25
Height: 5.8
Is Student: True

You can get the data type of any object by using the `type()` function.

Example - `type()` function

```
print(type(name))  
print(type(age))  
print(type(height))  
print(type(is_student))
```

Output:

```
<class 'str'>  
<class 'int'>  
<class 'float'>  
<class 'bool'>
```

We can concatenate strings using the `+` operator.

Example - Concatenating strings

```
name = "Mary"  
message = "Hello, " + name + "!"  
print(message)
```

Output:

Hello, Mary!

2.3 Built-in Data Types

The 4 built-in data types in Python that are used to store collections of data include Lists, Tuples, Set, and Dictionary, all with different qualities and usage.

There is a special type, `NoneType` (`None`), used to indicate the absence of a value.

2.3.1 Lists

Lists are used to store multiple items in a single variable.

Definition: A list in Python is an ordered collection of items. Each item can be of any data type (e.g., numbers, strings, other objects). Lists are mutable, meaning you can change their content (add, modify, or remove items).

Characteristics:

- Ordered: Items in a list have a specific order, and you can access them by their indices (position in the list).
- Mutable: You can modify the items in a list after it's created.
- Allows Duplicates: Lists can contain duplicate items.

Example - Lists

```
programming_language = ["python", "java", "C"]
print("Programming languages:", programming_language)

# Accessing list items
first_programming_language = programming_language[0]
print("First programming language:", first_programming_language)
print("Variable type:", type(programming_language))
```

Output:

```
Programming languages: ['python', 'java', 'C']
First programming language: python
Variable type: <class 'list'>
```

`programming_language` is a list containing three string items.

Lists are indexed, and we can access items using indices (0-based).

Example - Creating empty list using `list()` constructor

```
empty_list1 = list()
empty_list1
```

Output:

```
[]
```

Example - Creating empty list using square brackets `[]`

```
empty_list2 = []
empty_list2
```

Output:

```
[]
```

2.3.2 Tuples

Tuples are used to store multiple items in a single variable.

A tuple is a collection that is ordered and unchangeable, or immutable as it also is called.

Tuples are written with round brackets.

Definition: A tuple is similar to a list, but it is immutable, meaning you cannot change its content after creation. Tuples are typically used when you want to group related items together.

Characteristics:

- Ordered: Like lists, items in a tuple have a specific order and can be accessed by indices.
- Immutable: Once created, the elements in a tuple cannot be modified, added, or removed.
- Allows Duplicates: Tuples can contain duplicate items.

Example - Tuple

```
point = (3, 4)
print("Point:", point)
print(type(point))
```

Output:

```
Point: (3, 4)
<class 'tuple'>
```

point is a tuple containing two integers.

2.3.3 Sets

Sets are used to store unique items without duplicates.

Definition: A set is an unordered collection of unique items. Sets are used when you want to store multiple items without duplicates and don't care about their order.

Characteristics:

- Unordered: Items in a set have no specific order, so you can't access them by indices.

- **Mutable** (in terms of adding/removing items): You can add or remove items from a set.
- **Unique Elements**: Sets automatically remove duplicates, ensuring each item is unique.

Example - Sets

```
# Sets
colors = {"red", "green", "blue", "blue"}
print("Colors:", colors)

# Adding an item to a set
colors.add("yellow")
print("Updated Colors:", colors)
```

Output:

```
Colors: {'red', 'blue', 'green'}
Updated Colors: {'red', 'blue', 'green', 'yellow'}
```

colors is a set containing unique string items. Sets are defined using curly braces

2.3.4 Dictionaries

Dictionaries store key-value pairs.

Definition: A dictionary is an unordered collection of key-value pairs. Each key is associated with a value, and you can use the key to look up the corresponding value. Dictionaries are used to store and retrieve data using keys.

Characteristics:

- **Unordered**: Like sets, dictionaries have no specific order for their key-value pairs.
- **Mutable**: You can add, modify, or remove key-value pairs in a dictionary.
- **Key-Value Pairs**: Data is stored in the form of key-value pairs, where each key is unique within a dictionary.

Example - Dictionaries

```
student = {"name": "Mary", "age": 25, "grade": "A"}
print("Student:", student)
```

```
# Accessing dictionary values by key
student_name = student["name"]
print("Student Name:", student_name)
```

Output:

```
Student: {'name': 'Mary', 'age': 25, 'grade': 'A'}
Student Name: Mary
```

`student` is a dictionary with key-value pairs. Dictionaries are defined using curly braces with key-value pairs separated by colons ":".

2.4 Built-in Functions for Operations with Data Types

Operations with data types in Python involve various built-in functions and methods to manipulate and work with data effectively.

2.4.1 Built-in Functions for Operations with Strings

Most commonly used built-in functions for operations with strings:

`len(string)`: Returns the length (number of characters) of a string.

`string.upper()`: Converts the string to uppercase.

`string.lower()`: Converts the string to lowercase.

`string.strip()`: Removes leading and trailing whitespace.

`string.split(delimiter)`: Splits the string into a list of substrings based on the delimiter.

`string.replace(old, new)`: Replaces occurrences of a substring with another substring.

`string.find(substring)`: Searches for a substring and returns its index or -1 if not found.

Example - `len()`, `upper()`

```
text = "Hello, World!"
length = len(text) # Returns 13
uppercase_text = text.upper() # "HELLO, WORLD!"
print(length)
print(uppercase_text)
```

Output:

```
13
HELLO, WORLD!
```

2.4.2 Built-in Functions for Operations with Lists

Most commonly used built-in functions for operations with lists:

- `len(list)`: Returns the number of elements in a list.
- `list.append(item)`: Adds an item to the end of the list.
- `list.insert(index, item)`: Inserts an item at a specific index in the list.
- `list.remove(item)`: Removes the first occurrence of the specified item.
- `list.pop(index)`: Removes and returns the item at the specified index.
- `list.sort()`: Sorts the list in ascending order.
- `list.reverse()`: Reverses the order of elements in the list.

Example - `append()`, `sort()`

```
numbers = [3, 1, 4, 1, 5, 9, 2]
numbers.append(6) # [3, 1, 4, 1, 5, 9, 2, 6]
numbers.sort()   # [1, 1, 2, 3, 4, 5, 6, 9]
```

2.4.3 Built-in Functions for Operations with Tuples

Most commonly used built-in functions for operations with tuples:

- `len(tuple)`: Returns the number of elements in a tuple.
- `tuple[index]`: Accesses an element at a specific index.
- `tuple.count(item)`: Counts the number of occurrences of an item in the tuple.

Example - Accessing the first element of tuple

```
point = (3, 4)
x = point[0]
```

2.4.4 Built-in Functions for Operations with Sets

Most commonly used built-in functions for operations with sets: `len(set)`: Returns the number of elements in a set.

- `set.add(item)`: Adds an item to the set.
- `set.remove(item)`: Removes an item from the set.

`set.discard(item)`: Removes an item if it exists in the set, without raising an error.

`set.union(other_set)`: Returns the union of two sets.

`set.intersection(other_set)`: Returns the intersection of two sets.

`set.difference(other_set)`: Returns the elements in the set that are not in the other set.

Example - add()

```
colors = {"red", "green", "blue"}
colors.add("yellow")
colors
```

Output:

```
{'blue', 'green', 'red', 'yellow'}
```

Example - remove()

```
colors.remove('orange')
colors
```

Output:

```
-----
KeyError    Traceback (most recent call last)
<ipython-input-35-8f5c5a0abd98> in <cell line: 1>()
----> 1 colors.remove('orange')
```

KeyError: 'orange'

The above code gives an error because the "orange" is not in the set of "colors".

Example - discard()

```
colors.discard('orange')
colors
```

Output:

```
{'blue', 'green', 'red', 'yellow'}
```

2.4.5 Built-in Functions for Operations with Dictionaries

Most commonly used built-in functions for operations with dictionaries:

- `len(dict)`: Returns the number of key-value pairs in a dictionary.
- `dict[key]`: Accesses the value associated with a specific key.
- `dict.keys()`: Returns a list of all keys.
- `dict.values()`: Returns a list of all values.
- `dict.items()`: Returns a list of key-value pairs as tuples.

Example - Accessing value with key

```
student = {"name": "Mary", "age": 25}
student_name = student["name"]
student_name
```

Output:

Mary

2.5 Fundamental Built-in Functions in Python

Python provides a wide range of built-in functions that are commonly used in general programming. Here are some of the fundamental built-in functions in Python:

`range()`: Generates a sequence of numbers within a specified range, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.

`zip()`: Used for combining multiple iterables (e.g., lists, tuples, or other sequences) element-wise into a single iterable. It pairs elements from each input iterable together, creating a new iterable of tuples where each tuple contains elements from the corresponding positions in the input iterables.

`enumerate()`: Used to iterate over an iterable (such as a list, tuple, or string) while keeping track of both the elements and their respective indices. It returns an iterator that generates pairs of index and element values as you loop through the iterable.

`sum()`: Calculates the sum of elements in an iterable.

`min()`: Find the minimum value in an iterable.

`max()`: Find the maximum value in an iterable.

`abs()`: Returns the absolute value of a number.

`round()`: Rounds a floating-point number to the nearest integer or a specified number of decimal places.

`str()`, `int()`, `float()`: Used for type conversion between data types.

Iterable can be anything lists, tuples, or dictionaries, but most importantly it should be numbers.

Example - range()

```
numbers = list(range(1, 6)) # Generates [1, 2, 3, 4, 5]
```

Example - range()

```
range_ = range(3) # Note: stop parameter is required
```

Example - Sum of range

```
number_range = range(1, 10, 2)  
total = sum(number_range)
```

Example - Converting range to list

```
number_range = list(number_range)
```

Example - Range with empty sequence

```
len(range(1, 0))
```

Example - Data type conversion

```
number_as_string = "42"  
number_as_int = int(number_as_string)
```

Example - zip()

```
names = ['Mary', 'John', 'Matthew']  
scores = [85, 92, 78]  
  
zipped_data = zip(names, scores)  
  
for name, score in zipped_data:
```

```
print(name, score)
```

Output:

```
Mary 85  
John 92  
Matthew 78
```

In this example, `zip()` combines the `names` and `scores` lists element-wise, creating an iterable of tuples. Then, a for loop is used to iterate through the zipped data and print out the pairs.

It's important to note that if the input iterables are of different lengths, `zip()` will only combine elements up to the length of the shortest iterable. Any remaining elements in longer iterables will be ignored.

You can also use the `zip()` function to unzip data by applying the `zip(*zipped_data)` pattern, which essentially transposes the data back into separate iterables.

Example - Converting zipped data to original form

```
zipped_data = [('Mary', 85), ('John', 92), ('Matthew', 78)]  
  
names, scores = zip(*zipped_data)  
  
print(names)    # ('Mary', 'John', 'Matthew')  
print(scores)   # (85, 92, 78)
```

This allows you to easily convert the zipped data back into its original form.

Example - Converting zipped data to list

```
programming_languages = ['Java', 'Python', 'JavaScript']  
versions = [14, 3, 6]  
  
result = zip(programming_languages, versions)  
result = list(result)  
print(result)
```

Output:

```
[('Java', 14), ('Python', 3), ('JavaScript', 6)]
```

Example - enumerate

```
fruits = ['apple', 'banana', 'cherry', 'orange']

for i in enumerate(fruits):
    print(i)
```

Output:

```
(0, 'apple')
(1, 'banana')
(2, 'cherry')
(3, 'orange')
```

Example - enumerate

```
fruits = ['apple', 'banana', 'cherry', 'orange']

for index, fruit in enumerate(fruits):
    print(f"Index {index}: {fruit}")
```

Output:

```
Index 0: apple
Index 1: banana
Index 2: cherry
Index 3: orange
```

2.6 Loops

Loops are used to repeat a block of code multiple times.

2.7 For Loop

In Python, a for loop is a control structure that allows you to iterate (loop) over a sequence of elements, such as a list, tuple, string, or other iterable objects. The for loop is used to repeatedly execute a block of code for each item in the sequence until the sequence is exhausted.

The basic syntax of a for loop in Python:

Syntax - for loop

```
for element in sequence:
    # Code to be executed for each element
    # ...
```

Example - for loop

```
for i in range(5):
    print("Iteration", i)
```

Output:

```
Iteration 0
Iteration 1
Iteration 2
Iteration 3
Iteration 4
```

The for loop here iterates over the values from 0 to 4 (5 iterations), `range(5)` generates a sequence of numbers from 0 to 4.

Example - Decreasing range

```
for i in range(5, 0, -1): # Starts at 5, ends before 0, counts backward
    print(i)
```

Example - Empty sequence

```
for i in range(1, 0):
    print(i)
```

Example - Decreasing sequence

```
my_range = range(0, -10, -1)
for num in my_range:
    print(num, end=' ')
```

Output:

```
0 -1 -2 -3 -4 -5 -6 -7 -8 -9
```

Example - Decreasing sequence

```
my_range = range(0, -10, -2)
for num in my_range:
    print(num, end=' ')
```

Output:

0 -2 -4 -6 -8

2.8 While Loop

In Python, a while loop is a control structure that allows you to repeatedly execute a block of code as long as a specified condition remains True. Unlike for loops, which iterate over a sequence of elements, while loops continue executing as long as the given condition is met.

The basic syntax of a while loop in Python:

Syntax - while loop

```
while condition:
    # Code to be executed while the condition is True
    # ...
```

Example - while loop

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

Output:

Count: 0
Count: 1
Count: 2
Count: 3
Count: 4

The while loop continues to execute as long as the condition `count < 5` is True. The count variable is incremented in each iteration.

2.9 Functions

Functions are blocks of code that can be reused by calling them with different inputs.

In Python, you can define a function using the `def` keyword, followed by the function name, a pair of parentheses for the function's parameters, a colon, and an indented block of code that represents the function's body. The basic syntax for defining a Python function is as follows:

Syntax - Defining function

```
def function_name(parameters):  
    """  
    Docstring: Optional documentation for the function.  
    """  
    # Function body (code to be executed when the function is called)  
    # ...  
    return return_value # Optional return statement
```

Example - Defining and calling function

```
# Function definition  
def greet(name):  
    """  
    This function greets a person by name.  
    """  
    message = "Hello, " + name + "!"  
    return message  
  
# Function call  
person_name = "John"  
greeting = greet(person_name)  
print(greeting)
```

Output:

Hello, John!

2.10 If Statement

If statements are used for conditional execution of code.

Example - if, else

```
# If Statement
x = 10
if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")
```

Output:

```
x is greater than 5
```

The if statement checks the condition $x > 5$.

If the condition is True, the code block under if is executed; otherwise, the code block under else is executed.

Example - if, elif, else

```
x = 10
y = 5

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")
```

Output:

```
x is greater than 5
```

2.11 List Comprehension

List comprehension is a concise and efficient way to create lists in Python. It allows you to create a new list by applying an expression to each item in an existing iterable (e.g., a list, tuple, string, or range) and optionally filtering items based on a condition. List comprehensions are a powerful and readable way to perform operations on lists in a single line of code.

The basic syntax of a list comprehension is as follows:

Syntax - List Comprehension

```
new_list = [expression for item in iterable if condition]
```

- **expression:** This is the operation or transformation you want to apply to each item in the iterable to create the elements of the new list.
- **item:** This is a variable that takes on the value of each item in the iterable one by one.
- **iterable:** This is the source iterable (e.g., a list) that you want to loop through.
- **condition (optional):** This is an optional filter that allows you to include only items that meet a certain condition in the new list. If omitted, all items from the iterable will be included.

Example - List Comprehension

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

Output:

```
[1, 4, 9, 16, 25]
```

Example - List Comprehension with Range

```
my_range = range(5)
double_range = [x * 2 for x in my_range]
print(double_range)
```

Output:

```
[0, 2, 4, 6, 8]
```

2.12 Lambda Function

A lambda function in Python, also known as an anonymous function, is a small, unnamed function defined using the `lambda` keyword. Lambda functions are used when you need a simple, one-line function for a short period of time and don't want to define a formal function using the `def` keyword. Lambda functions can be assigned to variables and used like regular function

The basic syntax of a lambda function is as follows:

Syntax - Lambda

```
lambda arguments: expression
```

- **lambda**: This keyword is used to define a lambda function.
- **arguments**: These are the input parameters or arguments that the lambda function takes. You can have zero or more arguments separated by commas.
- **expression**: This is a single expression that the lambda function evaluates and returns as its result. It is typically a simple operation or computation.

Lambda functions are often used in situations where you need to pass a small function as an argument to a higher-order function like `map()`, `filter()`, or `sorted()`. They are particularly useful for tasks like data transformation, filtering, and sorting.

Here's an example of a lambda function that calculates the square of a number:

Example - Lambda

```
square = lambda x: x ** 2
result = square(5)  # Calling the lambda function
print(result)
```

Output:

25

Example - Lambda

```
list_of_range = lambda x: list(range(x))
result = list_of_range(5)  # Calling the lambda function
print(result)
```

Output:

[0, 1, 2, 3, 4]

Example - Lambda function with more arguments

```
product_of_two_numbers = lambda x,y: x * y
result = product_of_two_numbers(5,6)  # Calling the lambda function
print(result)
```

Output:

30

2.13 Exercise Set

2.13.1 Exercise 1: Working with Lists and Functions

Write Python code that calculates the sum of all even numbers in a given list.

- Implement a function called `sum_even_numbers` that takes a list of integers as input and returns the sum of all even numbers in the list.
- Use a for loop to iterate through the list.
- Use an if statement to check if a number is even.
- Test your function with different lists to ensure it works correctly.

Exercise 1: Sum of Even Numbers

```
# Function to calculate the sum of even numbers in a list
def sum_even_numbers(numbers):
    even_sum = 0
    for num in numbers:
        if num % 2 == 0:
            even_sum += num
    return even_sum

# Test the function
number_list = [1, 2, 3, 4, 5, 6]
result = sum_even_numbers(number_list)
print("Sum of even numbers:", result)
```

Output:

Sum of even numbers: 12

2.13.2 Exercise 2: Working with Strings and Functions

Write Python code that counts the number of vowels (*a, e, i, o, u*) in a given string.

- Implement a function called `count_vowels` that takes a string as input and returns the count of vowels in the string.
- Use a for loop to iterate through the characters in the string.

- Use an if statement to check if a character is a vowel.
- Test your function with different strings to ensure it works correctly.

Exercise 2: Counting Vowels

```
# Function to count the number of vowels in a string
def count_vowels(text):
    vowels = "aeiouAEIOU"
    count = 0
    for char in text:
        if char in vowels:
            count += 1
    return count

# Test the function
input_string = "Hello, World!"
vowel_count = count_vowels(input_string)
print("Number of vowels:", vowel_count)
```

Output:

Number of vowels: 3

2.13.3 Exercise 3: Working with Dictionaries and Functions

Write Python code that finds the key with the maximum value in a given dictionary.

- Implement a function called `find_max_key` that takes a dictionary as input and returns the key with the maximum value.
- Use a for loop to iterate through the dictionary's items.
- Use a variable to keep track of the key with the maximum value.
- Test your function with different dictionaries to ensure it works correctly.

Exercise 3: Finding Max Key in Dictionary

```
# Function to find the key with the maximum value in a dictionary
def find_max_key(data_dict):
    max_key = None
    max_value = float('-inf')
    for key, value in data_dict.items():
        if value > max_value:
            max_key = key
```

```
        max_value = value
    return max_key

# Test the function
sample_dict = {"apple": 5, "banana": 8, "cherry": 3, "date": 10}
max_key_result = find_max_key(sample_dict)
print("Key with maximum value:", max_key_result)
```

Output:

Key with maximum value: date

2.13.4 Exercise 4: Working with Sets and Functions

Write Python code that finds the intersection of two sets.

- Implement a function called `find_intersection` that takes two sets as input and returns their intersection as a new set.
- Use the intersection method of sets to perform the operation.
- Test your function with different sets to ensure it works correctly.

Exercise 4: Finding Intersection of Sets

```
# Function to find the intersection of two sets
def find_intersection(set1, set2):
    intersection = set1.intersection(set2)
    return intersection

# Test the function
set_a = {1, 2, 3, 4, 5}
set_b = {3, 4, 5, 6, 7}
intersection_result = find_intersection(set_a, set_b)
print("Intersection of sets:", intersection_result)
```

Output:

Intersection of sets: {3, 4, 5}

2.13.5 Exercise 5: Working with Numbers and Functions

Write Python code that calculates the factorial of a given integer.

- Implement a function called `calculate_factorial` that takes an integer as input and returns its factorial.

- Use a for loop to calculate the factorial.
- Test your function with different integers to ensure it works correctly.

Exercise 5: Factorial of given integer

```
def calculate_factorial(n):  
    if n == 0:  
        return 1  
    factorial = 1  
    for i in range(1, n + 1):  
        factorial *= i  
    return factorial  
  
# Test the function  
num = 5  
result_factorial = calculate_factorial(num)  
print("Factorial of", num, "is", result_factorial)
```

Output:

Factorial of 5 is 120

2.13.6 Exercise 6: Working with zip() and for loop

Write Python code to analyze student test scores. You will use the `zip()` function and a `for` loop to work with two lists containing student names and their test scores.

- Create two lists, `students` and `test_scores`, containing the following data:
`students = ["Alice", "Bob", "Charlie", "David", "Eva"]`; `test_scores = [85, 92, 78, 88, 90]`
- Use the `zip()` function to combine the `students` and `test_scores` lists into a list of tuples, where each tuple contains a student's name and their corresponding test score.
- Write a `for` loop to iterate over the list of tuples created in step 2 and perform the following tasks within the loop: Calculate and print the average test score of the students; Identify and print the name of the student with the highest test score.; Determine and print the number of students who scored above 85.
- Create a new list, `adjusted_scores`, by adding 5 to each test score in the `test_scores` list using a `for` loop.
- Print the `adjusted_scores` list.

Exercise 6: Student Test Scores Analysis

```
# Step 1: Create two lists
students = ["Alice", "Bob", "Charlie", "David", "Eva"]
test_scores = [85, 92, 78, 88, 90]

# Step 2: Combine lists using zip()
student_scores = list(zip(students, test_scores))

# Step 3: Initialize variables
total_scores = 0
highest_score = 0
top_student = ""
above_85_count = 0

# Step 4: Iterate over student_scores and perform tasks
for student, score in student_scores:
    total_scores += score

    if score > highest_score:
        highest_score = score
        top_student = student

    if score > 85:
        above_85_count += 1

# Calculate the average test score
average_score = total_scores / len(student_scores)

# Print the results
print(f"Average Test Score: {average_score:.2f}")
print(f"Top Student: {top_student} (Score: {highest_score})")
print(f"Number of Students Above 85: {above_85_count}")

# Step 5: Create adjusted_scores and print
adjusted_scores = [score + 5 for score in test_scores]
print("Adjusted Test Scores:", adjusted_scores)
```

Output:

```
Average Test Score: 86.60
Top Student: Bob (Score: 92)
Number of Students Above 85: 3
```

Adjusted Test Scores: [90, 97, 83, 93, 95]

2.13.7 Exercise 7: List Comprehension

In this exercise, you will work with list comprehension to perform various tasks on data.

- Create a list called `numbers` containing the integers from 1 to 10 (inclusive).
- Use list comprehension to create a new list called `squared_numbers` that contains the squares of each number in the `numbers` list.
- Use list comprehension to create a new list called `even_numbers` that contains only the even numbers from the `numbers` list.
- Create a list called `words` containing the following strings: `words = ["apple", "banana", "cherry", "date", "elderberry"]`
- Use list comprehension to create a new list called `long_words` that contains only the words from the `words` list that have a length of 6 or more characters.
- Create a list called `temperatures` containing the following temperature values in Celsius: `temperatures = [20, 25, 30, 35, 40]`
- Use list comprehension to create a new list called `temperatures_fahrenheit` that converts the temperatures from Celsius to Fahrenheit using the formula: $F = (C * 9/5) + 32$, where F is the temperature in Fahrenheit and C is the temperature in Celsius.

Exercise 7: List Comprehension

```
# Step 1: Create a list of numbers from 1 to 10
numbers = [i for i in range(1, 11)]

# Step 2: Create a list of squared numbers using list comprehension
squared_numbers = [num ** 2 for num in numbers]

# Step 3: Create a list of even numbers using list comprehension
even_numbers = [num for num in numbers if num % 2 == 0]

# Step 4: Create a list of words
words = ["apple", "banana", "cherry", "date", "elderberry"]

# Step 5: Create a list of long words (length >= 6) using list
↪ comprehension
long_words = [word for word in words if len(word) >= 6]
```

```
# Step 6: Create a list of temperatures in Celsius
temperatures = [20, 25, 30, 35, 40]

# Step 7: Convert temperatures to Fahrenheit using list comprehension
temperatures_fahrenheit = [(temp * 9/5) + 32 for temp in temperatures]

# Print the results
print("Original Numbers:", numbers)
print("Squared Numbers:", squared_numbers)
print("Even Numbers:", even_numbers)
print("Original Words:", words)
print("Long Words:", long_words)
print("Temperatures in Celsius:", temperatures)
print("Temperatures in Fahrenheit:", temperatures_fahrenheit)
```

Output:

```
Original Numbers: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Squared Numbers: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
Even Numbers: [2, 4, 6, 8, 10]
Original Words: ['apple', 'banana', 'cherry', 'date', 'elderberry']
Long Words: ['banana', 'cherry', 'elderberry']
Temperatures in Celsius: [20, 25, 30, 35, 40]
Temperatures in Fahrenheit: [68.0, 77.0, 86.0, 95.0, 104.0]
```

Chapter 3

NumPy

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, basic statistical operations, random simulation, etc.

NumPy is a cornerstone of the Python scientific ecosystem and is often used in conjunction with other libraries like SciPy, Matplotlib, and pandas for various data analysis and scientific computing tasks.

NumPy documentation: <https://numpy.org/>.

NumPy is widely used in fields such as scientific research, data analysis, machine learning, and engineering for tasks involving numerical computations. It forms the foundation for many other Python libraries and tools that rely on numerical data processing. To use NumPy in your Python projects, you typically need to import it like this:

Importing library

```
import numpy as np
```

3.1 NumPy Arrays

The fundamental data structure in NumPy is the ndarray (n-dimensional array).

Example - Creating array from list

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
```

Output:

```
[1 2 3 4 5]
```

Example - Creating multi-dimensional array by using list comprehension

```
np.array([range(i, i + 3) for i in [2, 4, 6]])
```

Output:

```
array([[2, 3, 4],
       [4, 5, 6],
       [6, 7, 8]])
```

Example - Various ways to create NumPy arrays

```
arr_zeros = np.zeros(10, dtype=int)
print('length-10 integer array filled with zeros:\n', arr_zeros, '\n')

arr_ones = np.ones((3, 5), dtype=float)
print('3x5 floating-point array filled with ones:\n', arr_ones, '\n')

arr_full = np.full((3, 5), 3.14)
print('3x5 array filled with 3.14:\n', arr_full, '\n')

arr_full = np.eye(3)
print('3x3 identity matrix:\n', arr_full, '\n')

arr_empty = np.empty(3)
print('uninitialized array of three integers:\n', arr_empty, '\n')
```

Output:

```
length-10 integer array filled with zeros:
[0 0 0 0 0 0 0 0 0 0]
```

```
3x5 floating-point array filled with ones:
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

```
3x5 array filled with 3.14:
[[3.14 3.14 3.14 3.14 3.14]
 [3.14 3.14 3.14 3.14 3.14]
 [3.14 3.14 3.14 3.14 3.14]]
```

3x3 identity matrix:

```
[[1. 0. 0.]  
 [0. 1. 0.]  
 [0. 0. 1.]]
```

uninitialized array of three integers:

```
[1. 1. 1.]
```

The `arange()` function is similar to the built-in `range()` function, it returns evenly spaced values within a given interval.

Syntax - `arange()`

```
numpy.arange(start = 0, stop, step = 1, dtype = None)
```

Example - `np.arange`

```
# Starting at 0, ending at 20, stepping by 2  
np.arange(0, 20, 2)
```

Output:

```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

The `linspace()` function is used to create an array of evenly spaced numbers within a specified range.

Syntax - `linspace()`

```
numpy.linspace(start, stop, num=50, endpoint=True, retstep=False,  
↪ dtype=None)
```

Example - `np.linspace`

```
# array of five values evenly spaced between 0 and 1  
np.linspace(0, 1, 5)
```

Output:

```
array([0. , 0.25, 0.5 , 0.75, 1.  ])
```

3.1.1 Array Information Functions

Basic NumPy attributes for getting information about arrays:

- `ndarray.shape`: Get the dimensions of the array (rows, columns, etc.).
- `ndarray.ndim`: Get the number of dimensions of the array.
- `ndarray.size`: Get the total number of elements in the array.
- `ndarray.dtype`: Get the data type of the elements in the array.
- `ndarray.itemsize`: Get the size (in bytes) of each element in the array.

Examples of getting information from an array:

Example - Multidimensional array

```
import numpy as np

# Create a NumPy array
arr = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])
```

Example - shape

```
shape = arr.shape
print("Shape of the array:", shape)
```

Output:

Shape of the array: (3, 3)

Example - ndim

```
# Using ndarray.ndim to get the number of dimensions
dimensions = arr.ndim
print("Number of dimensions:", dimensions) # Output: 2
```

Output:

Shape of the array: (3, 3)

Example - size

```
# Using ndarray.size to get the total number of elements
element_count = arr.size
print("Total number of elements:", element_count)
```

Output:

Total number of elements: 9

Example - dtype

```
# Using ndarray.dtype to get the data type of elements
data_type = arr.dtype
print("Data type of elements:", data_type)
```

Output:

Data type of elements: int64

Example - itemsize

```
# Using ndarray.itemsize to get the size (in bytes) of each element
element_size = arr.itemsize
print("Size (in bytes) of each element:", element_size) # Output: 8
↪ (for int64, as 64 bits or 8 bytes)
```

Output:

Size (in bytes) of each element: 8

3.1.2 Array Reshaping and Manipulation Functions

Array reshaping and manipulation in NumPy involves changing the shape or structure of a NumPy array without modifying the data it contains. This is a fundamental operation when working with arrays, as it allows you to prepare data for various computations or adapt it to the requirements of specific algorithms.

- `ndarray.reshape()`: Reshape an array into a new shape.
- `ndarray.flatten()`: Flatten an array into a 1D array.
- `np.transpose()`, `ndarray.T`: Transpose an array.
- `np.concatenate()`: Concatenate multiple arrays along specified axes.
- `np.vstack()`: Stack arrays vertically (along the rows).

- `np.hstack()`: Stack arrays horizontally (along the columns).
- `np.split()`: Split an array into multiple subarrays along a specified axis.
- `np.hsplit()`: Split a 2D array horizontally.
- `np.vsplit()`: Split a 2D array vertically.

Example - Reshaping array

```
arr = np.array([1, 2, 3, 4, 5, 6])
reshaped_arr = arr.reshape(2, 3) # Reshape to a 2x3 array
print(reshaped_arr)
```

Output:

```
[[1 2 3]
 [4 5 6]]
```

Example - Flattening an array

```
flat_arr = reshaped_arr.flatten()
print(flat_arr)
```

Output:

```
[1 2 3 4 5 6]
```

Example - Transposing an array

```
transposed_arr = reshaped_arr.T # Transpose the 2x3 array
print(transposed_arr)
```

Output:

```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

The `np.transpose()` function in the NumPy module can be used to transpose an array as well, similarly to `ndarray.T`. The primary difference is that `np.transpose()` can accept an optional argument, `axes`, which allows you to specify the axes to be transposed.

Example - Transposing an array using np.transpose()

```
transposed_arr = np.transpose(reshaped_arr, axes=(1,0))
transposed_arr
```

Output:

```
array([[1, 4],
       [2, 5],
       [3, 6]])
```

Example - Concatenating arrays

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
concatenated_arr = np.concatenate((arr1, arr2)) # Concatenate along
↪ the default axis 0 (rows)
print(concatenated_arr)
```

Output:

```
array([1, 2, 3, 4, 5, 6])
```

Example - Stacking arrays

```
arr1 = np.array([1, 2])
arr2 = np.array([3, 4])
vertically_stacked = np.vstack((arr1, arr2)) # Stack vertically
print('Vertically stacked array:\n', vertically_stacked)

horizontally_stacked = np.hstack((arr1, arr2)) # Stack horizontally
print('Horizontally stacked array:\n', horizontally_stacked)
```

Output:

Vertically stacked array:

```
[[1 2]
 [3 4]]
```

Horizontally stacked array:

```
[1 2 3 4]
```

`np.split`, `np.hsplit`, and `np.vsplit` are NumPy functions used for splitting arrays along different axes. Here are the key differences between them:

- `np.split`: This function is a more general method for splitting arrays along specified axis. It allows you to specify the number of equally-sized subarrays or the specific indices at which to split the array along a given axis.
- `np.hsplit`: This function is specifically designed for horizontal splitting along the second axis (`axis=1`). It is equivalent to `np.split` with `axis=1`. It splits an array into subarrays along the horizontal axis.
- `np.vsplit`: This function is specifically designed for vertical splitting along the first axis (`axis=0`). It is equivalent to `np.split` with `axis=0`. It splits an array into subarrays along the vertical axis.

Example - Splitting arrays by `np.split`

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
# Split into three equal parts
result = np.split(arr, 3)
print(result)
```

Output:

```
[array([1, 2, 3]), array([4, 5, 6]), array([7, 8, 9])]
```

Example - Splitting arrays by `np.hsplit`

```
arr = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])

# Split horizontally
result = np.hsplit(arr, 3)
print(result)
```

Output:

```
[array([[1],
        [4],
        [7]]), array([[2],
        [5],
        [8]]), array([[3],
        [6],
        [9]])]
```

```
[9]])]
```

Example - Splitting arrays by np.vsplit

```
arr = np.array([[1, 2, 3],  
               [4, 5, 6],  
               [7, 8, 9]])
```

```
# Split vertically  
result = np.vsplit(arr, 3)  
print(result)
```

Output:

```
[array([[1, 2, 3]]), array([[4, 5, 6]]), array([[7, 8, 9]])]
```

3.2 Random sampling

The `numpy.random` module implements pseudo-random number generators with the ability to draw samples from a variety of probability distributions.

Example - Random integer between a specified range

```
random_int = np.random.randint(1, 11) # Generates a random integer  
↪ between 1 and 10 (inclusive)  
print("Random Integer:", random_int)
```

Output:

```
Random Integer: 2
```

Example - Array of random integers with specific size

```
random_integers = np.random.randint(1, 101, size=5) # Generates an  
↪ array of 5 random integers between 1 and 100  
print("Random Integers Array:", random_integers)
```

Output:

```
Random Integers Array: [97 68 45 99 45]
```

Example - Random float between 0 and 1

```
random_float = np.random.rand()
print("Random Float between 0 and 1:", random_float)
```

Output:

Random Float between 0 and 1: 0.0301203495224317

Example - Array of random float between 0 and 1

```
random_floats = np.random.rand(5)  # Generates an array of 5 random
↳ floats between 0 and 1
print("Random Floats Array between 0 and 1:", random_floats)
```

Output:

Random Floats Array between 0 and 1:
[0.68534915 0.96283918 0.87293053 0.43772285 0.26360645]

Example - Random floats from a Gaussian (normal) distribution

```
mean = 0
std_deviation = 1
random_normal = np.random.normal(mean, std_deviation)  # Generates a
↳ random float from a normal distribution
print("Random Normal Float:", random_normal)
```

Output:

Random Normal Float: -1.4042822616581505

Example - Array of random floats from a Gaussian (normal) distribution

```
random_normals = np.random.normal(mean, std_deviation, size=5)  #
↳ Generates an array of 5 random floats from a normal distribution
print("Random Normal Floats Array:", random_normals)
```

Output:

Random Normal Floats Array:
[0.52845209 0.71718004 1.31563386 0.00142485 -0.84052511]

Example - Shuffling elements in an array randomly

```
original_array = np.array([1, 2, 3, 4, 5])
shuffled_array = np.random.permutation(original_array) # Shuffles the
↪ elements randomly
print("Shuffled Array:", shuffled_array)
```

Output:

Shuffled Array: [4 1 5 3 2]

Example - Seed for reproducibility

```
# set a seed to get the same random numbers every time
np.random.seed(42) # Setting the seed to 42
random_int_42 = np.random.randint(1, 101) # Generates a random integer
print("Random Integer with Seed 42:", random_int_42)
```

Output:

Random Integer with Seed 42: 52

Example - Shuffling elements by the same seed

```
# Reshuffle with the same seed (to get the same shuffle order)
shuffled_array_42 = np.random.permutation(original_array) # Shuffles
↪ the elements using the same seed
print("Shuffled Array with Seed 42:", shuffled_array_42)
```

Output:

Shuffled Array with Seed 42: [1 2 4 3 5]

Example - np.random.choice()

```
# Example 1: Randomly choose elements from a 1-D array
colors = np.array(['red', 'green', 'blue', 'yellow', 'orange'])

# Randomly choose one color
random_color = np.random.choice(colors)
print("Randomly chosen color:", random_color)
```

```
# Randomly choose three colors with replacement
random_colors = np.random.choice(colors, size=3)
print("Randomly chosen colors with replacement:", random_colors)

# Randomly choose three colors without replacement
random_unique_colors = np.random.choice(colors, size=3, replace=False)
print("Randomly chosen unique colors:", random_unique_colors)

# Example 2: Randomly choose integers within a specified range
random_integers = np.random.choice(10, size=5)
print("Randomly chosen integers:", random_integers)
```

Output:

```
Randomly chosen color: blue
Randomly chosen colors with replacement: ['blue' 'red' 'green']
Randomly chosen unique colors: ['green' 'yellow' 'orange']
Randomly chosen integers: [4 5 8 0 0]
```

3.3 Slicing and Indexing

Array indexing and slicing are essential techniques for working with multidimensional arrays (ndarrays). NumPy provides a powerful and efficient way to access, manipulate, and extract data from these arrays.

Indexing Array indexing allows you to access individual elements within a NumPy array. NumPy arrays are zero-indexed, which means the first element is accessed with index 0, the second with index 1, and so on. You can use square brackets ‘[]’ to index into an array.

Example - Indexing

```
arr = np.array([1, 2, 3, 4, 5])
print(arr[0]) # Access the first element (1)
print(arr[2]) # Access the third element (3)
```

You can modify individual elements of a NumPy array by simply indexing the element and assigning a new value to it.

```
arr = np.array([1, 2, 3, 4, 5])
arr[2] = 10 # Modify the third element to be 10
```

Slicing

Array slicing allows you to extract a portion of a NumPy array, creating a new array containing the selected elements. Slicing is done using a colon ':' within the square brackets, with the format `[start:stop:step]`, where start is the inclusive starting index, stop is the exclusive stopping index, and step is the interval between elements.

Example - Slicing

```
arr = np.array([1, 2, 3, 4, 5])

# Slice from index 1 (inclusive) to 4 (exclusive), step by 1
sliced_arr = arr[1:4]
print(sliced_arr) # Output: [2 3 4]

# Slice from the beginning to index 3 (exclusive), step by 2
sliced_arr2 = arr[:3:2]
print(sliced_arr2) # Output: [1 3]
```

You can omit any of the components, and the defaults will be used:

- If you omit start, it defaults to the beginning of the array.
- If you omit stop, it defaults to the end of the array.
- If you omit step, it defaults to 1.

Example - Multidimensional array indexing and slicing

```
# Create a 2D array (matrix)
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Accessing elements in a 2D array
print(matrix[0, 0]) # Access the element at row 0, column 0 (1)
print(matrix[1, 2]) # Access the element at row 1, column 2 (6)

# Slicing a 2D array
```

```
print(matrix[0:2, 1:]) # Slice rows 0 and 1, and columns 1 and 2
# Output:
# [[2 3]
# [5 6]]
```

Example - Advanced slicing

```
arr = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

# Slicing with :: to skip elements
print(arr[::2]) # Output: [0 2 4 6 8] (every 2nd element)

# Reverse the array
print(arr[::-1]) # Output: [9 8 7 6 5 4 3 2 1 0]

# Reverse and skip elements
print(arr[::-2]) # Output: [9 7 5 3 1] (every 2nd element in reverse
↪ order)
```

In the above examples, ‘::’ is used to specify the step size for the slicing operation, allowing you to skip elements in the sequence while creating a new sliced sequence.

Example - Reverse indexing using -1

```
import numpy as np

arr = np.array([11, 22, 33, 44, 55])

# Access the last element
print(arr[-1]) # Output: 55

# Access the second-to-last element
print(arr[-2]) # Output: 44

# Slicing from the end
print(arr[-3:]) # Output: [33 44 55] (last 3 elements)

# Reverse the array using slicing
print(arr[::-1]) # Output: [55 44 33 22 11]
```

3.4 Mathematical Operations

NumPy is a powerful library for numerical computations in Python, and it provides a wide range of mathematical operations and functions for working with arrays.

Basic Arithmetic Operations NumPy allows you to perform basic arithmetic operations on arrays, element-wise. These operations include addition, subtraction, multiplication, division, and more.

Example - Basic arithmetic operations:

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

# Addition
result_add = arr1 + arr2 # [5, 7, 9]

# Subtraction
result_sub = arr1 - arr2 # [-3, -3, -3]

# Multiplication
result_mul = arr1 * arr2 # [4, 10, 18]

# Division
result_div = arr1 / arr2 # [0.25, 0.4, 0.5]
```

Element-wise Mathematical Functions

NumPy provides many built-in mathematical functions that can be applied element-wise to arrays. Examples include `np.sin()`, `np.cos()`, `np.exp()`, `np.sqrt()`, and more.

Example - sine and cosine

```
arr = np.array([1, 2, 3])

# Compute sine of each element
result_sin = np.sin(arr) # [0.84147098, 0.90929743, 0.14112001]

# Compute square root of each element
result_sqrt = np.sqrt(arr) # [1.          1.41421356 1.73205081]
```

3.5 Statistical Functions

NumPy provides a wide range of statistical functions for analyzing data in arrays. These functions can help you compute various statistical measures, such as mean, median, standard deviation, variance, and more.

Example - Mean and median

```
data = np.array([1, 2, 2, 3, 4, 4, 4, 5])

mean = np.mean(data) # Compute mean
median = np.median(data) # Compute median
```

Example - Standard deviation and variance

```
data = np.array([1, 2, 2, 3, 4, 4, 4, 5])

std_deviation = np.std(data) # Compute standard deviation
variance = np.var(data) # Compute variance
```

Example - Percentiles

```
data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# Calculate the 25th and 75th percentiles
q25 = np.percentile(data, 25)
q75 = np.percentile(data, 75)
```

In the above example, the q th percentile of the data is calculated, where q is a value between 0 and 100.

3.6 Aggregation Functions

NumPy provides functions for aggregating data, such as `np.sum()`, `np.mean()`, `np.max()`, `np.min()`, and `np.std()`, to calculate various statistics on arrays.

Example - Aggregation functions

```
arr = np.array([1, 2, 3, 4, 5])

# Compute the sum of all elements
total = np.sum(arr) # 15

# Compute the mean (average)
avg = np.mean(arr) # 3.0

# Find the maximum and minimum values
max_val = np.max(arr) # 5
min_val = np.min(arr) # 1

# Find the indices of the minimum and maximum values in an array.
max_idx = np.argmax(arr) # 4
min_idx = np.argmin(arr) # 0
```

3.7 Other Useful Functions

`np.where(condition, x, y)` is a function that returns elements from `x` where the condition is `True`, and elements from `y` where the condition is `False`. It is used to create a new array by selecting values from two input arrays based on a condition.

Example - np.where

```
arr = np.array([1, 2, 3, 4, 5])
result = np.where(arr > 3, arr, 0)

# result will be [0, 0, 0, 4, 5] (values greater than 3 are replaced
↪ with 0)
```

`np.all(condition, axis=None)` is a function that checks if all elements along the specified axis (or all elements if axis is not provided) evaluate to `True`. It returns `True` if all elements satisfy the condition; otherwise, it returns `False`.

```
arr = np.array([True, True, False, True])
result = np.all(arr)
```

```
# result will be False (not all elements are True)
```

Array Sorting

- `np.sort(arr)`: Sorts elements along an axis.
- `np.argsort(arr)`: Returns the indices that would sort an array.
- `np.argmax(arr)` and `np.argmin(arr)`: Returns the indices of the maximum and

Example - Array sorting

```
# Create a NumPy array
arr = np.array([7, 3, 9, 1, 5, 2])

# Sort the array in ascending order
sorted_arr = np.sort(arr)

# Find the indices that would sort the array
sorted_indices = np.argsort(arr)

# Print the results
print("Original Array:", arr)
print("Sorted Array (Ascending Order):", sorted_arr)
print("Indices for Sorting:", sorted_indices)
```

Output:

```
Original Array: [7 3 9 1 5 2]
Sorted Array (Ascending Order): [1 2 3 5 7 9]
Indices for Sorting: [3 5 1 4 0 2]
```

3.8 Boolean Arrays

Working with boolean arrays in NumPy is essential for various tasks, including filtering data, conditional operations, and logical comparisons. Boolean arrays are arrays of Boolean values (True or False), often created as a result of applying conditions or comparisons to existing arrays.

You can create a boolean array by applying conditions or comparisons to an existing array.

Example - Applying condition to an array

```
arr = np.array([1, 2, 3, 4, 5])
is_greater_than_3 = arr > 3
print(is_greater_than_3)
```

Output:

```
array([False, False, False,  True,  True])
```

You can use a boolean array to filter data from another array. For example, to select values greater than 3:

Example - Filtering data

```
filtered_data = arr[is_greater_than_3]
print(filtered_data)
```

Output:

```
array([4, 5])
```

Example - Logical operations

```
arr1 = np.array([True, False, True])
arr2 = np.array([False, True, False])
result_and = arr1 & arr2  # Element-wise AND
result_or = arr1 | arr2   # Element-wise OR
result_not = ~arr1        # Element-wise NOT
print(result_and)
print(result_or)
print(result_not)
```

Output:

```
[False False False]
[ True  True  True]
[False  True False]
```

Example - Counting True values

```
count_true = np.count_nonzero(is_greater_than_3)
print(count_true)
```

Output:

2

Example - Checking if All or Any are True

```
all_true = np.all(is_greater_than_3)
any_true = np.any(is_greater_than_3)
print(all_true)
print(any_true)
```

Output:

False

True

Example - Combining Boolean Arrays

```
condition1 = arr > 3
condition2 = arr < 6
combined_condition = condition1 & condition2
print(combined_condition)
```

Output:

[False False False True True]

Understanding and working with boolean arrays is essential for data filtering, conditional operations, and data analysis in NumPy. These operations allow you to manipulate and analyze data based on specified conditions, which is a fundamental aspect of data manipulation in scientific computing and data science.

3.9 Exercise Set

3.9.1 Exercise 1: Analyzing Student Scores

You have been given a dataset containing information about students' scores in different subjects. Your task is to use NumPy to perform various data analysis tasks on this dataset.

You are provided with a NumPy array called `student_scores` representing the scores of 8 students in 4 subjects (Math, English, Science, and History). Each row represents a student, and each column represents a subject.

```
# Student scores data (8 students, 4 subjects)
student_scores = np.array([[85, 90, 78, 92],
                           [90, 87, 84, 88],
                           [78, 88, 93, 76],
                           [92, 85, 89, 90],
                           [85, 76, 92, 94],
                           [88, 79, 82, 85],
                           [76, 90, 91, 88],
                           [89, 82, 75, 90]])
```

- Find and display the scores of students 2, 4, and 6 for all subjects.
- Calculate and display the average score for each subject.
- Find and display the scores of all students in the "Science" subject.
- Create a new array called `pass_fail` where each element is 'Pass' if the student's average score is greater than or equal to 85 in all subjects, and 'Fail' otherwise.

Complete the tasks using NumPy functions, indexing, slicing, and list comprehension.

Exercise 1: Analyzing Student Scores

```
import numpy as np

# Student scores data (8 students, 4 subjects)
student_scores = np.array([[85, 90, 78, 92],
                           [90, 87, 84, 88],
                           [78, 88, 93, 76],
                           [92, 85, 89, 90],
                           [85, 76, 92, 94],
                           [88, 79, 82, 85],
                           [76, 90, 91, 88],
                           [89, 82, 75, 90]])

# Task 1: Find and display the scores of students 2, 4, and 6 for all
↪ subjects.
selected_students = student_scores[[1, 3, 5], :]
print("Task 1: Scores of Students 2, 4, and 6 for All Subjects:\n",
↪ selected_students)
```

```

# Task 2: (way 1) Calculate and display the average score for each
↪ subject.
average_scores = np.mean(student_scores, axis=0)
subject_names = ["Math", "English", "Science", "History"]
print("\nTask 2: Average Score for Each Subject:")
for i, subject in enumerate(subject_names):
    print(subject + ":", average_scores[i])

# Task 2: (way 2) Calculate and display the average score for each
↪ subject without using enumerate.
subject_names = ["Math", "English", "Science", "History"]
num_subjects = len(subject_names)

print("Task 2: Average Score for Each Subject:")
for i in range(num_subjects):
    subject = subject_names[i]
    average_score = np.mean(student_scores[:, i])
    print(subject + ":", average_score)

# Task 3: Find and display the scores of all students in the "Science"
↪ subject.
science_scores = student_scores[:, 2]
print("\nTask 3: Scores of All Students in the \"Science\" Subject:\n",
↪ science_scores)

# Task 4: (way 1) Create a new array called pass_fail where each
↪ element is 'Pass' if the student's average score is greater than or
↪ equal to 85 in all subjects, and 'Fail' otherwise.
pass_fail = np.where(np.all(student_scores >= 85, axis=1), 'Pass',
↪ 'Fail')
print("\nTask 4: Pass/Fail Status:\n", pass_fail)

# Task 4: (way 2) Create a new array called pass_fail where each
↪ element is 'Pass' if the student's average score is greater than or
↪ equal to 85 in all subjects, and 'Fail' otherwise.
pass_fail = []
for student in student_scores:
    if np.all(student >= 85):
        pass_fail.append('Pass')
    else:
        pass_fail.append('Fail')

```

```
pass_fail = np.array(pass_fail)
print("\nTask 4: Pass/Fail Status:\n", pass_fail)
```

Output:

Task 1: Scores of Students 2, 4, and 6 for All Subjects:

```
[[90 87 84 88]
 [92 85 89 90]
 [88 79 82 85]]
```

Task 2: Average Score for Each Subject:

```
Math: 85.375
English: 84.625
Science: 85.5
History: 87.875
```

Task 3: Scores of All Students in the "Science" Subject:

```
[78 84 93 89 92 82 91 75]
```

Task 4: Pass/Fail Status:

```
['Fail' 'Fail' 'Fail' 'Pass' 'Fail' 'Fail' 'Fail' 'Fail']
```

3.9.2 Exercise 2: Analyzing Sales Data

You have been given a dataset containing sales data for a store over a week. Your task is to use NumPy to perform various data analysis tasks on this dataset. You are provided with two NumPy arrays:

- `daily_sales`: Represents the daily sales of the store for a week.
- `product_prices`: Contains the prices of four different products.

```
# Daily sales data (7 days)
daily_sales = np.array([[150, 200, 180, 220],
                        [180, 220, 190, 210],
                        [160, 210, 170, 230],
                        [170, 230, 200, 250],
                        [190, 220, 180, 240],
                        [200, 250, 210, 270],
                        [220, 270, 230, 290]])
```

```
# Product prices (4 products)
product_prices = np.array([10.0, 12.0, 15.0, 8.0])
```

- Reshape the `daily_sales` array to have the shape (7, 1, 4). Each row should represent sales data for a single day, and each day should have a shape of (1, 4) to represent the sales of four products.
- Calculate the total sales for each day and display them.
- Calculate the total sales for each product over the week and display them.
- Calculate the average daily sales for each product.
- Create a new array called `sales_report` by stacking the `daily_sales` array on top of the `product_prices` array. The resulting array should have two sections: one for daily sales and one for product prices.

Exercise 1: Analyzing Sales Data

```
import numpy as np

# Daily sales data (7 days)
daily_sales = np.array([[150, 200, 180, 220],
                        [180, 220, 190, 210],
                        [160, 210, 170, 230],
                        [170, 230, 200, 250],
                        [190, 220, 180, 240],
                        [200, 250, 210, 270],
                        [220, 270, 230, 290]])

# Product prices (4 products)
product_prices = np.array([10.0, 12.0, 15.0, 8.0])

# Task 1: Reshape the daily_sales array to have the shape (7, 1, 4).
reshaped_sales = daily_sales.reshape(7, 1, 4)
print("Task 1: Reshaped Daily Sales Array:")
print(reshaped_sales)

# Task 2: Calculate the total sales for each day and display them.
total_sales_per_day = np.sum(daily_sales, axis=1)
print("\nTask 2: Total Sales for Each Day:")
for i, total in enumerate(total_sales_per_day, 1):
    print(f"Day {i}: {total}")
```

```

# Task 3: Calculate the total sales for each product over the week and
↪ display them.
total_sales_per_product = np.sum(daily_sales, axis=0)
print("\nTask 3: Total Sales for Each Product:")
for i, total in enumerate(total_sales_per_product, 1):
    print(f"Product {i}: {total}")

# Task 4: Calculate the average daily sales for each product.
average_daily_sales = np.mean(daily_sales, axis=0)
print("\nTask 4: Average Daily Sales for Each Product:")
for i, average in enumerate(average_daily_sales, 1):
    print(f"Product {i}: {average}")

# Task 5: Create a new array called sales_report by stacking
↪ daily_sales on top of product_prices.
sales_report = np.vstack((daily_sales, [product_prices]))
print("\nTask 5: Sales Report:")
print(sales_report)

```

Output:

Task 1: Reshaped Daily Sales Array:

```

[[[150 200 180 220]]

 [[180 220 190 210]]

 [[160 210 170 230]]

 [[170 230 200 250]]

 [[190 220 180 240]]

 [[200 250 210 270]]

 [[220 270 230 290]]]

```

Task 2: Total Sales for Each Day:

```

Day 1: 750
Day 2: 800
Day 3: 770
Day 4: 850

```

Day 5: 830
Day 6: 930
Day 7: 1010

Task 3: Total Sales for Each Product:

Product 1: 1270
Product 2: 1600
Product 3: 1360
Product 4: 1710

Task 4: Average Daily Sales for Each Product:

Product 1: 181.42857142857142
Product 2: 228.57142857142858
Product 3: 194.28571428571428
Product 4: 244.28571428571428

Task 5: Sales Report:

```
[[150. 200. 180. 220.]  
 [180. 220. 190. 210.]  
 [160. 210. 170. 230.]  
 [170. 230. 200. 250.]  
 [190. 220. 180. 240.]  
 [200. 250. 210. 270.]  
 [220. 270. 230. 290.]  
 [ 10.  12.  15.   8.]]
```

Chapter 4

Pandas

4.1 Introduction to Pandas

Pandas is a popular open-source Python library designed for data manipulation and analysis. It provides easy-to-use data structures and functions for working with structured data, making it a fundamental tool in data science, machine learning, and data analysis.

Pandas provides two primary data structures: **DataFrame** and **Series**. These data structures are essential for storing and manipulating structured data efficiently.

DataFrame

A **DataFrame** is a two-dimensional, tabular data structure. It consists of rows and columns, where each column can contain data of a different data type. The **DataFrame** is the most commonly used data structure in Pandas and is ideal for storing and working with structured data.

Syntax - DataFrame

```
pd.DataFrame(data=None, index=None, columns=None, dtype=None, copy=None)
```

Key characteristics:

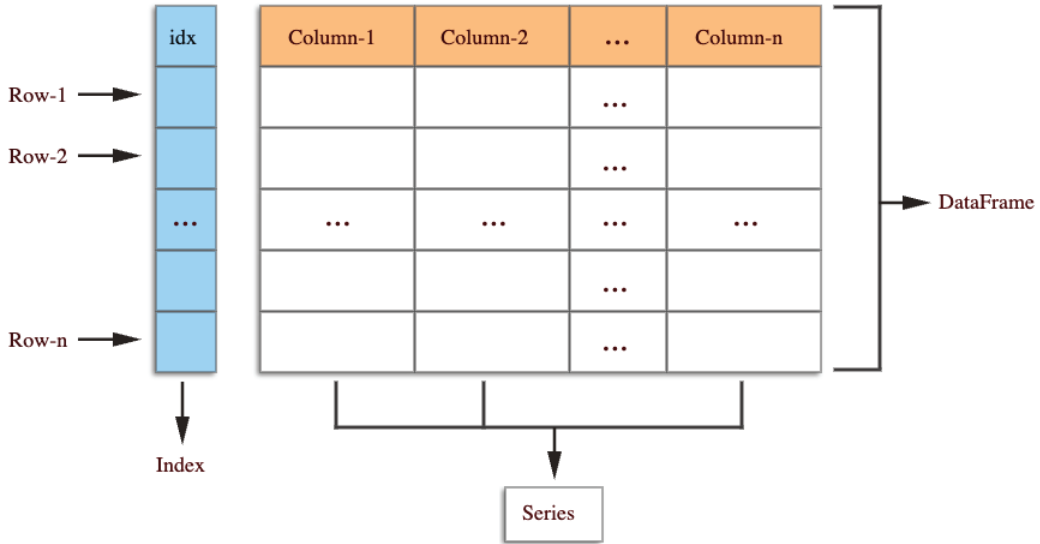
- Columns are named and can be accessed using column labels.
- Rows are indexed, and each row has a unique index label.
- You can think of a **DataFrame** as a collection of **Series** objects, where each **Series** represents a column.

Example of creating a **DataFrame**:

Example - DataFrame

```
import pandas as pd
import numpy as np

# Create a dictionary with different data types
data = {
```



```

'Integer': [1, 2, 3, 4, 5, None, 7, 8, 9, 10, 11, 12, 13, 14, 15],
'Float': [1.1, 2.2, 3.3, None, 5.5, 6.6, 7.7, 8.8, 9.9, 10.1,
↪ 11.11, None, 13.13, 14.14, 15.15],
'String': ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', None,
↪ 'L', 'M', 'N', 'O'],
'Boolean': [True, False, True, None, True, False, True, False,
↪ None, True, False, True, None, True, False]
}

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

# Display the DataFrame
print(df)

```

Output:

	Integer	Float	String	Boolean
0	1.0	1.10	A	True
1	2.0	2.20	B	False
2	3.0	3.30	C	True
3	4.0	NaN	D	None
4	5.0	5.50	E	True

5	NaN	6.60	F	False
6	7.0	7.70	G	True
7	8.0	8.80	H	False
8	9.0	9.90	I	None
9	10.0	10.10	J	True
10	11.0	11.11	None	False
11	12.0	NaN	L	True
12	13.0	13.13	M	None
13	14.0	14.14	N	True
14	15.0	15.15	O	False

DataFrame comes with several attributes that allow you to access and manipulate the data within the DataFrame.

Basic attributes:

- **shape:** returns a tuple representing the dimensions (number of rows, number of columns) of the DataFrame.
- **index:** gives access to the index labels.
- **columns:** attribute is an Index object holding the column labels.
- **values:** attribute returns a NumPy array containing the actual data values stored in the DataFrame.
- **dtypes:** returns a Series that displays the data type of each column in the DataFrame.

Example - Data Frame shape

```
df.shape # (n_rows, n_columns)
```

Output:

```
(15, 4)
```

Example - Data Frame index

```
df.index
```

Output:

```
RangeIndex(start=0, stop=15, step=1)
```

Example - Data Frame columns

```
df.columns
```

Output:

```
Index(['Integer', 'Float', 'String', 'Boolean'], dtype='object')
```

Example - Data Frame values

```
df.values
```

Output:

```
array([[1.0, 1.1, 'A', True],
       [2.0, 2.2, 'B', False],
       [3.0, 3.3, 'C', True],
       [4.0, nan, 'D', None],
       [5.0, 5.5, 'E', True],
       [nan, 6.6, 'F', False],
       [7.0, 7.7, 'G', True],
       [8.0, 8.8, 'H', False],
       [9.0, 9.9, 'I', None],
       [10.0, 10.1, 'J', True],
       [11.0, 11.11, None, False],
       [12.0, nan, 'L', True],
       [13.0, 13.13, 'M', None],
       [14.0, 14.14, 'N', True],
       [15.0, 15.15, 'O', False]], dtype=object)
```

Example - Data Frame data type

```
df.dtypes
```

Output:

```
Integer    float64
Float      float64
String     object
Boolean    object
dtype: object
```

Pandas offers a wide range of functions and methods for data manipulation and analysis. Some basic and commonly used functions and methods to get information about the `DataFrame`.

Basic functions:

- `head()`: display the first n rows of the `DataFrame`. By default, it returns the first 5 rows.
- `tail()`: display the last n rows of the `DataFrame`. By default, it returns the last 5 rows.
- `info()`: provide summary information about the `DataFrame`, including data types. This is useful for quickly inspecting the structure of the `DataFrame`.
- `describe()`: generates summary statistics for numerical columns in the `DataFrame`, such as count, mean, standard deviation, minimum, and maximum values.

Example - `.head()`

```
df.head()
```

Output:

	Integer	Float	String	Boolean
0	1.0	1.1	A	True
1	2.0	2.2	B	False
2	3.0	3.3	C	True
3	4.0	NaN	D	None
4	5.0	5.5	E	True

Example - `.tail()`

```
df.tail()
```

Output:

	Integer	Float	String	Boolean
10	11.0	11.11	None	False
11	12.0	NaN	L	True
12	13.0	13.13	M	None
13	14.0	14.14	N	True
14	15.0	15.15	O	False

Example - `.info()`

```
df.info()
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 15 entries, 0 to 14
Data columns (total 4 columns):
#   Column      Non-Null Count  Dtype
---  ---
0   Integer     14 non-null    float64
1   Float       13 non-null    float64
2   String      14 non-null    object
3   Boolean     12 non-null    object
dtypes: float64(2), object(2)
memory usage: 608.0+ bytes
```

Example - `.describe()`

```
df.describe()
```

Output:

	Integer	Float
count	14.000000	13.000000
mean	8.142857	8.363846
std	4.605300	4.503611
min	1.000000	1.100000
25%	4.250000	5.500000
50%	8.500000	8.800000
75%	11.750000	11.110000
max	15.000000	15.150000

Series

A Series is a one-dimensional labeled array that can hold data of various types, similar to a column in a DataFrame.

Syntax - Series

```
pd.Series(data=None, index=None, dtype=None, name=None, copy=None,  
↪ fastpath=False)
```

It has two components: the data and an associated index.

Key characteristics:

- Each element in a Series has a label (index) that allows for easy data retrieval.
- Series supports various data types (e.g., integers, floats, strings, etc.).

Example - Series

```
import pandas as pd  
  
data = [10, 20, 30, 40, 50]  
series = pd.Series(data, name='Numbers')  
series
```

Output:

```
0    10  
1    20  
2    30  
3    40  
4    50
```

Name: Numbers, dtype: int64

Basic attributes:

- **shape:** returns a tuple representing the dimensions of the Series. For a Series, it will be the length of the Series.
- **index:** returns the index labels of the Series.
- **name:** returns the name of the Series (if assigned).
- **values:** return Series as ndarray or ndarray-like depending on the dtype. However, it is recommended to use `Series.array` or `Series.to_numpy()`
- **dtype:** returns the data type of the elements in the Series.

Example - Series shape

```
series.shape
```

Output:

```
(5,)
```

Example - Series index

```
series.index
```

Output:

```
angeIndex(start=0, stop=5, step=1)
```

Example - Series name

```
series.name
```

Output:

```
'Numbers'
```

Example - Series values

```
series.values
```

Output:

```
array([10, 20, 30, 40, 50])
```

Example - Series array

```
series.array
```

Output:

```
<PandasArray>  
[10, 20, 30, 40, 50]  
Length: 5, dtype: int64
```

Example - Series array

```
series.to_numpy
```

Output:

```
<bound method IndexOpsMixin.to_numpy of 0      10
1      20
2      30
3      40
4      50
Name: Numbers, dtype: int64>
```

Example - Series dtype

```
series.dtype
```

Output:

```
dtype('int64')
```

Basic functions:

- **head(n)**: returns the first n elements of the Series. By default, it returns the first 5 elements.
- **tail(n)**: returns the last n elements of the Series. By default, it returns the last 5 elements.
- **info()**: prints information about a Series including the index dtype, non-null values and memory usage.
- **describe()**: generates summary statistics for the Series, including count, mean, std, min, max, and quantiles for numeric data.
- **unique()**: returns an array of unique values in the Series.
- **value_counts()**: returns a Series with counts of unique values in descending order.
- **sort_values()**: sorts the Series in ascending or descending order.
- **sort_index()**: sorts the index in Series in ascending or descending order.
- **apply(func)**: applies a function to each element in the Series.
- **map(mapping_dict)**: maps values in the Series to new values based on a dictionary.
- **isin(values)**: returns a boolean Series indicating whether each element is in the specified list of values.

Example - Series head(n)

```
series.head(2) # Returns the first 2 elements
```

Output:

```
0    10
1    20
Name: Numbers, dtype: int64
```

Example - Series tail(n)

```
series.tail(3) # Returns the last 3 elements
```

Output:

```
2    30
3    40
4    50
Name: Numbers, dtype: int64
```

Example - Series info()

```
series.info()
```

Output:

```
<class 'pandas.core.series.Series'>
RangeIndex: 5 entries, 0 to 4
Series name: Numbers
Non-Null Count  Dtype
-----  -----
5 non-null      int64
dtypes: int64(1)
memory usage: 168.0 bytes
```

Example - Series describe()

```
series.describe()
```

Output:

```
count    5.000000
mean     30.000000
std      15.811388
min      10.000000
```

```
25%      20.000000
50%      30.000000
75%      40.000000
max       50.000000
Name: Numbers, dtype: float64
```

Example - Series unique()

```
series.unique()
```

Output:

```
array([10, 20, 30, 40, 50])
```

Example - Series value_counts()

```
series.value_counts()
```

Output:

```
10    1
20    1
30    1
40    1
50    1
Name: Numbers, dtype: int64
```

Example - Series sort_values()

```
series.sort_values(ascending=False) # Sort in descending order
```

Output:

```
4    50
3    40
2    30
1    20
0    10
Name: Numbers, dtype: int64
```

Example - Series apply(func)

```
series.apply(lambda x: x * 2) # Multiply each element by 2
```

Output:

```
0    20
1    40
2    60
3    80
4   100
```

Name: Numbers, dtype: int64

Example - Series map(mapping_dict)

```
mapping_dict = {10: 'A', 20: 'B', 30: 'C', 40: 'D'}
series.map(mapping_dict)
```

Output:

```
0    A
1    B
2    C
3    D
4   NaN
```

Name: Numbers, dtype: object

Example - Series isin(values)

```
series.isin([10, 20])
```

Output:

```
0    True
1    True
2   False
3   False
4   False
```

Name: Numbers, dtype: bool

In addition to these two fundamental data structures, Pandas also offers several other data structures, such as Index, MultiIndex, and Categorical, to support more

advanced data manipulation and analysis tasks.

4.2 Data Cleaning

Data cleaning is a critical step in the data preparation process for any data analysis or machine learning project. It involves identifying and addressing issues or inconsistencies in the dataset to ensure that the data is accurate, complete, and suitable for analysis. Common data cleaning techniques and tasks in Pandas:

- Handling missing values
- Removing duplicates
- Handling outliers
- Correcting data types
- Dealing with inconsistent data

Data cleaning is often an iterative process, and the specific tasks you perform will depend on the nature of your dataset and the goals of your analysis or machine learning project. It is crucial to thoroughly clean and preprocess your data to ensure that your models are built on reliable and accurate data.

We will create a sample DataFrame with missing values, duplicates, outliers, and inconsistent data.

Example - Sample DataFrame

```
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eve', 'Frank',
    ↪ 'Alice'],
    'Age': [25, 30, np.nan, 40, 35, 28, 25],
    'Income': ['50,000', '60,000', '70,000', '85,000', '55,000',
    ↪ '65,000', '50,000'],
    'Height (cm)': [160, 175, 168, 185, 150, 190, 160],
}

df = pd.DataFrame(data)
```

Output:

	Name	Age	Income	Height (cm)
0	Alice	25.0	50,000	160
1	Bob	30.0	60,000	175
2	Charlie	NaN	70,000	168
3	David	40.0	85,000	185

4	Eve	35.0	55,000	150
5	Frank	28.0	65,000	190
6	Alice	25.0	50,000	160

Handling missing values

Identify missing values using `df.isnull()` or `df.isna()`. If the dataset has missing values, you need to decide the appropriate strategy for handling missing values. You can use: `df.dropna()` to remove rows or columns with missing values, `df.fillna(value)` to fill missing values with a specified value. Impute missing values using statistical methods like mean, median, or regression.

Example - Identify missing vlaues

```
# Using isnull() or isna() to identify missing values
missing_age = df['Age'].isnull()
# or use df['Age'].isna()
missing_age
```

Output:

```
0    False
1    False
2     True
3    False
4    False
5    False
6    False
```

Name: Age, dtype: bool

Example - Handling missing vlaues

```
mean_age = df['Age'].mean()
df['Age'].fillna(mean_age, inplace=True)
df
```

Output:

	Name	Age	Income	Height (cm)
0	Alice	25.0	50,000	160

1	Bob	30.0	60,000	175
2	Charlie	30.5	70,000	168
3	David	40.0	85,000	185
4	Eve	35.0	55,000	150
5	Frank	28.0	65,000	190
6	Alice	25.0	50,000	160

Alternatively, you can drop NA values along a different axis; `axis=1` drops all columns containing a null value.

Example - Dropping missing values

```
# Drop rows with any missing values
df.dropna(axis=0, inplace=True)

# Drop columns with any missing values
df.dropna(axis=1, inplace=True)
```

Removing duplicates

`df.drop_duplicates()`: remove duplicate rows.

Example - Removing duplicates

```
df.drop_duplicates(inplace=True)
```

Handling outliers

Example - Handling outliers

```
# assuming we consider 2 standard deviations as an outlier
mean_height = df['Height (cm)'].mean()
std_height = df['Height (cm)'].std()
df = df[(df['Height (cm)'] >= mean_height - 2 * std_height) &
        (df['Height (cm)'] <= mean_height + 2 * std_height)]
```

Dealing with inconsistent data

Example - Dealing with inconsistent data

```
# Dealing with Inconsistent Data (Capitalization)
df['Name'] = df['Name'].str.capitalize()
```

4.3 Basic Data Operations

In pandas, slicing, indexing, sorting, and filtering data are fundamental operations for manipulating and analyzing tabular data structures, typically in the form of DataFrames.

Commonly used methods for basic operations:

- `.iloc[]`
- `.loc[]`
- `.apply()`
- `.drop()`
- `.sort_values()`
- `.sort_index()`

Slicing and indexing

- **Slicing:** involves selecting a subset of rows or columns from a DataFrame based on their position (integer index). You can use `.iloc[]` to achieve this.
- **Indexing:** refers to selecting rows or columns using labels. You can use `.loc[]` for this purpose.

Example - Slicing and indexing

```
data = {'A': [1, 2, 3], 'B': [4, 5, 6]}
df = pd.DataFrame(data, index=['row1', 'row2', 'row3'])

# Slicing rows using integer index
sliced_rows = df.iloc[1:3]

# Indexing rows using labels
indexed_row = df.loc['row2']

# Slicing columns using integer index
sliced_cols = df.iloc[:, 0]
```

```
# Indexing columns using labels  
indexed_col = df.loc[:, 'A']
```

Filtering

Filtering involves selecting rows that meet certain conditions. You can use boolean indexing for this purpose.

Example - Filtering

```
# Filter rows where column 'A' is greater than 2  
filtered_df = df[df['A'] > 2]  
  
# Combine multiple conditions using & (AND) and | (OR)  
filtered_df = df[(df['A'] > 1) & (df['B'] < 6)]
```

Selecting columns

You can select specific columns from a DataFrame by passing a list of column names as a subscript.

Example - Selecting columns

```
# Select columns 'A' and 'B'  
selected_cols = df[['A', 'B']]
```

Creating new columns

You can create new columns based on existing data by performing operations on them.

Example - Adding new column to existing DataFrame

```
# Create a new column 'C' by adding columns 'A' and 'B'  
df['C'] = df['A'] + df['B']
```

Performing operations on data

You can perform various mathematical and statistical operations on DataFrame columns.

Example - Performing operations on data

```
# Calculate the mean of column 'A'
mean_a = df['A'].mean()

# Apply a custom function to a column
def custom_function(x):
    return x * 2

df['D'] = df['A'].apply(custom_function)
```

Dropping columns and rows

To drop one or more columns, you can use the `.drop()` method along with the axis parameter set to 1. You need to specify the column(s) you want to drop using the labels parameter.

Example - Dropping column(s)

```
data = {'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]}
df = pd.DataFrame(data)

# Drop a single column
df = df.drop(columns=['B'])

df.drop(['B'], axis=1) # without specifying the parameter columns, the
↳ axis should be defined explicitly

# Drop multiple columns
df = df.drop(columns=['B', 'C'])
```

Alternatively, you can use the `del` statement to drop a column in-place.

Example - Dropping column by del

```
del df['B']
```

To drop one or more rows, you can use the `.drop()` method with the axis parameter set to 0. Specify the row(s) you want to drop using the labels parameter.

Example - Dropping row(s)

```
data = {'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]}
df = pd.DataFrame(data)

# Drop a single row (by index)
df = df.drop(index=1)

# Drop multiple rows (by index)
df = df.drop(index=[0, 2])
```

Sorting data

You can sort a DataFrame by one or more columns using the `.sort_values()` method.

Example - Sorting data

```
# Creating a sample DataFrame
data = {'Name': ['Alice', 'Bob', 'Eve', 'David'],
        'Age': [25, 32, 25, 29],
        'Salary': [50000, 60000, 45000, 70000]}

df = pd.DataFrame(data)

# Sorting by a single column (e.g., 'Age') in ascending order
sorted_df_age_asc = df.sort_values(by='Age')

# Sorting by a single column (e.g., 'Salary') in descending order
sorted_df_salary_desc = df.sort_values(by='Salary', ascending=False)

# Sorting by multiple columns (first by 'Age' in ascending order and
→ then by 'Salary' in descending order)
sorted_df_age_salary = df.sort_values(by=['Age', 'Salary'],
→ ascending=[True, False])

print("Original DataFrame:")
print(df)

print("\nSorted by Age (Ascending):")
print(sorted_df_age_asc)
```

```
print("\nSorted by Salary (Descending):")
print(sorted_df_salary_desc)

print("\nSorted by Age (Ascending) and then Salary (Descending):")
print(sorted_df_age_salary)
```

Output:

Original DataFrame:

	Name	Age	Salary
0	Alice	25	50000
1	Bob	32	60000
2	Eve	25	45000
3	David	29	70000

Sorted by Age (Ascending):

	Name	Age	Salary
2	Eve	25	45000
0	Alice	25	50000
3	David	29	70000
1	Bob	32	60000

Sorted by Salary (Descending):

	Name	Age	Salary
3	David	29	70000
1	Bob	32	60000
0	Alice	25	50000
2	Eve	25	45000

Sorted by Age (Ascending) and then Salary (Descending):

	Name	Age	Salary
2	Eve	25	50000
0	Alice	25	45000
3	David	29	70000
1	Bob	32	60000

In pandas, the `sort_index()` method is used to sort the rows of a DataFrame or Series based on the index values. It allows you to rearrange the rows in either ascending or descending order based on the index.

Example - Sorting by index

```
# Creating a sample DataFrame with a custom index
data = {'Name': ['Alice', 'Bob', 'Eve', 'David'],
        'Age': [25, 32, 25, 29]}

custom_index = ['C', 'A', 'D', 'B']
df = pd.DataFrame(data, index=custom_index)

# Sorting the DataFrame by index in ascending order
sorted_df_asc = df.sort_index()

# Sorting the DataFrame by index in descending order
sorted_df_desc = df.sort_index(ascending=False)

print("Original DataFrame:")
print(df)

print("\nSorted by Index (Ascending):")
print(sorted_df_asc)

print("\nSorted by Index (Descending):")
print(sorted_df_desc)
```

Output:

Original DataFrame:

	Name	Age
C	Alice	25
A	Bob	32
D	Eve	25
B	David	29

Sorted by Index (Ascending):

	Name	Age
A	Bob	32
B	David	29
C	Alice	25
D	Eve	25

Sorted by Index (Descending):

	Name	Age
D	Eve	25
C	Alice	25
B	David	29
A	Bob	32

4.4 Data Integration

Data integration is the process of combining data from multiple sources or manipulating data within a single source to create a unified dataset for analysis. In pandas, you can perform data integration using various methods such as:

- `merge()`
- `concat()`
- `join()`
- `pivot()`
- `melt()`

`merge` is used to combine two or more DataFrames based on a common column (or index) known as the "key."

You can specify different types of joins (inner, outer, left, right) and handle how the merge handles overlapping column names.

Example - `.merge()`

```
# Create two sample DataFrames
df1 = pd.DataFrame({'key': ['A', 'B', 'C'], 'value1': [1, 2, 3]})
df2 = pd.DataFrame({'key': ['B', 'C', 'D'], 'value2': [4, 5, 6]})

# Merge based on the 'key' column
merged_df = pd.merge(df1, df2, on='key', how='inner')
```

The `concat` is used to concatenate two or more DataFrames along a particular axis (either rows or columns).

You can stack DataFrames on top of each other (along rows) or side by side (along columns).

Example - .concat()

```
# Create two sample DataFrames
df1 = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
df2 = pd.DataFrame({'A': [5, 6], 'B': [7, 8]})

# Concatenate along rows (axis=0)
concatenated_df = pd.concat([df1, df2], axis=0)
```

The `join` is a convenient method for merging DataFrames based on their index or common columns.

It's similar to `merge`, but it's primarily used when you want to join DataFrames based on their indices.

Example - .join()

```
# Create two sample DataFrames with the same index
df1 = pd.DataFrame({'value1': [1, 2]}, index=['A', 'B'])
df2 = pd.DataFrame({'value2': [3, 4]}, index=['A', 'B'])

# Join based on the index
joined_df = df1.join(df2)
```

The `pivot` is used to reshape your DataFrame by converting columns into index rows and vice versa.

It's useful when you have data in a long format and want to transform it into a wide format.

Example - .pivot()

```
data = {'Date': ['2023-01-01', '2023-01-02', '2023-01-01'],
        'Variable': ['A', 'B', 'C'],
        'Value': [10, 20, 15]}

df = pd.DataFrame(data)

# Pivot the DataFrame
pivoted_df = df.pivot(index='Date', columns='Variable', values='Value')
print(pivoted_df)
```

Output:

Variable	A	B	C
Date			
2023-01-01	10.0	NaN	15.0
2023-01-02	NaN	20.0	NaN

The `melt` is used to transform a wide DataFrame into a long format by "unpivoting" it.

It's useful when you have data in a wide format and want to make it more suitable for analysis.

Example - `.melt()`

```
data = {'Date': ['2023-01-01', '2023-01-02', '2023-01-01'],
        'Variable': ['A', 'B', 'C'],
        'Value': [10, 20, 15]}

df = pd.DataFrame(data)

# Melt the DataFrame
melted_df = df.melt(id_vars='Date', var_name='var_col',
                    value_name='value_col')
print(melted_df)
```

Output:

	Date	var_col	value_col
0	2023-01-01	Variable	A
1	2023-01-02	Variable	B
2	2023-01-01	Variable	C
3	2023-01-01	Value	10
4	2023-01-02	Value	20
5	2023-01-01	Value	15

These operations are essential for data preprocessing and transformation in pandas, allowing you to adapt your data to the specific requirements of your analysis or modeling tasks.

4.5 Data Aggregation

In pandas, data aggregation involves summarizing or computing statistics on data within a DataFrame. Here are some common methods for data aggregation:

- `groupby()`
- `agg()` or `aggregate()`
- `pivot_table()`
- `crosstab()`

The `groupby()` method is used to group rows in a `DataFrame` based on one or more columns. You can then apply aggregation functions like `sum()`, `mean()`, `count()`, `min()`, `max()`, etc., to compute summary statistics for each group.

Example - `groupby()`

```
data = {'Category': ['A', 'B', 'A', 'B', 'A'],
        'Value': [10, 20, 15, 25, 30]}

df = pd.DataFrame(data)

# Group by 'Category' and calculate the mean of 'Value' within each
↪ group
aggregated_df = df.groupby('Category')['Value'].mean()
print(aggregated_df)
```

Output:

```
Category
A      18.333333
B      22.500000
Name: Value, dtype: float64
```

The `agg()` or `aggregate()` method allows you to apply multiple aggregation functions simultaneously to one or more columns within each group.

Example - `agg()`

```
data = {'Category': ['A', 'B', 'A', 'B', 'A'],
        'Value': [10, 20, 15, 25, 30]}

df = pd.DataFrame(data)

# Group by 'Category' and calculate mean and sum of 'Value' within each
↪ group
```

```
aggregated_df = df.groupby('Category')['Value'].agg(['mean', 'sum'])
print(aggregated_df)
```

Output:

	mean	sum
Category		
A	18.333333	55
B	22.500000	45

Example - aggregate()

```
data = {'Category': ['A', 'B', 'A', 'B', 'A'],
        'Value': [10, 20, 15, 25, 30]}

df = pd.DataFrame(data)

# Group by 'Category' and calculate mean and sum of 'Value' within each
↪ group
aggregated_df = df.groupby('Category')['Value'].aggregate(['mean',
↪ 'sum'])
print(aggregated_df)
```

Output:

	mean	sum
Category		
A	18.333333	55
B	22.500000	45

Ultimately, you can choose to use either `aggregate()` or `agg()` based on your coding style and preference. They are functionally equivalent, and you can achieve the same results with both methods.

The `pivot_table()` method is used for creating pivot tables, which can aggregate data in a tabular format.

You can specify the index, columns, and aggregation functions.

Example - pivot_table()

```
data = {'Category': ['A', 'B', 'A', 'B', 'A'],
        'Value': [10, 20, 15, 25, 30]}
```

```
df = pd.DataFrame(data)

# Create a pivot table to calculate the mean of 'Value' for each
↪ 'Category'
pivot_table = df.pivot_table(index='Category', values='Value',
↪ aggfunc='mean')
print(pivot_table)
```

Output:

	Value
Category	
A	18.333333
B	22.500000

The `crosstab()` function computes a cross-tabulation table, which is essentially a contingency table showing the frequency of occurrences of different categories. It's useful for aggregating data when you want to compare two or more categorical variables.

Example - `crosstab()`

```
data = {'Category1': ['A', 'B', 'A', 'B', 'A'],
        'Category2': ['X', 'Y', 'X', 'X', 'Y']}

df = pd.DataFrame(data)

# Create a cross-tabulation table to count occurrences of Category1 and
↪ Category2 combinations
crosstab_table = pd.crosstab(df['Category1'], df['Category2'])
print(crosstab_table)
```

Output:

	X	Y
Category2		
Category1		
A	2	1
B	1	1

These are some of the methods for data aggregation in pandas. You can choose the appropriate method based on your data structure and the specific summary statistics you need to compute.

4.6 Time Series Data

Pandas is a powerful library for handling time series data, which is a series of data points indexed or listed in chronological order. Time series data is commonly encountered in various domains, including finance, economics, weather forecasting, and more. Pandas provides several methods and attributes that are important for working with time series data.

DateTime index

One of the most critical aspects of time series data in pandas is the DateTime index. A DataFrame or Series with a DateTime index allows you to easily perform time-based operations.

To create a DateTime index, you can use the `pd.to_datetime()` function or specify it when reading data from external sources.

Example - DateTime index

```
# Creating a DateTime index
date_rng = pd.date_range(start='2023-01-01', end='2023-01-10', freq='D')
df = pd.DataFrame(date_rng, columns=['date'])
```

Output:

```
DatetimeIndex(['2023-01-01', '2023-01-02',
               '2023-01-03', '2023-01-04',
               '2023-01-05', '2023-01-06',
               '2023-01-07', '2023-01-08',
               '2023-01-09', '2023-01-10'],
              dtype='datetime64[ns]', freq='D')
```

The following example uses `to_datetime()` to convert a list of date strings into DateTime objects, then a DataFrame is created with these DateTime objects as the index. The resulting DataFrame has a DateTime index, which is essential for working with time series data in pandas.

Example - to_datetime()

```
# Creating a DataFrame with a DateTime index using pd.to_datetime()
date_rng = pd.to_datetime(['2023-01-01', '2023-01-02', '2023-01-03',
                           '2023-01-04', '2023-01-05'])
data = {'values': [10, 20, 15, 25, 30]}
df = pd.DataFrame(data, index=date_rng)
```

```
# Now, the DataFrame has a DateTime index
print(df)
```

Output:

	values
2023-01-01	10
2023-01-02	20
2023-01-03	15
2023-01-04	25
2023-01-05	30

The most commonly used frequencies for `freq` parameter are listed in the table 4.6.

Frequency	Description
'D'	Daily frequency.
'B'	Business day frequency (weekdays).
'W'	Weekly frequency.
'M'	Month end frequency.
'Q'	Quarter end frequency.
'A'	Year end frequency.
'H'	Hourly frequency.
'T'	Minute frequency.
'S'	Second frequency.

Table 4.1: Frequencies for the `freq` parameter in `pd.date_range()`

Resampling

Resampling is the process of changing the frequency of your time series data. You can resample data to a lower frequency (downsampling) or a higher frequency (upsampling).

The `resample()` method allows you to apply aggregation functions over specific time intervals.

Example - Resampling

```
# create date range from 2023-01-01 to 2023-12-31 with daily ('D')
↳ frequency
# the resulted range will have 365 time points
data_rng = pd.date_range(start='2023-01-01', end='2023-12-31', freq='D')
# create a data frame by previously generated range of time points
df = pd.DataFrame(data_rng, columns=['date'])
# add 365 random values between 10 (inclusive) and 25 (exclusive) to
↳ the data frame
df['value'] = np.random.randint(low=10, high=25, size=df.shape[0])
# set the date column as index
df = df.set_index('date')
print('Original data frame shape: ', df.shape)

#downsampling:
resampled_df_down = df.resample('M').count()
print('Downsampled data frame shape: ', resampled_df_down.shape)

#upsampling
resampled_df_up = df.resample('H').count()
print('Upsampled data frame shape: ', resampled_df_up.shape)
```

Output:

```
Original data frame shape: (365, 1)
Downsampled data frame shape: (12, 1)
Upsampled data frame shape: (8737, 1)
```

Time-Based Slicing

Pandas allows you to easily slice and filter time series data based on date and time criteria.

Example - Select data between two specific dates

```
data_rng = pd.date_range(start='2023-01-01', end='2023-12-31', freq='D')
df = pd.DataFrame(data_rng, columns=['date'])
df_filtered = df[(df['date'] >= '2023-10-09') & (df['date'] <=
↪ '2023-10-15')]
print(df_filtered)
```

Output:

```
      date
281 2023-10-09
282 2023-10-10
283 2023-10-11
284 2023-10-12
285 2023-10-13
286 2023-10-14
287 2023-10-15
```

Shifting and Lagging

You can use the `shift()` method to create lagged versions of time series data, which can be useful for calculating differences or percentage changes between consecutive data points.

Example - shift()

```
data_rng = pd.date_range(start='2023-01-01', end='2023-12-31', freq='M')
df = pd.DataFrame(data_rng, columns=['date'])
df['value'] = np.random.randint(low=10, high=25, size=df.shape[0])
df['lagged_value'] = df['value'].shift(2)
print(df)
```

Output:

```
      date  value  lagged_value
0 2023-01-31     10          NaN
1 2023-02-28     13          NaN
2 2023-03-31     10         10.0
3 2023-04-30     17         13.0
4 2023-05-31     24         10.0
5 2023-06-30     23         17.0
6 2023-07-31     24         24.0
```

7	2023-08-31	22	23.0
8	2023-09-30	11	24.0
9	2023-10-31	12	22.0
10	2023-11-30	20	11.0
11	2023-12-31	11	12.0

Rolling Windows

Rolling window calculations are important for calculating moving averages or other statistics over a fixed-size time window.

The `rolling()` method followed by aggregation functions like `mean()`, `sum()`, etc., is used for this purpose.

Example - Rolling windows

```
data_rng = pd.date_range(start='2023-01-01', end='2023-12-31', freq='M')
df = pd.DataFrame(data_rng, columns=['date'])
df['value'] = np.random.randint(low=10, high=25, size=df.shape[0])

# Calculate the 3-day rolling mean of a time series
df['rolling_mean'] = df['value'].rolling(window=3).mean()

print(df)
```

Output:

	date	value	rolling_mean
0	2023-01-31	23	NaN
1	2023-02-28	18	NaN
2	2023-03-31	20	20.333333
3	2023-04-30	17	18.333333
4	2023-05-31	11	16.000000
5	2023-06-30	11	13.000000
6	2023-07-31	13	11.666667
7	2023-08-31	12	12.000000
8	2023-09-30	17	14.000000
9	2023-10-31	18	15.666667
10	2023-11-30	14	16.333333
11	2023-12-31	19	17.000000

4.7 Data Visualization

Data visualization is a critical aspect of data analysis and presentation, allowing you to understand data patterns, relationships, and trends more easily. In Python, there are several libraries for data visualization, with two of the most popular ones being Matplotlib and Seaborn.

Matplotlib

Matplotlib is a versatile and widely-used library for creating static, animated, and interactive visualizations in Python.

Importing matplotlib library

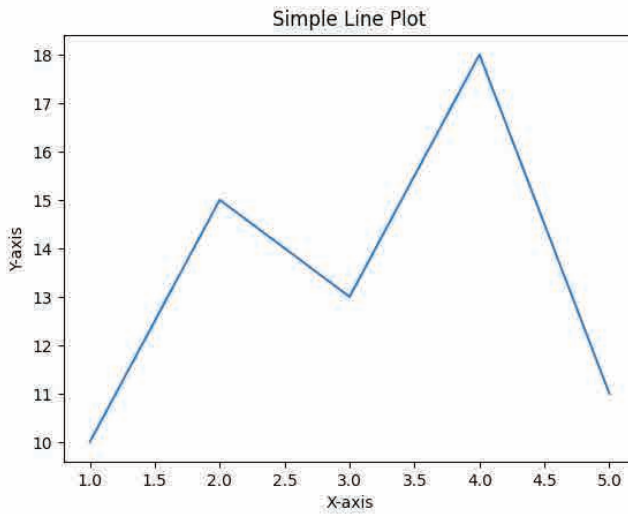
```
import matplotlib.pyplot as plt
```

Example - Creating simple line plot

```
x = [1, 2, 3, 4, 5]
y = [10, 15, 13, 18, 11]

plt.plot(x, y)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Simple Line Plot')
plt.show()
```

Output:

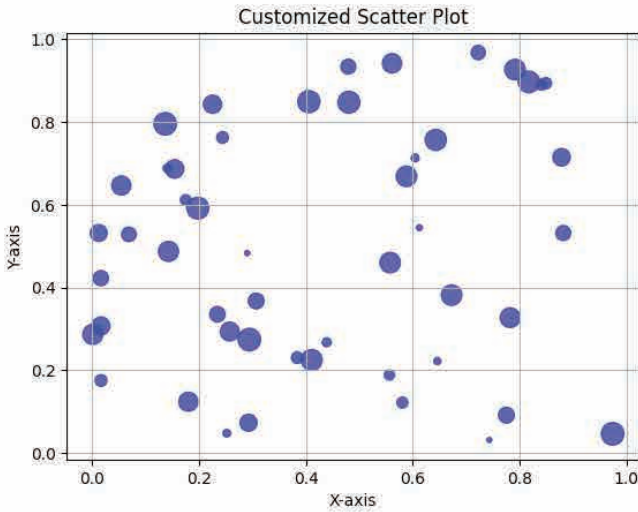


Example - Creating scatter plot

```
x = np.random.rand(50)
y = np.random.rand(50)
sizes = np.random.randint(10, 200, 50)

plt.scatter(x, y, s=sizes, c='blue', alpha=0.6)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Customized Scatter Plot')
plt.grid(True)
plt.show()
```

Output:



Seaborn

Seaborn is built on top of Matplotlib and provides a high-level interface for creating attractive and informative statistical graphics. It's particularly useful for complex visualizations and data exploration.

Importing seaborn library

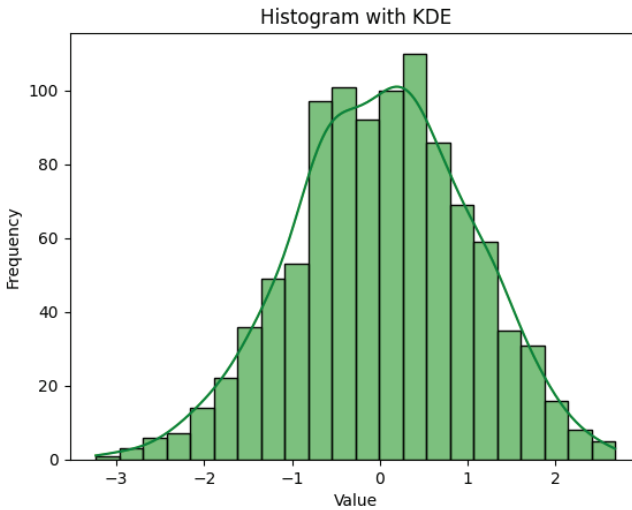
```
import seaborn as sns
```

Example - Seaborn histogram

```
import pandas as pd

data = pd.Series(np.random.randn(1000))
sns.histplot(data, kde=True, color='green')
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('Histogram with KDE')
plt.show()
```

Output:



4.8 Data Input/Output

In Pandas, "Data Input/Output" refers to the process of reading data from external sources, such as files or databases, and writing data to these external sources. This is a fundamental aspect of data manipulation and analysis, as you often need to import data into Pandas DataFrames for analysis and then export the results or modified data to external storage.

Example - Read data from csv file

```
import pandas as pd

# Read data from a CSV file into a DataFrame
df = pd.read_csv('data.csv')
```

Example - Read data from Excel file

```
import pandas as pd

# Read data from an Excel file into a DataFrame
df = pd.read_excel('data.xlsx', sheet_name='Sheet1')
```

Example - Write data to csv file

```
import pandas as pd

# Write DataFrame data to a CSV file
df.to_csv('output.csv', index=False)
```

Example - Write data to Excel file

```
import pandas as pd

# Write DataFrame data to an Excel file
df.to_excel('output.xlsx', sheet_name='Sheet1', index=False)
```

4.9 Exercise Set

4.9.1 Exercise 1: Basic Data Operations with Pandas

Objective The objective of this assignment is to practice basic data operations using the Pandas library in Python. You will work with the Iris dataset, perform various operations, and present the results.

Tasks:

1. Load the Iris dataset from the Seaborn library into a Pandas DataFrame.
2. Display the first 5 rows of the dataset to get an overview.
3. Calculate and display the following statistics for the "sepal_length" column:
 - The mean (average) sepal length.
 - The highest sepal length.
 - The lowest sepal length.
 - The standard deviation of sepal lengths.
4. Create a new column named "Flower_Type" in the DataFrame based on the "species" column. Map the species names as follows: - "setosa" -> "Setosa" - "versicolor" -> "Versicolor" - "virginica" -> "Virginica"
5. Sort the DataFrame in ascending order based on sepal length.
6. Filter and display the rows where sepal length is greater than 5.5.
7. Calculate and display the total count of each "Flower_Type."

Here's a sample solution for the assignment tasks:

Exercise 1: Basic Data Operations with Pandas

```
import pandas as pd
import seaborn as sns

# Task 1: Load the Iris dataset into a Pandas DataFrame
iris = sns.load_dataset('iris')

# Task 2: Display the first 5 rows of the dataset
print("First 5 rows of the Iris dataset:")
print(iris.head())

# Task 3: Calculate and display statistics for sepal length
mean_sepal_length = iris['sepal_length'].mean()
highest_sepal_length = iris['sepal_length'].max()
lowest_sepal_length = iris['sepal_length'].min()
std_deviation_sepal_length = iris['sepal_length'].std()

print(f"\nMean Sepal Length: {mean_sepal_length}")
print(f"Highest Sepal Length: {highest_sepal_length}")
print(f"Lowest Sepal Length: {lowest_sepal_length}")
print(f"Standard Deviation of Sepal Length:
↪ {std_deviation_sepal_length}")

# Task 4: Create a new "Flower_Type" column
flower_type_mapping = {
    'setosa': 'Setosa',
    'versicolor': 'Versicolor',
    'virginica': 'Virginica'
}
iris['Flower_Type'] = iris['species'].map(flower_type_mapping)

# Task 5: Sort the DataFrame in ascending order based on sepal length
iris_sorted = iris.sort_values(by='sepal_length', ascending=True)

# Task 6: Filter and display rows where sepal length is greater than 5.5
sepal_length_above_5_5 = iris[iris['sepal_length'] > 5.5]

# Task 7: Calculate and display the count of each "Flower_Type"
flower_type_counts = iris['Flower_Type'].value_counts()

print("\nCounts of each Flower Type:")
```

```
print(flower_type_counts)
```

Output:

First 5 rows of the Iris dataset:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Mean Sepal Length: 5.8433333333333334

Highest Sepal Length: 7.9

Lowest Sepal Length: 4.3

Standard Deviation of Sepal Length: 0.828066127977863

Counts of each Flower Type:

Setosa 50

Versicolor 50

Virginica 50

Name: Flower_Type, dtype: int64

4.9.2 Exercise 2: Data Integration, Aggregation, and Visualization

Objective The objective of this assignment is to practice data integration and aggregation using the Pandas library in Python. You will work with the "Tips" dataset from Seaborn, perform various integration and aggregation operations, and present the results.

Task 1: Data Integration

1. Load the "Tips" dataset from the Seaborn library into a Pandas DataFrame. You can load it using `seaborn.load_dataset('tips')`.
2. Create a synthetic dataset named "synthetic_data" using NumPy. This dataset should contain two columns, "ID" and "Synthetic_Value," with at least 15 rows. You can generate random data for this dataset.
3. Merge the "Tips" DataFrame with the "synthetic_data" DataFrame based on a common column or attribute you find relevant. You should use an appropriate merging method (e.g., inner, left, right, or outer) based on your analysis goals.

Task 2: Data Aggregation

1. Group the merged DataFrame by the "sex" column from the "Tips" dataset and calculate the following aggregate statistics:
 - The mean (average) of the "total_bill" column.
 - The median of "Synthetic_Value" (from the synthetic dataset).
 - The total count of samples.
2. Create a new DataFrame that contains the aggregated statistics for each gender category.

Task 3: Data Presentation

1. Create a bar chart using a library like Matplotlib or Seaborn to visualize the average total bill for each gender category. Label the chart appropriately.
2. Create a box plot to visualize the distribution of synthetic values for each gender category.

Exercise 2: Data Integration, Aggregation and Visualization

```
import pandas as pd
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt

# Task 1: Data Integration
tips_data = sns.load_dataset('tips')
synthetic_data = pd.DataFrame({
    'ID': range(1, 16),
    'Synthetic_Value': np.random.rand(15)
})
merged_data = pd.concat([tips_data, synthetic_data], axis=1)

# Task 2: Data Aggregation
agg_data = merged_data.groupby('sex').agg({
    'total_bill': 'mean',
    'Synthetic_Value': 'median',
    'sex': 'count'
}).rename(columns={'total_bill': 'Average_Total_Bill',
    ↪ 'Synthetic_Value': 'Median_Synthetic_Value', 'sex':
    ↪ 'Sample_Count'}).reset_index()

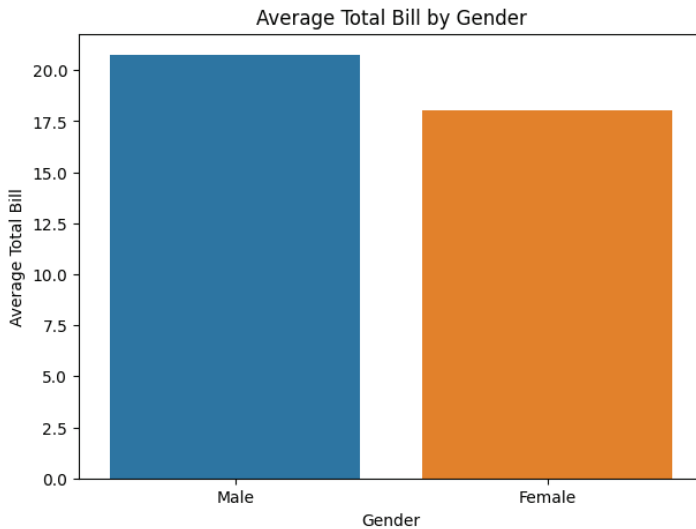
# Task 3: Data Presentation
```

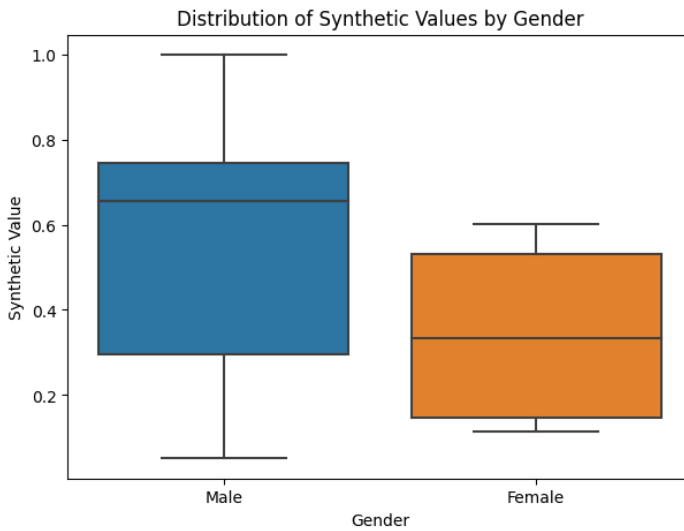
```
plt.figure(figsize=(7, 5))
sns.barplot(x='sex', y='Average_Total_Bill', data=agg_data)
plt.title('Average Total Bill by Gender')
plt.xlabel('Gender')
plt.ylabel('Average Total Bill')
plt.show()

plt.figure(figsize=(7, 5))
sns.boxplot(x='sex', y='Synthetic_Value', data=merged_data)
plt.title('Distribution of Synthetic Values by Gender')
plt.xlabel('Gender')
plt.ylabel('Synthetic Value')
plt.show()

# Display the aggregated data
print("Aggregated Data:")
print(agg_data)
```

Output:





Aggregated Data:

	sex	Average_Total_Bill	Median_Synthetic_Value	Sample_Count
0	Male	20.744076	0.654744	157
1	Female	18.056897	0.332319	87

4.9.3 Exercise 3: Analyzing Time Series Data with Pandas

Objective The objective of this assignment is to practice working with time series data using the Pandas library in Python. You will analyze the "Air Passengers" dataset, perform various operations, and present the results.

Task 1: Data Loading and Inspection

1. Load the "Air Passengers" dataset into a Pandas DataFrame. You can load it using Seaborn's built-in dataset function.
2. Inspect the dataset by displaying the first few rows, checking data types, and identifying any missing values.

Task 2: Data Preprocessing

1. Ensure that the date column is in the correct data type (datetime).
2. Set the date column as the DataFrame's index.

Task 3: Data Exploration

1. Calculate and display the following statistics for the number of passengers:
 - Mean (average)
 - Standard deviation
 - Minimum

- Maximum
2. Resample the data to have annual frequency (each year) and calculate the mean number of passengers for each year.

Task 4: Data Visualization

1. Create a line chart to visualize the monthly number of passengers over time.
2. Create a bar chart to visualize the mean annual number of passengers.

Task 5: Data Analysis

1. Calculate the rolling 12-month average of the number of passengers and create a line chart to visualize it.

Here's a sample solution for the assignment tasks using the "Air Passengers" dataset:

Exercise 3: Pandas Time Series

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

# Task 1: Data Loading and Inspection
# Load the "Air Passengers" dataset
df = sns.load_dataset('flights')

# Display the first few rows
print("Task 1: Data Loading and Inspection")
print(df.head())

# Check data types and missing values
print("\nData Types:")
print(df.dtypes)
print("\nMissing Values:")
print(df.isnull().sum())

# Task 2: Data Preprocessing
# Convert the date column to datetime
df['year'] = pd.to_datetime(df['year'], format='%Y')

# Set the date column as the DataFrame's index
df.set_index('year', inplace=True)

# Task 3: Data Exploration
```

```
print("\nTask 3: Data Exploration")
print("Statistics for Number of Passengers:")
print(df['passengers'].describe())

# Resample to annual frequency and calculate mean annual number of
↳ passengers
annual_mean = df['passengers'].resample('Y').mean()
print("\nMean Annual Number of Passengers:")
print(annual_mean.head())

# Task 4: Data Visualization
# Line chart for monthly number of passengers
plt.figure(figsize=(12, 6))
plt.plot(df.index, df['passengers'], label='Monthly Number of
↳ Passengers', color='blue')
plt.title('Monthly Number of Passengers Over Time')
plt.xlabel('Year')
plt.ylabel('Number of Passengers')
plt.legend()
plt.show()

# Bar chart for mean annual number of passengers
plt.figure(figsize=(12, 6))
plt.bar(annual_mean.index.year, annual_mean.values, label='Mean Annual
↳ Number of Passengers', color='green')
plt.title('Mean Annual Number of Passengers Over Time')
plt.xlabel('Year')
plt.ylabel('Mean Number of Passengers')
plt.legend()
plt.show()

# Task 5: Data Analysis
# Calculate rolling 12-month average
rolling_average = df['passengers'].rolling(window=12).mean()

# Line chart for rolling 12-month average
plt.figure(figsize=(12, 6))
plt.plot(df.index, df['passengers'], label='Monthly Number of
↳ Passengers', color='blue', alpha=0.5)
plt.plot(rolling_average.index, rolling_average.values, label='12-Month
↳ Rolling Average', color='red')
```

```
plt.title('Monthly Number of Passengers and 12-Month Rolling Average')
plt.xlabel('Year')
plt.ylabel('Number of Passengers')
plt.legend()
plt.show()
```

Output:

Task 1: Data Loading and Inspection

	year	month	passengers
0	1949	Jan	112
1	1949	Feb	118
2	1949	Mar	132
3	1949	Apr	129
4	1949	May	121

Data Types:

year	int64
month	category
passengers	int64

dtype: object

Missing Values:

year	0
month	0
passengers	0

dtype: int64

Task 3: Data Exploration

Statistics for Number of Passengers:

count	144.000000
mean	280.298611
std	119.966317
min	104.000000
25%	180.000000
50%	265.500000
75%	360.500000
max	622.000000

Name: passengers, dtype: float64

Mean Annual Number of Passengers:

year

1949-12-31 126.666667

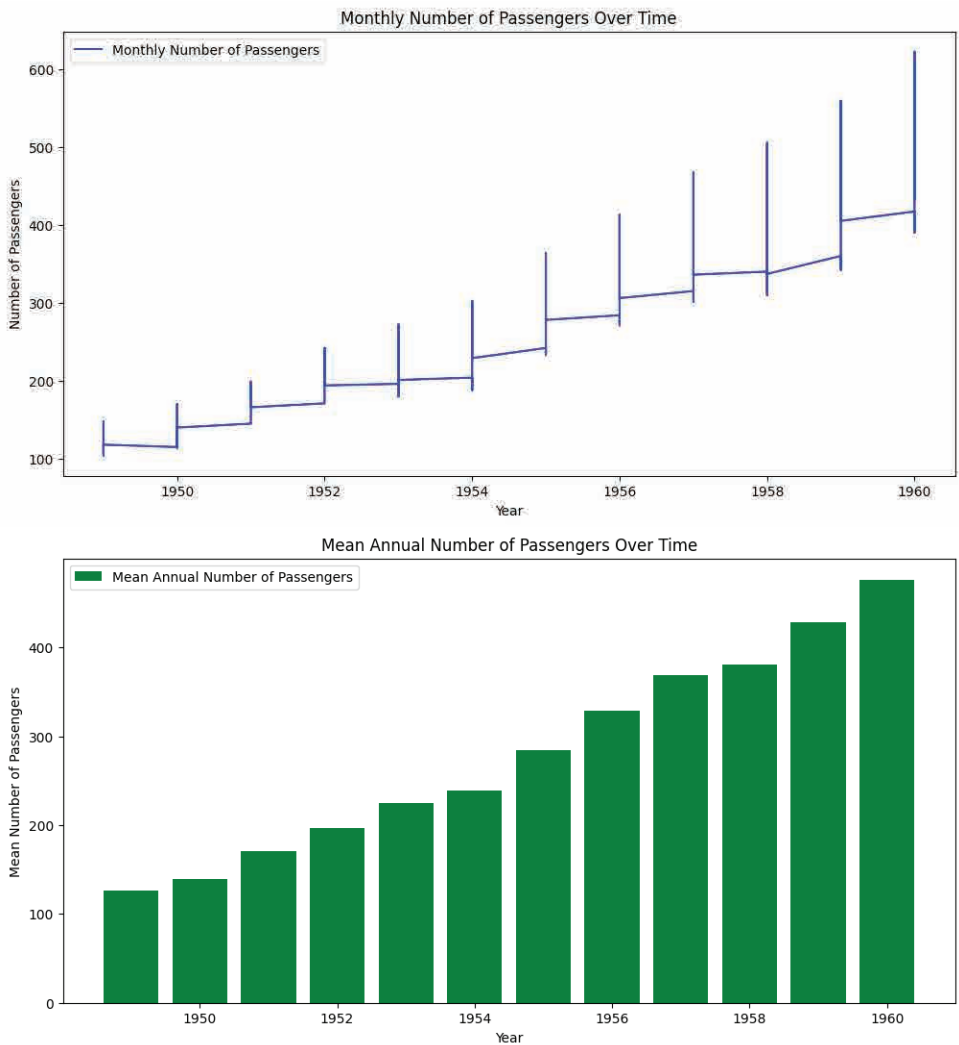
1950-12-31 139.666667

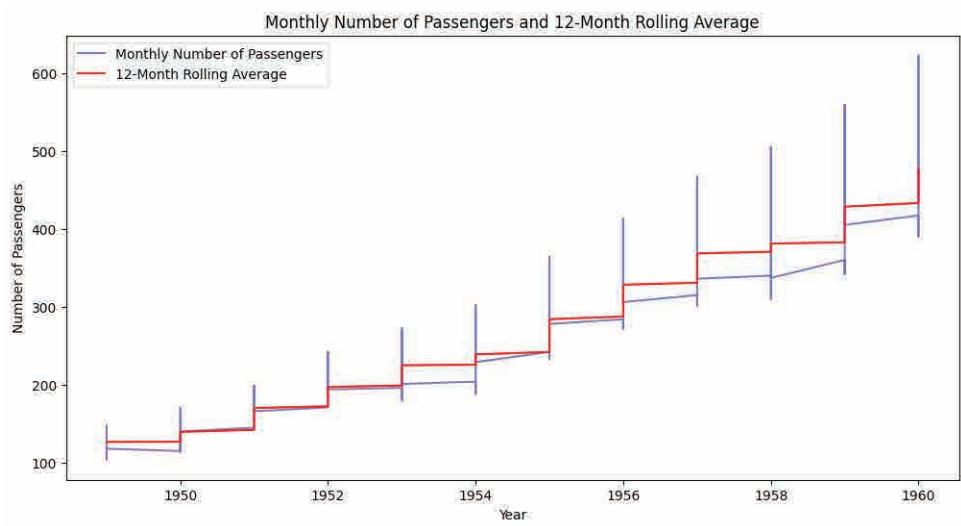
1951-12-31 170.166667

1952-12-31 197.000000

1953-12-31 225.000000

Freq: A-DEC, Name: passengers, dtype: float64





Chapter 5

Exploratory Data Analysis

Exploratory Data Analysis (EDA) is a crucial initial step in the data analysis process. It involves the thorough examination and visualization of data to understand its characteristics, patterns, and relationships. EDA helps data analysts and scientists gain insights, detect anomalies, and make informed decisions about subsequent data processing, modeling, or hypothesis testing.

Seaborn provides built-in datasets that are often used for data visualization and statistical analysis. These datasets are accessible through the seaborn library and are useful for learning and practicing data visualization techniques without needing to load external datasets. They cover a wide range of data types and scenarios. To load a Seaborn dataset, you can use the `seaborn.load_dataset()` function.

Iris Dataset: The Iris dataset is a classic dataset frequently used in machine learning. It comprises measurements of sepal and petal length and width for three different species of iris flowers. This dataset is often employed for tasks like classification and clustering.

Titanic Dataset: The Titanic dataset contains information about passengers who were aboard the Titanic, including details such as their survival status, class, gender, age, and more. It's commonly used for data analysis and visualization, exploring factors that influenced passenger survival.

Tips Dataset: The Tips dataset includes data from restaurant bills, including the total bill amount, tip amount, gender of the payer, whether the party was a smoker or non-smoker, the day of the week, and the time of day. This dataset is useful for analyzing tipping behavior and other restaurant-related insights.

Planets Dataset: The Planets dataset provides information about planets outside our solar system (exoplanets). It includes details about the discovery year, method of discovery, orbital period, and planet radius. Researchers use this dataset to study exoplanetary systems and their properties.

We will use the well-known Titanic dataset for basic EDA.

5.1 Part 1: Data Overview and Summary Statistics

In this part, we load the Titanic dataset and provide an overview of the first few rows of the data. We also calculate summary statistics for numerical features like

age, fare, etc.

EDA - part 1

```
# Import necessary libraries
import pandas as pd

# Load the Titanic dataset (you can load it from seaborn)
titanic = sns.load_dataset("titanic")

# Overview of the dataset
print(titanic.head())

# Summary statistics for numerical features
print(titanic.describe())
```

Output:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class \
0	0	3	male	22.0	1	0	7.2500	S	Third
1	1	1	female	38.0	1	0	71.2833	C	First
2	1	3	female	26.0	0	0	7.9250	S	Third
3	1	1	female	35.0	1	0	53.1000	S	First
4	0	3	male	35.0	0	0	8.0500	S	Third

	who	adult_male	deck	embark_town	alive	alone
0	man	True	NaN	Southampton	no	False
1	woman	False	C	Cherbourg	yes	False
2	woman	False	NaN	Southampton	yes	True
3	woman	False	C	Southampton	yes	False
4	man	True	NaN	Southampton	no	True

	survived	pclass	age	sibsp	parch	fare
count	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

5.2 Part 2: Missing Values

In this part, we check for missing values in the dataset. This helps us identify which columns have missing data.

EDA - part 2

```
# Check for missing values
print(titanic.isnull().sum())
```

Output:

```
survived      0
pclass        0
sex            0
age           177
sibsp         0
parch         0
fare          0
embarked      2
class         0
who           0
adult_male    0
deck         688
embark_town   2
alive         0
alone         0
dtype: int64
```

5.3 Part 3: Data Distribution Visualization

In this part, we visualize data distributions using histograms and countplots. We explore the distribution of the "Age," "Sex," "Class," and "Fare" columns.

EDA - part 3

```
# Data distribution visualization
plt.figure(figsize=(12, 6))
plt.subplot(2, 2, 1)
sns.histplot(titanic['age'].dropna(), kde=True)
```

```
plt.title('Distribution of Age')

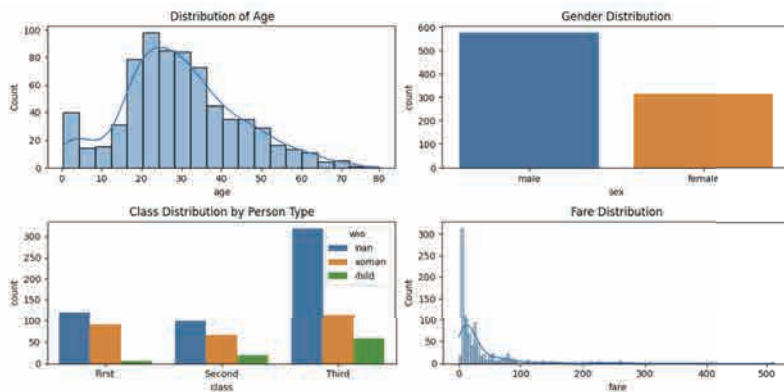
plt.subplot(2, 2, 2)
sns.countplot(data=titanic, x='sex')
plt.title('Gender Distribution')

plt.subplot(2, 2, 3)
sns.countplot(data=titanic, x='class', hue='who')
plt.title('Class Distribution by Person Type')

plt.subplot(2, 2, 4)
sns.histplot(data=titanic, x='fare', kde=True)
plt.title('Fare Distribution')

plt.tight_layout()
plt.show()
```

Output:



Interpreting histograms with Kernel Density Estimation (KDE) involves understanding the distribution of data and the smoothing effect that KDE adds to the histogram. Here are the steps to interpret histograms with KDE:

- A histogram is a graphical representation of the distribution of data. It consists of bins or bars that show the frequency or count of data points within each bin.
- Kernel Density Estimation (KDE) is a technique used to estimate the probability density function of a continuous random variable from a finite data sample. It essentially smooths out the histogram by adding a curve that represents the estimated underlying probability distribution.
- Observe the shape of the histogram. Is it symmetric, skewed to the left or right, or bimodal (having two distinct peaks)?
- The KDE curve shows the likelihood of a data point occurring at different values along the x-axis.
- Areas under the KDE curve represent the relative probability of different data values.
- Peaks in the KDE curve correspond to modes or clusters in the data. These peaks can indicate significant data points or modes within the distribution.
- The width of the KDE curve at any point represents the uncertainty or spread in the data. Wider parts of the curve suggest a broader range of possible data values.
- Symmetric KDE curves suggest a normal (bell-shaped) distribution.
- Skewed KDE curves (left or right) indicate a skew in the data, with a longer tail on one side, which may imply an asymmetric distribution.
- Outliers can be identified as data points that fall far from the main body of the KDE curve. These are often data points that are relatively rare or unusual.
- To gain a more precise understanding, you can use statistical measures such as mean, median, standard deviation, and percentiles to describe the distribution.
- Interpretation should also consider the context of the data. Is the distribution consistent with what you expected based on the nature of the data and the problem you're trying to solve?

In summary, interpreting histograms with KDE involves visually analyzing both the histogram and the overlaid KDE curve to understand the shape, central tendencies, spread, and modes of the data distribution. It allows for a more informative representation of the data compared to a simple histogram.

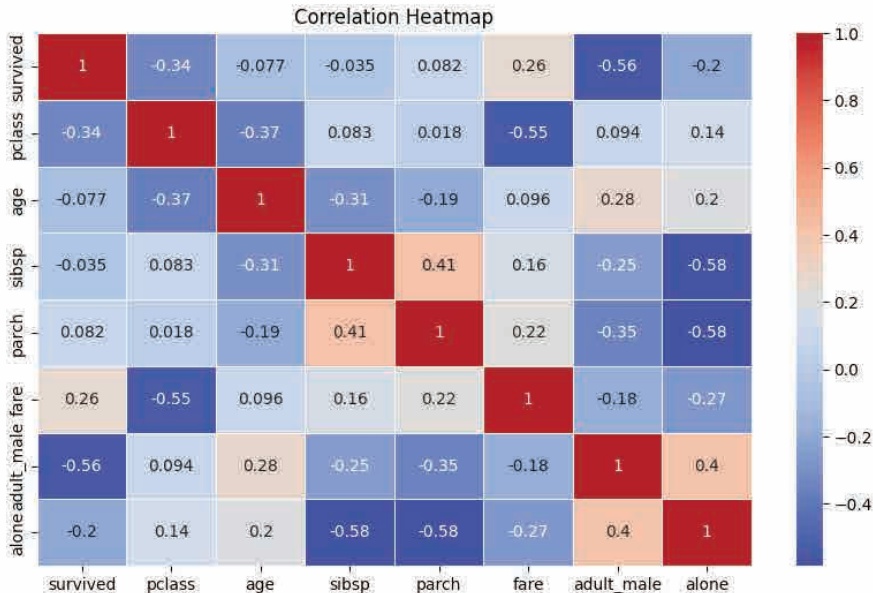
5.4 Part 4: Relationship Between Features

In this part, we create a correlation heatmap to visualize relationships between numerical features. It helps us understand how features are correlated with each other.

EDA - part 4

```
# Relationship between features
plt.figure(figsize=(10, 6))
sns.heatmap(titanic.corr(), annot=True, cmap='coolwarm', linewidths=0.5)
plt.title('Correlation Heatmap')
plt.show()
```

Output:



A correlation heatmap is a graphical representation of the correlation coefficients between variables in a dataset. It is a visual tool used in data analysis and data visualization to quickly identify and understand the relationships between pairs of variables. Correlation coefficients quantify the strength and direction of the linear relationship between two variables, and a heatmap provides an intuitive way to display these correlations.

- The correlation coefficients between all pairs of variables in a dataset results in a symmetric matrix where each cell represents the correlation between two

variables.

- The heatmap is typically created using a color scale, where colors represent the strength and direction of the correlation. Common color scales range from cool (e.g., blue) to warm (e.g., red), with a neutral color (e.g., white) indicating no correlation.
- Look at the diagonal line of cells from the top left to the bottom right. These cells represent the correlations of each variable with itself, which is always perfect (correlation of 1.0).
- Focus on the off-diagonal cells. These cells represent the correlations between pairs of different variables.
- The color of each cell in the heatmap indicates the strength and direction of the correlation. Dark blue or dark red colors (near -1 or 1) indicate a strong negative or positive correlation, respectively. Lighter colors (closer to 0) indicate weaker or no correlation. Colors closer to white suggest little to no correlation.
- The closer the correlation coefficient is to -1 or 1, the stronger the linear relationship between the variables. A correlation of -1 indicates a perfect negative linear relationship, while a correlation of 1 indicates a perfect positive linear relationship. Correlations close to 0 suggest little or no linear relationship between the variables.
- A positive correlation (values closer to 1) means that as one variable increases, the other tends to increase as well.
- A negative correlation (values closer to -1) means that as one variable increases, the other tends to decrease.
- Look for patterns and clusters of variables with high or low correlations. High correlations between groups of variables may suggest redundancy or multicollinearity, while low correlations indicate independence.

In summary, a correlation heatmap is a useful tool for visually assessing the relationships between variables in a dataset. It helps identify which variables are strongly related, which are weakly related, and which are independent. This information can guide subsequent data analysis and modeling decisions.

Chapter 6

Basic Concepts in Machine Learning

Machine learning (ML) is a subfield of artificial intelligence (AI) that focuses on developing algorithms and models that enable computers to learn from and make predictions or decisions based on data.

- **Data:** Data is the foundation of machine learning. It can be structured or unstructured and consists of features (input variables) and a target variable (the variable you want to predict or classify). Good data quality is essential for ML success.
- **Features:** Features are the attributes or variables used as input to an ML model. They can be numeric, categorical, or even text data. Proper feature selection and engineering can significantly impact model performance.
- **Target Variable:** The target variable, also known as the label or outcome variable, is what you're trying to predict or classify using your ML model. In supervised learning, you train the model to make predictions based on historical data.
- **Training Data:** The training data is the portion of your dataset used to train an ML model. The model learns patterns and relationships in the data during this phase.
- **Testing Data:** The testing data is a separate portion of your dataset used to evaluate the model's performance. It helps you assess how well the model generalizes to unseen data.
- **Model:** A model is the algorithm or mathematical function that learns patterns and relationships in the training data and uses that knowledge to make predictions or classifications on new, unseen data.
- **Supervised Learning:** In supervised learning, the ML model is trained on labeled data, where the target variable is known. It aims to learn the mapping from inputs (features) to outputs (target variable).
- **Unsupervised Learning:** In unsupervised learning, the model works with unlabeled data. It aims to discover patterns, structures, or relationships in the data without explicit target variables.
- **Classification:** Classification is a type of supervised learning where the goal is to assign input data points to predefined classes or categories. It's used for

tasks like spam detection, image recognition, and sentiment analysis.

- **Regression:** Regression is another type of supervised learning where the goal is to predict a continuous numerical value. It's used for tasks like predicting house prices or stock prices.
- **Clustering:** Clustering is a type of unsupervised learning that groups similar data points together based on their features. It's used for tasks like customer segmentation or anomaly detection.
- **Evaluation Metrics:** Evaluation metrics are used to assess the performance of ML models. Common metrics include accuracy, precision, recall, F1-score (for classification), and mean squared error (for regression).
- **Overfitting and Underfitting:** Overfitting occurs when a model learns the training data too well, capturing noise and making it perform poorly on unseen data. Underfitting, on the other hand, occurs when a model is too simple to capture the underlying patterns in the data.
- **Bias-Variance Tradeoff:** The bias-variance tradeoff refers to the balance between model simplicity (bias) and model flexibility (variance). Finding the right balance is crucial for model performance.
- **Feature Engineering:** Feature engineering involves creating new features or transforming existing ones to improve a model's performance. It's a critical step in ML.
- **Cross-Validation:** Cross-validation is a technique for assessing how well a model will generalize to new data. It involves splitting the data into multiple subsets for training and testing, providing a more robust evaluation of the model.
- **Hyperparameters:** Hyperparameters are settings that are not learned from the data but are set prior to training. Examples include learning rates, the number of hidden layers in a neural network, and regularization parameters.

Chapter 7

Linear Regression

Linear regression is a statistical method used for modeling the relationship between a dependent variable (y) and one or more independent variables (X) by fitting a linear equation to the observed data. It is one of the most widely used techniques in statistics and machine learning for predictive modeling and understanding the relationship between variables.

The primary goal of linear regression is to find the best-fitting linear equation that represents the relationship between the variables. In its simplest form, linear regression involves fitting a straight line to the data.

Simple Linear Regression

$$y = b + \omega X$$

where:

- y is the dependent variable (the one you want to predict).
- X is the independent variable (the one used for prediction).
- b is the intercept (the point where the line crosses the Y-axis when X is 0).
- ω is the slope (the change in y for a unit change in X).

Multiple Linear Regression

The equation for multiple linear regression extends the concept of simple linear regression to include multiple independent variables. In multiple linear regression, you have a dependent variable (y) that you want to predict based on two or more independent variables ($X_1, X_2, X_3, \dots, X_n$). The equation is as follows:

$$y = b + \omega_1 X_1 + \omega_2 X_2 + \omega_3 X_3 + \dots + \omega_n X_n + \varepsilon$$

where:

- y is the dependent variable you want to predict.
- $X_1, X_2, X_3, \dots, X_n$ are the independent variables.
- b is the intercept, representing the value of y when all independent variables ($X_1, X_2, X_3, \dots, X_p$) are zero.
- $\omega_1, \omega_2, \omega_3, \dots, \omega_n$ are the coefficients associated with each independent variable,

representing the change in y for a one-unit change in the corresponding X variable while holding all other variables constant.

- ε represents the error term, which accounts for the variability in y that is not explained by the model. It includes all other factors affecting y that are not included in the model.

The goal of multiple linear regression is to estimate the values of the coefficients $(b, \omega_1, \omega_2, \dots, \omega_n)$ that best fit the observed data. These estimates are typically obtained through the method of least squares, which minimizes the sum of the squared differences between the observed and predicted values of y .

In practice, multiple linear regression can be used for various purposes, such as predicting outcomes based on multiple predictors, understanding the relationships between variables, and modeling complex real-world phenomena.

It's important to note that when working with multiple linear regression, assumptions, and considerations such as multicollinearity (high correlations between independent variables), normality of residuals, and homoscedasticity should be taken into account to ensure the validity of the model and its interpretations.

The main steps involved in linear regression are as follows:

1. **Data Collection:** Gather a dataset containing values of both the dependent variable (y) and one or more independent variables (X).
2. **Data Exploration:** Visualize and explore the data to understand the relationships and check for linearity.
3. **Model Selection:** Choose the appropriate type of linear regression based on the problem. Simple Linear Regression when there is only one independent variable. Multiple Linear Regression when there are multiple independent variables.
4. **Parameter Estimation:** Use statistical techniques to estimate the values of b and ω that best fit the data. Common methods include the least squares method, which minimizes the sum of squared differences between the observed and predicted values.
5. **Model Evaluation:** Assess the quality of the linear regression model using various metrics, such as the coefficient of determination (R^2), mean squared error (MSE), or root mean squared error (RMSE).
6. **Interpretation:** Interpret the coefficients (b and ω) to understand the relationship between the variables. The sign of ω indicates the direction of the relationship (positive or negative), and the magnitude of ω quantifies the strength of the relationship.
7. **Prediction:** Use the fitted linear equation to make predictions for the dependent variable based on new or unseen values of the independent variable(s).

Linear regression has several assumptions, including linearity (the relationship

between variables is linear), independence of errors (residuals), homoscedasticity (constant variance of residuals), and normality of residuals. Violations of these assumptions may affect the validity of the model and require additional analysis or transformations.

Linear regression is widely applied in various fields for tasks like predicting stock prices, housing prices, customer demand, and more. It serves as a foundational tool for understanding and modeling relationships between variables in a straightforward and interpretable manner.

In the following example, a synthetic dataset is created and single linear regression is implemented from scratch.

Calculating gradients with respect to the error in the context of linear regression is a fundamental step in gradient-based optimization algorithms, such as gradient descent. The gradients represent the rate of change of the error with respect to each of the model's parameters (in this case, the slope and intercept). By adjusting the parameters in the direction opposite to the gradients, you can iteratively minimize the error and find the best-fit line for your data.

Here's how to calculate the gradients with respect to the error for linear regression:

Define the Error Function (Cost Function): In linear regression, the most common error function is the Mean Squared Error (MSE), which is defined as:

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

where

- n is the number of data points.
- Y_i is the actual output for data point i .
- $\hat{Y}_i$ is the predicted output for data point i using the current model parameters.

Calculate the Gradients: To find the gradients with respect to the error, you need to compute the partial derivatives of the error function with respect to each parameter (slope and intercept).

Partial Derivative with Respect to Slope ($\frac{\partial \text{MSE}}{\partial \text{Slope}}$):

$$\frac{\partial \text{MSE}}{\partial \text{Slope}} = -\frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i) \cdot X_i$$

Partial Derivative with Respect to Intercept ($\frac{\partial \text{MSE}}{\partial \text{Intercept}}$):

$$\frac{\partial \text{MSE}}{\partial \text{Intercept}} = -\frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)$$

These partial derivatives indicate how much the error would change with a small change in the respective parameters.

Update Parameters Using Gradients: In gradient descent, you adjust the parameters (slope and intercept) in the opposite direction of the gradients to minimize the error. The update rule for each parameter is as follows:

New Slope: $\text{Slope} = \text{Slope} - \text{Learning Rate} \cdot \frac{\partial \text{MSE}}{\partial \text{Slope}}$

New Intercept: $\text{Intercept} = \text{Intercept} - \text{Learning Rate} \cdot \frac{\partial \text{MSE}}{\partial \text{Intercept}}$

The learning rate determines the step size in each parameter update.

By iteratively applying these parameter updates while calculating the gradients at each step, you can converge to the values of the slope and intercept that minimize the MSE and provide the best linear fit to your data. This process continues until the error converges to a minimum or reaches a predefined number of iterations.

Example - Creating synthetic data

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

# Set a random seed for reproducibility
np.random.seed(0)

# Define the parameters of the linear relationship
slope = 2.5
intercept = 10
num_samples = 100

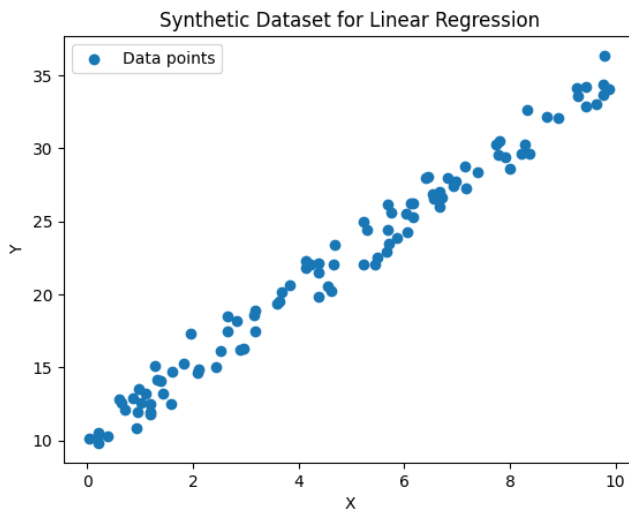
# Generate random input values (X)
X = np.random.rand(num_samples, 1) * 10

# Generate corresponding output values (Y) using the linear
↪ relationship Y = slope*X + intercept
Y = slope * X + intercept

# Add some random noise to the output to make it more realistic
noise = np.random.randn(num_samples, 1)
Y += noise
```

Example - Plot the dataset

```
plt.scatter(X, Y, label='Data points')
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Synthetic Dataset for Linear Regression')
plt.legend()
plt.show()
```

Output:

Now that we have our dataset, let's implement single linear regression from scratch using gradient descent.

Example - Parameter initialization

```
# Initialize the parameters for linear regression
learning_rate = 0.01
num_iterations = 1000
initial_slope_guess = 0
initial_intercept_guess = 0

# Perform gradient descent to find the optimal parameters (slope and
↪ intercept)
slope_guess = initial_slope_guess
intercept_guess = initial_intercept_guess
```

Example - Training

```

for iteration in range(num_iterations):
    # Calculate the predicted values using the current parameters
    predictions = slope_guess * X + intercept_guess

    # Calculate the error (mean squared error)
    error = np.mean((predictions - Y) ** 2)

    # Calculate the gradients with respect to the error
    slope_gradient = 2 * np.mean((predictions - Y) * X)
    intercept_gradient = 2 * np.mean(predictions - Y)

    # Update the parameters using gradient descent
    slope_guess -= learning_rate * slope_gradient
    intercept_guess -= learning_rate * intercept_gradient

    # Print the progress
    if iteration % 100 == 0:
        print(f'Iteration {iteration}: Error = {error:.4f}, Slope =
        ↪ {slope_guess:.4f}, Intercept = {intercept_guess:.4f}')

    # Print the final parameters
    print(f'Final Parameters: Slope = {slope_guess:.4f}, Intercept =
    ↪ {intercept_guess:.4f}')

```

Output:

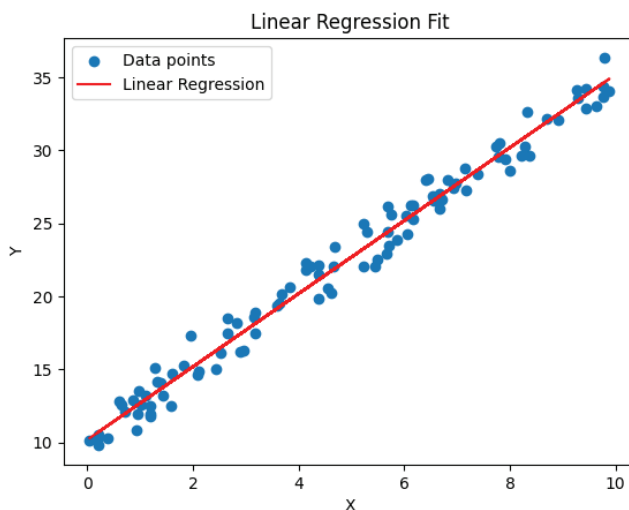
```

Iteration 0: Error = 537.2154, Slope = 2.4960, Intercept = 0.4402
Iteration 100: Error = 9.6384, Slope = 3.3673, Intercept = 4.6048
Iteration 200: Error = 3.9830, Slope = 3.0075, Intercept = 6.9184
Iteration 300: Error = 2.0269, Slope = 2.7959, Intercept = 8.2791
Iteration 400: Error = 1.3502, Slope = 2.6714, Intercept = 9.0794
Iteration 500: Error = 1.1162, Slope = 2.5982, Intercept = 9.5501
Iteration 600: Error = 1.0352, Slope = 2.5552, Intercept = 9.8269
Iteration 700: Error = 1.0072, Slope = 2.5298, Intercept = 9.9897
Iteration 800: Error = 0.9976, Slope = 2.5150, Intercept = 10.0854
Iteration 900: Error = 0.9942, Slope = 2.5062, Intercept = 10.1417
Final Parameters: Slope = 2.5011, Intercept = 10.1746

```

```
# Plot the dataset and the linear regression line
plt.scatter(X, Y, label='Data points')
plt.plot(X, slope_guess * X + intercept_guess, color='red',
         ↪ label='Linear Regression')
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Linear Regression Fit')
plt.legend()
plt.show()
```

Output:



Example - Prediction on new data point

```
# Test the model on a new data point
new_data_point = 0.2 # Replace with your new data point
new_prediction = slope_guess * new_data_point + intercept_guess
print(f'Prediction for new data point: {new_prediction:.4f}')
```

Output:

Prediction for new data point: 10.6748

Example of linear regression using scikit-learn (sklearn), a popular Python library for machine learning:

Example - Linear regression using scikit-learn

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

# Generate random data with a linear relationship:  $y = 2x + 1$ 
np.random.seed(0)
X = np.random.rand(100, 1) # Random values for the independent variable
y = 2 * X + 1 + 0.1 * np.random.randn(100, 1) # Adding random noise to
↳ the dependent variable

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
↳ test_size=0.2, random_state=0)

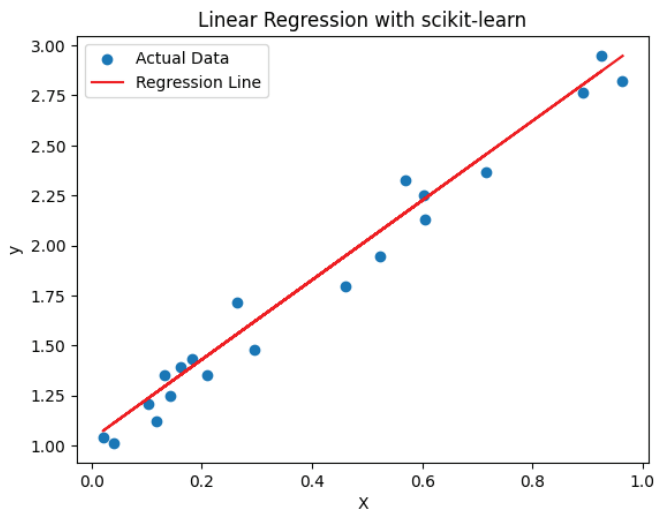
# Create a Linear Regression model
model = LinearRegression()

# Fit the model to the training data
model.fit(X_train, y_train)

# Make predictions on the test data
y_pred = model.predict(X_test)

# Plot the original data and the regression line
plt.scatter(X_test, y_test, label='Actual Data')
plt.plot(X_test, y_pred, color='red', label='Regression Line')
plt.xlabel('X')
plt.ylabel('y')
plt.title('Linear Regression with scikit-learn')
plt.legend()
plt.show()

# Test the model on new data
new_data_point = np.array([[0.2]]) # Replace with your new data point
new_prediction = model.predict(new_data_point.reshape(-1, 1))
print(f'Prediction for new data point: {new_prediction[0][0]}')
```



Output:

Prediction for new data point: 1.4296947133551845

Chapter 8

Logistic Regression

Logistic regression is the baseline supervised machine learning algorithm for classification and also has a very close relationship with neural networks. Logistic regression is used for binary classification tasks, where the goal is to predict one of two possible outcomes based on one or more independent variables. Despite its name, logistic regression is actually a type of generalized linear model used for classification, not regression.

In logistic regression, the dependent variable (the variable you're trying to predict) is binary, meaning it can take one of two possible values, typically coded as 0 and 1. For example, it might be used to predict whether a customer will buy a product (1) or not (0) based on certain features or attributes.

Logistic regression models the relationship between the independent variables (also called features or predictors) and the log odds of the binary outcome. It does this by calculating a linear combination of the independent variables, similar to linear regression. The linear combination is then transformed using the logistic function (sigmoid function) to constrain the predicted values between 0 and 1.

The logistic function, also known as the sigmoid function, is an S-shaped curve that maps any real-valued number to a value between 0 and 1.

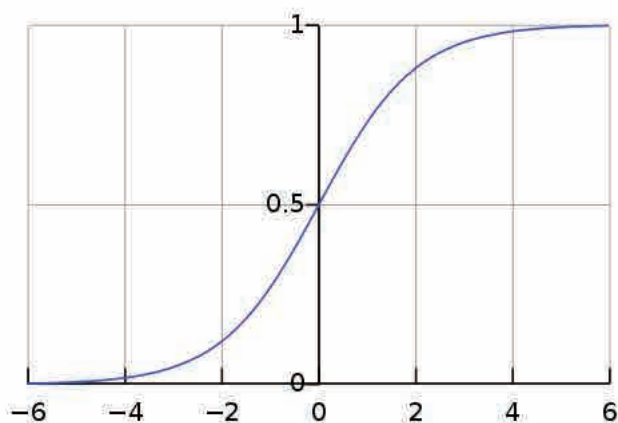


Figure 8.1: Sigmoid function

The formula for the Sigmoid function is:

$$g(z) = \frac{1}{1 + e^{-z}} \quad (8.1)$$

where $g(z)$ represents the output (probability) of the logistic function for a given input z . z is the input to the function, which is the linear combination of variables and their associated coefficients (weights).

The linear combination z is defined as:

$$z = w_0 + w_1 \cdot x_1 + w_2 \cdot x_2 + \dots + w_n \cdot x_n \quad (8.2)$$

where w_0 is the intercept term (bias). $w_1, w_2, \dots, w_n$ are the weights for the features $x_1, x_2, \dots, x_n$.

In practice, logistic regression learns the values of the weights (w_i) from the training data using optimization techniques (e.g., gradient descent) to maximize the likelihood of the observed outcomes.

Once the model parameters are estimated, you can use the logistic regression model to make predictions. Given a set of independent variables, the model calculates the probability of the binary outcome being 1. If this probability is greater than a certain threshold (typically 0.5), the model predicts class 1; otherwise, it predicts class 0.

Neural networks and logistic regression are related in the sense that logistic regression can be thought of as a simplified, single-layer neural network. Understanding this connection can provide insights into both logistic regression and neural networks.

In a neural network, you have layers of interconnected neurons (also called nodes or units). A single-layer neural network, often referred to as a "perceptron," "single-layer perceptron," or "single-layer neural network," consists of just one layer of neurons. Each neuron in this layer is connected to all the input features, and it computes a weighted sum of the inputs, similar to logistic regression.

In a single-layer neural network, each neuron typically applies an activation function to the weighted sum of its inputs. The logistic regression model also uses an activation function, the Sigmoid function, as it is described earlier. The logistic function "squashes" the linear combination of inputs into the range $[0, 1]$, just like the activation function used in neural networks introduces non-linearity into the model. Note, that there are many other activation functions.

In logistic regression, the output of the logistic function $g(z)$ represents the probability that the binary outcome is 1. In a single-layer neural network, the output from the activation function can also be interpreted as a probability when using a logistic activation function.

Both logistic regression and neural networks can be trained using gradient-based

optimization algorithms to adjust the weights (coefficients in logistic regression) to minimize a loss function. The common optimization technique used for both is gradient descent or its variants.

Neural networks, especially deep neural networks with multiple layers, are capable of modeling highly complex and nonlinear relationships in data. Logistic regression, being a linear model, is relatively simpler and can represent only linear decision boundaries. Neural networks, on the other hand, can capture both linear and nonlinear patterns in the data.

In summary, logistic regression can be seen as a special case of a single-layer neural network with a logistic activation function. Neural networks extend beyond logistic regression by allowing the composition of multiple layers and a wide variety of activation functions, making them more powerful for complex tasks like image recognition, natural language processing, and more. However, logistic regression remains a useful and interpretable model for binary classification tasks when the relationship between input features and the output is assumed to be approximately linear.

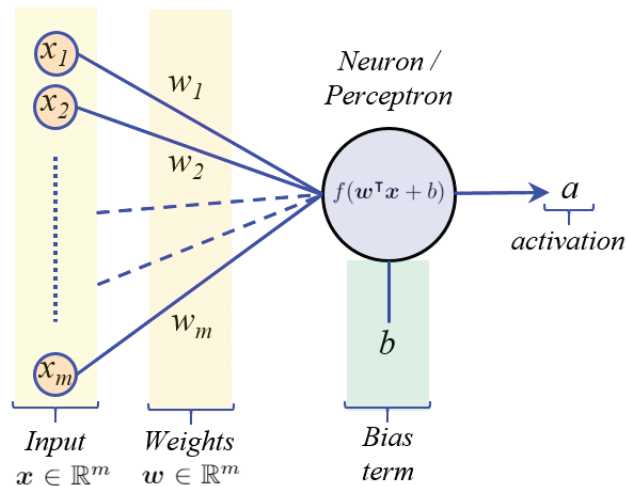


Figure 8.2: Single neuron

Example - Logistic regression (part 1)

```
# Import necessary libraries
import pandas as pd
import numpy as np
```

```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report,
↳ confusion_matrix
from sklearn.impute import SimpleImputer
```

Example - Logistic regression (part 2)

```
# Load the Titanic dataset from Seaborn
titanic_df = sns.load_dataset('titanic')

# Select relevant features and target variable
X = titanic_df[['pclass', 'age', 'fare', 'sex', 'sibsp', 'parch']]
y = titanic_df['survived']

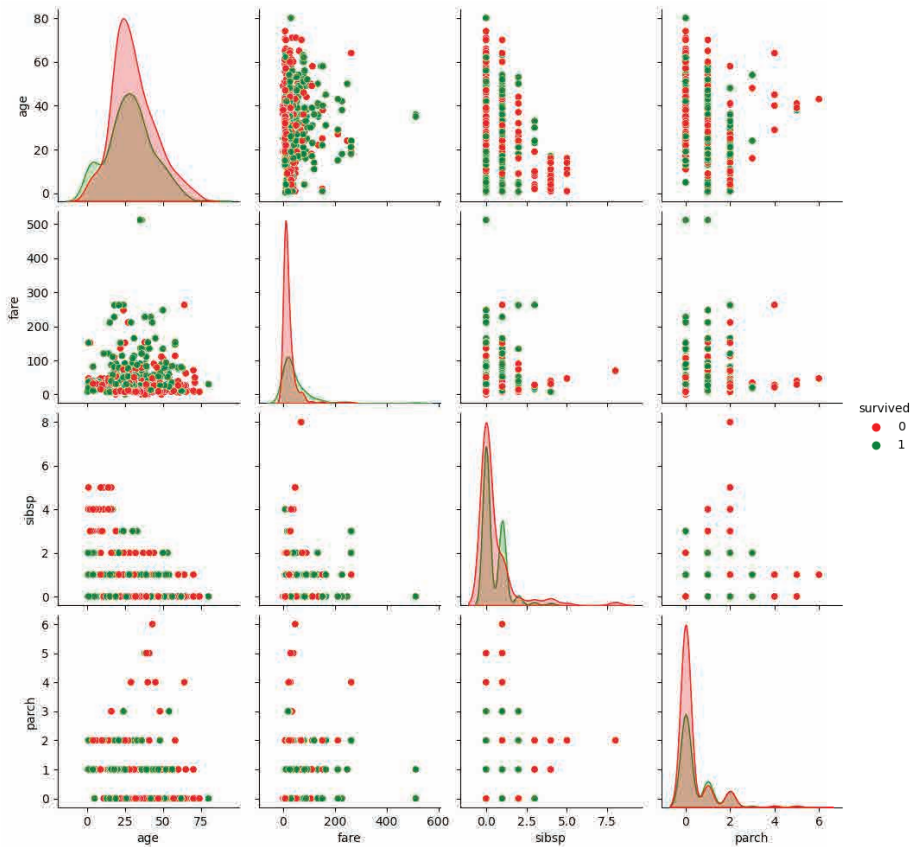
# Use LabelEncoder to encode the 'sex' column
label_encoder = LabelEncoder()
X.loc[:, 'sex'] = label_encoder.fit_transform(X['sex'])

# Handle missing values (fill with mean for simplicity, you can use
↳ more sophisticated methods)
# With these lines:
imputer = SimpleImputer(strategy='mean')
X = pd.DataFrame(imputer.fit_transform(X), columns=X.columns)
```

Example - Logistic regression (part 3)

```
# Visualize the data (pair plot for feature relationships)
sns.pairplot(titanic_df[['age', 'fare', 'sex', 'sibsp', 'parch',
↳ 'survived']], hue='survived', palette={0: 'red', 1: 'green'})
plt.title("Pair Plot of Features")
plt.show()
```

Output:



Interpretation of Pair Plot for Titanic Dataset

Diagonal Plots (Histograms or Kernel Density Estimation):

- The diagonal plots display histograms (or kernel density estimation plots) for each of the selected variables: age, fare, sex, sibsp (number of siblings/spouses aboard), and parch (number of parents/children aboard).
- For continuous variables like age and fare, you can observe their distributions. For example, you can see the age distribution skewed towards younger passengers.
- For categorical variables like sex, you can see the distribution of male and female passengers.
- For count variables like sibsp and parch, you can see the distribution of the number of siblings/spouses and parents/children aboard the Titanic.

Off-Diagonal Scatterplots:

- The off-diagonal elements show scatterplots for pairs of variables.
- The scatterplots are colored by survival status: red for passengers who did not survive (0) and green for passengers who survived (1).
- Key insights from the scatterplots:
 - In the scatterplot of age vs. fare, there doesn't appear to be a strong linear relationship between age and fare, and survival status does not show a clear separation.
 - In the scatterplot of age vs. sibsp, there's no strong linear correlation, and survival status overlaps.
 - In the scatterplot of fare vs. sibsp, there's no strong linear correlation, and survival status overlaps.
 - In the scatterplot of age vs. parch, there's no strong linear correlation, and survival status overlaps.
 - In the scatterplot of fare vs. parch, there's no strong linear correlation, and survival status overlaps.
 - The scatterplot of sex vs. sex (the diagonal) shows two separate bars for males and females. This indicates that the 'sex' variable is categorical.

Coloring by Survival Status:

- The use of color (red and green) distinguishes passengers who survived from those who did not.
- It allows you to visually compare the distribution of variables for survivors and non-survivors.
- For instance, you can observe that the scatterplots for age vs. fare and other variables do not show a clear separation between survivors and non-survivors, suggesting that these variables alone may not be strong predictors of survival.

Pattern Recognition:

- While the scatterplots show some overlap in points, they also reveal complex relationships and variations within the data, which can be valuable for feature engineering and further analysis.

Overall, the pair plot helps you get a sense of how the selected variables relate to each other and how they are associated with the survival status of Titanic passengers. It provides a visual foundation for understanding the data's characteristics and can guide more in-depth analysis and modeling decisions.

Example - Logistic regression (part 4)

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

Data Splitting in Machine Learning

Data splitting is a crucial step in machine learning and data analysis, where you divide your dataset into multiple subsets for different purposes. The most common splits include:

1. **Training Set:** The training set is the portion of your dataset used to train your machine learning model. It constitutes the majority of your data and is essential for the model to learn patterns and relationships.
2. **Validation Set:** The validation set is used during the model training process to evaluate and fine-tune model performance. It helps in adjusting hyperparameters and assessing how well the model generalizes to unseen data.
3. **Test Set:** The test set is an independent dataset that the model has never seen during training or validation. It serves as the final evaluation to estimate how well the model will perform on new, unseen data. It provides an unbiased estimate of the model's generalization performance.

Data splitting is essential for the following reasons:

- **Model Evaluation:** It allows for accurate assessment of the model's performance. Using the same data for both training and evaluation can lead to overfitting.
- **Hyperparameter Tuning:** Data splitting enables experimentation with different hyperparameters (e.g., learning rate, depth of a decision tree) on the validation set to optimize model performance.
- **Generalization Assessment:** The test set provides an unbiased estimate of how well the model will perform on new, real-world data, which is the ultimate goal of machine learning.

Example - Logistic regression (part 5)

```
# Standardize features (optional but can help with model performance,  
↳ to speed up training)  
scaler = StandardScaler()  
X_train = scaler.fit_transform(X_train)
```

```
X_test = scaler.transform(X_test)
```

Feature Standardization and Normalization in Machine Learning

Feature standardization and normalization are critical data preprocessing techniques used in machine learning to scale and transform the features (variables) in a dataset. They play a vital role in ensuring that the data is in a suitable form for modeling and can lead to improved model performance.

Feature Standardization

Feature standardization (also known as feature scaling or z-score normalization) is the process of transforming numerical features in a dataset to have a standard scale with a mean of 0 and a standard deviation of 1. It involves subtracting the mean value of the feature from each data point and then dividing by the standard deviation:

$$\text{Standardized Feature} = \frac{\text{Original Feature} - \text{Mean of Feature}}{\text{Standard Deviation of Feature}}$$

Mathematically:

$$\text{Standardized Feature} = \frac{X - \mu}{\sigma} \quad (8.3)$$

where Standardized Feature is the transformed feature after standardization. X is the original feature value. μ is the mean (average) of the feature values. σ is the standard deviation of the feature values.

- **Importance:** Standardization ensures that all features are on the same scale, preventing some features from dominating the learning process in machine learning algorithms. This is particularly important for models that rely on distances, such as k-nearest neighbors and support vector machines.
- **Benefits:** It can improve the convergence speed of gradient-based optimization algorithms and make models more robust.

Feature Normalization

Feature normalization scales the values of numerical features to a specified range, typically between 0 and 1. It is achieved by transforming the feature values based on their minimum and maximum values:

$$\text{Normalized Feature} = \frac{\text{Original Feature} - \text{Minimum Value of Feature}}{\text{Maximum Value of Feature} - \text{Minimum Value of Feature}}$$

Mathematically:

$$\text{Normalized Feature} = \frac{X - \min(X)}{\max(X) - \min(X)} \quad (8.4)$$

where Normalized Feature is the transformed feature after normalization. X is the original feature value. $\min(X)$ is the minimum value of the feature among all data points. $\max(X)$ is the maximum value of the feature among all data points.

- **Importance:** Normalization ensures that features with different ranges are brought to a common scale, making the dataset more suitable for models that are sensitive to the magnitude of values, such as neural networks and clustering algorithms.
- **Benefits:** It can improve the model's convergence, accuracy, and generalization performance.

Both feature standardization and normalization help in achieving better model performance, especially when the features have varying scales or are measured in different units. The choice between standardization and normalization depends on the specific requirements of the machine learning algorithm and the characteristics of the dataset.

Example - Logistic regression (part 6)

```
# Create and train the Logistic Regression model
logistic_reg = LogisticRegression()
logistic_reg.fit(X_train, y_train)

# Make predictions on the test set
y_pred = logistic_reg.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)
```

Example - Logistic regression (part 7)

```
# Print model performance metrics
print("Accuracy:", accuracy)
print("Confusion Matrix:\n", conf_matrix)
print("Classification Report:\n", classification_rep)
```

Output:

Accuracy: 0.7988826815642458

Confusion Matrix:

[[90 15]
 [21 53]]

Classification Report:

	precision	recall	f1-score	support
0	0.81	0.86	0.83	105
1	0.78	0.72	0.75	74
accuracy			0.80	179
macro avg	0.80	0.79	0.79	179
weighted avg	0.80	0.80	0.80	179

Explanation of Classification Metrics

In machine learning, classification tasks involve assessing the performance of models using various metrics. Here are explanations of commonly used classification metrics:

1. Precision:

- Precision measures how many of the predicted positive instances were actually positive.
- Formula: $\frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$
- Important for tasks where minimizing false positives is crucial, like medical diagnoses.

2. Recall (Sensitivity or True Positive Rate):

- Recall measures how many of the actual positive instances were correctly predicted as positive.
- Formula: $\frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$
- Important for tasks where minimizing false negatives is critical, such as fraud

detection.

3. F1-Score:

- The F1-score is the harmonic mean of precision and recall, providing a balanced measure of performance.
- Formula: $2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$
- Useful when class distribution is uneven in the dataset.

4. Support:

- Support represents the number of instances in each class, giving an idea of class distribution.
- Useful for understanding class importance and interpreting metrics.

5. Accuracy:

- Accuracy measures overall model performance, calculating the proportion of correct predictions.
- Formula: $\frac{\text{Correct Predictions}}{\text{Total Predictions}}$
- Common but may not be suitable for imbalanced datasets.

6. Macro Average:

- Macro average computes the unweighted average of precision, recall, and F1-score across all classes.
- Useful for assessing performance without considering class imbalance.

7. Weighted Average:

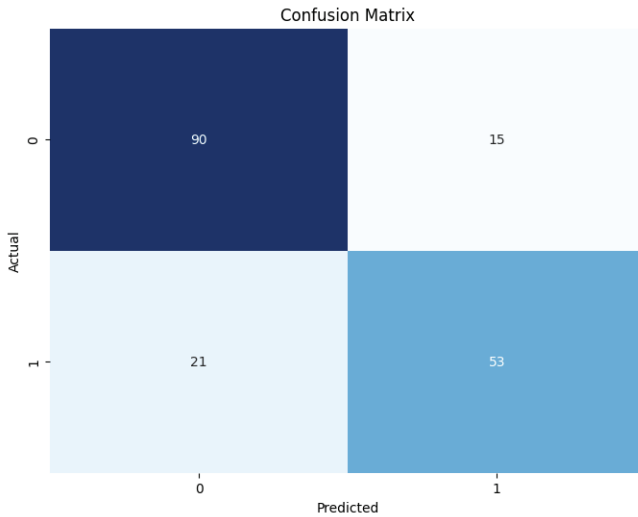
- Weighted average computes the average of precision, recall, and F1-score with more weight given to classes with more instances.
- Helpful for addressing class imbalance in the dataset.

These metrics provide a comprehensive evaluation of classification model performance, considering various aspects of accuracy and error based on the problem's specific requirements.

Example - Logistic regression (part 8)

```
# Visualize the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', cbar=False)
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

Output:



Explanation of Confusion Matrix

A confusion matrix is a table used to evaluate the performance of a classification model. It provides a detailed breakdown of predictions versus actual class labels and consists of four components:

- **True Positives (TP):**
 - The number of instances correctly predicted as the positive class.
 - These are cases where the model correctly identified the positive class.
- **True Negatives (TN):**
 - The number of instances correctly predicted as the negative class.
 - These are cases where the model correctly identified the negative class.
- **False Positives (FP):**
 - The number of instances incorrectly predicted as the positive class when they are actually negative.
 - Also known as Type I errors or false alarms.
- **False Negatives (FN):**
 - The number of instances incorrectly predicted as the negative class when they are actually positive.
 - Also known as Type II errors or misses.

The confusion matrix helps assess the model's performance and calculate various classification metrics, such as accuracy, precision, recall, and F1-score.

	Predicted Positive	Predicted Negative
Actual Positive	True Positives (TP)	False Negatives (FN)
Actual Negative	False Positives (FP)	True Negatives (TN)

Table 8.1: Confusion Matrix

Chapter 9

Decision Trees

A decision tree is a popular machine learning algorithm used for both classification and regression tasks. It is a graphical representation of a decision-making process that resembles an upside-down tree. Decision trees are used to model decisions or make predictions by recursively splitting a dataset into subsets based on the most significant attributes or features. These splits are determined by asking a series of questions, each of which helps narrow down the possibilities until a decision or prediction can be made.

In a decision tree, each internal node represents a decision or a test on an attribute, and each branch represents an outcome of that test. The leaves of the tree represent the final decisions or predictions. Decision trees are easy to understand and interpret, making them valuable tools for both machine learning beginners and experts.

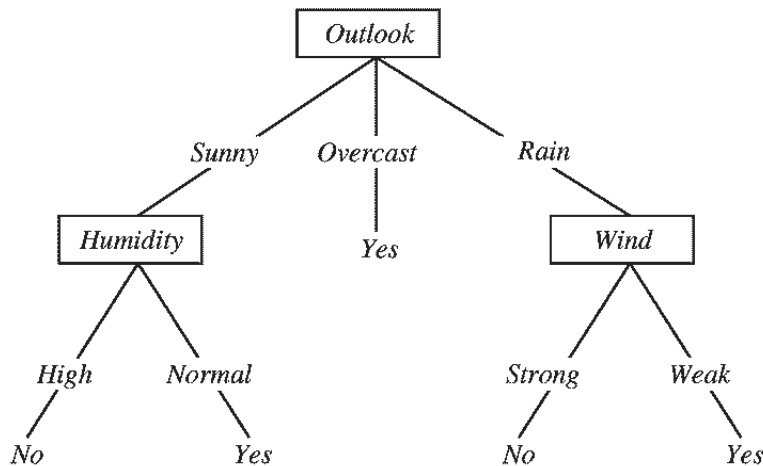


Figure 9.1: Example of Decision Tree

Decision Tree Algorithm

1. **Dataset:** Let D denote the dataset, where D consists of pairs $(\mathbf{X}, y)$, where $\mathbf{X}$ is a vector of features, and y is the target label or value.
2. **Tree Structure:** A decision tree is a binary tree. Each internal node represents a feature test, and it has two child nodes representing the outcomes of that test. Leaf nodes contain the final decision or prediction.

3. Splitting Criteria: At each internal node, the algorithm selects a feature and a threshold value to split the dataset into two subsets: D_{left} and D_{right} . The selection is based on criteria like Gini impurity, entropy, or mean squared error, depending on the task.

4. Recursive Splitting: The splitting process continues recursively on the subsets D_{left} and D_{right} until a stopping criterion is met (e.g., maximum depth, minimum samples per leaf) to prevent overfitting.

5. Label Assignment: When a leaf node is reached, it assigns a label or value to all data points in that leaf. In classification tasks, it could be the majority class; in regression tasks, it could be the mean or median of target values.

6. Prediction: To make predictions for a new input vector $\mathbf{X}_{\text{new}}$, the algorithm traverses the tree from the root node, following feature tests until a leaf node is reached. The label or value of the leaf node is the prediction.

Gini impurity measures the level of impurity or disorder in a dataset. In the context of a decision tree, it quantifies the probability of misclassifying a randomly chosen element in the dataset. The Gini impurity for a node with multiple classes or categories is given by:

$$\text{Gini}(p) = 1 - \sum_{i=1}^k (p_i)^2$$

where p is a vector of class probabilities for each class i and k is the number of classes.

The Gini impurity ranges from 0 (pure node, all samples belong to one class) to 0.5 (maximum impurity, samples are evenly distributed among all classes).

Entropy is a measure of impurity and disorder as well. It quantifies the uncertainty or randomness in the dataset. For a node with multiple classes, entropy is calculated as follows:

$$\text{Entropy}(p) = - \sum_{i=1}^k p_i \log_2(p_i)$$

where p is a vector of class probabilities for each class i and k is the number of classes.

Entropy ranges from 0 (pure node, all samples belong to one class) to $\log_2(k)$ (maximum entropy, all classes are equally distributed).

Mean Squared Error is used as a splitting criterion in regression decision trees. It measures the average squared difference between predicted values and actual values. For a node with regression targets y_i , MSE is calculated as:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \bar{y})^2$$

where n is the number of data points in the node, y_i are the actual target values, and $\bar{y}$ is the mean of the target values in the node.

A lower MSE indicates a better split in regression trees, as it represents less variance in the predicted values.

In the Decision Tree classification example, we will perform Cross-validation (CV) and Leave-One-Out Cross-Validation (LOOCV).

Cross-validation (CV) and Leave-One-Out Cross-Validation (LOOCV) are techniques used to assess the performance and generalization of machine learning models.

Cross-Validation (CV): Cross-validation is a resampling procedure used to evaluate machine learning models on a limited dataset. In traditional k-fold cross-validation, the dataset is divided into 'k' subsets or folds of approximately equal size. The model is trained on 'k-1' of these subsets and tested on the remaining fold. This process is repeated 'k' times, with each fold serving as the test set exactly once. The k accuracy scores obtained are then averaged to provide a more robust estimate of the model's performance. Common choices for 'k' are 3, 5, or 10, but they can be adjusted based on the dataset size and computational resources.

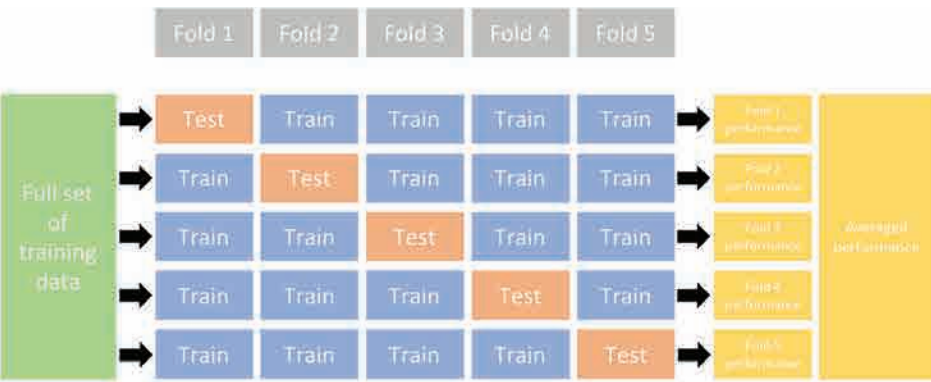


Figure 9.2: CV

Leave-One-Out Cross-Validation (LOOCV): LOOCV is a special case of cross-validation where 'k' is equal to the number of samples in the dataset. In LOOCV, for each fold, a single data point (sample) is used as the test set, while the rest of the data is used as the training set. This process is repeated for every data point in the dataset. LOOCV is particularly useful when you have a small dataset, but it can be computationally expensive for larger datasets. It provides a rigorous assessment of the model's performance because it tests the model on every single data point. The results are averaged to obtain a mean performance score.

The main purpose of both cross-validation and LOOCV is to provide a more accurate estimate of a model's performance and its ability to generalize to unseen

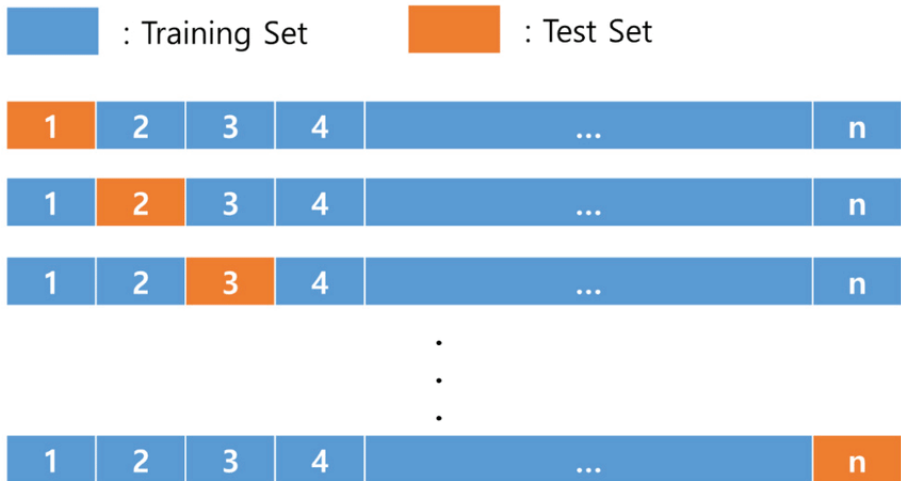


Figure 9.3: LOOCV

data. They help in detecting issues like overfitting (where a model performs well on the training data but poorly on new data) by assessing the model's performance on different subsets of the dataset. Cross-validation and LOOCV are valuable tools for model evaluation and selection, as they provide a more comprehensive and reliable assessment of a model's performance compared to a single train-test split.

Example - Decision Tree classification

```

import seaborn as sns
import pandas as pd
from sklearn.model_selection import train_test_split, cross_val_score,
↳ LeaveOneOut
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Load the "tips" dataset from Seaborn
tips = sns.load_dataset("tips")

# Create a binary classification target variable (tip > 5)
tips['above_5'] = (tips['tip'] > 5).astype(int)

# Select features and target
X = tips[['total_bill', 'size']]
  
```

```

y = tips['above_5']

# Split the dataset into a training and testing set
X_train, X_test, y_train, y_test = train_test_split(X, y,
    ↪ test_size=0.2, random_state=42)

# Create a decision tree classifier
clf = DecisionTreeClassifier(random_state=42)

# Fit the classifier to the training data
clf.fit(X_train, y_train)

# Make predictions on the test data
y_pred = clf.predict(X_test)

# Calculate accuracy on the test data
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy (Test Set): {accuracy:.2f}")

# Perform k-fold cross-validation
k = 3 # You can change the number of folds as needed
cv_scores = cross_val_score(clf, X, y, cv=k)
print(f"Cross-Validation Scores: {cv_scores}")
print(f"Mean Accuracy (Cross-Validation): {cv_scores.mean():.2f}")

# Perform Leave-One-Out Cross-Validation (LOOCV)
loo = LeaveOneOut()
loo_scores = cross_val_score(clf, X, y, cv=loo)
print(f"Mean Accuracy (Leave-One-Out CV): {loo_scores.mean():.2f}")

```

Output:

```

Accuracy (Test Set): 0.96
Cross-Validation Scores: [0.92682927 0.90123457 0.88888889]
Mean Accuracy (Cross-Validation): 0.91
Mean Accuracy (Leave-One-Out CV): 0.90

```

The provided results consist of accuracy scores for different evaluation methods. Here's how to interpret them:

Accuracy (Test Set): This is the accuracy of your model when evaluated on the test set. It tells you that your decision tree classifier correctly predicted the binary classification (tip > \$5) with an accuracy of 96%. In other words, 96% of the

test set samples were classified correctly.

Cross-Validation Scores: These are accuracy scores from k-fold cross-validation (in this case, k=3). The values in the array represent the accuracy for each fold. The cross-validation accuracy scores show how well your model generalizes to different subsets of the data. The values are quite close to each other and indicate that your model is performing consistently across different partitions of the data.

Mean Accuracy (Cross-Validation): This is the mean (average) accuracy of your model across all folds in the k-fold cross-validation. It provides an estimate of how well your model is likely to perform on new, unseen data.

Mean Accuracy (Leave-One-Out CV): This is the mean accuracy of your model when using Leave-One-Out Cross-Validation (LOOCV). LOOCV is a special case of cross-validation where each data point is left out as a separate test set. The mean accuracy of 0.90 suggests that your model performs well when tested on data points that were not used during training, providing a good estimate of its generalization performance.

In summary, the high accuracy on the test set (96%) and the consistent cross-validation results (around 91%) indicate that your decision tree classifier is performing well in this binary classification task, and it's likely to generalize well to new, unseen data. The results are in line with each other, which is a positive sign of model consistency.

Example - Decision Tree classification

```
import seaborn as sns
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import mean_squared_error

# Load the "tips" dataset from Seaborn
tips = sns.load_dataset("tips")

# Select features and target
X = tips[['total_bill', 'size']]
y = tips['tip']

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

```

# Create a decision tree regressor
regressor = DecisionTreeRegressor(random_state=42)

# Fit the regressor to the training data
regressor.fit(X_train, y_train)

# Make predictions on the test data
y_pred = regressor.predict(X_test)

# Calculate Mean Squared Error (MSE)
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse:.2f}")

```

Output:

Mean Squared Error: 1.00

Example - Other regression metrics

```

def mean_absolute_percentage_error(y_true, y_pred):
    y_true, y_pred = np.array(y_true), np.array(y_pred)
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100

def median_symmetric_accuracy(y_true, y_pred):
    Q = y_pred/y_true
    return 100 * (np.exp(np.median(np.abs(np.log(Q)))) - 1)

def symmetric_signed_percentage_bias(y_true, y_pred):
    Q = y_pred/y_true
    MdLQ = np.median(np.log(Q))

    return 100 * np.sign(MdLQ) * (np.exp(np.abs(MdLQ)) - 1)

def calculate_metrics(y_test, y_pred):

    mae = round(metrics.mean_absolute_error(y_test, y_pred),2)
    mse = round(metrics.mean_squared_error(y_test, y_pred),2)
    rmse = round(metrics.mean_squared_error(y_test, y_pred,
        ↳ squared=False),2)
    evs = round(metrics.explained_variance_score(y_test, y_pred),2)
    me = round(metrics.max_error(y_test, y_pred),2)

```

```

msle = round(metrics.mean_squared_log_error(y_test, y_pred),2)
med_ae = round(metrics.median_absolute_error(y_test, y_pred),2)
r2 = round(metrics.r2_score(y_test, y_pred),2)
mpd = round(metrics.mean_poisson_deviance(y_test, y_pred), 2)
mgd = round(metrics.mean_gamma_deviance(y_test, y_pred),2)
mape = round(mean_absolute_percentage_error(y_test, y_pred), 2)
msa = round(median_symmetric_accuracy(y_test, y_pred),2)
sspb = round(symmetric_signed_percentage_bias(y_test, y_pred),2)

print('Mean Absolute Error (MAE):', mae)
print('Mean Squared Error (MSE):', mse)
print('Root Mean Squared Error (RMSE):', rmse)
print('Explained Variance Score:', evs)
print('Max Error:', me)
print('Mean Squared Log Error:', msle)
print('Median Absolute Error:', med_ae)
print('R^2:', r2)
print('Mean Poisson Deviance:', mpd)
print('Mean Gamma Deviance:', mgd)
print('Mean Absolute Percentage Error (MAPE):', mape)
print('Median symmetric accuracy:', msa)
print('Symmetric Signed Percentage Bias (SSPB):', sspb)

return mae, mse, rmse, evs, me, msle, med_ae, r2, mpd, mgd, mape,
    ↪ msa, sspb

mae, mse, rmse, evs, me, msle, med_ae, r2, mpd, mgd, mape, msa, sspb =
    ↪ calculate_metrics(y_test, y_pred)

```

Output:

```

Mean Absolute Error (MAE): 0.82
Mean Squared Error (MSE): 1.0
Root Mean Squared Error (RMSE): 1.0
Explained Variance Score: 0.22
Max Error: 2.55
Mean Squared Log Error: 0.08
Median Absolute Error: 0.71
R^2: 0.2
Mean Poisson Deviance: 0.39
Mean Gamma Deviance: 0.17

```

Mean Absolute Percentage Error (MAPE): 35.12

Median symmetric accuracy: 33.33

Symmetric Signed Percentage Bias (SSPB): 10.0

Regression Metrics

Mean Absolute Error (MAE)

MAE measures the average absolute difference between the predicted and actual values. It provides a sense of the magnitude of error in the predictions.

$$\text{MAE} = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i| \quad (9.1)$$

Mean Squared Error (MSE)

MSE measures the average of the squared differences between the predicted and actual values. It emphasizes larger errors more than MAE.

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad (9.2)$$

Root Mean Squared Error (RMSE)

RMSE is the square root of the MSE. It provides a measure of the typical size of the error.

$$\text{RMSE} = \sqrt{\text{MSE}} \quad (9.3)$$

Explained Variance Score

Explained Variance quantifies the proportion of the variance in the dependent variable that is predictable from the independent variables.

$$\text{Explained Variance} = 1 - \frac{\text{Var}(y - \hat{y})}{\text{Var}(y)} \quad (9.4)$$

Max Error

Max Error measures the maximum absolute difference between predicted and actual values.

$$\text{Max Error} = \max_i |y_i - \hat{y}_i| \quad (9.5)$$

Mean Squared Log Error (MSLE)

MSLE measures the average of the squared differences between the logarithms of the predicted and actual values. It's often used when dealing with data with exponential growth patterns.

$$\text{MSLE} = \frac{1}{n} \sum_{i=1}^n (\log(1 + y_i) - \log(1 + \hat{y}_i))^2 \quad (9.6)$$

Median Absolute Error

Median Absolute Error is similar to MAE but uses the median instead of the mean. It's less sensitive to outliers.

$$\text{Median Absolute Error} = \text{median}(|y_1 - \hat{y}_1|, |y_2 - \hat{y}_2|, \dots, |y_n - \hat{y}_n|) \quad (9.7)$$

R-squared (R^2)

R-squared measures the proportion of the variance in the dependent variable explained by the independent variables. It's often called the coefficient of determination.

$$R^2 = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2} \quad (9.8)$$

Mean Poisson Deviance

Mean Poisson Deviance is used in Poisson regression to assess the goodness of fit by comparing the predicted values to actual values.

$$\text{Mean Poisson Deviance} = \frac{1}{n} \sum_{i=1}^n (2\hat{y}_i - 2y_i + 2y_i \log(\frac{y_i}{\hat{y}_i})) \quad (9.9)$$

Mean Gamma Deviance

Mean Gamma Deviance is used in Gamma regression to evaluate model performance.

$$\text{Mean Gamma Deviance} = \frac{1}{n} \sum_{i=1}^n \frac{2y_i - 2\hat{y}_i}{\hat{y}_i} \quad (9.10)$$

Mean Absolute Percentage Error (MAPE)

MAPE measures the average percentage difference between predicted and actual values, which is useful when dealing with data with varying scales.

$$\text{MAPE} = \frac{1}{n} \sum_{i=1}^n \frac{|y_i - \hat{y}_i|}{|y_i|} \times 100\% \quad (9.11)$$

Chapter 10

Deep Learning and AI Applications

Deep learning is a subfield of machine learning that focuses on neural networks with multiple layers, known as deep neural networks. It is a subset of artificial intelligence (AI) and is inspired by the structure and function of the human brain.

Neural Networks

Deep learning relies on artificial neural networks, computational models inspired by biological neurons. Neural networks consist of interconnected layers of artificial neurons, and deep networks have multiple hidden layers between the input and output layers.

Deep Architectures

Deep learning models are characterized by their deep architectures, which enable them to automatically learn hierarchical features and representations from data. This allows them to capture complex patterns and relationships in datasets.

Representation Learning

One of the key features of deep learning is representation learning. Deep neural networks learn to extract relevant features directly from raw input data during training. This eliminates the need for manual feature engineering.

Supervised and Unsupervised Learning

Deep learning supports both supervised learning, where models are trained on labeled data for tasks like predictions and classifications, and unsupervised learning, where models discover patterns in unlabeled data.

Large Datasets

Deep learning algorithms often require large datasets to generalize effectively. The availability of big data has been a significant driver of deep learning's success.

Backpropagation

Training deep neural networks involves backpropagating errors through the network to update the model's parameters (weights and biases). This optimization process minimizes a loss function that measures the difference between predicted and actual values.

Activation Functions

Deep neural networks use non-linear activation functions to introduce non-linearity into the model. Common activation functions include ReLU (Rectified Linear Unit), sigmoid, and tanh.

Convolutional Neural Networks (CNNs)

CNNs are a type of deep neural network designed for tasks involving grid-like data, such as images and videos. They use convolutional layers to learn local patterns and hierarchies of features.

Recurrent Neural Networks (RNNs)

RNNs are used for sequential data, such as natural language or time series. They have recurrent connections that allow them to maintain and utilize information from previous time steps.

Applications

Deep learning has a wide range of applications, including image and video analysis, natural language processing, speech recognition, autonomous vehicles, healthcare, and more.

In the following example, we will use CNN for MNIST digit classification.

Example - TensorFlow and Keras to perform image classification on MNIST dataset

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np
import matplotlib.pyplot as plt

# Load the MNIST dataset
```

```

mnist = keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) =
    ↪ mnist.load_data()

# Preprocess the data
train_images = train_images.reshape((60000, 28, 28, 1))
test_images = test_images.reshape((10000, 28, 28, 1))
train_images, test_images = train_images / 255.0, test_images / 255.0

# Create a simple convolutional neural network (CNN) model
model = keras.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28,
    ↪ 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])

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

# Train the model
model.fit(train_images, train_labels, epochs=5)

# Evaluate the model
test_loss, test_acc = model.evaluate(test_images, test_labels,
    ↪ verbose=2)
print(f'\nTest accuracy: {test_acc*100:.2f}%')

# Display some sample images
sample_images = test_images[:5] # Take the first 5 test images
sample_labels = test_labels[:5] # Corresponding labels

# Predict the labels for the sample images
sample_predictions = model.predict(sample_images)

```

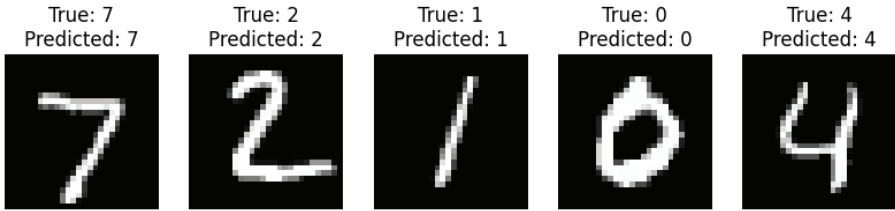
```
# Plot the sample images with their predicted and true labels
plt.figure(figsize=(10, 5))
for i in range(5):
    plt.subplot(1, 5, i + 1)
    plt.imshow(sample_images[i].reshape(28, 28), cmap='gray')
    plt.title(f"True: {sample_labels[i]}\nPredicted:
    ↪ {np.argmax(sample_predictions[i])}")
    plt.axis('off')

plt.show()
```

Output:

```
Downloading data from https://storage.googleapis.com/tensorflow/
tf-keras-datasets/mnist.npz
11490434/11490434 [=====] - 0s 0us/step
Epoch 1/5
1875/1875 [=====] - 54s 28ms/step
- loss: 0.1446 - accuracy: 0.9564
Epoch 2/5
1875/1875 [=====] - 54s 29ms/step
- loss: 0.0474 - accuracy: 0.9853
Epoch 3/5
1875/1875 [=====] - 54s 29ms/step
- loss: 0.0334 - accuracy: 0.9895
Epoch 4/5
1875/1875 [=====] - 51s 27ms/step
- loss: 0.0246 - accuracy: 0.9923
Epoch 5/5
1875/1875 [=====] - 51s 27ms/step
- loss: 0.0195 - accuracy: 0.9939
313/313 - 3s - loss: 0.0362 - accuracy: 0.9888 - 3s/epoch - 8ms/step

Test accuracy: 98.88%
1/1 [=====] - 0s 112ms/step
```



A **Convolutional Neural Network (CNN)** is a type of deep neural network designed for tasks involving structured grid-like data, such as images and videos. CNNs are particularly effective in computer vision tasks due to their ability to capture spatial dependencies in the data.

Convolution Operation

The fundamental building block of a CNN is the convolution operation. Given an input feature map I and a learnable kernel K , the convolution operation is defined as:

$$(S * I)(x, y) = \sum_i \sum_j I(x + i, y + j) \cdot K(i, j)$$

where $I(x, y)$ is the value at position (x, y) in the input feature map, $K(i, j)$ is the value of the kernel at position (i, j) , S represents the output of the convolution operation.

The convolution operation slides the kernel over the input feature map, element-wise multiplying and summing the values within the kernel's receptive field. This operation captures local patterns in the data.

Pooling

Pooling is used to downsample feature maps, reducing the spatial dimensions of the data. Common pooling operations include max pooling and average pooling. For max pooling, the output value is the maximum value within a fixed-size window. For average pooling, it's the average.

Activation Function

An activation function, typically ReLU (Rectified Linear Unit), is applied element-wise to the feature maps. It introduces non-linearity to the model.

$$\text{ReLU}(x) = \max(0, x)$$

Fully Connected Layers

After several convolution and pooling layers, CNNs typically have one or more fully connected layers for making final predictions.

Hyperparameters

The key parameters in a CNN include:

- **Kernel Size:** The size of the convolutional kernel, which determines the receptive field.
- **Number of Kernels:** The number of convolutional kernels in a layer.
- **Stride:** The step size at which the kernel moves during convolution.
- **Padding:** Padding with zeros to control the spatial dimensions of the feature map.
- **Pooling Size:** The size of the pooling window.
- **Activation Function:** Typically ReLU.
- **Learning Rate:** The step size for updating the model's parameters during training.
- **Loss Function:** The function used to calculate the error between predicted and actual values.
- **Optimization Algorithm:** Algorithms like Stochastic Gradient Descent (SGD) or Adam for updating model weights during training.

Bibliography

- [1] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 4th ed. Pearson, 2021.
- [2] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, et al. “Array programming with NumPy”. In: *Nature* 585.7825 (2020), pp. 357–362. DOI: [10.1038/s41586-020-2649-2](https://doi.org/10.1038/s41586-020-2649-2).
- [3] Wes McKinney. “Data Structures for Statistical Computing in Python”. In: *Proceedings of the 9th Python in Science Conference*. 2010, pp. 51–56.
- [4] J. D. Hunter. “Matplotlib: A 2D graphics environment”. In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: [10.1109/MCSE.2007.55](https://doi.org/10.1109/MCSE.2007.55).
- [5] F. Pedregosa, G. Varoquaux, A. Gramfort, et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.
- [6] Martín Abadi, Paul Barham, Jianmin Chen, et al. “TensorFlow: A system for large-scale machine learning”. In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016, pp. 265–283.
- [7] Adam Paszke, Sam Gross, Francisco Massa, et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32 (NeurIPS)*. 2019, pp. 8024–8035.

CIP - Каталогизација у публикацији
Народна библиотека Србије, Београд

004.43PYTHON(075.8)(076)
004.42:004.8(075.8)(076)

BEZDAN, Timea, 1987-

Python Workbook for Artificial Intelligence / Timea Bezdán. - 1. izd. - Beograd
: Singidunum University, 2025 (Belgrade : Birograf). - VI, 150 str. : ilustr. ; 24 cm
Tiraž 300. - Bibliografija: str. 149.

ISBN 978-86-7912-851-5

а) Програмски језик "Python" -- Вежбе б) Вештачка интелигенција -
- Програмирање -- Вежбе

COBISS.SR-ID 177504521

© 2025

All rights reserved. No part of this work covered by the copyright herein may be reproduced, transmitted, stored or used in any form or by any means graphic, electronic, or mechanical, including but not limited to photocopying, recording, scanning, digitizing, taping, Web distribution, information networks, or information storage and retrieval systems, without the prior written permission of the publisher.



Timea Bezdan

Python

Workbook for Artificial Intelligence

Artificial Intelligence (AI) is one of the fastest growing areas in computer science, mathematics, and engineering. Artificial Intelligence is a discipline that studies how to design systems capable of reasoning, learning, and problem solving. AI applications are now embedded in everyday technologies such as search engines, social media, recommender systems, medical diagnostics, and autonomous vehicles. Because of its broad influence, AI has become an essential subject of study for students in computing, engineering, and applied sciences.

In this book, Python is used as the primary programming language for exploring AI concepts and methods. Python is widely used in artificial intelligence and data science because of its simplicity and the availability of powerful libraries such as NumPy, Pandas, and Matplotlib. It also provides machine learning and deep learning frameworks such as scikit-learn, TensorFlow, and PyTorch. These advantages make Python an excellent choice for both beginners and advanced learners in AI.

