

AI & ML





Table of Contents

[**Chapter One: Python Refresher 3**](#_heading=h.90wvikz4j5p1)

[1. Python Basics: Variables, Data Types, and Control Structures 3](#_heading=h.8ot7nr4vx7k)

[2. Functions and Modules 8](#_heading=h.90vecjggh0nk)

[3. Introduction to NumPy for Numerical Operations 11](#_heading=h.1ysli4x95ufw)

[4. Introduction to Pandas for Data Manipulation 13](#_heading=h.3pbpnkk9uaxu)

[5. Basic Data Visualization with Matplotlib 15](#_heading=h.hih502yt24yk)

[6. Writing and Using Python Scripts and Modules 16](#_heading=h.5uxu2ooul0qc)

[7. Common Python Pitfalls and Best Practices 18](#_heading=h.2x27bpy67isa)

[**Chapter Two: Data Preprocessing and Feature Engineering 19**](#_heading=h.z33q04qwpqa4)

[1. Data manipulation and viewing with pandas 19](#_heading=h.wyx4om19ug0c)

[2. Handling Missing Data: Deletion, Imputation Techniques 20](#_heading=h.b1lo2iutiyu8)

[3. Feature Scaling: Normalization and Standardization 24](#_heading=h.am3bkhr44cyw)

[4. Encoding Categorical Variables: One-Hot Encoding, Label Encoding 26](#_heading=h.cjjsezghjf3o)

[5. Basic Feature Engineering: Creating New Features, Handling Outliers 27](#_heading=h.9zfkhwafs95)

[6. Detecting and Treating Outliers in Data 28](#_heading=h.d3sp0haqlgjb)

[7. Data Transformation Techniques for Improving Model Performance 29](#_heading=h.z21ibv1frjtg)

[**Chapter Three: Introduction to Machine Learning 31**](#_heading=h.cff8g4vetgxt)

[1. True Positive and True Negative 31](#_heading=h.imt6t3m9omqg)

[2. Confusion Matrix Example: 32](#_heading=h.ht1iwcxncyb4)

[3. Metrics Derived from These Concepts: 33](#_heading=h.r0zyru4svwjj)

[4. ROC Curve and AUC: 34](#_heading=h.j2bopwaxfvq4)

[5. Linear Equation 36](#_heading=h.clzdnu61v21q)

[6. Mean Squared Error (MSE) 36](#_heading=h.x8dp072x0925)

[7. R-squared (R²) 37](#_heading=h.xp0j6p6b8bki)

[8. Logistic Function 38](#_heading=h.b275zk8avaoi)

[9. Model Evaluation Metrics: Accuracy, Precision, Recall 38](#_heading=h.25l826hjupmp)

[10. Overfitting and Cross-Validation 39](#_heading=h.7or3xqo33l9u)

[11. Bias-Variance Tradeoff 42](#_heading=h.nvghr1vj20hs)

[12. Summary 44](#_heading=h.bt3cwfp9agla)

[**Chapter Four: Machine Learning Models 44**](#_heading=h.1soh41scve6o)

[1. Overview of Machine Learning: Definitions and Types 44](#_heading=h.918p0njofw76)

[2. Creating a New Dataset 47](#_heading=h.fnzqoupsk4it)

[3. Supervised Learning: Linear and Logistic Regression 49](#_heading=h.5yyxm6kdq844)

[4. Model Evaluation Metrics: Accuracy, Precision, Recall 53](#_heading=h.rc5rgsw706nq)

[5. Introduction to Overfitting and Cross-Validation 53](#_heading=h.fa1yj75cuncp)

[6. Understanding the Bias-Variance Tradeoff 54](#_heading=h.a5erul4m1orx)

[7. Best Practices for Training and Evaluating ML Models 55](#_heading=h.3uxvv8dsenp3)

# 

# 

# **Chapter One: Python Refresher**

## 1. Python Basics: Variables, Data Types, and Control Structures

**Introduction to Python**

Python is a high-level, interpreted programming language known for its readability and versatility. It is used in various domains such as web development, data science, machine learning, automation, and more.

**Variables and Data Types:**

* **Variables:** Containers for storing data values. In Python, you don’t need to declare a variable’s type explicitly. The interpreter infers the type based on the value assigned to the variable.

x = 10 # integer

y = 15.5 # float

name = "Alice" # string

is\_active = True # boolean

**Data Types:**

* **Integers**: Whole numbers, positive or negative.

a = 5

b = -3

* **Floats**: Numbers with a decimal point.

pi = 3.14159

gravity = 9.8

* **Strings**: Sequences of characters.

greeting = "Hello, World!"

* **Booleans**: `True` or `False`.

is\_open = True

* **Complex** **Numbers**: Numbers with a real and imaginary part.

complex\_num = 2 + 3j

**Lists**

* **Introduction:** Lists are ordered, mutable collections that can hold items of different data types.

fruits = ["apple", "banana", "cherry"]

* **Accessing Elements:** Use indices to access list elements.

print(fruits[0]) # Output: apple

* **Modifying** **Elements**: Lists are mutable, so elements can be changed.

fruits[1] = "blueberry"

* **List** **Methods**:

fruits.append("orange") # Adds an item to the end

fruits.remove("banana") # Removes an item by value

fruits.pop(1) # Removes an item by index

* **List** **Comprehensions**: Concise way to create lists.

squares = [x\*\*2 for x in range(10)]

**Tuples**

* **Introduction:** Tuples are ordered, immutable collections.

point = (10, 20)

* **Accessing** **Elements**: Use indices to access tuple elements.

print(point[0]) # Output: 10

* **Immutability**: Tuples cannot be changed after creation.

# point[1] = 30 # This will raise an error

* **Tuple Packing and Unpacking:**

coordinates = 1, 2

x, y = coordinates

**Sets**

* **Introduction:** Sets are unordered collections of unique elements.

fruits = {"apple", "banana", "cherry"}

* **Adding Elements:** Use `add()` method.

python

fruits.add("orange")

* **Set Operations:** Union, intersection, difference.

python

set1 = {1, 2, 3}

set2 = {3, 4, 5}

union\_set = set1.union(set2) # {1, 2, 3, 4, 5}

intersection\_set = set1.intersection(set2) # {3}

**Dictionaries**

* **Introduction:** Dictionaries are collections of key-value pairs.

python

person = {"name": "Alice", "age": 25}

* **Accessing Elements:** Use keys to access values.

print(person["name"]) # Output: Alice

* **Modifying Elements:** Dictionaries are mutable.

person["age"] = 26

* **Dictionary Methods:**

person.keys() # Returns all keys

person.values() # Returns all values

person.items() # Returns all key-value pairs

* **Dictionary Comprehensions**: Concise way to create dictionaries.

squares = {x: x\*\*2 for x in range(10)}

**Type Conversion:** Python allows for type conversion using functions like `int()`, `float()`, `str()`, etc.

x = 5 # integer

y = 3.2 # float

z = x + y # automatically converted to float

print(z) # Output: 8.2

# Explicit conversion

x = 5

y = "10"

z = x + int(y) # converting string to int

print(z) # Output: 15

**Control Structures:** Control structures allow you to control the flow of your program.

* **Conditional Statements:**

age = 20

if age >= 18:

print("You are an adult.")

elif age < 18 and age >= 13:

print("You are a teenager.")

else:

print("You are a child.")

**Loops:**

* **For Loop:**

for i in range(5): # 0 to 4

print(i)

* **While Loop:**

count = 0

while count < 5:

print(count)

count += 1

**Loop Control Statements:**

for i in range(10):

if i == 3:

continue # skips the rest of the loop for i = 3

if i == 8:

break # exits the loop when i = 8

print(i)

**List Comprehensions:** A concise way to create lists.

squares = [x\*\*2 for x in range(10)]

print(squares) # Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

**Exercises:**

1. Create a variable `radius` with a value of 7.5, calculate the area of a circle using this radius, and print the result. (Use the formula: `Area = π \* radius^2`).
2. Write a program that prints all the numbers from 1 to 50 that are divisible by 3.
3. Write a list comprehension that creates a list of the squares of all even numbers between 1 and 20.
4. Create a list of your favorite fruits and print each fruit.
5. Create a tuple with your top three favorite numbers and print the second number.
6. Create a set of your favorite hobbies and add a new hobby to the set.
7. Create a dictionary with keys as subjects and values as your scores in those subjects. Print the score of a particular subject.

## 2. Functions and Modules

**Functions**

Functions are blocks of reusable code that perform a specific task. They help to make your code more organized and modular.

**Defining a Function:**

def greet(name):

return f"Hello, {name}!"

print(greet("Alice")) # Output: Hello, Alice!

**Function Arguments:**

* **Positional Arguments**: Arguments that are passed in a specific order.

def add(a, b):

return a + b

print(add(5, 3)) # Output: 8

* **Keyword Arguments:** Arguments passed by explicitly naming the parameter.

def introduce(name, age):

return f"My name is {name} and I am {age} years old."

print(introduce(age=25, name="Bob")) # Output: My name is Bob and I am 25 years old.

* **Default Arguments:** Arguments that have a default value if not provided.

def greet(name, message="Hello"):

return f"{message}, {name}!"

print(greet("Charlie")) # Output: Hello, Charlie!

print(greet("Charlie", "Hi")) # Output: Hi, Charlie!

* **Variable-Length Arguments:** Allows you to pass an arbitrary number of arguments.

def sum\_all(\*args):

return sum(args)

print(sum\_all(1, 2, 3, 4)) # Output: 10

**Returning Values:** Functions can return a value using the `return` statement.

def multiply(x, y):

return x \* y

result = multiply(6, 7)

print(result) # Output: 42

```

**Anonymous Functions (Lambda Expressions):** Lambda functions are small, unnamed functions defined using the `lambda` keyword.

add = lambda x, y: x + y

print(add(2, 3)) # Output: 5

**Recursion:** A function that calls itself to solve a problem.

def factorial(n):

if n == 1:

return 1

else:

return n \* factorial(n - 1)

print(factorial(5)) # Output: 120

**Modules**

Modules are files containing Python code (functions, variables, etc.) that can be imported into other Python programs.

**Importing Modules:**

import math

print(math.sqrt(16)) # Output: 4.0

**Custom Modules:** You can create your own module by saving a `.py` file and importing it into another script.

# In my\_module.py

def greet(name):

return f"Hello, {name}!"

# In another script

import my\_module

print(my\_module.greet("Diana")) # Output: Hello, Diana!

**Exercises:**

1. Write a function `is\_even()` that checks if a number is even. Use this function to filter all even numbers from a list of numbers.
2. Create a custom module `calculator.py` with functions for addition, subtraction, multiplication, and division. Import this module into another script and use its functions.

## 3. Introduction to NumPy for Numerical Operations

**Introduction to NumPy**

NumPy is a powerful library for numerical computing in Python. It provides support for arrays, matrices, and a wide range of mathematical functions.

**Creating Arrays:**

import numpy as np

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

print(arr) # Output: [1 2 3 4 5]

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

print(matrix)

# Output:

# [[1 2 3]

# [4 5 6]]

**Array Operations:** NumPy allows for element-wise operations on arrays.

arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

print(arr1 + arr2) # Output: [5 7 9]

print(arr1 \* arr2) # Output: [ 4 10 18]

**Broadcasting:** Broadcasting allows you to perform operations on arrays of different shapes.

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

print(arr + 5) # Output: [6 7 8]

**Array Reshaping:** Reshaping allows you to change the shape of an array without changing its data.

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

6])

reshaped\_arr = arr.reshape((2, 3))

print(reshaped\_arr)

# Output:

# [[1 2 3]

# [4 5 6]]

**Statistical Operations:**

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

print(np.mean(data)) # Output: 3.0

print(np.std(data)) # Output: 1.4142135623730951

**Exercises:**

1. Create a NumPy array containing numbers from 1 to 50. Reshape this array into a 5x10 matrix.
2. Use NumPy to create an array of random numbers and compute the mean, median, and standard deviation.

## 4. Introduction to Pandas for Data Manipulation

**Introduction to Pandas**

Pandas is a powerful data analysis library that provides data structures like Series and DataFrames, which are essential for handling and analyzing structured data.

**Creating DataFrames:**

import pandas as pd

data = {

'Name': ['Alice', 'Bob', 'Charlie', 'David'],

'Age': [24, 27, 22, 32],

'City': ['New York', 'Los Angeles', 'Chicago', 'Houston']

}

df = pd.DataFrame(data)

print(df)

**DataFrame Operations:** Accessing data, modifying data, and performing operations.

print(df['Name']) # Accessing a column

print(df.iloc[1]) # Accessing a row by index

print(df.loc[df['Age'] > 25]) # Filtering data based on a condition

df['Age'] += 1 # Modifying data

print(df)

**Data Cleaning:** Handling missing values, removing duplicates, and transforming data.

df = pd.DataFrame({

'A': [1, 2, np.nan, 4],

'B': [5, np.nan, np.nan, 8],

'C': [10, 11, 12, 13]

})

df.fillna(0, inplace=True) # Replace NaN with 0

df.dropna(inplace=True) # Drop rows with NaN

print(df)

**Grouping and Aggregation:** Grouping data and performing aggregate functions.

df = pd.DataFrame({

'Department': ['HR', 'Engineering', 'HR', 'Engineering'],

'Employee': ['Alice', 'Bob', 'Charlie', 'David'],

'Salary': [50000, 60000, 55000, 65000]

})

grouped = df.groupby('Department').mean()

print(grouped)

**Exercises:**

1. Create a Pandas DataFrame from a CSV file. Filter the data to show only rows where a certain condition is met.
2. Group data in a DataFrame by a categorical column and calculate the mean for another column.

## 5. Basic Data Visualization with Matplotlib

**Introduction to Matplotlib**

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

**Creating Basic Plots:**

import matplotlib.pyplot as plt

# Line Plot

plt.plot([1, 2, 3, 4], [10, 20, 25, 30])

plt.title("Simple Line Plot")

plt.xlabel("X-axis")

plt.ylabel("Y-axis")

plt.show()

# Scatter Plot

plt.scatter([1, 2, 3, 4], [10, 20, 25, 30])

plt.title("Simple Scatter Plot")

plt.show()

**Customizing Plots:** Adding titles, labels, legends, and customizing the appearance.

plt.plot([1, 2, 3, 4], [10, 20, 25, 30], marker='o', linestyle='--', color='r')

plt.title("Customized Line Plot")

plt.xlabel("X-axis")

plt.ylabel("Y-axis")

plt.grid(True)

plt.show()

**Saving Plots:**

plt.plot([1, 2, 3, 4], [10, 20, 25, 30])

plt.savefig("plot.png") # Save as PNG

plt.savefig("plot.pdf") # Save as PDF

**Exercises:**

1. Create a bar plot that shows the number of students in different classes.
2. Create a scatter plot of two variables from a dataset and customize the plot with labels, title, and a grid.

## 6. Writing and Using Python Scripts and Modules

**Writing Python Scripts**

Scripts are files containing Python code that can be executed from the command line.

**Basic Script:**

# script.py

def greet(name):

return f"Hello, {name}!"

if \_\_name\_\_ == "\_\_main\_\_":

print(greet("World"))

**Run this script from the command line:**

python script.py

**Using Python Modules and Packages**

**Creating a Custom Module:** Save a Python file (e.g., `my\_module.py`) with some functions or classes, and import it in another script.

# my\_module.py

def add(a, b):

return a + b

# main.py

import my\_module

print(my\_module.add(5, 3)) # Output: 8

**Virtual Environments:** Virtual environments are isolated Python environments that allow you to manage dependencies for different projects.

**Creating and Activating a Virtual Environment:**

python -m venv myenv

source myenv/bin/activate # On Windows: myenv\Scripts\activate

**Installing Packages in a Virtual Environment:**

pip install numpy pandas matplotlib

**Exercises:**

1. Create a Python script that reads a CSV file, processes the data, and saves the result to a new file.
2. Set up a virtual environment for a project, install the necessary packages, and create a simple script that uses those packages.

## 7. Common Python Pitfalls and Best Practices

**Common Pitfalls**

**Mutable vs Immutable Data Types:** Understand how mutable (e.g., lists, dictionaries) and immutable (e.g., tuples, strings) types behave in Python.

# Mutable example

list\_a = [1, 2, 3]

list\_b = list\_a

list\_b.append(4)

print(list\_a) # Output: [1, 2, 3, 4]

# Immutable example

str\_a = "Hello"

str\_b = str\_a

str\_b += " World"

print(str\_a) # Output: "Hello"

**Best Practices:**

**Writing Clean Code:**

* Follow PEP 8 guidelines for code style.
* Use meaningful variable names.
* Comment your code where necessary.

**Exercises:**

1. Write a Python script that includes both mutable and immutable data types. Modify the variables and observe the differences.

# 

# **Chapter Two: Data Preprocessing and Feature Engineering**

## 1. Data manipulation and viewing with pandas

To replace a value in a DataFrame, you can use the replace() function or directly assign a new value to a specific cell using loc[]. Below is an example where we replace the NaN value in the 'Magic Power' column for 'Legolas' with a specific value, say 85.

import pandas as pd

import numpy as np

# Create a sample dataset

data = {

'Character': ['Gandalf', 'Legolas', 'Gimli', 'Saruman', 'Frodo', np.nan, 'Aragorn'],

'Magic Power': [95, np.nan, 10, 90, np.nan, 75, 50],

'Agility': [80, 95, 60, 70, 85, np.nan, 80],

'Wisdom': [100, 85, 70, 95, np.nan, 60, 90]

}

df = pd.DataFrame(data)

# Replace NaN value in 'Magic Power' column for 'Legolas' with 85

df.loc[df['Character'] == 'Legolas', 'Magic Power'] = 85

print(df)

**Simple Example of groupby**

You can group the data by a specific column and apply aggregation functions. Here's an example where we group by the 'Character' column and calculate the mean of the other columns:

# Group by 'Character' and calculate the mean of 'Magic Power', 'Agility', and 'Wisdom'

grouped\_df = df.groupby('Character').mean()

print(grouped\_df)

**Aggregate Examples**

You can use the agg() function to apply multiple aggregation functions at once. Here's an example where we group by the 'Character' column and apply different aggregation functions:

# Group by 'Character' and apply different aggregation functions

aggregate\_df = df.groupby('Character').agg({

'Magic Power': ['mean', 'max'],

'Agility': ['mean', 'min'],

'Wisdom': ['mean', 'sum']

})

print(aggregate\_df)

These examples should help you understand how to replace values, group data, and perform aggregation operations using pandas.

## 

## 2. Handling Missing Data: Deletion, Imputation Techniques

**Understanding Missing Data**

In datasets related to fantasy characters, missing data can occur due to incomplete records or information that was never collected. Just like in any other dataset, it is important to understand the nature of missing data and handle it appropriately to avoid biased models.

For example, suppose we have a dataset of characters from different fantasy races (Wizards, Elves, Dwarves) with attributes such as Magic Power, Agility, and Wisdom. Missing data in this context could mean missing values for some characters’ abilities.

**Techniques for Handling Missing Data**

* **Deletion Methods**

Deletion is one approach to handling missing data. In our fantasy characters dataset, we might want to remove characters with incomplete data, but this could lead to a loss of valuable information, especially if many records are missing data.

**Example:** Continuing with our fantasy characters dataset:

import pandas as pd

import numpy as np

# continuing with the previously created dataframe (df)

# Load the CSV to demonstrate deletion

df = pd.read\_csv('fantasy\_characters.csv')

# Listwise deletion: Remove rows with any missing values

listwise\_deleted\_df = df.dropna()

print("Listwise Deletion:\n", listwise\_deleted\_df)

# Pairwise deletion: Drop specific columns with too many missing values

pairwise\_deleted\_df = df.dropna(axis=1)

print("\nPairwise Deletion:\n", pairwise\_deleted\_df)

**Imputation Techniques**

Imputation is a more sophisticated method where missing data is filled with substituted values. In our fantasy dataset, we can use various imputation techniques to estimate the missing Magic Power, Agility, or Wisdom of characters.

* **Imputation using mean and median**

We will work with certain columns here

**Example:** Continuing with our fantasy characters dataset:

from sklearn.impute import SimpleImputer

from sklearn.impute import KNNImputer

# Mean Imputation for 'Magic Power'

mean\_imputer = SimpleImputer(strategy='mean')

df['Magic Power'] = mean\_imputer.fit\_transform(df[['Magic Power']])

# Median Imputation for 'Agility'

median\_imputer = SimpleImputer(strategy='median')

df['Agility'] = median\_imputer.fit\_transform(df[['Agility']])

print("After Mean and Median Imputation:\n", df)

* **Imputation using KNN**

Let us try & work by imputing the entire dataframe at one go, but KNNImputer in scikit-learn works only with numerical data like strings (e.g., the 'Character' column in your dataframe). To resolve this issue, you can either drop the non-numeric columns before applying the KNN imputer or separately handle the imputation for numeric and non-numeric columns. Here’s how you can do it:

**Approach 1: Drop the Non-Numeric Columns for KNN Imputation**

import pandas as pd

import numpy as np

from sklearn.impute import KNNImputer

# Create a sample dataset

data = {

'Character': ['Gandalf', 'Legolas', 'Gimli', 'Saruman', 'Frodo', np.nan, 'Aragorn'],

'Magic Power': [95, np.nan, 10, 90, np.nan, 75, 50],

'Agility': [80, 95, 60, 70, 85, np.nan, 80],

'Wisdom': [100, 85, 70, 95, np.nan, 60, 90]

}

df = pd.DataFrame(data)

# Separating numeric data

numeric\_cols = df.select\_dtypes(include=[np.number])

# KNN Imputation for numeric columns

knn\_imputer = KNNImputer(n\_neighbors=2)

df\_knn = pd.DataFrame(knn\_imputer.fit\_transform(numeric\_cols), columns=numeric\_cols.columns)

# Combine the imputed numeric data with the original non-numeric data

df\_final = df.copy()

df\_final.update(df\_knn)

print("\nAfter KNN Imputation:\n", df\_final)

**Approach 2: Impute Non-Numeric Data Separately**

If you want to keep the 'Character' column intact and handle the imputation only for numeric columns:

import pandas as pd

import numpy as np

from sklearn.impute import KNNImputer

# Create a sample dataset

data = {

'Character': ['Gandalf', 'Legolas', 'Gimli', 'Saruman', 'Frodo', np.nan, 'Aragorn'],

'Magic Power': [95, np.nan, 10, 90, np.nan, 75, 50],

'Agility': [80, 95, 60, 70, 85, np.nan, 80],

'Wisdom': [100, 85, 70, 95, np.nan, 60, 90]

}

df = pd.DataFrame(data)

# Separating numeric data

numeric\_cols = df.select\_dtypes(include=[np.number])

# KNN Imputation for numeric columns

knn\_imputer = KNNImputer(n\_neighbors=2)

df\_knn = pd.DataFrame(knn\_imputer.fit\_transform(numeric\_cols), columns=numeric\_cols.columns)

# Combine the imputed numeric data with the original non-numeric data

df\_combined = df.drop(columns=numeric\_cols.columns).join(df\_knn)

print("\nAfter KNN Imputation:\n", df\_combined)

In both approaches, the non-numeric columns like 'Character' are preserved as is, while the numeric columns are imputed using the KNN method.

## 

## 3. Feature Scaling: Normalization and Standardization

**Importance of Feature Scaling**

In machine learning, feature scaling is critical for ensuring that all features contribute equally to the model. For example, in a dataset of fantasy characters, Magic Power might range from 0 to 100, while Wisdom might have a narrower range. Without scaling, Magic Power could dominate the model, leading to biased predictions.

**Normalization**

Normalization scales features to a fixed range, typically [0, 1]. This is particularly useful for algorithms that rely on distance metrics, such as KNN or neural networks.

**Example:** Let's normalize the features in our modified fantasy characters dataset:

from sklearn.preprocessing import MinMaxScaler

# Normalization

scaler = MinMaxScaler()

df[['Magic Power', 'Agility', 'Wisdom']] = scaler.fit\_transform(df[['Magic Power', 'Agility', 'Wisdom']])

print("After Normalization:\n", df)

In this example, all numerical features are scaled to the range [0, 1].

**Standardization**

Standardization centers the data around zero with a standard deviation of one. This is especially important for algorithms like SVM or linear regression that assume normally distributed data.

**Example:** Standardizing the fantasy characters dataset:

from sklearn.preprocessing import StandardScaler

# Standardization

scaler = StandardScaler()

df[['Magic Power', 'Agility', 'Wisdom']] = scaler.fit\_transform(df[['Magic Power', 'Agility', 'Wisdom']])

print("After Standardization:\n", df)

In this example, the features are standardized so that they have a mean of 0 and a standard deviation of 1.

## 4. Encoding Categorical Variables: One-Hot Encoding, Label Encoding

**Understanding Categorical Variables**

Categorical variables in a fantasy dataset might include character types (Wizard, Elf, Dwarf) or weapon types (Sword, Bow, Staff). These need to be converted into a numerical format for machine learning models to process them.

**One-Hot Encoding**

One-Hot Encoding is useful for converting categorical variables into binary columns. For instance, in our fantasy dataset, we might want to create binary columns for each character type.

**Example:** Encoding character types:

# Add a categorical column for 'Character Type'

df['Character Type'] = ['Wizard', 'Elf', 'Dwarf', 'Wizard', 'Hobbit', 'Elf', 'Human']

# One-Hot Encoding

df = pd.get\_dummies(df, columns=['Character Type'])

print("After One-Hot Encoding:\n", df)

Here, each character type is converted into a binary column, ensuring that the model treats each type independently.

**Label Encoding**

Label Encoding assigns an integer to each category. This is suitable for ordinal data where the categories have a meaningful order.

**Example:** Encoding the ranks of fantasy characters:

from sklearn.preprocessing import LabelEncoder

# Add a categorical column for 'Rank'

df['Rank'] = ['A', 'B', 'C', 'A', 'B', 'C', 'A']

# Label Encoding

le = LabelEncoder()

df['Rank'] = le.fit\_transform(df['Rank'])

print("After Label Encoding:\n", df)

In this example, the ranks are converted into integers.

## 

## 5. Basic Feature Engineering: Creating New Features, Handling Outliers

**Creating New Features**

Feature engineering involves creating new features that capture additional information. For instance, combining Magic Power and Wisdom might give us a new feature called "Magical Intelligence," which could be a better predictor of a character's effectiveness.

Example: Creating new features on our modified dataset (df\_combined):

Note: We are using the dataset after KNN imputation (df\_combined)

# Create a new feature 'Magical Intelligence' by combining 'Magic Power' and 'Wisdom'

df\_combined['Magical Intelligence'] = df\_combined['Magic Power'] \* df\_combined['Wisdom']

print("New Feature - Magical Intelligence:\n", df\_combined)

**Handling Outliers**

Outliers in a fantasy dataset could be characters with extreme abilities that differ significantly from the rest. Detecting and handling outliers is important to prevent them from skewing the model.

**Example:** Handling outliers in the dataset continued from earlier example (df\_combined):

# Detecting Outliers using IQR

Q1 = df\_combined['Magic Power'].quantile(0.25)

Q3 = df\_combined['Magic Power'].quantile(0.75)

IQR = Q3 - Q1

outliers = df\_combined[(df\_combined['Magic Power'] < (Q1 - 1.5 \* IQR)) | (df\_combined['Magic Power'] > (Q3 + 1.5 \* IQR))]

print("Outliers Detected:\n", outliers)

# Handling Outliers by Capping

df\_combined['Capped Magic Power'] = np.where(df\_combined['Magic Power'] > df\_combined['Magic Power'].quantile(0.95),

df\_combined['Magic Power'].quantile(0.95), df\_combined['Magic Power'])

print("\nAfter Capping Outliers:\n", df\_combined)

## 6. Detecting and Treating Outliers in Data

**Advanced Outlier Detection Methods**

Advanced outlier detection methods, such as Isolation Forest, are useful in datasets with complex interactions between features, like in fantasy characters where a combination of abilities might create outliers.

**Example:** Using Isolation Forest to detect outliers in the dataset continued from earlier example (df\_combined):

from sklearn.ensemble import IsolationForest

# Detecting Outliers using Isolation Forest

iso\_forest = IsolationForest(contamination=0.1)

df\_combined['Outlier'] = iso\_forest.fit\_predict(df\_combined[['Magic Power', 'Agility', 'Wisdom']])

print("Outliers Detected by Isolation Forest:\n", df\_combined)

## 7. Data Transformation Techniques for Improving Model Performance

**Logarithmic Transformation**

Logarithmic transformation can reduce skewness in data, such as when dealing with highly skewed features like Magic Power in a dataset of fantasy characters.

**Example:** Applying logarithmic transformation in the dataset continued from earlier example (df\_combined):

# Logarithmic Transformation of 'Magic Power'

df\_combined['Log Magic Power'] = np.log(df\_combined['Magic Power'] + 1)

print("After Logarithmic Transformation:\n", df\_combined)

**Box-Cox Transformation**

The Box-Cox transformation can stabilize variance and make the data more normal, which is useful for improving the performance of models like linear regression.

**Example:** Applying Box-Cox transformation in the dataset continued from earlier example (df\_combined):

from scipy.stats import boxcox

# Box-Cox Transformation of 'Magic Power'

df\_combined['Magic Power BoxCox'], lam = boxcox(df\_combined['Magic Power'] + 1)

print("After Box-Cox Transformation:\n", df\_combined)

print("Lambda Value:", lam)

**Feature Interactions**

Creating interaction terms between features can capture complex relationships. For instance, multiplying Magic Power by Agility could create a feature representing "Combat Effectiveness."

**Example:** Creating interaction terms in the dataset continued from earlier example (df\_combined):

# Create an interaction term between 'Magic Power' and 'Agility'

df\_combined['Combat Effectiveness'] = df\_combined['Magic Power'] \* df\_combined['Agility']

print("Interaction Term - Combat Effectiveness:\n", df\_combined)

**Binning and Discretization**

Binning continuous variables into categories can make the data easier to interpret. For example, binning Magic Power into "Low", "Medium", and "High" categories.

**Example:** Binning power levels in the dataset continued from earlier example (df\_combined):

# Binning 'Magic Power' into categories

bins = [0, 30, 70, np.inf]

labels = ['Low', 'Medium', 'High']

df\_combined['Magic Power Category'] = pd.cut(df\_combined['Magic Power'], bins=bins, labels=labels)

print("Binned Magic Power:\n", df\_combined)

# **Chapter Three: Introduction to Machine Learning**

Before we delve deeper into machine Learning let us cover some terminologies and concepts which are essential in understanding the implementation and evaluation of ML models.

## 1. True Positive and True Negative

Imagine we are developing a model to predict whether a customer will default on a loan. The possible outcomes are:

* **Positive:** The customer defaults on the loan.
* **Negative:** The customer does not default on the loan.

**True Positives (TP):**

* **Definition:** True positives occur when the model correctly predicts that a customer will default on the loan.
* **Example:** Out of 100 customers, suppose 20 actually defaulted on their loans. If the model correctly predicts 15 of these 20 customers as defaulters, these 15 are true positives.

**True Negatives (TN):**

* **Definition:** True negatives occur when the model correctly predicts that a customer will not default on the loan.
* **Example:** Out of the 100 customers, suppose 80 did not default on their loans. If the model correctly predicts 70 of these 80 customers as non-defaulters, these 70 are true negatives.

**False Positives (FP):**

* **Definition:** False positives occur when the model incorrectly predicts that a customer will default on the loan when they actually will not.
* **Example:** In the same set of 100 customers, if the model predicts that 10 customers will default but they actually pay back their loans, these 10 are false positives. This is also known as a "Type I error."

**False Negatives (FN):**

* **Definition:** False negatives occur when the model incorrectly predicts that a customer will not default on the loan when they actually will.
* **Example:** If the model predicts that 5 customers will not default on their loans, but they end up defaulting, these 5 are false negatives. This is also known as a "Type II error."

## 

## 2. Confusion Matrix Example:

Let’s assume the following outcomes for the model's predictions:

* **Actual Defaults (Positive):** 20 customers
  + Model predicts 15 correctly (True Positives)
  + Model predicts 5 incorrectly (False Negatives)
* **Actual Non-Defaults (Negative):** 80 customers
  + Model predicts 70 correctly (True Negatives)
  + Model predicts 10 incorrectly (False Positives)

The confusion matrix would look like this:

|  | **Predicted Default (Positive)** | **Predicted Non-Default (Negative)** |
| --- | --- | --- |
| **Actual Default (Positive)** | True Positives (15) | False Negatives (5) |
| **Actual Non-Default (Negative)** | False Positives (10) | True Negatives (70) |

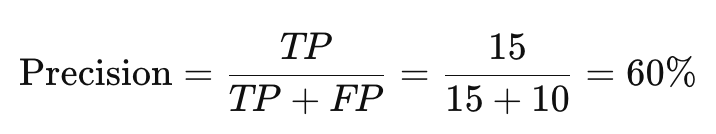
## 

## 3. Metrics Derived from These Concepts:

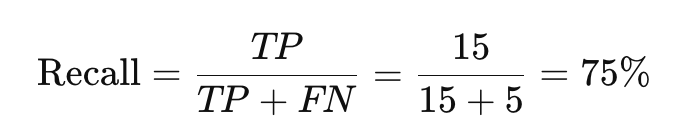
1. **Accuracy:** Measures the overall correctness of the model.



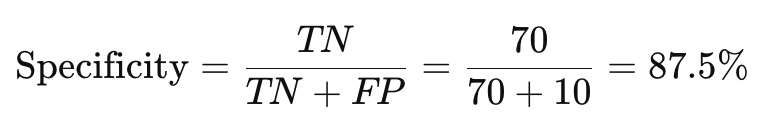
1. **Precision:** Measures the correctness of positive predictions.



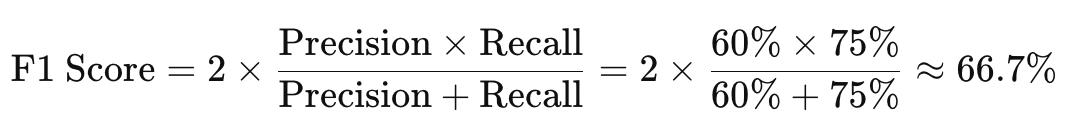
1. **Recall (Sensitivity or True Positive Rate):** Measures how well the model captures actual positives.



1. **Specificity (True Negative Rate):** Measures how well the model captures actual negatives.



1. **F1 Score:** Harmonic mean of precision and recall.

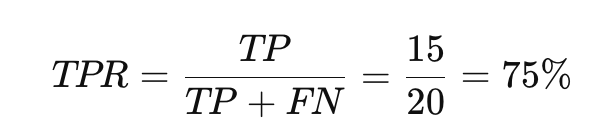


## 

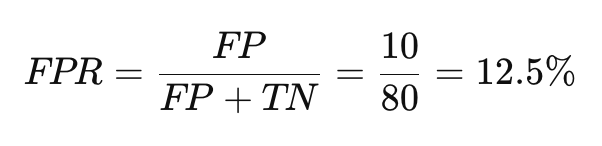
## 4. ROC Curve and AUC:

**ROC Curve:** The ROC curve is a graphical plot that shows the trade-off between the True Positive Rate (Recall) and the False Positive Rate as the decision threshold of the model changes.

* **True Positive Rate (TPR):**



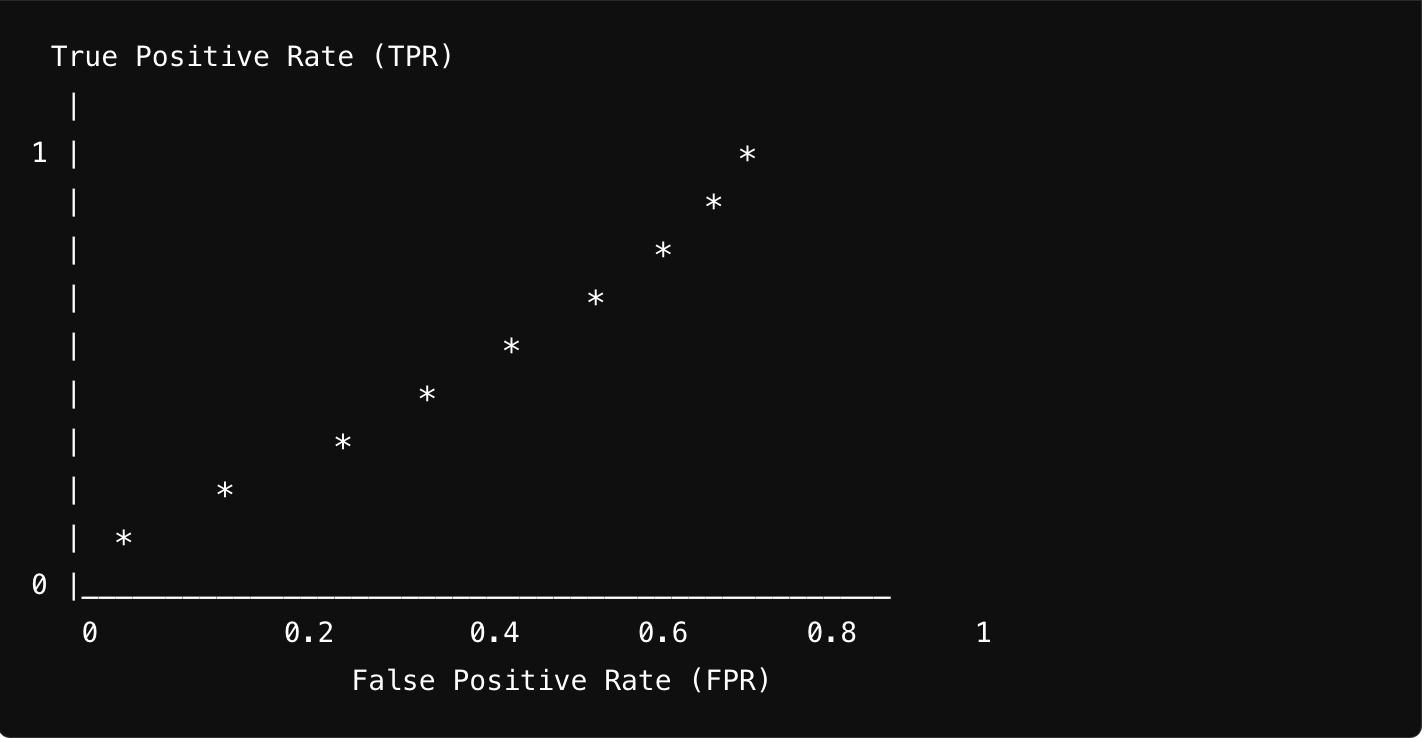
* **False Positive Rate (FPR):**



By varying the threshold at which a customer is classified as a defaulter, you can get different values for TPR and FPR, which can be plotted to form the ROC curve.

**Visual Representation of ROC Curve:**

Imagine a scenario where you have multiple thresholds. For each threshold, you calculate the TPR and FPR:



* The diagonal line represents a random model with no predictive power.
* A curve above the diagonal indicates better performance, with the model distinguishing between the two classes.
* The ideal curve would pass close to the top-left corner, representing high TPR and low FPR.

**AUC (Area Under the Curve):**

* **AUC = 0.5:** The model is no better than random guessing.
* **AUC > 0.5:** The model is better than random, with higher AUC values indicating better performance.
* **AUC = 1.0:** The model perfectly distinguishes between defaulters and non-defaulters.

**Example of AUC Calculation:**Suppose the ROC curve for your loan default model has an AUC of 0.85. This means that if you randomly choose one customer who defaulted and one who didn’t, there’s an 85% chance that the model will correctly assign a higher score to the defaulter.

**Importance of AUC:**

* **AUC as a Performance Metric:** AUC gives a single value that summarizes the model's ability to distinguish between classes across all thresholds.
* **Choosing Models:** A model with a higher AUC is generally preferred because it indicates better overall performance.

## 5. Linear Equation

A linear equation can be represented as y = mx + c, where m is the slope, x is the independent variable, and c is the intercept.

**Example:**

Let's plot a simple linear equation y = 2x + 1.

import numpy as np

import matplotlib.pyplot as plt

# Generate data

x = np.linspace(-10, 10, 100)

y = 2 \* x + 1

# Plotting the linear equation

plt.figure(figsize=(8, 6))

plt.plot(x, y, label='y = 2x + 1')

plt.title('Linear Equation: y = 2x + 1')

plt.xlabel('x')

plt.ylabel('y')

plt.axhline(0, color='black', linewidth=0.5)

plt.axvline(0, color='black', linewidth=0.5)

plt.grid(True)

plt.legend()

plt.show()

* We used np.linspace to generate 100 points between -10 and 10 for x.
* The corresponding y values are calculated using the equation y = 2x + 1.
* The plot visually represents the linear relationship between x and y.

## 

## 6. Mean Squared Error (MSE)

MSE measures the average of the squares of the errors between actual and predicted values. It's commonly used to evaluate regression models.

**Example:**

Let's calculate the MSE for a simple set of predictions.

# Actual values

y\_true = np.array([3, -0.5, 2, 7])

# Predicted values

y\_pred = np.array([2.5, 0.0, 2, 8])

# Calculate MSE

mse = np.mean((y\_true - y\_pred) \*\* 2)

print(f'Mean Squared Error: {mse}')

* y\_true represents the actual values.
* y\_pred represents the predicted values.
* We calculate MSE by taking the mean of the squared differences between y\_true and y\_pred.

## 7. R-squared (R²)

R-squared is a statistical measure that represents the proportion of the variance for a dependent variable that's explained by an independent variable(s) in a regression model.

**Example:**

Let's calculate the R-squared value for a simple set of predictions.

# Calculate R-squared

ss\_total = np.sum((y\_true - np.mean(y\_true)) \*\* 2)

ss\_residual = np.sum((y\_true - y\_pred) \*\* 2)

r\_squared = 1 - (ss\_residual / ss\_total)

print(f'R-squared: {r\_squared}')

* ss\_total is the total sum of squares, measuring the variance in the actual values.
* ss\_residual is the residual sum of squares, measuring the variance between the actual and predicted values.
* R-squared is calculated by the formula 1 - (ss\_residual / ss\_total), indicating how well the model explains the variance in the data.

## 8. Logistic Function

The logistic function is used in logistic regression and is defined as f(x) = 1 / (1 + e^(-x)). It outputs values between 0 and 1, often interpreted as probabilities.

**Example:**

Let's plot a logistic function.

# Logistic function

def logistic(x):

return 1 / (1 + np.exp(-x))

# Generate data

x = np.linspace(-10, 10, 100)

y = logistic(x)

# Plotting the logistic function

plt.figure(figsize=(8, 6))

plt.plot(x, y, label='Logistic Function')

plt.title('Logistic Function')

plt.xlabel('x')

plt.ylabel('f(x)')

plt.grid(True)

plt.legend()

plt.show()

* The logistic function converts input values to a probability between 0 and 1.
* We plot the function over a range of x values to visualize its S-shaped curve.

## 9. Model Evaluation Metrics: Accuracy, Precision, Recall

These metrics are used to evaluate classification models.

* **Accuracy**: The proportion of true results (both true positives and true negatives) among the total number of cases.
* **Precision**: The proportion of true positive results in the positive predicted results.
* **Recall**: The proportion of true positive results among all actual positive cases.

**Example:**

Let's calculate these metrics for a simple classification problem.

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score

# Actual labels

y\_true = np.array([1, 0, 1, 1, 0, 1, 0, 0, 1, 1])

# Predicted labels

y\_pred = np.array([1, 0, 1, 0, 0, 1, 1, 0, 1, 1])

# Calculate metrics

accuracy = accuracy\_score(y\_true, y\_pred)

precision = precision\_score(y\_true, y\_pred)

recall = recall\_score(y\_true, y\_pred)

print(f'Accuracy: {accuracy}')

print(f'Precision: {precision}')

print(f'Recall: {recall}')

* y\_true contains the actual class labels.
* y\_pred contains the predicted class labels.
* We use accuracy\_score, precision\_score, and recall\_score from sklearn to calculate these metrics.

Each of these concepts is fundamental in understanding machine learning and statistical modeling. Although these examples don't involve machine learning directly, they provide the mathematical and conceptual foundation needed to evaluate models effectively.

## 10. Overfitting and Cross-Validation

Overfitting occurs when a model learns not only the underlying pattern but also the noise in the training data, leading to poor generalization on unseen data. Cross-Validation is a technique used to assess the model's performance and prevent overfitting by dividing the data into training and validation sets multiple times.

**Example: Polynomial Regression with Overfitting**

Let's create a synthetic dataset and apply polynomial regression with different degrees to visualize overfitting and how cross-validation helps in selecting the right model.

import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import PolynomialFeatures

from sklearn.linear\_model import LinearRegression

from sklearn.model\_selection import train\_test\_split, cross\_val\_score

# Generate synthetic data

np.random.seed(42)

X = np.random.normal(0, 1, 100)

y = X \*\* 2 + np.random.normal(0, 0.1, 100)

# Split the data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

X\_train = X\_train.reshape(-1, 1)

X\_test = X\_test.reshape(-1, 1)

# Visualize overfitting with different polynomial degrees

degrees = [1, 2, 10]

plt.figure(figsize=(16, 4))

for i, degree in enumerate(degrees):

plt.subplot(1, 3, i+1)

# Polynomial features

poly = PolynomialFeatures(degree)

X\_train\_poly = poly.fit\_transform(X\_train)

X\_test\_poly = poly.fit\_transform(X\_test)

# Linear regression model

model = LinearRegression()

model.fit(X\_train\_poly, y\_train)

# Predict and plot

y\_pred = model.predict(X\_test\_poly)

plt.scatter(X\_test, y\_test, color='black', label='Test Data')

plt.plot(np.sort(X\_test, axis=0), model.predict(poly.fit\_transform(np.sort(X\_test, axis=0))), color='red', label=f'Degree {degree}')

plt.title(f'Polynomial Degree {degree}')

plt.xlabel('X')

plt.ylabel('y')

plt.legend()

plt.show()

* We generate synthetic data that follows a quadratic relationship (i.e., y = X^2).
* We fit polynomial regression models of different degrees (1, 2, and 10) to the data.
* The degree 1 model underfits, as it doesn't capture the quadratic nature of the data.
* The degree 2 model fits well, capturing the underlying relationship.
* The degree 10 model overfits, as it captures noise in the training data, leading to poor generalization.

**Using Cross-Validation to Prevent Overfitting**

# Cross-validation with different degrees

degrees = [1, 2, 10]

cv\_scores = []

for degree in degrees:

poly = PolynomialFeatures(degree)

X\_poly = poly.fit\_transform(X.reshape(-1, 1))

model = LinearRegression()

# Perform 5-fold cross-validation

scores = cross\_val\_score(model, X\_poly, y, cv=5, scoring='neg\_mean\_squared\_error')

cv\_scores.append(-scores.mean())

plt.figure(figsize=(8, 6))

plt.plot(degrees, cv\_scores, marker='o')

plt.title('Cross-Validation Score vs Polynomial Degree')

plt.xlabel('Polynomial Degree')

plt.ylabel('Negative Mean Squared Error')

plt.grid(True)

plt.show()

* We perform 5-fold cross-validation for polynomial models of different degrees.
* The plot shows the cross-validation scores (negative mean squared error) for each degree.
* A lower cross-validation error indicates a better model that generalizes well without overfitting.

## 11. Bias-Variance Tradeoff

The Bias-Variance Tradeoff describes the tradeoff between the error due to bias (error from incorrect assumptions in the model) and variance (error from sensitivity to small fluctuations in the training set).

* **High Bias:** Model is too simple, leading to underfitting.
* **High Variance:** Model is too complex, leading to overfitting.

**Example: Visualizing Bias-Variance Tradeoff**

We can visualize the bias-variance tradeoff by plotting the training and validation errors for models with varying complexity.

# Function to calculate training and validation errors

def plot\_bias\_variance\_tradeoff(max\_degree=15):

train\_errors = []

val\_errors = []

degrees = list(range(1, max\_degree+1))

for degree in degrees:

poly = PolynomialFeatures(degree)

X\_poly = poly.fit\_transform(X\_train)

X\_test\_poly = poly.fit\_transform(X\_test)

model = LinearRegression()

model.fit(X\_poly, y\_train)

train\_error = np.mean((model.predict(X\_poly) - y\_train) \*\* 2)

val\_error = np.mean((model.predict(X\_test\_poly) - y\_test) \*\* 2)

train\_errors.append(train\_error)

val\_errors.append(val\_error)

plt.figure(figsize=(8, 6))

plt.plot(degrees, train\_errors, label='Training Error', marker='o')

plt.plot(degrees, val\_errors, label='Validation Error', marker='o')

plt.title('Bias-Variance Tradeoff')

plt.xlabel('Model Complexity (Polynomial Degree)')

plt.ylabel('Mean Squared Error')

plt.legend()

plt.grid(True)

plt.show()

# Plotting the Bias-Variance Tradeoff

plot\_bias\_variance\_tradeoff()

* We fit polynomial models of varying degrees to the data.
* We plot the training and validation errors for each degree.
* Low-degree models (left side of the plot) have high bias and underfit, shown by high training and validation errors.
* High-degree models (right side of the plot) have high variance and overfit, shown by a low training error but a high validation error.
* The sweet spot is where the validation error is minimized, indicating the best tradeoff between bias and variance.

## 12. Summary

* Overfitting occurs when a model is too complex, leading to poor generalization. Cross-validation helps in selecting the model with the right complexity.
* The Bias-Variance Tradeoff is about finding the right balance between underfitting (high bias) and overfitting (high variance). By visualizing training and validation errors, you can select a model that generalizes well to unseen data.

These visualizations and scenarios will help you understand how to choose the right model complexity and avoid overfitting in practical applications.

# **Chapter Four: Machine Learning Models**

## 1. Overview of Machine Learning: Definitions and Types

**Machine Learning (ML)** is a subset of artificial intelligence (AI) that focuses on developing systems that can learn from data, identify patterns, and make decisions with minimal human intervention. Unlike traditional programming, where rules are explicitly coded, machine learning models learn from data to improve their performance over time.

**Types of Machine Learning**

Machine learning can be broadly categorized into three types based on the nature of the learning process:

1. Supervised Learning
2. Unsupervised Learning
3. Reinforcement Learning

Each category is associated with specific types of problems and models.

**Supervised Learning**

In supervised learning, the model is trained on a labeled dataset, which means that the input data is paired with the correct output. The goal is for the model to learn a mapping from inputs to outputs so that it can predict the output for new, unseen data.

**Common Models in Supervised Learning**

* **Linear Regression:** Predicts a continuous output (regression problems).
* **Logistic Regression:** Predicts categorical outcomes (binary or multiclass classification).
* **Support Vector Machines (SVM):** Used for classification and regression tasks, especially effective in high-dimensional spaces.
* **Decision Trees:** Used for both classification and regression; model decisions as a tree structure.
* **Random Forest:** An ensemble of decision trees, used for classification and regression.
* **K-Nearest Neighbors (KNN):** A non-parametric method used for classification and regression.
* **Naive Bayes:** Based on Bayes' theorem, used for classification tasks.
* **Neural Networks:** Inspired by the human brain, used for both classification and regression tasks.
* **Gradient Boosting Machines (GBM):** An ensemble technique that builds models sequentially, used for both classification and regression (e.g., XGBoost, LightGBM).
* **Ridge and Lasso Regression:** Extensions of linear regression that include regularization, used for regression tasks.

**Unsupervised Learning**

Unsupervised learning deals with unlabeled data, meaning the model tries to identify patterns or structures in the input data without reference to known outcomes. The model's goal is to infer the natural structure present within a set of data points.

**Common Models in Unsupervised Learning**

* **K-Means Clustering:** Partitions the data into K distinct clusters based on feature similarity.
* **Hierarchical Clustering:** Builds a tree of clusters, used to understand the hierarchy of clusters in the data.
* **Principal Component Analysis (PCA):** A dimensionality reduction technique that transforms the data into a set of orthogonal components.
* **Independent Component Analysis (ICA):** Similar to PCA, but focuses on making the components statistically independent.
* **Gaussian Mixture Models (GMM):** A probabilistic model that assumes the data is generated from a mixture of several Gaussian distributions.
* **Autoencoders:** A type of neural network used for unsupervised learning, primarily for dimensionality reduction or feature learning.
* **t-SNE (t-Distributed Stochastic Neighbor Embedding):** Used for visualizing high-dimensional data by reducing it to two or three dimensions.

**Reinforcement Learning**

Reinforcement learning is concerned with how an agent should take actions in an environment to maximize cumulative reward. The agent learns by interacting with the environment, receiving feedback in the form of rewards or punishments.

**Common Models in Reinforcement Learning**

* **Q-Learning:** A model-free algorithm that seeks to learn the value of actions in states to maximize the cumulative reward.
* **Deep Q-Networks (DQN):** A combination of Q-learning and deep learning, used for more complex tasks like playing video games.
* **Policy Gradient Methods:** Instead of learning the value of actions, these methods learn a policy that maps states to actions.
* **Actor-Critic Methods:** A hybrid approach that combines policy gradients (actor) and value-based methods (critic).
* **SARSA (State-Action-Reward-State-Action):** Similar to Q-learning but updates the Q-value based on the action taken in the next state.

**Hybrid Models and Other Techniques**

Some models and techniques don't fall strictly into one category. For example:

* **Semi-Supervised Learning:** Combines a small amount of labeled data with a large amount of unlabeled data.
* **Self-Supervised Learning:** The model generates its own labels based on the input data, often used in natural language processing (NLP).
* **Transfer Learning:** A model trained on one task is adapted to perform a different but related task.
* **Ensemble Methods:** Combine multiple models to improve performance, such as stacking or boosting (e.g., Random Forest, Gradient Boosting).

## 2. Creating a New Dataset

For this tutorial, we’ll simulate a dataset that could be used to predict house prices based on various features. This dataset will include information such as the number of bedrooms, bathrooms, square footage, location, and whether the house has a garden.

**Dataset Creation: House Prices**

import numpy as np

import pandas as pd

# Set random seed for reproducibility

np.random.seed(42)

# Create a dataset

n\_samples = 1000

# Number of Bedrooms: Randomly generated between 1 and 5

bedrooms = np.random.randint(1, 6, n\_samples)

# Number of Bathrooms: Randomly generated between 1 and 4

bathrooms = np.random.randint(1, 5, n\_samples)

# Square Footage: Normally distributed with mean 2000 and standard deviation 500

square\_footage = np.random.normal(2000, 500, n\_samples)

# Location: Categorical variable representing different locations (0: Suburban, 1: Urban, 2: Rural)

location = np.random.choice([0, 1, 2], n\_samples, p=[0.5, 0.3, 0.2])

# Garden: Binary variable indicating whether the house has a garden (0: No, 1: Yes)

garden = np.random.choice([0, 1], n\_samples, p=[0.7, 0.3])

# Price: Based on bedrooms, bathrooms, square footage, and location with some noise

price = (50000 + (bedrooms \* 30000) + (bathrooms \* 20000) + (square\_footage \* 100) +

(location \* 50000) + (garden \* 25000) + np.random.normal(0, 20000, n\_samples))

# Create a DataFrame

data = pd.DataFrame({

'Bedrooms': bedrooms,

'Bathrooms': bathrooms,

'SquareFootage': square\_footage,

'Location': location,

'Garden': garden,

'Price': price

})

# Display the first few rows

data.head()

* **Bedrooms**: Number of bedrooms in the house, ranging from 1 to 5.
* **Bathrooms**: Number of bathrooms in the house, ranging from 1 to 4.
* **Square Footage**: The area of the house in square feet, modeled with a normal distribution.
* **Location**: Categorical variable representing different locations (0: Suburban, 1: Urban, 2: Rural).
* **Garden**: Binary variable indicating whether the house has a garden.
* **Price**: The target variable, representing the house price, influenced by the features above and some added noise.

## 3. Supervised Learning: Linear and Logistic Regression

**Linear Regression**

* **Linear Regression** models the relationship between a dependent variable (target) and one or more independent variables (features) using a linear equation.

**Example: Predicting House Price Based on Features**

from sklearn.model\_selection import train\_test\_split

from sklearn.linear\_model import LinearRegression

from sklearn.metrics import mean\_squared\_error, r2\_score

# Define features and target

X = data[['Bedrooms', 'Bathrooms', 'SquareFootage', 'Location', 'Garden']]

y = data['Price']

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Create and train the linear regression model

lin\_reg = LinearRegression()

lin\_reg.fit(X\_train, y\_train)

# Predict on the test data

y\_pred = lin\_reg.predict(X\_test)

# Evaluate the model

mse = mean\_squared\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

# Display the results

print(f"Mean Squared Error: {mse}")

print(f"R-squared: {r2}")

# Display predicted vs actual values

predicted\_df = pd.DataFrame({'Actual': y\_test, 'Predicted': y\_pred})

print(predicted\_df.head())

* **Mean Squared Error (MSE)**: Measures the average squared difference between the predicted and actual prices. A lower MSE indicates a more accurate model.
* **R-squared (R²)**: Represents the proportion of the variance in the dependent variable that is predictable from the independent variables. An R² value close to 1 indicates a strong model.

**Visualizing the Regression Line**:

Since we have multiple features, we’ll visualize the regression results using a scatter plot of actual vs. predicted prices.

import matplotlib.pyplot as plt

# Plotting the actual vs predicted prices

plt.scatter(y\_test, y\_pred, color='blue')

plt.xlabel('Actual Prices')

plt.ylabel('Predicted Prices')

plt.title('Actual vs Predicted House Prices')

plt.plot([min(y\_test), max(y\_test)], [min(y\_test), max(y\_test)], color='red') # line of best fit

plt.show()

**Logistic Regression**

In this case, let’s predict whether a house is likely to have a high price (above $300,000) based on its features.

* **Logistic Regression** is used for binary classification tasks. It predicts the probability that a given input belongs to a certain class.

**Example: Predicting High Price Based on Features**

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, precision\_score, recall\_score

# Create a binary target variable: 1 if Price > 300,000, else 0

data['HighPrice'] = (data['Price'] > 300000).astype(int)

# Define features and target

X = data[['Bedrooms', 'Bathrooms', 'SquareFootage', 'Location', 'Garden']]

y = data['HighPrice']

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

# Create and train the logistic regression model

log\_reg = LogisticRegression()

log\_reg.fit(X\_train, y\_train)

# Predict on the test data

y\_pred = log\_reg.predict(X\_test)

# Evaluate the model using Accuracy, Precision, and Recall

accuracy = accuracy\_score(y\_test, y\_pred)

precision = precision\_score(y\_test, y\_pred)

recall = recall\_score(y\_test, y\_pred)

# Display the results

print(f"Accuracy: {accuracy}")

print(f"Precision: {precision}")

print(f"Recall: {recall}")

# Display predicted vs actual values

predicted\_df = pd.DataFrame({'Actual': y\_test, 'Predicted': y\_pred})

print(predicted\_df.head())

* **Accuracy**: The proportion of correctly predicted instances among the total number of instances.
* **Precision**: The ratio of correctly predicted positive observations to the total predicted positives.
* **Recall**: The ratio of correctly predicted positive observations to all observations in the actual class.

## 4. Model Evaluation Metrics: Accuracy, Precision, Recall

**Accuracy:** The proportion of correct predictions among the total number of predictions.

**Precision:** The proportion of true positive predictions out of all positive predictions.

**Recall:** The proportion of true positives out of all actual positives.

**Example Using Logistic Regression Predictions**:

# Precision and Recall calculation was already done in the Logistic Regression section

# We can print them again to emphasize their importance

print(f"Precision: {precision}")

print(f"Recall: {recall}")

## 5. Introduction to Overfitting and Cross-Validation

**Overfitting:** Overfitting occurs when a model learns not only the underlying patterns but also the noise in the training data. This results in excellent performance on the training data but poor generalization to new, unseen data.

**Cross-Validation:** Cross-validation is a technique used to assess the generalizability of a model. It involves splitting the dataset into multiple folds and training/testing the model on different combinations of these folds.

**5-Fold Cross-Validation Example**:

from sklearn.model\_selection import cross\_val\_score

# 5-fold cross-validation on the logistic regression model

cv\_scores = cross\_val\_score(log\_reg, X, y, cv=5)

print(f"Cross-Validation Scores: {cv\_scores}")

print(f"Mean CV Score: {np.mean(cv\_scores)}")

* **Cross-Validation**: The data is split into 5 folds. The model is trained on 4 folds and tested on the remaining fold. This process is repeated 5 times, each time with a different fold as the test set. The cross-validation scores help us assess the model's ability to generalize to new data.
* **Mean CV Score**: The average of the cross-validation scores gives a more reliable estimate of the model’s performance.

## 6. Understanding the Bias-Variance Tradeoff

**Bias:** Bias is the error introduced by approximating a real-world problem by a simplified model. High bias can lead to underfitting, where the model is too simple to capture the underlying patterns in the data.

**Variance:** Variance refers to the model's sensitivity to small fluctuations in the training dataset. High variance can lead to overfitting, where the model captures noise as well as the signal, resulting in poor generalization to new data.

**Tradeoff:** The bias-variance tradeoff is the balance between a model's complexity and its ability to generalize. A model with too much bias (high bias, low variance) will be overly simplistic and may underfit. A model with too much variance (low bias, high variance) will be overly complex and may overfit.

**Illustrating Bias-Variance Tradeoff Using Polynomial Regression**:

Let's create a polynomial regression model to see how increasing model complexity affects bias and variance.

from sklearn.preprocessing import PolynomialFeatures

# Creating a higher-degree polynomial regression model

poly = PolynomialFeatures(degree=5)

X\_poly = poly.fit\_transform(X)

# Train a linear regression model on the polynomial features

model = LinearRegression()

model.fit(X\_poly, y)

# Predict on training data

y\_train\_pred = model.predict(X\_poly)

# Evaluate on training data

train\_mse = mean\_squared\_error(y, y\_train\_pred)

print(f"Training MSE (Polynomial Degree 5): {train\_mse}")

# Plotting the actual vs predicted prices for polynomial regression

plt.scatter(y, y\_train\_pred, color='blue')

plt.xlabel('Actual Prices')

plt.ylabel('Predicted Prices')

plt.title('Actual vs Predicted House Prices (Polynomial Regression)')

plt.plot([min(y), max(y)], [min(y), max(y)], color='red') # line of best fit

plt.show()

* **Polynomial Regression**: By increasing the degree of the polynomial, the model becomes more flexible and can fit the training data more closely. However, this also increases the risk of overfitting.
* **Training MSE**: A very low training MSE indicates that the model is fitting the training data very well, but this might be at the cost of generalization. The actual vs. predicted plot can reveal how well the model performs on training data and whether overfitting is present.

## 

## 7. Best Practices for Training and Evaluating ML Models

**Data Preprocessing**

**Standardization**: Always standardize or normalize your features if they have different scales. This ensures that the model treats each feature equally.

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

**Encoding Categorical Variables**: Convert categorical variables into a format that can be provided to ML algorithms, like one-hot encoding.

**X\_encoded = pd.get\_dummies(data[['Location', 'Garden']], drop\_first=True)**

**Model Selection**

Start with simpler models (e.g., linear regression) and gradually move to more complex ones if needed (e.g., random forests, gradient boosting).

**Regularization**

Use regularization techniques like L1 (Lasso) and L2 (Ridge) to prevent overfitting by penalizing large coefficients.

from sklearn.linear\_model import Ridge

ridge\_reg = Ridge(alpha=1.0)

ridge\_reg.fit(X\_train, y\_train)

**Cross-Validation**

Always use cross-validation to ensure that your model generalizes well to unseen data.

**Feature Engineering**

* Spend time understanding your data and creating relevant features that capture the underlying patterns. Feature engineering can often be more impactful than choosing a more complex model.

**Monitoring and Maintenance**

* Continuously monitor model performance in production and update the model as necessary. Real-world data can change over time, leading to model degradation if not monitored.

**Example: Combining Regularization with Cross-Validation**

from sklearn.model\_selection import GridSearchCV

# Define the model

ridge\_reg = Ridge()

# Define the parameter grid

param\_grid = {'alpha': [0.1, 1.0, 10.0]}

# Implement grid search with cross-validation

grid\_search = GridSearchCV(ridge\_reg, param\_grid, cv=5, scoring='neg\_mean\_squared\_error')

grid\_search.fit(X\_train, y\_train)

# Best parameters and score

print(f"Best parameters: {grid\_search.best\_params\_}")

print(f"Best cross-validation score: {np.abs(grid\_search.best\_score\_)}")

* **GridSearchCV:** This method is used to find the optimal hyperparameters by testing different values (in this case, for the regularization parameter alpha) and using cross-validation to evaluate performance.
* **Best Parameters:** The parameter setting that provides the best performance across the cross-validation folds.