Artificial Intelligence & Machine Learning

Industrial Training Report

On

Project "TexifySympy Solver"

Undertaken at NIELIT, Ropar

SUBMITTED TO

National Institute of Electronics and Information Technology

(NIELIT),

Village Bada Phull, Ropar, Punjab

SUBMITTED BY

Name of Student	Roll No.
Prabhsimran Singh Mandare	15

SUPERVISED BY

Dr. Sarwan Singh

(Joint Director)

June 2025

ACKNOWLEDGEMENT

I express my sincere gratitude to **Dr. Sarwan Singh**, **Joint Director at NIELIT Ropar,** my industry mentor, for his invaluable guidance, continuous support, and encouragement throughout my industrial training in **Artificial Intelligence and Machine Learning (AI/ML)**. His expertise and mentorship have greatly enhanced my understanding of the subject, and I am truly grateful for the opportunity to learn under his supervision.

Signatures

Prabhsimran Singh Mandare
(**Roll No.: 15, 6 Month AIML Industrial Training, NIELIT, Ropar**)

Dr. Sarwan Singh
(**Joint Director at NIELIT, Ropar**)

DECLARATION

I, Prabhsimran Singh Mandare, a student of **4th year, ECE, CCET, Chandigarh,** hereby declare that the Industrial Training Report on **"Artificial Intelligence and Machine Learning"** conducted at **NIELIT Ropar** is my original work. The information presented in this report is accurate and complete to the best of my knowledge.

This report has been prepared under the supervision of **Dr. Sarwan Singh, Joint Director at NIELIT Ropar** and is submitted as part of my academic requirements.

(Signature)

Prabhsimran Singh Mandare

Roll No.: 15

Date: _____

Abstract

This report outlines the outcomes of an extensive industrial training program at NIELIT Ropar, focused on the domain of Artificial Intelligence and Machine Learning (AI/ML). The training comprised four key modules: Python programming, data analysis using NumPy and Pandas, machine learning and deep learning principles, and web development with HTML. Through guided mentorship and project-driven learning, the program emphasized real-world application of AI concepts, using tools such as Google Colab, Jupyter Notebook, and Matplotlib.

A strong emphasis was placed on practical project development, resulting in a portfolio of impactful AI solutions. The Deepfake Detection System utilized CNN-LSTM architectures and Grad-CAM to detect facial manipulation in images and videos, integrating face detection for accurate region targeting. The TexifySymPy Solver combined OCR with symbolic computation, enabling handwritten mathematical expression parsing and step-by-step solving using SymPy.

In the area of computer vision, projects were implemented for face detection and recognition, with deployment on platforms such as Hugging Face, Gradio and Streamlit. Another notable project, Quad-Predict, applied supervised learning techniques to predict and classify Parkinson's disease based on medical and behavioural data, offering a functional example of AI in healthcare diagnostics.

Together, these projects reflect the robust practical understanding acquired during training, aligning theoretical learning with hands-on application. The program successfully developed industry-ready skills in AI, machine learning, deep learning, computer vision, and data science positioning the trainee for future contributions in cutting-edge technological domains.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction to NIELIT Ropar

National Institute of Electronics & Information Technology (NIELIT) (erstwhile DOEACC Society), an Autonomous Scientific Society under the administrative control of Ministry of Electronics & Information Technology (MoE&IT), Government of India, was set up to carry out Human Resource Development and related activities in the area of Information, Electronics & Communications Technology (IECT). NIELIT is engaged both in Formal & Non-Formal Education in the area of IECT besides development of industry-oriented quality education and training programmes in the state-of-the-art areas. NIELIT has endeavoured to establish standards to be the country's premier institution for Examination and Certification in the field of IECT. It is also one of the National Examination Body, which accredits institutes/organizations for conducting courses in IT in the non-formal sector.

Figure 1.1: NIELIT Ropar

As on date, NIELIT has forty seven (47) centers located at Agartala, Aizawl, Ajmer, Alawalpur (Saksharta Kendra), Aurangabad, Bhubaneswar, Calicut, Chandigarh, Chennai, Chuchuyimlang, Churachandpur, Daman, Delhi, Dibrugarh, Dimapur, Gangtok, Gorakhpur, Guwahati, Haridwar, Imphal, Itanagar, Jammu, Jorhat, Kargil, Kohima, Kolkata, Kokrajhar, Kurukshetra, Lakhanpur (Saksharta Kendra), Leh, Lucknow, Lunglei, Majuli, Mandi, Pasighat, Patna, Pali, Ranchi, Ropar, Senapati, Shillong, Shimla, Silchar, Srinagar, Tezpur, Tura and Tezu with its Headquarters at New Delhi. It is also well networked throughout India with the presence of about 700 + institutes.

Figure 1.2: NIELIT Logo

Over the last two decades, NIELIT has acquired very good expertise in IT training, through its wide repertoire of causes, ranging from 'O' Level (Foundation), 'A' Level (Advance Diploma), 'B' Level (MCA equivalent), 'C' Level (M-Tech level), IT literacy courses such as CCC (Course on Computer Concept), BCC (Basic Computer Course) and other such long term and short term course in the non formal sector like courses on Information Security, ITeS-BPO(Customer Care/Banking), Computer Hardware Maintenance (CHM-O/A level), Bio-Informatics(BI-O/A/B level), ESDM etc, besides, high end courses offered by NIELIT Centres at Post-Graduate level (M.Tech) in Electronics Design & Technology, Embedded Systems etc. which are not normally offered by Universities/Institutions in the formal sector, in association with the respective state Universities.

The basket of activities of NIELIT is further augmented by the wide range of projects that it undertakes. NIELIT has demonstrated its capability and capacity to undertake R&D projects, consultancy services, turnkey projects in office automation, software development, website development etc. NIELIT is also the nodal implementing agency on behalf of MeitY for Data Digitization of the population of 15 assigned States and 2 Union Territories for the creation of National Population Register (NPR) project of Registrar General of India (RGI).

NIELIT is also successfully executing the Agriculture Census and Input Survey project under which tabulation of about 10 crore data records have to be done. NIELIT has planned a roadmap for adopting appropriate pedagogy for metamorphosing NIELIT into an Institute of National Importance.

1.1. About AIML Course

Artificial Intelligence is the intelligence exhibited by machines or software. The application areas of artificial intelligence are very vast and so this is a field of study which is gaining importance day by day. This branch of engineering emphasizes on creating intelligent machines that work and react like humans.

Course Highlights

- Joint assessment and certification
- Lectures by faculty of NIELIT and IIT Ropar
- Major thrust on Hands-on training
- Course curriculum jointly designed by NIELIT and IIT Ropar
- Exposure & access to high standards of IIT & NIELIT's industry-oriented approach

Course Outcome

- Industry Ready

- In depth practical knowledge of AI and ML
- Enhance Employability
- Visit to IIT Ropar

Objective of course: This joint certification programme is being undertaken in since 2019 and covers not only the Python Programming and its fundamental data structures, rather participants also learn how to program and work on data science libraries like Numpy and pandas, apply data analysis, data cleaning techniques, data visualization.

Four module programme followed by Industry relevant Project:

- Python Associate
- Data Analyst
- AI and ML Expert
- Neural Networks and Deep Learning Professional

1.2. Team Members

Dr Sarwan Singh
(Joint Director)

Industry Certified in: Software Development Certification held Microsoft Certified Professional (MCP), Oracle Database Administrator (OCP-DBA), Sun Certified Java Programmer (SCJP) Academy Lead at Cisco Local Academy (NIELIT Chandigarh) Certification held Cisco Certified Network Associate (CCNA), Cisco Certified Academy Instructor (CCAI)

Area of expertise: Artificial Intelligence and Machine Learning, Blockchain, Cloud computing, Augmented and Virtual Reality, Internet of Things (IoT)

Anita Budhiraja

(Scientist D)

25 years of teaching Experience. Taught various subjects like: Artificial Intelligence, Data Science in Python, Blockchain Technology, Web Application using ASP.Net and C#.Net, Data Structures, Database Management System, Network Security, Operating Systems, Unix.

Area of Expertise: Artificial Intelligence and Machine Learning ➢ Data Science in Python ➢ Blockchain Technology ➢ .Net Technology ➢ Software Engineering ➢ DBMS ➢ Data Structures ➢ Operating Systems

Dr. Arun Kumar

(Associate Professor)

Dr. Arun Kumar received his PhD degree in Mathematics from IIT Bombay in 2012. He also holds a Master degree in Industrial Mathematics from IIT Roorkee. His PhD work is related to Subordinated Stochastic Processes that have applications in finance, fractional partial differential equations and statistical physics. During his stint in financial industry as a research analyst, he worked on all the major asset classes i.e. Fixed Income, Equity, Currency, and Commodity. Further, he possesses a good experience in pricing and analysis of Bonds, Swaps, Total Return Swaps, Swap Curve Construction, Bonds Portfolio, Interest Rate Swaps Portfolio, Bond Futures, Cheapest to Deliver Calculations.

Chapter 2: Artificial Intelligence

2.1. What is Artificial Intelligence (A.I.)?

Artificial Intelligence (AI) is a branch of computer science that aims to create machines capable of mimicking human intelligence. The core idea behind AI is to enable computers to perform tasks that typically require human cognition, such as learning, reasoning, problem-solving, perception, and language understanding. AI systems are designed to interpret external data correctly, learn from such data, and use that learning to achieve specific goals and tasks through flexible adaptation. The field draws heavily from disciplines such as mathematics, psychology, linguistics, neuroscience, and computer engineering.

At the heart of AI is machine learning (ML), a subset that involves training algorithms to recognize patterns in data and make decisions without being explicitly programmed for every scenario. These algorithms improve over time with more data and feedback. Deep learning, a more advanced form of ML, uses artificial neural networks inspired by the structure of the human brain to process complex data like images, audio, and natural language.

2.2. Nomenclature of Artificial Intelligence

- **Big Data** refers to extremely large data sets that are computationally analyzed to reveal patterns, trends and unique associations related to human interactions and behaviors.
- **Collaborative systems** related to those models and algorithms for development of autonomous systems that can work collaboratively with other systems as well as with human entities.
- **Computer vision** is the most prominent form of machine perception currently available. This sub-area of AI has been significantly transformed by “deep learning.” Many computers are now able to perform some vision tasks better than humans. At present, significant research is underway on the further advancement of “computer vision” in the areas of automatic image and video captioning.
- **Deep learning** is a form of learning that has facilitated object recognition via images and video, along with activity recognition. Research is underway into other areas of perception, including audio, speech and natural language processing.
- **The Internet of Things (IoT)** encompasses concepts related to an array of devices, many of which are used in our everyday lives. Things like your appliances, home, office building, cameras and vehicles may all be (or soon will be) connected via the internet to permit the collection and sharing of information for intelligent purposes.

- **Natural Language Processing** is sometimes referred to, or coupled with, speech recognition. This form of AI is quickly developing for widely spoken languages associated with large data sets. At the same time, developers are refining NLP systems so that they can interact directly with people through simple dialog rather than specifically stylized requests. As part of this emerging form of AI, multi-lingual forms of NLP are being designed so that systems can interact with anyone speaking any language on the planet.
- **Reinforcement learning** shifts the focus of machine learning from pattern recognition to experience-driven decision-making. This technology will bring AI to the real world, and in doing so, impact millions of lives. Strides continue to be made in the practical implementation of this form of learning as part of a broadening of AI real-world environments.
- **Robotics** the process of developing and training robots to interact with the world in predictable ways. This includes the facilitation and manipulation of objects in interactive environments and with people. Substantial advances in robotics have been made in the past few years based upon the successes of other AI related technologies, including computer vision and other forms of machine perception.

2.3. Types of A.I.

The Figure below illustrates the four types of Artificial Intelligence (AI), categorized based on their capabilities and sophistication. These categories outline the progression of AI systems from basic to advanced levels, aiming to mimic human intelligence in increasingly complex ways.

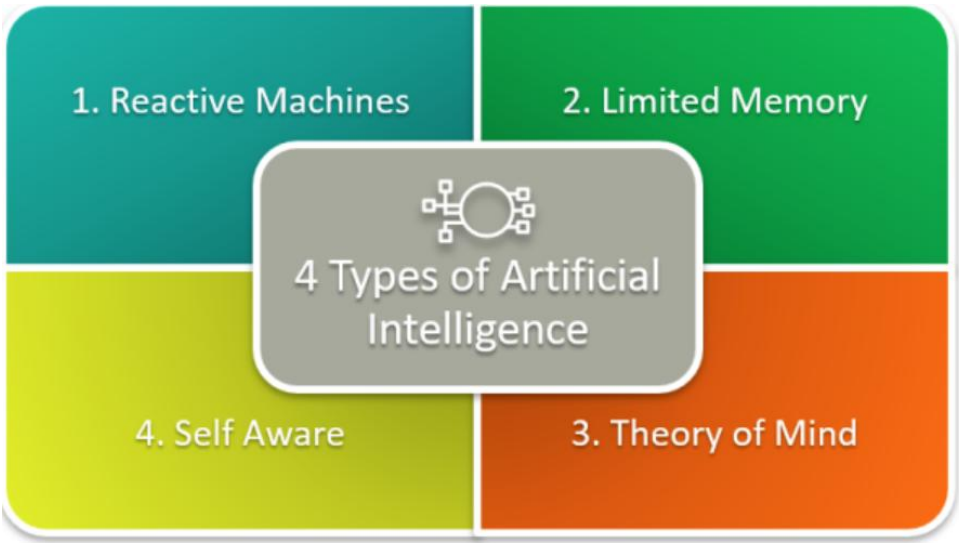

Figure 2.3: Different Types of A.I

1. Reactive Machines

Reactive machines are the most basic type of AI. These systems do not have memory or the ability to learn from past experiences. They simply react to specific inputs with programmed responses. They operate purely on the present data and cannot form inferences beyond their immediate task.

Example: IBM's Deep Blue chess-playing computer is a classic example. It could evaluate possible moves and their outcomes but had no memory of previous games or ability to improve over time. It functioned entirely by evaluating each move in the moment.

2. Limited Memory

This type of AI can store past data and use it to make better decisions. It is a more advanced version of reactive machines and forms the basis for many modern AI applications. These systems use historical data to learn and improve over time, though the memory is typically short-term or task-specific.

Example: Self-driving cars use limited memory AI to observe other vehicles, traffic signs, and road conditions. They store this information temporarily to make immediate decisions like changing lanes or stopping at traffic lights.

3. Theory of Mind

This is a conceptual and still largely theoretical form of AI. Theory of Mind AI aims to understand human emotions, beliefs, intentions, and thought processes. If achieved, such systems could interact with humans more naturally, recognizing and adapting to their mental states.

Example: In future applications, robots with Theory of Mind might be able to serve as companions, teachers, or caregivers, recognizing when a person is sad, distracted, or angry, and adjusting their behavior accordingly. However, we have not yet developed machines that exhibit true Theory of Mind.

4. Self-Aware

The most advanced and hypothetical form of AI, self-aware machines possess consciousness, self-understanding, and awareness of their internal states. They can think, reason, and potentially experience emotions like humans. This type of AI goes beyond simply understanding others—it understands itself.

Example: No real-world example exists yet. Self-aware AI is still in the realm of science fiction, appearing in movies like *Ex Machina* or *Her*, where machines are portrayed with human-like consciousness and emotional depth.

2.4. A.I. Types and Business Use Cases

The Figure showcases five main branches of Artificial Intelligence (AI) based on application domains. These categories illustrate how AI is embedded into different technologies to perform specialized tasks.

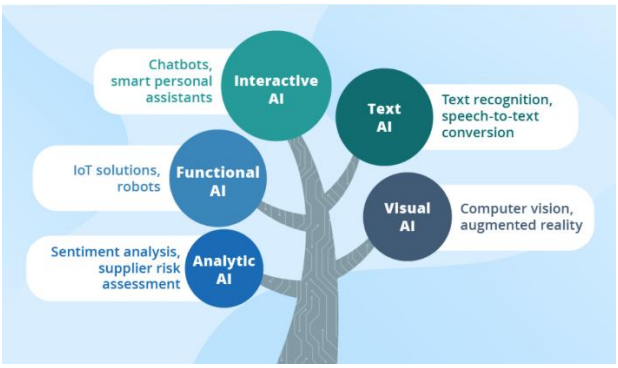

Figure 2.4. A.I. Types and Business use Cases

1. Interactive AI

Interactive AI focuses on human-computer interaction. It powers systems that can understand and respond to users through natural communication channels such as voice or text. These systems aim to mimic human conversation and behavior as closely as possible to provide assistance or entertainment.

Examples:

- Chatbots: Automated conversational agents used in websites, customer support, or messaging apps.
- Smart Personal Assistants: Devices like Amazon Alexa, Google Assistant, or Apple's Siri that can perform tasks like setting reminders, playing music, or answering questions.

Interactive AI relies heavily on natural language processing (NLP), speech recognition, and contextual understanding.

2. Text AI

Text AI specializes in understanding, generating, and manipulating human language in written form. It enables machines to read and understand text-based information, perform linguistic analysis, and convert speech into text or vice versa.

Examples:

- Text Recognition: Identifying and extracting text from images or scanned documents (OCR – Optical Character Recognition).
- Speech-to-Text Conversion: Converting spoken words into written text, useful in transcription, virtual meetings, and accessibility tools.

Text AI plays a crucial role in language translation, summarization, sentiment detection, and document classification.

3. Visual AI

Visual AI deals with interpreting and analyzing visual inputs like images and videos. It mimics the human ability to recognize and understand visual elements, enabling automation in visual tasks.

Examples:

- Computer Vision: Detecting objects, recognizing faces, reading barcodes, and analyzing image content.
- Augmented Reality (AR): Enhancing real-world environments with virtual overlays, often used in gaming, training, and shopping apps.

Visual AI is powered by deep learning, especially convolutional neural networks (CNNs), and is widely used in surveillance, healthcare imaging, autonomous vehicles, and manufacturing inspection.

4. Analytic AI

Analytic AI focuses on data interpretation and decision-making. It uses large datasets to identify trends, assess risks, and provide insights that help organizations make informed decisions.

Examples:

- Sentiment Analysis: Evaluating user opinions or emotions from text, often used in marketing and social media analysis.
- Supplier Risk Assessment: Identifying potential risks in supply chains based on performance history, market trends, or geopolitical events.

Analytic AI is fundamental in business intelligence, finance, cybersecurity, and predictive analytics.

5. Functional AI

Functional AI refers to systems that perform physical or system-level tasks, often by combining AI with hardware and automation. It enables machines to act, move, or respond in real-time.

Examples:

- IoT Solutions (Internet of Things): Smart devices that gather and process data to perform automated functions like adjusting room temperature or monitoring equipment.
- Robots: Machines that can navigate environments, perform industrial tasks, or assist in medical procedures.

Functional AI integrates sensors, real-time data, and adaptive algorithms to create responsive systems.

2.5. Evolution of AI, ML and DL

In the **1950s and 1960s**, AI emerged as a theoretical and experimental discipline. This period, often called the dawn of AI, saw the creation of early algorithms that could perform basic logical reasoning and solve puzzles. Researchers were enthusiastic about building machines that could mimic human intelligence, but progress was slow due to limited computational power and lack of data. This era is referred to as the **Artificial Intelligence** phase, where the focus was primarily on rule-based systems and symbolic logic.

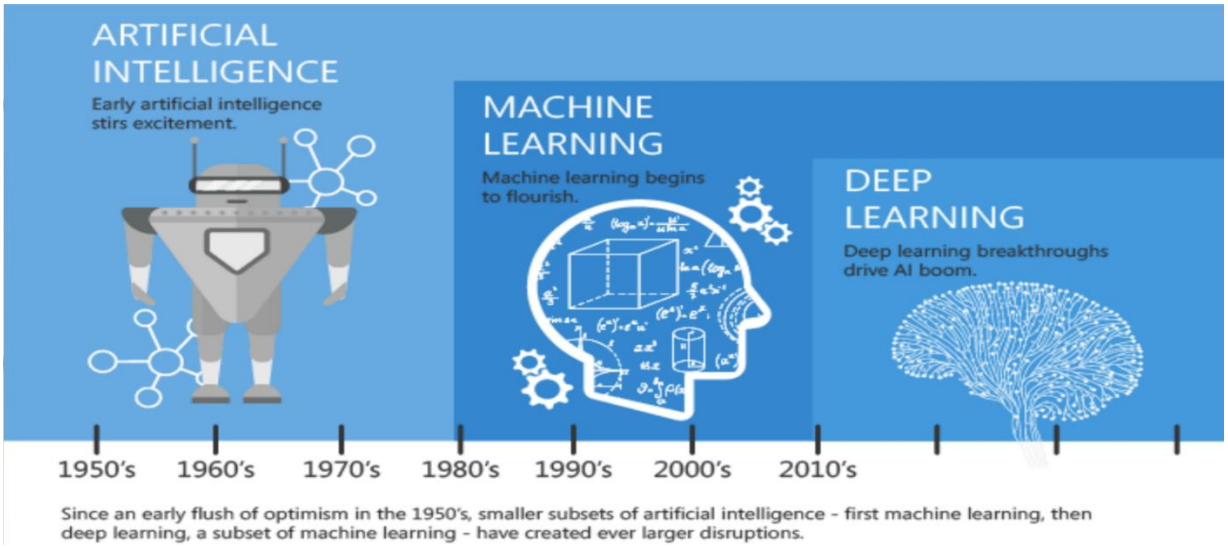

Figure 2.5: Evolution of AI, ML & DL

As we moved into the **1980s and 1990s**, the focus shifted toward **Machine Learning (ML)** - a subset of AI that allowed systems to learn from data instead of relying solely on explicit programming. Algorithms like decision trees, support vector machines, and Bayesian networks began to flourish. These advancements enabled computers to perform tasks such as spam detection, recommendation systems, and pattern recognition more effectively. ML marked a significant evolution because it introduced the idea that machines could improve their performance over time through exposure to data.

The **2010s and beyond** witnessed the explosive rise of **Deep Learning**, a specialized subset of machine learning that uses artificial neural networks to model and process complex patterns. With the advent of large datasets and powerful GPUs, deep learning has enabled breakthroughs in fields like image recognition, natural language processing, and autonomous driving. Technologies like convolutional neural networks (CNNs) and recurrent neural networks (RNNs) revolutionized how machines interpret visuals, language, and time-series data.

2.6. Relationship between AI, ML and DL

AI (Artificial Intelligence), ML (Machine Learning), and DL (Deep Learning) have a hierarchical relationship, where AI is the broadest term, ML is a subset of AI, and DL is a subset of ML. AI aims to create machines that mimic human intelligence, while ML focuses on enabling machines to learn from data without explicit programming. DL is a specific type of ML that uses neural networks to learn complex patterns from large datasets.

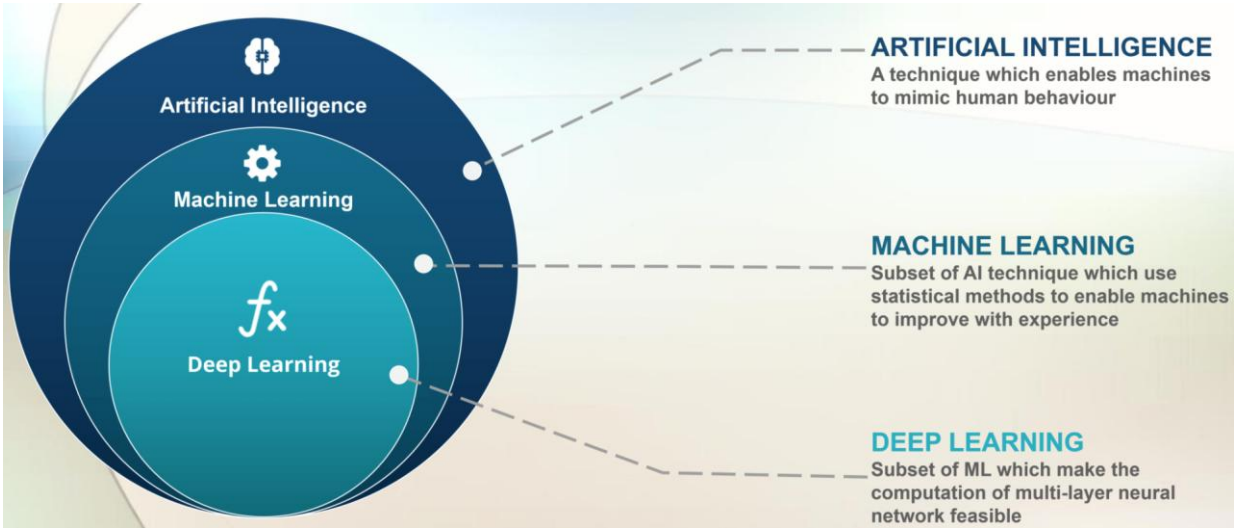

Figure 2.6: Relationship between AI, ML & DL

1. Artificial Intelligence (AI)

Artificial Intelligence is a broad field of computer science focused on creating machines that can mimic human intelligence and behaviour. It includes everything from simple rule-based systems to complex models capable of learning and decision-making. AI enables machines to perform tasks such as reasoning, problem-solving, language understanding, and perception—functions traditionally associated with human cognition. The goal is to build systems that can act intelligently and autonomously in various environments.

Examples:

- Self-driving cars
- Smart assistants (like Siri, Alexa)

- Facial recognition systems
- Spam filters

2. Machine Learning (ML)

Machine Learning is a subset of AI that allows systems to learn from data and improve their performance over time without being explicitly programmed for every task. It uses statistical techniques and algorithms to detect patterns in data, enabling machines to make decisions or predictions. As the system is exposed to more data, its accuracy and capability improve. ML is used in applications such as recommendation systems, fraud detection, and predictive analytics.

Examples:

- Recommendation engines (like Netflix or Amazon)
- Email spam detection
- Fraud detection systems
- Stock price prediction

3. Deep Learning (DL)

Deep Learning is a specialized subset of machine learning that uses multi-layered neural networks to model and analyze complex patterns in large datasets. It automates feature extraction and excels in processing unstructured data like images, audio, and text. Deep learning is the driving force behind modern AI breakthroughs in areas such as computer vision, speech recognition, and natural language processing. Its success relies heavily on high computational power and massive datasets.

Examples:

- Image classification (e.g., detecting objects in photos)
- Natural Language Processing (NLP) – chatbots, translators
- Voice recognition systems
- Autonomous vehicle perception systems

This layered relationship highlights how AI is the **umbrella concept**, with ML and DL being more focused, data-driven branches that have revolutionized the capabilities and real-world impact of AI technologies today.

Chapter 3: Introduction to Python

Python is a high-level, interpreted programming language widely used for web development, data analysis, artificial intelligence, automation, and scientific computing. It offers a simple syntax, extensive libraries, and strong community support, making it an ideal language for both beginners and experienced developers.

3.1. Jupyter Notebook – Installation & Function

Jupyter Notebook is an open-source interactive computing environment that allows users to write and execute Python code in a web-based interface. It is commonly used for data analysis, machine learning, and visualization. It can be installed using:

pip install jupyter

After installation, running jupyter notebook opens the web-based interface where users can create and execute Python scripts in separate cells.

3.2. Python - Operators, Expressions, and Statements

Python supports various operators that perform arithmetic, logical, and comparison operations. Expressions combine variables, constants, and operators to produce values, while statements are executable instructions in Python.

- **Arithmetic Operators**: +, -, *, /, **, %
- **Comparison Operators**: ==, !=, >, <, >=, <=
- **Logical Operators**: and, or, not

Example:

```
a = 10
b = 5
print(a + b)  # Output: 15
```

3.3. Conditional Statements and Loops

Conditional statements and loops allow for control flow in Python programs.

3.3.1 Conditional Statements

- if, elif, and else statements execute code based on conditions.

Example:

```
x = 20
if x > 10:
   print("Greater than 10")
```

```
elif x == 10:
    print("Equal to 10")
else:
    print("Less than 10")
```

3.3.2. Loops

- for loop iterates over sequences like lists and tuples.
- while loop runs until a specified condition is met.

Example:

```
for i in range(5):
    print(i)  # Outputs: 0 1 2 3 4
```

3.4. Sequence Data Types – List, Tuple, Set

Python provides built-in sequence data types to store collections of items.

- **List**: Mutable, ordered collection defined using [ ].
- **Tuple**: Immutable, ordered collection defined using ( ).
- **Set**: Unordered, unique collection defined using { }.

Example:

```
my_list = [1, 2, 3]
my_tuple = (4, 5, 6)
my_set = {7, 8, 9}
```

3.5. Input and Output in Python

Python uses input() to take user input and print() to display output.

Example:

```
name = input("Enter your name: ")
print("Hello, " + name)
```

3.6. Dictionary, Functions, and Lambda Functions

3.6.1. Dictionary

A dictionary is an unordered collection of key-value pairs defined using { }.

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

3.6.2. Functions

Functions are reusable blocks of code defined using def.

```
def greet(name):
    return "Hello, " + name
```

3.6.3. Lambda Functions

A lambda function is an anonymous function defined using lambda.

```
square = lambda x: x * x
print(square(5))  # Output: 25
```

3.7. Modules and Functions in Python

Modules are files containing Python code that can be imported into other programs.

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

3.8. Examples

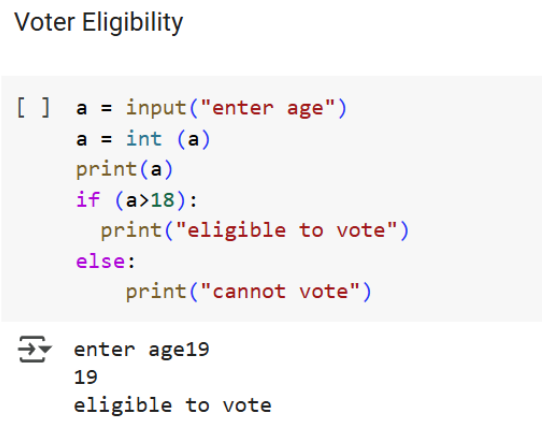

Figure 3.8.1: Checking Voter eligibility

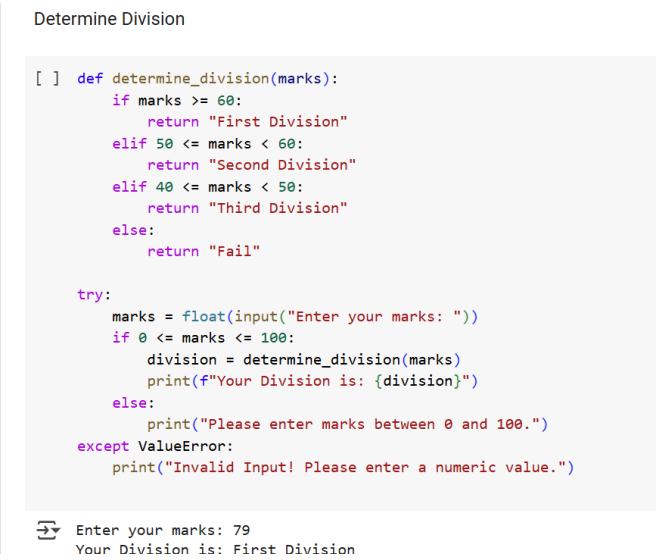

Figure 3.8.2: Determining Division Based Upon Marks

Reverse digits of a number

```
[ ]  n = 153
     d1 = n % 10
     d2 = (n%100)//10
     d3 = (n%1000)//100
     result = d1*100 + d2*10 + d3
     print(result)
```

```
351
```

Figure 3.8.3: Reversing the Digits of a Number

```
[ ]  pm1 = "Manhmohan Singh"
     for i in range (len(pm1)):
      print(pm1[i],end = "")
```

```
Manhmohan Singh
```

Figure 3.8.4: Printing a String

LIST

```
[ ]  l1 = ["venkat", 23, "deepanshu", 893.9]
     print(type(l1), l1)
```

```
<class 'list'> ['venkat', 23, 'deepanshu', 893.9]
```

Figure 3.8.5: Printing a List

```
[ ]  class1 = [[1, "amit"],[2, "sita"], [3, "laxman"]]
     class2 = [[1, "john"],[2, "shawn"], [3, "paul"]]
     school = [class1, class2 ]
     for i in school :
       print(i)
```

```
[[1, 'amit'], [2, 'sita'], [3, 'laxman']]
[[1, 'john'], [2, 'shawn'], [3, 'paul']]
```

Figure 3.8.6: Nested List

Sum of Nested list

```python
def sum_nested_list(nested_list):
    total = 0
    for item in nested_list:
        if isinstance(item, list):  # Check if the item is a list
            total += sum_nested_list(item)  # Recursive call for inner lists
        else:
            total += item  # Add the number directly if it's not a list
    return total

# Example usage
nested_list = [1, [2, 3], [4, [5, 6]], 7]
result = sum_nested_list(nested_list)
print(f"Sum of nested list: {result}")
```

Figure 3.8.7: Sum of Nested List using Recursion

3.9. Google Colab Notebook Link for Basics of Python:

https://colab.research.google.com/drive/1aY81Lhw687eBWgkMk6Gsd83XIRetcKy8?usp=sharing

Basics of Python

Chapter 4: Introduction to NumPy

Numerical Python (NumPy) has greater role for numerical computing in Python. It provides the data structures, algorithms, and library glue needed for most scientific applications involving numerical data in Python. It has fast and efficient multidimensional (N-dimensional) array object ndarray. Functions for performing element-wise computations with arrays or mathematical operations between arrays. It has tools for reading and writing array-based datasets to disk.
It is useful for Linear algebra operations, Fourier transform, and random number generation. It has sophisticated (broadcasting) functions. It has tools for integrating C/C++ and Fortran code. It provides an efficient interface to store and operate on dense data buffers. NumPy arrays form the core of nearly the entire ecosystem of data science tools in Python. Besides its obvious scientific uses, NumPy can also be used as an efficient multidimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases. NumPy is licensed under the BSD license.

4.1. Features of NumPy

1. **Multidimensional Arrays**: NumPy introduces the ndarray object, which is optimized for numerical operations.
2. **Broadcasting**: Allows operations on arrays of different shapes without explicit looping.
3. **Mathematical Functions**: Provides a vast collection of functions for mathematical computations such as trigonometry, algebra, and statistics.
4. **Linear Algebra Support**: Includes functions for matrix operations, eigenvalues, determinants, and more.
5. **Random Number Generation**: Supports generation of random numbers from various distributions.
6. **Interoperability**: Can integrate with other Python libraries like Pandas, SciPy, and TensorFlow.

4.2. Basic Operations with NumPy

- Every Python package that follows standard conventions includes a __version__ attribute. NumPy follows this convention, so you can check its version using:

```
[ ]  import numpy as np
     np.__version__

'1.26.4'
```

Figure 4.2.1: Checking Version of NumPy

This helps ensure that the correct version of NumPy is installed and compatible with your project.

- A NumPy array is a multi-dimensional container of elements, all of the same data type. It is much faster and more memory-efficient than a Python list. Arrays in NumPy are represented by the numpy.ndarray class.

```
[ ]  import numpy as np
     np1 = np.array([10, 20, 3, 4, 50])
     np1

array([10, 20,  3,  4, 50])
```

Figure 4.2.2: Creating an array

- In NumPy, the number of dimensions (ndim) of an array represents the number of axes (or levels) it has, such as 1D (vector), 2D (matrix), or 3D (tensor). The shape (shape) of an array is a tuple indicating the number of elements along each dimension, for example, a 2D array with 3 rows and 4 columns has a shape of (3, 4). The size (size) of an array refers to the total number of elements it contains, calculated as the product of its shape dimensions. These attributes help in understanding the structure of an array and are essential for performing numerical operations efficiently.

```
[ ]  import numpy as np

     np2 = np.array([[1, 2, 3, 4], [6, 7, 8, 9]])
     print(np2.size, np2.ndim, np2.shape)

8 2 (2, 4)
```

Figure 4.2.3: Checking Size, dimension and shape of array

- In NumPy, concatenation refers to joining two or more arrays along a specified axis using the numpy.concatenate() function. By default, arrays are concatenated along axis 0 (rows for 2D arrays), but this can be changed using the axis parameter. For example, concatenating two 1D arrays [1, 2, 3] and [4, 5, 6] results in [1, 2, 3, 4, 5, 6]. In the case of 2D arrays, concatenation along axis 0 stacks arrays row-wise, while along axis 1, it joins them column-wise. Additionally, functions like np.vstack() and np.hstack() provide specific ways to concatenate arrays vertically and horizontally. To concatenate arrays, they must have compatible shapes along the specified axis; otherwise, an error occurs.

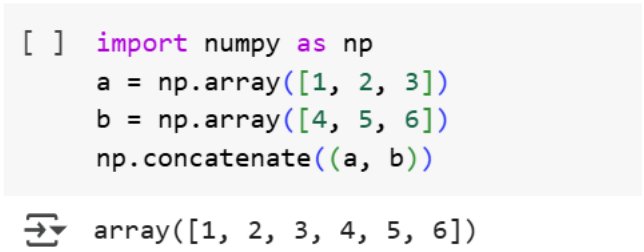

Figure 4.2.4: Concatenating Two Arrays

- In NumPy, the numpy.eye() function is used to create an identity matrix, which is a square matrix with ones on the main diagonal and zeros elsewhere. The function takes parameters N (number of rows) and M (optional, number of columns), where if M is not specified, it defaults to N, creating a square matrix. The optional k parameter shifts the diagonal, with k=0 placing ones on the main diagonal, k>0 shifting them above, and k<0 shifting them below. For example, np.eye(3) generates a 3×3 identity matrix, while np.eye(3, 4, k=1) creates a 3×4 matrix with ones on the first superdiagonal. This function is widely used in linear algebra, especially for identity transformations and solving matrix equations.

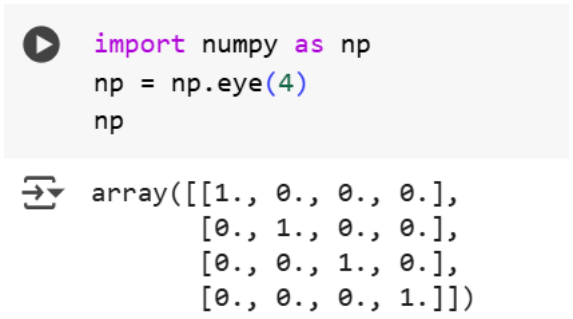

Figure 4.2.5: Creating Identity Matrix

- In NumPy, the numpy.mean() function is used to compute the arithmetic mean (average) of an array along a specified axis. The mean is calculated by summing all elements and dividing by the total number of elements. By default, np.mean() computes the mean of the entire array, but the axis parameter allows for row-wise (axis=1) or column-wise (axis=0) calculations in multi-dimensional arrays. For example, np.mean([1, 2, 3, 4]) returns 2.5. The function is useful in statistics, data analysis, and machine learning for understanding the central tendency of a dataset. Additionally, the dtype parameter can be specified to prevent overflow in integer arrays, ensuring accurate results in large datasets.

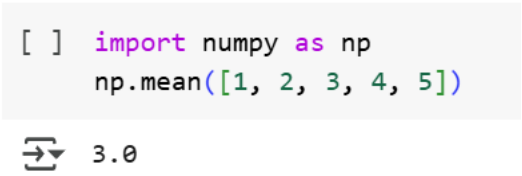

Figure 4.2.6: Calculating Mean of an Array

- In NumPy, the numpy.sum() function is used to compute the **sum of all elements** in an array along a specified axis. By default, np.sum() adds up all the elements in the array, but using the axis parameter, summation can be performed along rows (axis=1) or columns (axis=0) in multi-dimensional arrays. For example, np.sum([1, 2, 3, 4]) returns 10, while np.sum([[1, 2], [3, 4]], axis=0) results in [4, 6]. The function also supports a dtype parameter to prevent overflow in integer operations and ensure precision. numpy.sum() is widely used in numerical computing, data analysis, and scientific computations where aggregation of values is required.

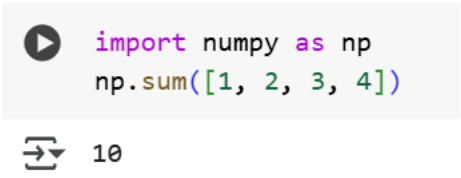

Figure 4.2.7: Calculating Sum of an array

- In NumPy, the numpy.max() function is used to find the maximum value in an array along a specified axis. By default, np.max() returns the highest value from the entire array, but by using the axis parameter, it can compute the maximum along rows (axis=1) or columns (axis=0) in multi-dimensional arrays. For example, np.max([1, 5, 3, 9]) returns 9, while np.max([[1, 2], [3, 4]], axis=0) results in [3, 4], giving the maximum values column-wise.

This function is commonly used in data analysis, machine learning, and image processing to identify peak values in datasets.

```
import numpy as np
np1 = np.array([10, 20, 3, 4, 50])
np1.max()
```

```
50
```

Figure 4.2.8: Finding Maximum of an Array

- In NumPy, indexing is used to access specific elements or subsets of an array. Similar to Python lists, NumPy arrays use zero-based indexing, meaning the first element is at index 0. For 1D arrays, elements can be accessed using their index, such as arr[2] to get the third element. In 2D arrays (matrices), indexing follows arr[row, column] notation, where arr[1, 2] retrieves the element from the second row and third column. Negative indexing allows access from the end, like arr[-1] for the last element. NumPy also supports slicing, such as arr[1:4] to get elements from index 1 to 3. Additionally, boolean indexing and fancy indexing (using index arrays) enable advanced selection of elements. Indexing is essential for efficient data manipulation and extraction in numerical computations.

```
import numpy as np

np2 = np.array([[1, 2, 3, 4], [6, 7, 8, 9]])
np2 [0, 3]
```

```
4
```

Figure 4.2.9: Indexing

4.3. Google Colab Notebook Link for NumPy

https://colab.research.google.com/drive/1zepm8t_hpsU2arAndMLcVNE78lpDPO3W#scrollTo=zhD7n8Q3NCax

NumPy

Chapter 5: Web Development Using HTML

HTML stands for Hyper Text Markup Language. It is the standard markup language for creating Web pages which is used to describe the structure of a Web page. It consists of a series of elements where elements tell the browser how to display the content. The elements label pieces of content such as "this is a heading", "this is a paragraph", "this is a link", etc.

5.1. Simple HTML Document

index.html × style.css × script.js ×

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <title>Page Title</title>
5  </head>
6  <body>
7
8  <h1>My First Heading</h1>
9  <p>My first paragraph.</p>
10
11 </body>
12 </html>
```

Figure 5.1: HTML Document

5.2. HTML Elements

- The `<!DOCTYPE html>` declaration defines that this document is an HTML5 document
- The `<html>` element is the root element of an HTML page
- The `<head>` element contains meta information about the HTML page
- The `<title>` element specifies a title for the HTML page (which is shown in the browser's title bar or in the page's tab)
- The `<body>` element defines the document's body, and is a container for all the visible contents, such as headings, paragraphs, images, hyperlinks, tables, lists, etc.
- The `<h1>` element defines a large heading
- The `<p>` element defines a paragraph

Figure 5.2: HTML Elements

An HTML element is defined by a start tag, some content, and an end tag:

<tagname> Content goes here... </tagname>

The HTML **element** is everything from the start tag to the end tag:

<h1>My First Heading</h1>
My first paragraph.

5.3. HTML Documents

- All HTML documents must start with a document type declaration: <!DOCTYPE html>.
- The HTML document itself begins with <html> and ends with </html>.
- The visible part of the HTML document is between <body> and </body>.
- The <!DOCTYPE> Declaration
- The <!DOCTYPE> declaration represents the document type, and helps browsers to display web pages correctly. It must only appear once, at the top of the page (before any HTML tags).
- The <!DOCTYPE> declaration is not case sensitive.
- The <!DOCTYPE> declaration for HTML5 is: <!DOCTYPE html>

5.4. Links

HTML links are defined with the <a> tag:

This is a link

5.5. Images

HTML images are defined with the <img> tag.

The source file (src), alternative text (alt), width, and height are provided as attributes:

<img src="w3schools.jpg" alt="W3Schools.com" width="104" height="142">

5.6. Nested HTML Elements

HTML elements can be nested (this means that elements can contain other elements).

All HTML documents consist of nested HTML elements.

```
<!DOCTYPE html>
<html>
<body>

<h1>My First Heading</h1>
<p>My first paragraph.</p>

</body>
</html>
```

Figure 5.6: Nested HTML Elements

5.7. Empty HTML Elements

HTML elements with no content are called empty elements.

The
 tag defines a line break, and is an empty element without a closing tag:

This is a
 paragraph with a line break.

5.8. HTML Attributes

- All HTML elements can have attributes
- Attributes provide additional information about elements
- Attributes are always specified in the start tag
- Attributes usually come in name/value pairs like: name="value"

5.9. The src Attribute

The <img> tag is used to embed an image in an HTML page. The src attribute specifies the path to the image to be displayed:

<img src="img_girl.jpg">

There are two ways to specify the URL in the src attribute:

1. Absolute URL - Links to an external image that is hosted on another website. Example: src="https://www.w3schools.com/images/img_girl.jpg".

2. Relative URL - Links to an image that is hosted within the website. Here, the URL does not include the domain name. If the URL begins without a slash, it will be relative to the current page. Example: src="img_girl.jpg". If the URL begins with a slash, it will be relative to the domain. Example: src="/images/img_girl.jpg".

5.10. The width and height Attributes

The <img> tag should also contain the width and height attributes, which specify the width and height of the image (in pixels):

<img src="img_girl.jpg" width="500" height="600">

5.11. The alt Attribute

The required alt attribute for the <img> tag specifies an alternate text for an image, if the image for some reason cannot be displayed. This can be due to a slow connection, or an error in the src attribute, or if the user uses a screen reader.

<img src="img_girl.jpg" alt="Girl with a jacket">

5.12. Text Formatting

It contains several elements for defining text with a special meaning.

This text is bold

This text is italic

This is $_{\text{subscript}}$ and $^{\text{superscript}}$

- Formatting Elements

  Formatting elements were designed to display special types of text:

  - ✓ - Bold text
  - ✓ - Important text
  - ✓ <i> - Italic text
  - ✓ - Emphasized text
  - ✓ <mark> - Marked text
  - ✓ <small> - Smaller text
  - ✓ <del> - Deleted text
  - ✓ <ins> - Inserted text
  - ✓ <sub> - Subscript text
  - ✓ <sup> - Superscript text
  - ✓ and Elements

  The HTML element defines bold text, without any extra importance.

  - ✓ **This text is bold**

  The HTML element defines text with strong importance. The content inside is typically displayed in bold.

  - ✓ **This text is important!<**/strong>

- **<small> Element**

  The HTML <small> element defines smaller text:

  <small>This is some smaller text.</small>

- **<mark> Element**

  The HTML <mark> element defines text that should be marked or highlighted:

  Do not forget to buy <mark>**milk**</mark> today.

- **<del> Element**

  The HTML <del> element defines text that has been deleted from a document.

  Browsers will usually strike a line through deleted text:

  My favorite color is ~~blue~~ red.

- **<ins> Element**

  The HTML <ins> element defines a text that has been inserted into a document. Browsers will usually underline inserted text

  My favorite color is ~~blue~~ <ins>red</ins>.

- **<sub> Element**

  The HTML <sub> element defines subscript text. Subscript text appears half a character below the normal line, and is sometimes rendered in a smaller font. Subscript text can be used for chemical formulas, like H_2O

- **<sup> Element**

  The HTML <sup> element defines superscript text. Superscript text appears half a character above the normal line, and is sometimes rendered in a smaller font. Superscript text can be used for footnotes, like WWW:

  This is ^{superscripted} text.

- **Comment Tag**: HTML comments are not displayed in the browser, but they can help document your HTML source code.

  Syntax: <!-- Write your comments here -->

5.13. HTML WEBPAGE

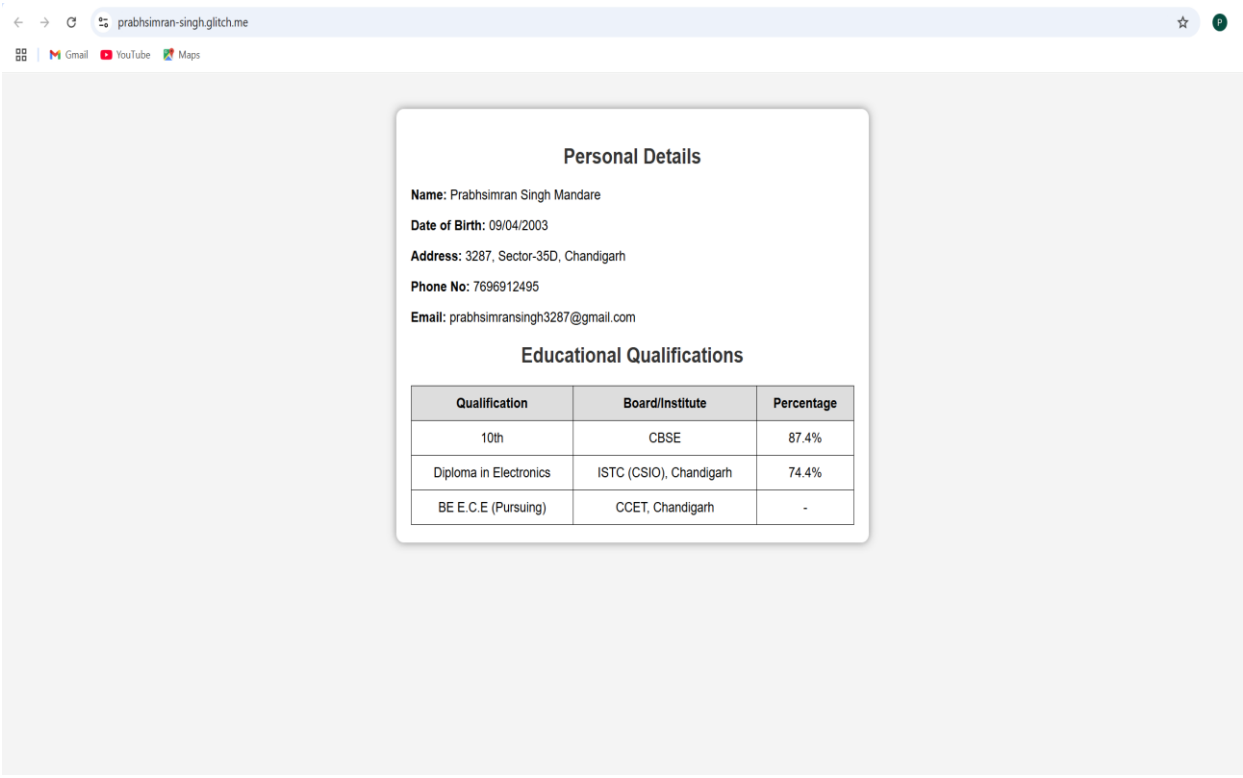

Figure 5.13: HTML Webpage Example

5.14. HTML Webpage Link

https://prabhsimran-singh.glitch.me/

HTML Webpage

Chapter 6: Pandas – Data frame, Series, EDA using python

Pandas is a powerful open-source data analysis and data manipulation library for Python. It provides high-performance, easy-to-use data structures and data analysis tools that simplify handling structured data. Pandas is built on top of NumPy and integrates well with many other Python libraries.

Pandas is a Python library used for working with data sets. It has functions for analyzing, cleaning, exploring, and manipulating data.

The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

Pandas are also able to delete rows that are not relevant, or contains wrong values, like empty or NULL values. This is called cleaning the data.

- To check the installed version of Pandas in Python, you can run the following command in your script or interactive environment:

Figure 6.1: Importing Pandas Library

This will output the version number of the Pandas library currently installed in your environment.

- In Python, the Pandas library provides a **Series** data structure, which is a one-dimensional labeled array capable of holding various data types, including integers, floats, and strings. When we use the statement s = pd.Series(), it creates an empty Series.

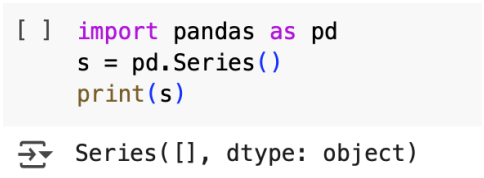

Figure 6.2: Series Data Structure

- When we execute S = pd.Series([1,2,3,4,5]), we create a **Series object** containing the values [1, 2, 3, 4, 5]. By default, Pandas assigns an index to each element, starting from 0 for the first value and increasing sequentially. When we print S, it displays the data along with the automatically assigned index. The output appears as:

```
[ ]  S = pd.Series([1,2,3,4,5])
     print(S)

     0    1
     1    2
     2    3
     3    4
     4    5
     dtype: int64
```

Figure 6.3: Creating a Series Object

Here, the left column represents the index, while the right column represents the actual data values. The dtype: int64 at the bottom indicates that the data type of the Series is **64-bit integer**. The Pandas Series is useful for handling and manipulating one-dimensional data efficiently, such as time series, sensor readings, or simple lists of values. Additionally, we can modify the index or apply operations such as filtering and aggregation on a Series for more advanced data manipulation.

6.1. Dataframes in Pandas

A **DataFrame** is a two-dimensional, table-like data structure in Pandas that consists of rows and columns, similar to a spreadsheet or a SQL table. It is one of the core data structures in Pandas and is widely used for handling structured data. A DataFrame can store data of different types (integers, floats, strings, etc.) in each column, making it a flexible tool for data analysis and manipulation.

- When we create an empty DataFrame using A = pd.DataFrame(), it initializes a DataFrame with no data, meaning it has neither rows nor columns. Printing A will display an output indicating that the DataFrame is empty, typically showing "Empty DataFrame Columns: [] Index: []".

```
[ ]  A = pd.DataFrame()
     print(A)

     Empty DataFrame
     Columns: []
     Index: []
```

Figure 6.1.1: Creating a Dataframe

This signifies that while the structure of a DataFrame is created, it contains no actual data. An empty DataFrame is often used as a placeholder, allowing data to be added dynamically later through various operations such as appending rows, reading data from files, or merging with

other DataFrames. It provides flexibility in data processing and is useful in cases where data is collected iteratively or conditionally during execution.

- In this code, a **Pandas DataFrame** is created using a nested list, where each inner list represents a row of data. The variable data contains three lists, each consisting of a name and an age. The pd.DataFrame() function is then used to convert this list into a structured tabular format. The columns=['Name', 'Age'] parameter assigns meaningful labels to the columns, where "Name" stores string values and "Age" stores integers. When print(df) is executed, the DataFrame is displayed in a table-like structure with an automatically assigned index for each row.

```
[ ]  data = [['Alex',10],['Bob',12],['Clarke',13]]
     df = pd.DataFrame(data,columns=['Name','Age'])
     print(df)
```

```
     Name  Age
0    Alex   10
1     Bob   12
2  Clarke   13
```

Figure 6.1.2.: Dataframe Created Using Nested List

This approach is useful for organizing small datasets in a structured format, making it easier to manipulate and analyze using Pandas functions.

The given code reads a dataset from a URL and stores it in a **Pandas DataFrame** using the pd.read_csv() function. The dataset is in CSV format, but instead of using a comma (,) as the default separator, it uses a semicolon (;), which is specified using the delimiter=";" parameter. This ensures that Pandas correctly interprets the structure of the data. Once loaded, the dataset is stored in the variable url, allowing for further manipulation and analysis. When executed in an interactive environment, the DataFrame displays its contents in a structured table format, where each row represents an observation of red wine, and each column corresponds to a specific characteristic such as acidity, alcohol content, or quality.

```
[ ] url=pd.read_csv("https://raw.githubusercontent.com/sarwansingh/Python/refs/heads/master/ClassExamples/data/wines/winequality-red.csv",delimiter=";")
    url
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	5
1	7.8	0.880	0.00	2.6	0.098	25.0	67.0	0.99680	3.20	0.68	9.8	5
2	7.8	0.760	0.04	2.3	0.092	15.0	54.0	0.99700	3.26	0.65	9.8	5
3	11.2	0.280	0.56	1.9	0.075	17.0	60.0	0.99800	3.16	0.58	9.8	6
4	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	5
...	...	...	...	...	...	...	...	...	...	...	...	...
1594	6.2	0.600	0.08	2.0	0.090	32.0	44.0	0.99490	3.45	0.58	10.5	5
1595	5.9	0.550	0.10	2.2	0.062	39.0	51.0	0.99512	3.52	0.76	11.2	6
1596	6.3	0.510	0.13	2.3	0.076	29.0	40.0	0.99574	3.42	0.75	11.0	6
1597	5.9	0.645	0.12	2.0	0.075	32.0	44.0	0.99547	3.57	0.71	10.2	5
1598	6.0	0.310	0.47	3.6	0.067	18.0	42.0	0.99549	3.39	0.66	11.0	6

1599 rows × 12 columns

Figure 6.1.3: Reading a CSV File

This approach is useful for importing and analyzing publicly available datasets directly from online sources without needing to download them manually.

6.2. Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA) is an important first step in data science projects. It involves looking at and visualizing data to understand its main features, find patterns, and discover how different parts of the data are connected.

Exploratory Data Analysis (EDA) is important for several reasons, especially in the context of data science and statistical modelling. Here are some of the key reasons why EDA is a critical step in the data analysis process:

- Helps to understand the dataset, showing how many features there are, the type of data in each feature, and how the data is spread out, which helps in choosing the right methods for analysis.
- EDA helps to identify hidden patterns and relationships between different data points, which help us in and model building.
- Allows to spot errors or unusual data points (outliers) that could affect your results.
- Insights that you obtain from EDA help you decide which features are most important for building models and how to prepare them to improve performance.
- By understanding the data, EDA helps us in choosing the best modelling techniques and adjusting them for better results.

Types of Exploratory Data Analysis

There are various sorts of EDA strategies based on nature of the records. Depending on the number of columns we are analyzing we can divide EDA into three types: Univariate, bivariate and multivariate.

1. Univariate Analysis

Univariate analysis focuses on studying one variable to understand its characteristics. It helps describe the data and find patterns within a single feature. Common methods include histograms to show data distribution, box plots to detect outliers and understand data, and bar charts for categorical data.

Summary statistics like mean, statistics like mean, median, mode, variance, and standard deviation help describe the central tendency and spread of the data.

2. Bivariate Analysis

Bivariate analysis focuses on exploring the relationship between two variables to find connections, correlations, and dependencies. It's an important part of exploratory data analysis that helps understand how two variables interact. Some key techniques used in bivariate analysis include scatter plots, which visualize the relationship between two continuous variables; correlation coefficient, which measures how strongly two variables are related, commonly using Pearson's correlation for linear relationships; and cross-tabulation, or contingency tables, which show the frequency distribution of two categorical variables and help understand their relationship.

Line graphs are useful for comparing two variables over time, especially in time series data, to identify trends or patterns. Covariance measures how two variables change together, though it's often supplemented by the correlation coefficient for a clearer, more standardized view of the relationship.

3. Multivariate Analysis

Multivariate analysis examines the relationships between two or more variables in the dataset. It aims to understand how variables interact with one another, which is crucial for most statistical modelling techniques. It include Techniques like pair plots, which show the relationships between multiple variables at once, helping to see how they interact. Another technique is Principal Component Analysis (PCA), which reduces the complexity of large datasets by simplifying them, while keeping the most important information.

In addition to univariate and multivariate analysis, there are specialized EDA techniques tailored for specific types of data or analysis needs:

Spatial Analysis: For geographical data, using maps and spatial plotting to understand the geographical distribution of variables.

Text Analysis: Involves techniques like word clouds, frequency distributions, and sentiment analysis to explore text data.

Time Series Analysis: This type of analysis is mainly applied to statistics sets that have a

temporal component. Time collection evaluation entails inspecting and modelling styles, traits, and seasonality inside the statistics through the years. Techniques like line plots, autocorrelation analysis, transferring averages, and ARIMA (Autoregressive Integrated Moving Average) fashions are generally utilized in time series analysis.Below are all the steps listed for analysing dataframes:

1. Understanding Data Structure

Before performing any operations, it is essential to understand the DataFrame's shape and content. The df.head()function displays the first few rows, while df.tail() shows the last few rows. The df.shape attribute returns the number of rows and columns, helping assess the dataset size.

```
[ ] url1=url.head(10)
    url1
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.9968	3.20	0.68	9.8	5
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.9970	3.26	0.65	9.8	5
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.9980	3.16	0.58	9.8	6
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.9978	3.51	0.56	9.4	5
5	7.4	0.66	0.00	1.8	0.075	13.0	40.0	0.9978	3.51	0.56	9.4	5
6	7.9	0.60	0.06	1.6	0.069	15.0	59.0	0.9964	3.30	0.46	9.4	5
7	7.3	0.65	0.00	1.2	0.065	15.0	21.0	0.9946	3.39	0.47	10.0	7
8	7.8	0.58	0.02	2.0	0.073	9.0	18.0	0.9968	3.36	0.57	9.5	7
9	7.5	0.50	0.36	6.1	0.071	17.0	102.0	0.9978	3.35	0.80	10.5	5

Figure 6.2.1: Using df.head() Command

The code url1 = url.head(10) extracts the first **10 rows** from the DataFrame url using the head() function. This function is useful for quickly previewing the dataset, allowing users to understand its structure, column names, and data distribution.

```
[ ] url2=url.tail()
    url2
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
1594	6.2	0.600	0.08	2.0	0.090	32.0	44.0	0.99490	3.45	0.58	10.5	5
1595	5.9	0.550	0.10	2.2	0.062	39.0	51.0	0.99512	3.52	0.76	11.2	6
1596	6.3	0.510	0.13	2.3	0.076	29.0	40.0	0.99574	3.42	0.75	11.0	6
1597	5.9	0.645	0.12	2.0	0.075	32.0	44.0	0.99547	3.57	0.71	10.2	5
1598	6.0	0.310	0.47	3.6	0.067	18.0	42.0	0.99549	3.39	0.66	11.0	6

Figure 6.2.2: Using df.tail() Command

The code url2 = url.tail() extracts the **last 5 rows** from the DataFrame url using the tail() function.

```
[ ] url5=url.shape
    url5
```

```
(1599, 12)
```

Figure 6.2.3: Using df.shape() Command

The code url5 = url.shape retrieves the **shape** of the DataFrame url, which provides information about the number of rows and columns in the dataset. The shape attribute returns a **tuple** in the format (number_of_rows, number_of_columns).

2. Getting Data Information

To understand the structure of the dataset, df.info() provides details about the columns, data types, and memory usage. This helps in identifying missing values and optimizing memory usage.

```
[ ] url7=url.info()
    url7

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1599 entries, 0 to 1598
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   fixed acidity         1599 non-null   float64
 1   volatile acidity      1599 non-null   float64
 2   citric acid           1599 non-null   float64
 3   residual sugar        1599 non-null   float64
 4   chlorides             1599 non-null   float64
 5   free sulfur dioxide   1599 non-null   float64
 6   total sulfur dioxide  1599 non-null   float64
 7   density               1599 non-null   float64
 8   pH                    1599 non-null   float64
 9   sulphates             1599 non-null   float64
 10  alcohol               1599 non-null   float64
 11  quality               1599 non-null   int64
dtypes: float64(11), int64(1)
memory usage: 150.0 KB
```

Figure 6.2.4: Using df.info() Command

The code url7 = url.info() is used to retrieve detailed information about the DataFrame url, including the number of rows and columns, column names, data types, non-null values, and memory usage. The .info() method is particularly useful for identifying missing data and understanding the overall structure of the dataset. However, since .info() only **prints** the information and does not return a value, assigning it to url7 will result in None.

3. Descriptive Statistics

Pandas provides statistical summaries of numerical columns using df.describe(), which returns metrics like mean, median, standard deviation, and quartiles. This helps in understanding the distribution of numerical data.

```
[ ] url8=url.describe()
    url8
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
count	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000	1599.000000
mean	8.319637	0.527821	0.270976	2.538806	0.087467	15.874922	46.467792	0.996747	3.311113	0.658149	10.422983	5.636023
std	1.741096	0.179060	0.194801	1.409928	0.047065	10.460157	32.895324	0.001887	0.154386	0.169507	1.065668	0.807569
min	4.600000	0.120000	0.000000	0.900000	0.012000	1.000000	6.000000	0.990070	2.740000	0.330000	8.400000	3.000000
25%	7.100000	0.390000	0.090000	1.900000	0.070000	7.000000	22.000000	0.995600	3.210000	0.550000	9.500000	5.000000
50%	7.900000	0.520000	0.260000	2.200000	0.079000	14.000000	38.000000	0.996750	3.310000	0.620000	10.200000	6.000000
75%	9.200000	0.640000	0.420000	2.600000	0.090000	21.000000	62.000000	0.997835	3.400000	0.730000	11.100000	6.000000
max	15.900000	1.580000	1.000000	15.500000	0.611000	72.000000	289.000000	1.003690	4.010000	2.000000	14.900000	8.000000

Figure 6.2.5: Using df.describe() Command

The code url8 = url.describe() generates summary statistics for the numerical columns in the DataFrame url. The .describe() method provides important statistical insights such as the **count** (number of non-null values), **mean**(average), **standard deviation**, **minimum and maximum values**, and **percentiles (25%, 50%, 75%)**. This function helps in understanding the distribution of numerical data, detecting outliers, and assessing data variability.

4. Checking for Missing Values

Missing data can affect analysis, and Pandas provides df.isnull().sum() to check how many missing values exist in each column. The df.dupicated().sun() function can be used to check duplicate values.

```
[ ]  ddf.duplicated().sum()

     np.int64(146)
```

Figure 6.2.6: Checking Duplicate Values

The code ddf.duplicated().sum() checks for and counts the number of **duplicate rows** in the DataFrame ddf.

```
[ ]  iris.isnull().sum()
```

	0
sepal_length	0
sepal_width	0
petal_length	0
petal_width	0
species	0

dtype: int64

Figure 6.2.7: Checking Missing Values

If all values are 0, it means the dataset has **no missing values**. Otherwise, columns with nonzero values have missing data that may require handling (e.g., filling missing values or dropping rows). This method is essential in **data preprocessing** before performing analysis or building machine learning models.

5. **Grouping and Aggregation**

The groupby() function helps in aggregating data based on specific columns.

```
[ ]  iris.groupby('species').size()
```

species	0
setosa	50
versicolor	50
virginica	50

dtype: int64

Figure 6.2.8: Aggregating Data Based on Columns

This output shows that there are **50 samples** for each species: **setosa, versicolor, and virginica**.

6. Data Correlation and Relationships

To understand relationships between numerical variables, df.corr() computes the correlation matrix, showing how strongly variables are related. This is useful in predictive modeling and statistical analysis.

```
[ ]  iris_numeric = iris.select_dtypes(include=np.number)
     iris_numeric.corr(method='pearson')
```

	sepal_length	sepal_width	petal_length	petal_width
sepal_length	1.000000	-0.117570	0.871754	0.817941
sepal_width	-0.117570	1.000000	-0.428440	-0.366126
petal_length	0.871754	-0.428440	1.000000	0.962865
petal_width	0.817941	-0.366126	0.962865	1.000000

Figure 6.2.9: Computing Correlation Matrix

The given code extracts numerical columns from the **Iris dataset** and calculates their **Pearson correlation coefficients**. The function iris.select_dtypes(include=np.number) selects only numerical columns, filtering out categorical data like species names. The resulting DataFrame, iris_numeric, is then analyzed using iris_numeric.corr(method='pearson'), which computes the **linear relationship** between numerical variables. Pearson's correlation ranges from **-1** (strong negative correlation) to **1** (strong positive correlation), with 0 indicating no correlation. The output matrix shows relationships between features such as **petal length and petal width**, which typically have a **strong positive correlation**, meaning as one increases, the other does too. In contrast, **sepal width** often has a weak or negative correlation with other features. This correlation analysis helps in **feature selection** for **machine learning models**, ensuring that redundant variables do not affect performance.

6.3. Google Colab Notebook Link for Pandas

https://colab.research.google.com/drive/12qwRBTdY0YaH9CCkCsgdOAhoIlNuht-_?usp=sharing

Pandas

Chapter 7: Data visualization using Matplotlib

Matplotlib is a powerful and widely used Python library for creating static, animated, and interactive visualizations. It provides a flexible interface for generating a variety of plots, ranging from simple line charts to complex 3D visualizations.

Matplotlib follows a procedural approach similar to MATLAB, but it also supports an object-oriented approach, offering greater flexibility and control.

Matplotlib provides a module called pyplot which offers a **MATLAB-like interface** for creating plots and charts. It simplifies the process of generating various types of visualizations by providing a collection of functions that handle common plotting tasks.

Matplotlib supports a variety of plots including line charts, bar charts, histograms, scatter plots, etc.

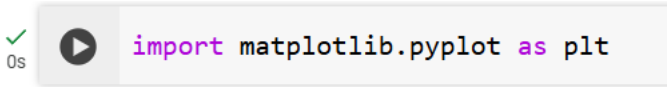

Figure 7.1: Importing matplotlib Library

import matplotlib.pyplot as plt is a common way to import the pyplot module from the Matplotlib library, providing a simple interface for creating and customizing plots in Python. The pyplot module follows a state-based approach, where functions like plt.plot(), plt.xlabel(), and plt.show() operate on the current figure, making it easy to generate visualizations without explicitly managing figure and axis objects. It supports various types of plots, including line charts, scatter plots, bar charts, and histograms.

7.1. Types of Plots

1. Line Chart

Line Chart is one of the basic plots and can be created using the plot() function. It is used to represent a relationship between two data X and Y on a different axis.

Example:

```
import matplotlib.pyplot as plt
x = [1,2,3,4,5,6,7,8]
y = [5,2,4,2,1,4,5,2]
plt.title("Line Chart")

plt.ylabel('Y-Axis')

plt.xlabel('X-Axis')
plt.plot(x,y)
```

Figure 7.1.1: Python Code for Line Chart

Output:

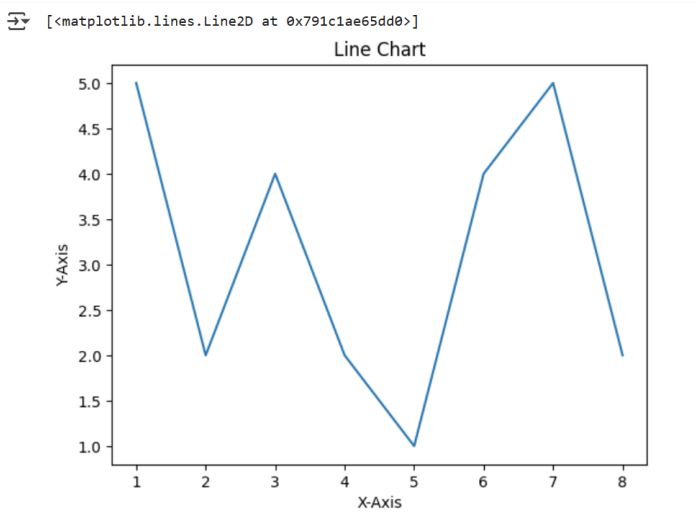

Figure 7.1.2: Line Chart

2. Bar Chart

A bar chart is a graph that represents the category of data with rectangular bars with lengths and heights that is proportional to the values which they represent. The bar plots can be plotted horizontally or vertically. A bar chart describes the comparisons between the different categories. It can be created using the bar() method.

Example:

```
[6]  categories = ['A', 'B', 'C', 'D']
     values = [3, 7, 1, 8]

     plt.bar(categories, values, color=['blue', 'green', 'red', 'purple'])
     plt.xlabel("Categories")
     plt.ylabel("Values")
     plt.title("Bar Chart Example")
     plt.show()
```

Figure 7.1.3: Python Code for Bar Chart

Output:

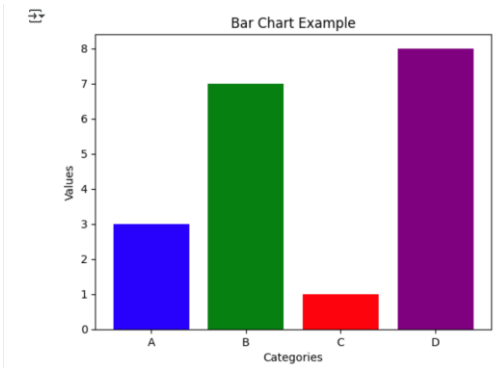

Figure 7.1.4: Bar Chart

3. Histogram

A histogram is basically used to represent data provided in a form of some groups. It is a type of bar plot where the X-axis represents the bin ranges while the Y-axis gives information about frequency. The hist() function is used to compute and create histogram of x.

Example:

```
[7]  data = np.random.randn(1000)  # Generate 1000 random numbers
     plt.hist(data, bins=30, color="orange", edgecolor="black")
     plt.xlabel("Value")
     plt.ylabel("Frequency")
     plt.title("Histogram Example")
     plt.show()
```

Figure 7.1.5: Python Code for Histogram

Output:

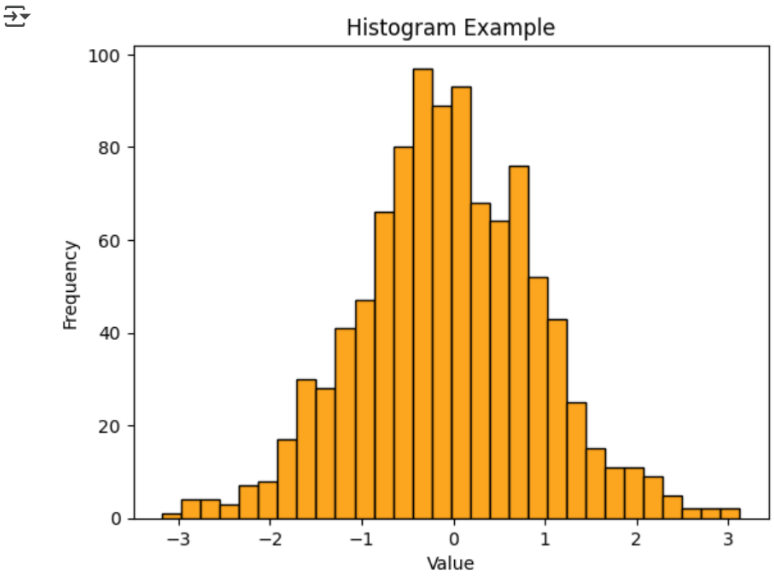

Figure 7.1.6: Histogram

4. Scatter Plot

Scatter plots are used to observe relationships between variables. The scatter() method in the matplotlib library is used to draw a scatter plot.

Example:

```
x = np.random.rand(50)
y = np.random.rand(50)

plt.scatter(x, y, color="red", marker="o")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.title("Scatter Plot Example")
plt.show()
```

Figure 7.1.7: Python Code for Scatter Plot

Output:

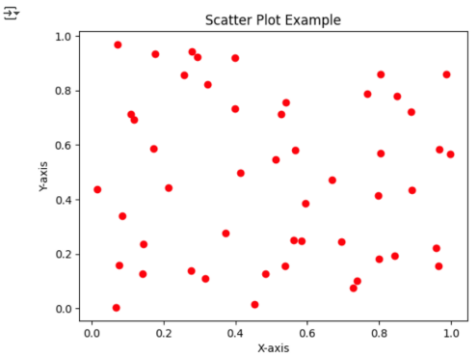

Figure 7.1.8: Scatter Plot

5. Pie Chart

Pie chart is a circular chart used to display only one series of data. The area of slices of the pie represents the percentage of the parts of the data. The slices of pie are called wedges. It can be created using the pie() method.

Syntax:

matplotlib.pyplot.pie(data, explode=None, labels=None, colors=None, autopct=None, shadow=False)

Example:

```
[8]  labels = ['Python', 'Java', 'C++', 'JavaScript']
     sizes = [40, 30, 20, 10]

     plt.pie(sizes, labels=labels, autopct="%1.1f%%", colors=['blue', 'green', 'red', 'purple'])
     plt.title("Pie Chart Example")
     plt.show()
```

Figure 7.1.9: Python Code for Pie Chart

Output:

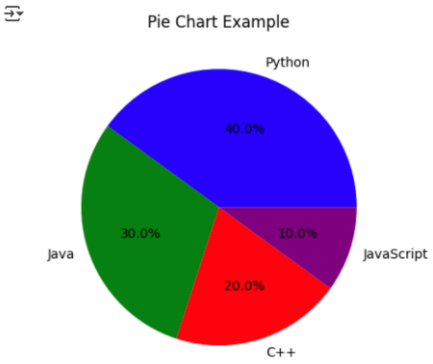

Figure 7.1.10: Pie Chart

7.2. Google Colab Notebook Link for Data Visualization

https://colab.research.google.com/drive/1aLtjgOISZbsasZKQV4z1gSnvKAsYi92x?usp=sharing

Chapter 8: Machine Learning

Machine Learning (ML) is a subset of Artificial Intelligence (AI) that enables computers to learn from data and make predictions or decisions without explicit programming. It has revolutionized various industries by automating tasks, improving decision-making, and enhancing efficiency. Machine learning is fundamentally built upon data, which serves as the foundation for training and testing models. Data consists of inputs (features) and outputs (labels). A model learns patterns during training and is tested on unseen data to evaluate its performance and generalization. In order to make predictions, there are essential steps through which data passes in order to produce a machine learning model that can make predictions.

8.1. Types of Machine Learning

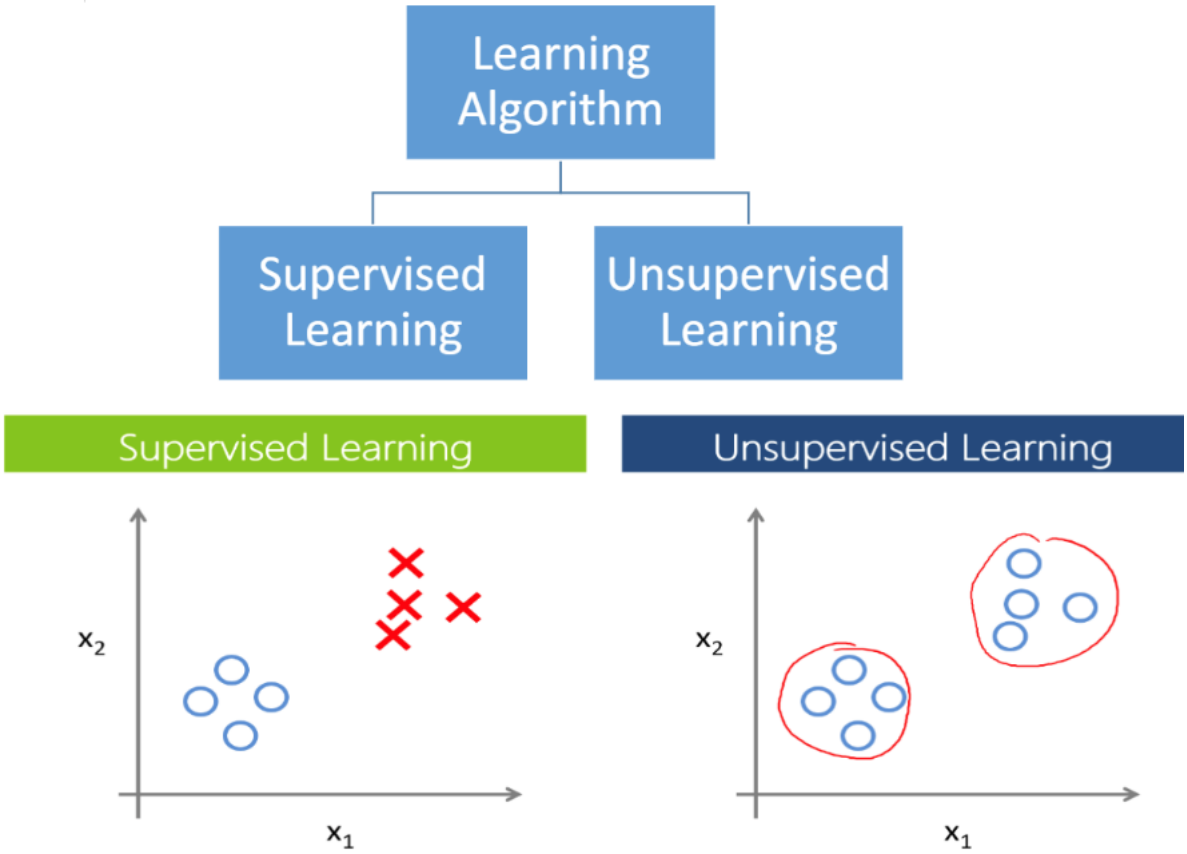

Figure 8.1.1: Types of Machine Learning

8.1.1. Supervised Learning

Supervised and unsupervised learning are two fundamental types of machine learning, each with distinct approaches and applications. Supervised learning involves training a model using labeled data, where each input is paired with a corresponding output. The model learns patterns from the labeled examples and generalizes this knowledge to make predictions on new, unseen data.

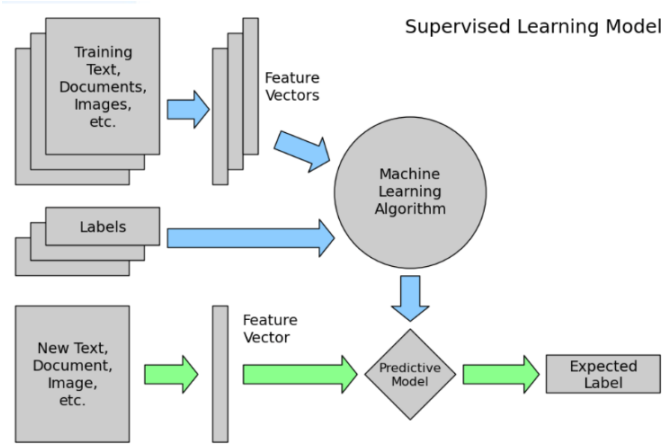

Figure 8.1.2: Supervised Learning Model

It is primarily used for classification and regression tasks. For example, in medical diagnosis, a supervised learning model can be trained on patient records, where features such as age, blood pressure, and test results are associated with a known diagnosis. Common supervised learning algorithms include Linear Regression, Logistic Regression, Decision Trees, Random Forest, Support Vector Machines (SVM), and Neural Networks. The effectiveness of supervised learning depends on the quality and quantity of labeled data, as well as the model's ability to generalize without overfitting or underfitting.

8.1.2. Unsupervised Learning

Unsupervised learning deals with unlabeled data, where the model identifies patterns and structures in the dataset without predefined outputs. Instead of predicting specific outcomes, unsupervised learning algorithms focus on discovering hidden patterns, grouping similar data points, or reducing the dimensionality of the dataset. A common example is customer segmentation in marketing, where businesses analyze purchasing behavior to categorize customers into different groups for targeted advertising. Popular unsupervised learning techniques include clustering algorithms (such as K-Means, DBSCAN, and Hierarchical Clustering) and dimensionality reduction techniques (such as Principal Component Analysis (PCA) and t-SNE).

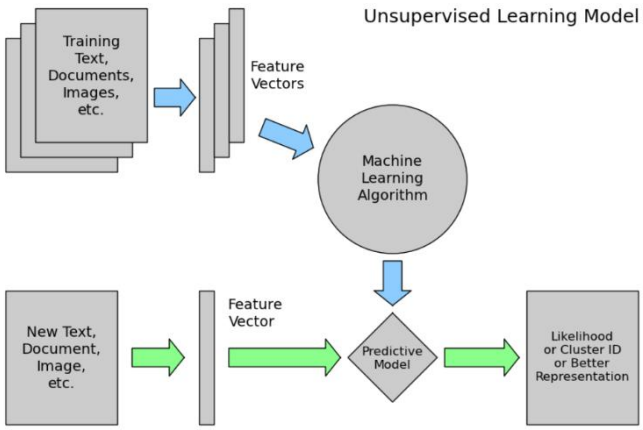

Figure 8.1.3: Unsupervised Learning Model

Since there are no explicit labels, evaluating the performance of unsupervised models is often more challenging compared to supervised learning. However, unsupervised learning is powerful for tasks like anomaly detection, recommendation systems, and feature learning, where manually labeling data is impractical or infeasible.

8.2. Category of Algorithms

The diagram below provides a clear classification of machine learning types and their associated algorithms.

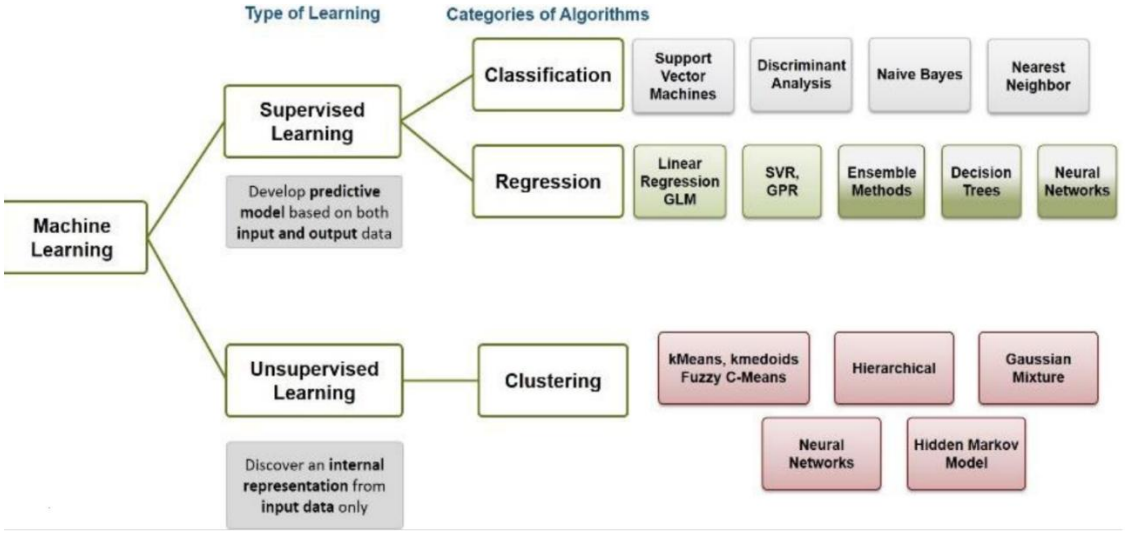

Figure 8.2: Category of Algorithms

8.2.1. Categories of Supervised Learning Algorithms

Supervised learning is further divided into Classification and Regression, based on the type of output:

(i) Classification

Classification problems involve categorizing input data into predefined classes or labels. The

output is discrete (e.g., spam or not spam, disease diagnosis).

Common Classification Algorithms:

- **Support Vector Machines (SVM):** Finds the optimal decision boundary (hyperplane) to separate classes.
- **Discriminant Analysis:** Uses statistical approaches to classify data points into different categories.
- **Naïve Bayes:** A probabilistic model based on Bayes' theorem, assuming feature independence.
- **Nearest Neighbour (KNN - K-Nearest Neighbours):** Classifies data based on the majority label of its nearest neighbours.

(ii) Regression

Regression problems involve predicting continuous values, such as predicting house prices or stock prices.

Common Regression Algorithms:

- **Linear Regression, GLM (Generalized Linear Models):** Establishes a linear relationship between input features and the target variable.
- **SVR (Support Vector Regression), GPR (Gaussian Process Regression):** Extensions of SVM for regression tasks.
- **Ensemble Methods:** Combine multiple models to improve predictive performance (e.g., Random Forest, Gradient Boosting).
- **Decision Trees:** A tree-based model that splits data based on feature conditions to make predictions.
- **Neural Networks:** A deep learning approach inspired by the human brain, effective for complex tasks.

8.2.2. Categories of Unsupervised Learning Algorithms

The primary category of unsupervised learning is Clustering, which involves grouping similar data points.

(i) Clustering

Clustering algorithms aim to partition data into distinct groups based on similarity. Common Clustering Algorithms:

- **K-Means, K-Medoids, Fuzzy C-Means:** Partition-based clustering methods that assign data points to clusters based on similarity.
- **Hierarchical Clustering:** Builds a tree-like hierarchy of clusters, useful for datasets where the number of clusters is unknown.

- **Gaussian Mixture Models (GMM):** A probabilistic approach that assumes data is generated from multiple Gaussian distributions.
- **Neural Networks:** Used in deep learning-based clustering methods like autoencoders.
- **Hidden Markov Model (HMM):** A statistical model that deals with time-series data, widely used in speech recognition and biological sequence analysis.

8.3. Regression Metrics in Machine Learning

Regression models predict continuous numerical values, and their performance is evaluated using regression metrics. These metrics measure the difference between actual values (ytrue) and predicted values (ypred).

8.3.1. Mean Absolute Error (MAE)

MAE calculates the average absolute difference between actual and predicted values. It measures how far, on average, predictions are from actual values.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_{\text{true},i} - y_{\text{pred},i}|$$

Interpretation:

- Lower MAE indicates better accuracy.
- It is simple and easy to interpret, but it does not penalize large errors more than small ones.

Example:
If the actual house price is $300,000 and the model predicts $320,000, the error is $20,000. MAE averages such absolute errors.

8.3.2. Mean Squared Error (MSE)

MSE calculates the average squared difference between actual and predicted values. Squaring the errors gives more weight to larger errors.

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

Interpretation:

- Lower MSE indicates better model performance.
- MSE penalizes large errors more heavily than MAE.
- Since it squares errors, the unit of MSE is not the same as the original variable (e.g., if predicting prices in dollars, MSE is in dollars squared).

8.3.3. Root Mean Squared Error (RMSE)

RMSE is simply the square root of MSE, bringing it back to the same unit as the original data.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_{\text{true},i} - y_{\text{pred},i})^2}$$

Interpretation:

- RMSE is more interpretable than MSE because it has the same units as the target variable.
- Like MSE, RMSE gives higher weight to large errors.

Example:

If predicting house prices, RMSE might be $15,000, meaning the model's average error is around $15,000.

8.3.4. R-Squared (R^2) - Coefficient of Determination

$\boldsymbol{R^2}$ measures how well the independent variables explain the variability of the dependent variable. It compares the model's performance to a baseline mean model.

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

Interpretation:

- $\boldsymbol{R^2}$ = 1 → Perfect model (all predictions are accurate).
- $\boldsymbol{R^2}$ = 0 → Model is no better than guessing the average.
- $\boldsymbol{R^2}$ < 0 → Model performs worse than a simple mean prediction.

Example:

If $\boldsymbol{R^2}$ = 0.85, the model explains 85% of the variance in the data.

8.3.5. Example of Regression Metrics:

```
[ ] # Define Ridge Regression model with alpha (regularization strength)
    ridge_model = Ridge(alpha=1.0)

    # Train the model
    ridge_model.fit(X_train, Y_train)

    # Make predictions
    y_pred_ridge = ridge_model.predict(X_validation)

    # Compute evaluation metrics
    mae_ridge = mean_absolute_error(Y_validation, y_pred_ridge)
    mse_ridge = mean_squared_error(Y_validation, y_pred_ridge)
    rmse_ridge = np.sqrt(mse_ridge)
    r2_ridge = r2_score(Y_validation, y_pred_ridge)
    print(f"Mean Absolute Error (MAE): {mae_ridge:}")
    print(f"Mean Squared Error (MSE): {mse_ridge:}")
    print(f"Root Mean Squared Error (RMSE): {rmse_ridge:}")
    print(f"R² Score: {r2_ridge:}")
```

Figure 8.3.5.1: Python Code for Finding Regression Metrics

Output:

```
Mean Absolute Error (MAE): 572.5490307843945
Mean Squared Error (MSE): 723101.9026721603
Root Mean Squared Error (RMSE): 850.3539866856393
R² Score: 0.8954706970803535
```

Figure 8.3.5.2: Resultant Regression Metrics

8.4. Classification Report in Machine Learning

A Classification Report is a performance evaluation metric used in classification problems. It provides a detailed analysis of a classification model's performance by calculating different metrics like precision, recall, F1-score, and support.

8.4.1. Precision

Precision is the proportion of correctly predicted positive instances out of all instances predicted as positive.

$$\text{Precision} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}}$$

- **High Precision** → Few false positives (FP).
- **Low Precision** → Many false positives (FP).

Example: In a spam email classifier, precision tells us how many emails flagged as "spam" are actually spam.

8.4.2. Recall (Sensitivity or True Positive Rate)

Recall measures how well the model identifies all actual positive cases.

$$\text{Recall} = \frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}}$$

- **High Recall** → Few false negatives (FN), meaning the model identifies most of the actual positive cases.
- **Low Recall** → Many false negatives (FN), meaning the model misses many actual positive cases.

Example: In cancer detection, recall is crucial because missing a positive case (False Negative) is dangerous.

8.4.3. F1-Score (Harmonic Mean of Precision & Recall)

F1-score balances precision and recall, especially when the dataset is imbalanced.

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

- **High F1-score** → Model has a good balance between precision and recall.
- **Low F1-score** → Model struggles in either precision or recall.

Example: If a classifier has high precision but low recall, it means it is very selective but missing many true positives. F1-score helps in such cases.

8.4.4. Support

- Support refers to the number of actual occurrences of each class in the dataset.
- It helps understand how well a model performs across different class distributions.

Example: If a classification model is tested on a dataset with 100 spam emails and 900 non-spam emails, the support for "spam" is 100, and for "non-spam" is 900.

8.5. Example of Classification Report

```
[ ]  print( "Classification Report : \n " , classification_report(Y_validation, predictedResult ))
```

Figure 8.5.1: Python Code for obtaining Classification Report

Output:

```
Classification Report :
                precision    recall  f1-score   support

      setosa       1.00      1.00      1.00         7
  versicolor       0.83      0.83      0.83        12
   virginica       0.82      0.82      0.82        11

    accuracy                           0.87        30
   macro avg       0.88      0.88      0.88        30
weighted avg       0.87      0.87      0.87        30
```

Figure 8.5.2: Classification Report

8.6. Google Colab Link for Regression Example (Diamond Price Prediction Dataset)

https://colab.research.google.com/drive/185IyhuaC2-3D8UqVXUuB1qlbHCCcuN0L?usp=sharing

Regression Example

8.7. Google Colab Link for Classification Example (Iris Dataset)

https://colab.research.google.com/drive/1gsf_vAO1xl_wi6Exg1f4S6lNsLL3Wn_R?usp=sharing

Classf Example

8.8. Google Colab Link for Unsupervised Learning

https://colab.research.google.com/drive/1HVfDRRIHdI1dYXTR2oW1p4v05VdI8CZY?usp=sharing

K-means

Chapter 9: Deep Learning

Deep learning is a subset of machine learning that focuses on artificial neural networks, particularly deep neural networks, to model complex patterns and relationships in data. It is inspired by the structure and functioning of the human brain, where multiple layers of interconnected neurons process information in a hierarchical manner.

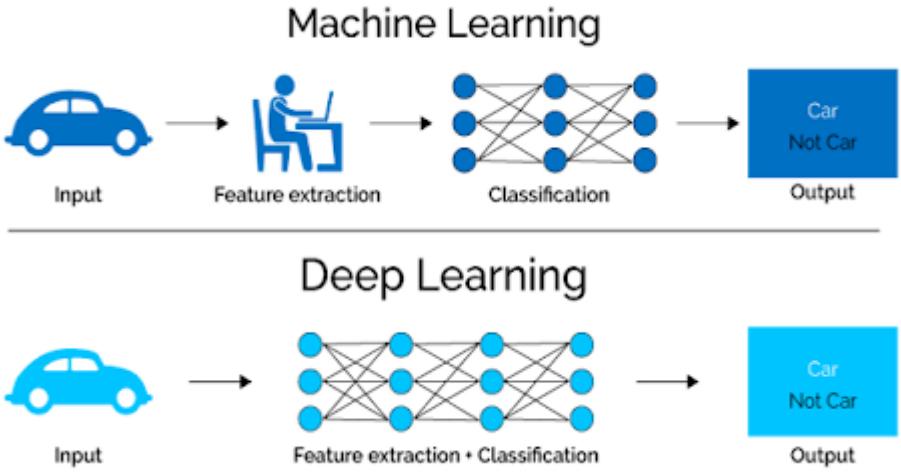

Figure 9.1: Comparison Between Machine & Deep Learning

The core idea behind deep learning is feature extraction through multiple layers of abstraction, allowing the model to automatically learn representations from raw data without the need for manual feature engineering. Deep learning models typically consist of an input layer, multiple hidden layers, and an output layer, with each layer transforming the data in a way that captures higher-level abstractions. Training these networks involves optimizing millions or even billions of parameters using backpropagation and gradient descent, where the model adjusts weights based on errors to improve predictions over time. Common architectures in deep learning include convolutional neural networks (CNNs) for image processing, recurrent neural networks (RNNs) for sequential data, and transformer models for natural language processing. The success of deep learning is largely attributed to the availability of large datasets, powerful computing resources like GPUs, and improved training algorithms. It has revolutionized various fields such as computer vision, speech recognition, autonomous systems, and medical diagnostics by enabling models to achieve human-like or even superhuman performance in complex tasks. However, deep learning also has limitations, such as high computational costs, large data requirements, and interpretability challenges, which researchers are actively working to address.

9.1. Steps Involved in Deep Learning

1. Importing Libraries

Start with the imports. Here, we are importing TensorFlow and calling it tf for convention and ease of use. The framework we will use to build a neural network as a sequence of layers is called keras and is contained within tensorflow, so you can access it using tf.keras.

We then import a library called numpy which helps to represent data as arrays easily and to optimize numerical operations.

```
[ ]  import tensorflow as tf
     import numpy as np
```

Figure 9.1.1: Importing Tensorflow & Numpy Libraries

2. Generating the Data

Before creating a neural network, we will generate the data that we will be working with. In this case, we taking 6 X's and 6 Y's. We can see that the relationship between these is y=2x-1.

y = 2x-1

x	-1	0	1	2	3	4	-2
y	**-3**	**-1**	**1**	**3**	**5**	**7**	**-5**

Figure 9.1.2: Relationship Between x & y

```
[ ]  # Declare model inputs and outputs for training
     xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0, -2.0], dtype=float)
     ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0, -5.0], dtype=float)
```

Figure 9.1.3: Generating the Data

3. Define and Compile the Neural Network

Next, we will create the simplest possible neural network. It has **1 layer with 1 neuron**. We will build this model using **Keras Sequential Class** which allows us to define the network as a sequence of layers. We can use a single dense layer to build this simple network. It is good practice to define the expected shape of the input to the model. In this case, we can see that each element in **xs** is a scalar, which we can also treat as a 1-dimensional vector. We can define this shape through the **tf.keras.Input()** object and its shape parameter as shown below. Take note that this is not a layer so even if there are two lines of code inside **Sequential** below, this is still a one-layer model.

```
[ ]  # Build a simple sequential model
     # model = tf.keras.Sequential([ tf.keras.Input(shape(1,)),tf.keras.layers.Dense(units=1)])

     model = tf.keras.Sequential([

                        #Define the input shape
                        tf.keras.Input(shape=(1,)),

                        #Add a Dense Layer
                        tf.keras.layers.Dense(units=1)
     ])
```

Figure 9.1.4: Defining the Neural Network

Now, we will compile the neural network. When we do so, we have to specify 2 functions:a loss and an optimizer. The loss function measures the guessed answers against the known correct answers and measures how well or how badly it did. It the uses the optimzer function to make another guess. Based on how the loss function went, it will try to minimize the loss. It will repeat this for the number of epochs.

```
[ ] # Compile the model
    model.compile(optimizer='sgd', loss='mean_squared_error') #Uses Stochastic Gradient Descent (SGD) as the optimization algorithm.
```

Figure 9.1.5: Compiling the Neural Network

4. Training the Neural Network

The process of training the neural network, where it learns the relationship between the x's and y's is in the model.fit() call. This is where it will go through the loop we spoke about above: making a guess, measuring how good or bad it is (loss), using the optimizer to make another guess etc. It will do it for the specified number of epochs.

```
[ ]  model.fit(xs, ys, epochs=1)

     1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 203ms/step - loss: 3.3176
     <keras.src.callbacks.history.History at 0x7b9e3a03ec10>
```

Figure 9.1.6: Training the Neural Network

5. Prediction

Prediction in deep learning refers to the process where a trained neural network model takes an input and generates an output based on learned patterns.

```
[ ]  # Make a prediction ( if x = 10, y should be 19)
     print(f"model predicted: {model.predict(np.array([10.0]), verbose=0).item():.5f}")

     model predicted: 11.61191
```

Figure 9.1.7: Making a Prediction

In the provided code snippet, the model predicts an output for an input value of 10 by applying

the trained function it has learned from the data. The predict() function takes the input, processes it through the network, and returns a numerical output. The accuracy of predictions depends on factors such as the quality of training data, model complexity, and hyperparameter tuning. Deep learning models improve their prediction capabilities by minimizing loss during training, using optimization algorithms like gradient descent.

Example:

Epoch 10

```
[ ] # Step 1. Build a Simple sequential Model

    model10 = tf.keras.Sequential([

                      #Define the input shape
                      tf.keras.Input(shape=(1,)),

                      #Add a Dense Layer
                      tf.keras.layers.Dense(units=1)
    ])

    # Step 2. Compile the model
    model10.compile(optimizer='sgd', loss='mean_squared_error') #Uses Stochastic Gradient Descent (SGD) as the optimization algorithm

    # Step 3. Train the model
    model10.fit(xs, ys, epochs=10)
```

Figure 9.1.8: An Example using 10 Epoch

Output:

```
Epoch 1/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 197ms/step - loss: 2.0436
Epoch 2/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 46ms/step - loss: 1.8031
Epoch 3/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step - loss: 1.6054
Epoch 4/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 47ms/step - loss: 1.4420
Epoch 5/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 66ms/step - loss: 1.3064
Epoch 6/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 51ms/step - loss: 1.1931
Epoch 7/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 46ms/step - loss: 1.0978
Epoch 8/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 48ms/step - loss: 1.0172
Epoch 9/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 59ms/step - loss: 0.9484
Epoch 10/10
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 62ms/step - loss: 0.8893
<keras.src.callbacks.history.History at 0x7b9e3a0c3b90>
```

```
# Make a prediction ( if x = 10, y should be 19)
print(f"model predicted: {model10.predict(np.array([10.0]), verbose=0).item():.5f}")
```

```
model predicted: 16.22886
```

Figure 9.1.9: Result of a Model with 10 Epochs

9.2. Comparison of Predicted Values and Loss on the basis of No. of Epochs used

Number of Epochs	Loss		Predicted Value (should be = 19)
	Maximum	Minimum	
1	3.3176	3.3176	11.61191
10	2.0436	0.8893	16.22886
50	29.9284	0.4190	17.94355
100	49.1916	0.1089	18.50709
150	1.6204	0.0096	18.85372
250	17.5144	7.4027e-04	18.95941
350	9.2544	2.8548e-05	18.99203
450	15.8699	1.5203e-06	18.99816
500	4.3956	2.3899e-07	18.99927

Table 9.2: Comparison of Predicted Values and Loss on the basis of Epochs

- **Observations:**

➢ Loss Decreases Over Time

➢ The maximum and minimum loss values decrease steadily as training progresses. Initially, the loss is high (e.g., 3.3176 at epoch 1), but by 500 epochs, it becomes ~2.39e-07, indicating excellent convergence. Predicted Value Approaches 19.

➢ At epoch 1, the predicted value is 11.61191 (far from 19). As epochs increase, the prediction gets closer to 19.

➢ By 500 epochs, the predicted value is 18.99927, almost perfect.

9.3. Google Colab Link for Deep Learning

https://colab.research.google.com/drive/1eR_j8AsltB17HO3eiivoSxdtWn0B-RR-?usp=sharing

Deep Learning

9.4. Neural Networks

A Neural Network is a type of machine learning model inspired by the human brain's structure and function. It is made up of layers of interconnected nodes, also known as neurons, which process data and identify patterns through a series of mathematical operations. The architecture of a neural network typically consists of three types of layers: the input layer, which receives the

raw data; one or more hidden layers, where intermediate computations take place; and the output layer, which provides the final prediction or decision. Each connection between neurons has a weight and each neuron has a bias, both of which are adjusted during training to reduce prediction errors.

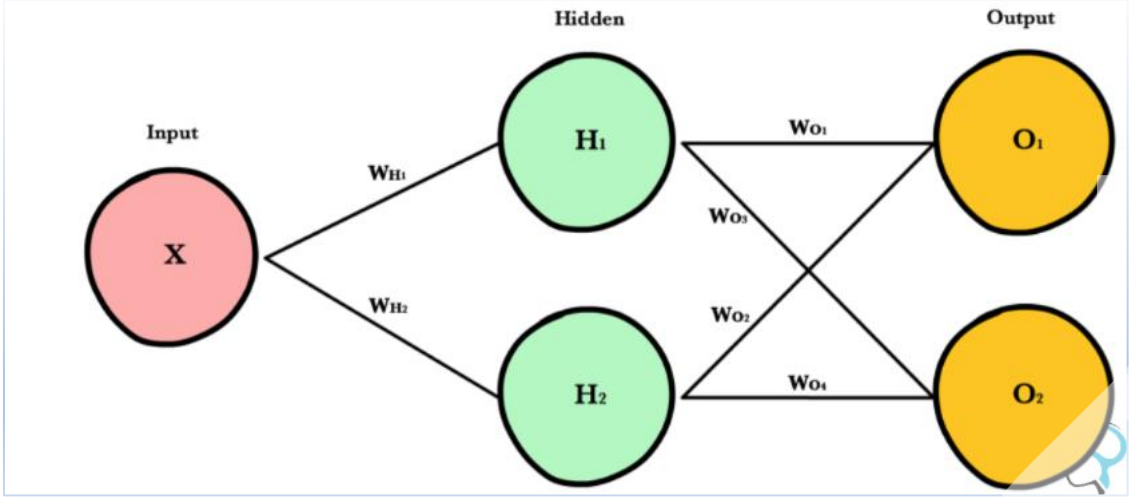

Figure 9.4: Architecture of Neural Network

Neurons in a neural network take multiple inputs, perform a weighted sum of those inputs, add a bias term, and then pass the result through an activation function to determine the output. This activation function introduces non-linearity, enabling the network to learn complex patterns and decision boundaries. Common activation functions include the sigmoid function, the hyperbolic tangent (tanh), and the Rectified Linear Unit (ReLU).

Training a neural network involves a process known as forward propagation, where data passes through the network to produce an output, and backpropagation, where the error between the predicted and actual outputs is calculated and used to update the weights and biases. The loss function quantifies the difference between actual and predicted outputs, and an optimization algorithm like gradient descent is used to minimize this loss by adjusting the model's parameters.

Neural networks come in various types suited for different tasks. Feedforward neural networks (FNNs) are the simplest, where information flows in one direction without loops. Convolutional neural networks (CNNs) are specialized for image processing, while recurrent neural networks (RNNs) handle sequential data such as text and time series. More advanced variants like autoencoders are used for unsupervised learning, and generative adversarial networks (GANs) are capable of generating new, synthetic data.

Neural networks have enabled groundbreaking applications in fields such as image and speech recognition, natural language processing, autonomous driving, medical diagnosis, and financial forecasting. Their ability to automatically learn representations and perform complex tasks with minimal feature engineering has made them a cornerstone of modern artificial intelligence.

9.4.1. Artificial Neural Networks (ANNs) Structure

Artificial Neural Networks (ANNs) are computational models inspired by the human brain's neural architecture. They consist of interconnected layers of nodes, or "neurons," which process input data to perform tasks like classification and regression. The basic structure includes:

- Input Layer: Receives the initial data.
- Hidden Layers: Intermediate layers that process inputs through weighted connections and activation functions.
- Output Layer: Produces the final result or prediction.

Each connection between neurons has an associated weight, and each neuron applies an activation function to its input to introduce non-linearity into the model.

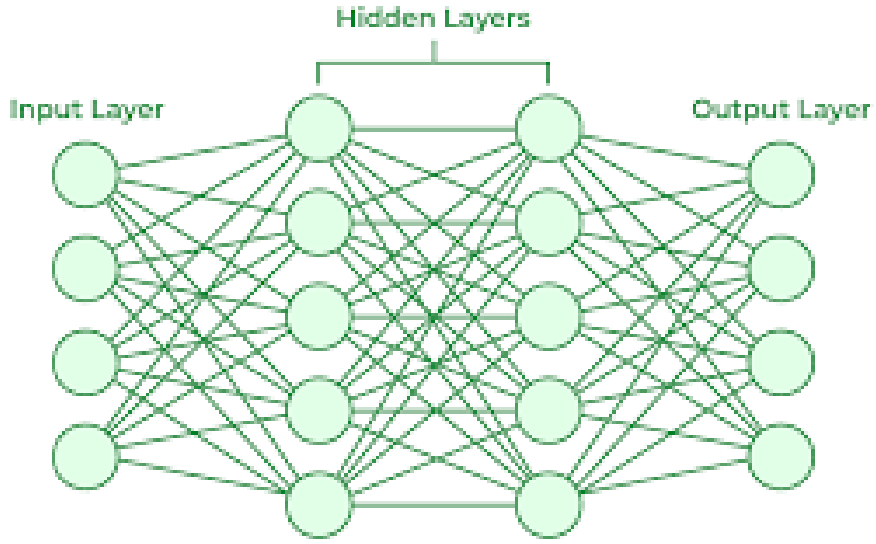

Figure 9.4.1: Architecture of Artificial Neural Network (ANN)

9.4.2. Feedforward Neural Network (FFNN)

A Feedforward Neural Network is the simplest type of ANN where connections between the nodes do not form cycles. Information moves in only one direction—from the input layer, through the hidden layers, to the output layer.

- Unidirectional Flow: Data flows forward without any feedback loops.
- Layered Structure: Comprises an input layer, one or more hidden layers, and an output layer.
- Activation Functions: Commonly used functions include ReLU, sigmoid, and tanh.

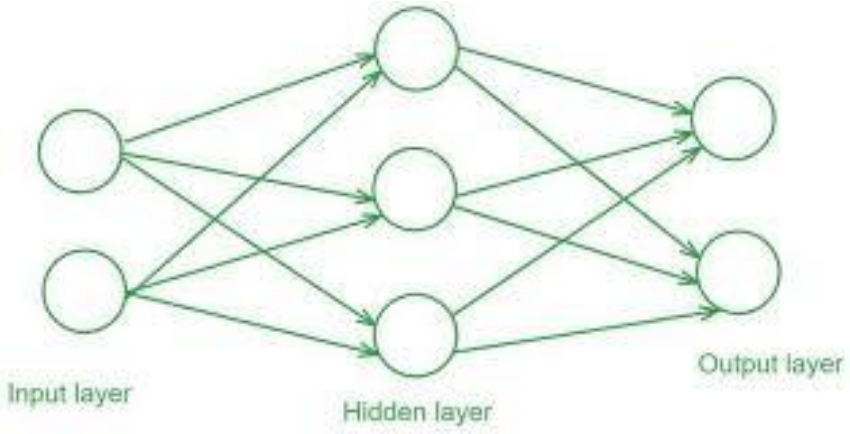

Figure 9.4.2: Architecture of Feedforward Neural Network (FFNN)

9.4.3. Backpropagation

Backpropagation is a supervised learning algorithm used for training ANNs. It calculates the gradient of the loss function with respect to each weight by the chain rule, allowing efficient computation of gradients for all weights in the network.

Process:

- Forward Pass: Compute the output of the network for a given input.
- Loss Calculation: Measure the difference between the predicted output and the actual output using a loss function (e.g., Mean Squared Error).
- Backward Pass: Propagate the error backward through the network, computing the gradient of the loss with respect to each weight.
- Weight Update: Adjust the weights using gradient descent to minimize the loss.

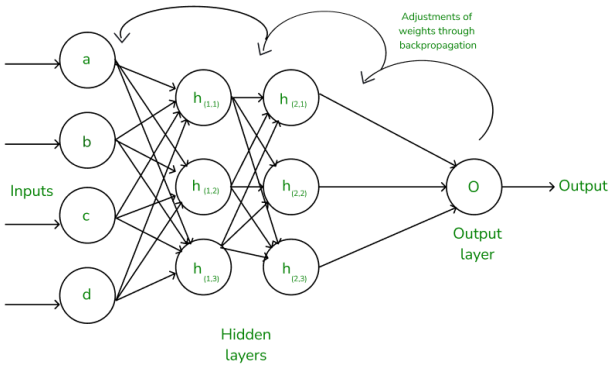

Figure 9.4.3: Architecture of Backpropagation

9.4.4. Convolutional Neural Networks (CNNs)

Convolutional Neural Networks are specialized deep neural networks designed for processing structured grid data like images. They are particularly effective for image and video recognition tasks.

Key Components:

- Convolutional Layers: Apply filters to the input to extract features such as edges, textures, and shapes.
- Pooling Layers: Reduce the spatial dimensions (width and height) of the feature maps, retaining the most important information.
- Fully Connected Layers: Perform high-level reasoning and output the final classification.

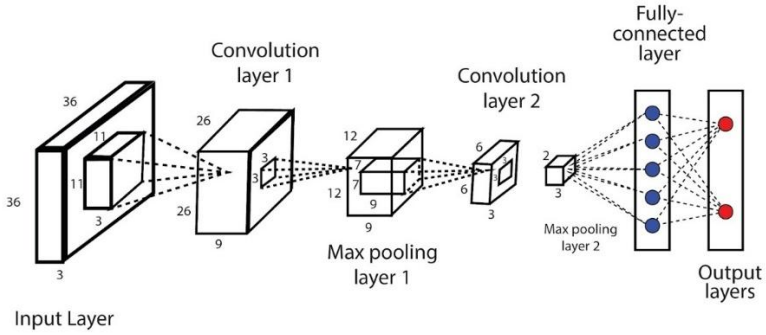

Figure 9.4.4: Architecture of Convolutional Neural Network (CNN)

The provided diagram represents the architecture of a Convolutional Neural Network (CNN), a deep learning model designed primarily for image recognition and classification tasks. At the very beginning is the input layer, which accepts a raw image of size 36×36 pixels with 3 color channels (usually representing RGB). This image is passed to the first major component, the convolutional layer 1, where 26 filters (kernels) of size 11×11×3 are applied. These filters scan across the image, detecting low-level features such as edges, textures, and corners. The result is a set of 26 feature maps, each capturing different aspects of the input image.

Following the first convolution is max pooling layer 1, which performs a dimensionality reduction operation. It selects the maximum value within small regions (often 2×2) across each feature map, reducing the spatial dimensions while retaining the most important information. This step decreases computational load and increases robustness to slight changes or distortions in the image.

Next, the pooled output is passed through convolutional layer 2, where another set of filters—smaller in size (likely 3×3×12)—further refine the extracted features, capturing more abstract or complex patterns like shapes or combinations of earlier features. The result again undergoes dimensionality reduction through max pooling layer 2, shrinking the feature map to a much smaller size (e.g., 2×2), which condenses the learned information significantly.

After the convolution and pooling operations, the output is flattened and fed into a fully connected layer. In this layer, each neuron is connected to every neuron in the previous layer, allowing the network to perform higher-level reasoning and decision-making based on the

features it has learned so far. Finally, the network reaches the output layer, where a softmax activation function is typically applied to produce probability values. Each node in this layer corresponds to a possible classification label (e.g., object categories), and the one with the highest probability is chosen as the model's final prediction.

This step-by-step architecture allows CNNs to effectively transform raw image data into meaningful decisions by gradually learning spatial hierarchies of features through convolution, pooling, and fully connected operations.

9.4.5. Neural Networks Using TensorFlow

Neural Networks using TensorFlow are built using one of the most powerful and widely adopted deep learning frameworks developed by Google. TensorFlow provides an extensive ecosystem for constructing, training, and deploying machine learning models efficiently, and it is particularly well-suited for building neural networks due to its support for automatic differentiation, GPU acceleration, and scalable deployment options. At the core of TensorFlow's neural network capabilities is its high-level API called Keras, which allows developers to define models in a concise and readable way, making rapid experimentation and prototyping easier.

To create a neural network in TensorFlow, one typically uses the Sequential API to stack layers in a linear fashion. For instance, a basic image classification model may consist of convolutional layers for feature extraction, pooling layers for dimensionality reduction, and dense layers for classification. TensorFlow also allows the use of the Functional API to define more complex, non-linear architectures such as multi-input/multi-output models or those with shared layers. Each layer in the network—whether it be a fully connected Dense layer, a convolutional Conv2D layer, or a Dropout layer for regularization—can be configured with activation functions (like ReLU or softmax), kernel initializations, and other hyperparameters.

During training, TensorFlow handles the forward pass, backpropagation, and weight updates internally. The compile() method is used to specify the optimizer (e.g., SGD, Adam), the loss function (e.g., categorical crossentropy), and metrics to monitor during training. The fit() method is then used to train the model on the dataset over a number of epochs. TensorFlow supports both CPU and GPU execution, allowing for significant speedups, especially for large models and datasets.

In addition to training, TensorFlow provides tools for model evaluation, saving/loading models, and exporting to production environments (e.g., via TensorFlow Lite or TensorFlow.js). Moreover, with built-in tools like TensorBoard, developers can visualize training performance metrics, inspect layer weights, and monitor model behavior in real time.

9.5. Learning Algorithms: Error Correction and Gradient Descent

Learning algorithms are the foundation of how neural networks improve their performance over time, and two of the most fundamental mechanisms involved are the Error Correction Rule and Gradient Descent. These algorithms govern how a neural network adjusts its internal parameters—namely, weights and biases—in response to errors made during prediction, enabling the network to learn from training data.

The Error Correction Rule is one of the earliest and simplest forms of weight adjustment used in neural networks, particularly in single-layer perceptrons. According to this rule, when a network makes an incorrect prediction, the weights are updated proportionally to the error between the actual output and the desired output. The goal is to reduce this error in future iterations. Mathematically, the weight update rule can be expressed as:

$$w := w + \eta \cdot (t - o) \cdot x$$

This formula adjusts the weight in the direction that reduces the error, effectively correcting the mistake the model made.

Gradient Descent, on the other hand, is a more general and powerful optimization algorithm used for training multi-layer neural networks. It works by minimizing a loss function, which quantifies the difference between the predicted outputs and the actual target values. The loss function is computed over all training examples, and its gradient (partial derivatives with respect to each weight) tells us the direction in which the weights should be adjusted to reduce the error.

In each iteration, the weights are updated using the following rule:

$$w := w - \eta \cdot \nabla L(w)$$

A properly chosen learning rate is crucial—if it's too small, learning is slow; if it's too large, the model may not converge.

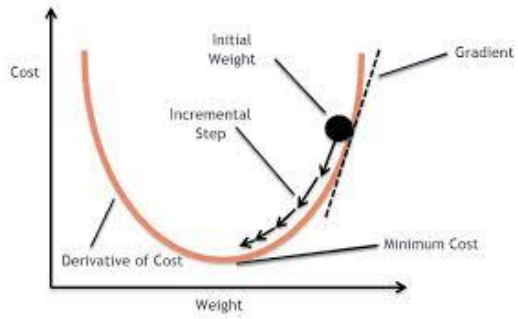

Figure 9.5: Concept of Gradient Descent

The diagram illustrates the fundamental concept of Gradient Descent, an optimization algorithm used to minimize the cost function in a neural network. The vertical axis represents the cost (or

error), while the horizontal axis shows the weight, which is a parameter of the model. The curve represents how the cost changes as the weight is adjusted. At any point on the curve, the gradient (or slope) of the cost function indicates the direction and steepness of the error. The initial weight is shown as a point on the curve, and the derivative of the cost at that point tells us which direction the weight should be adjusted to reduce the cost.

Gradient descent updates the weight by taking small incremental steps in the opposite direction of the gradient. These steps are guided by a learning rate, which controls how large each update is. As the weight is updated iteratively, the point moves downhill along the curve toward the minimum cost, which is the optimal value where the model makes the least prediction error. This process continues until convergence is achieved, or the cost function reaches its minimum point. These visual captures the essence of how neural networks learn and improve through weight adjustments driven by gradient information.

There are also advanced variants of gradient descent such as Stochastic Gradient Descent (SGD), Mini-batch Gradient Descent, and optimizers like Adam, RMSProp, and Adagrad. These optimizers use momentum, adaptive learning rates, and other techniques to improve convergence speed and stability.

Together, the Error Correction Rule and Gradient Descent form the backbone of how neural networks learn to map inputs to correct outputs. They allow the model to refine itself iteratively, reducing the gap between predictions and reality with every pass through the data.

9.5.1. Perceptron Learning Algorithm

The Perceptron Learning Algorithm is one of the earliest and simplest types of supervised learning algorithms used to train a binary classifier. Developed by Frank Rosenblatt in 1958, the perceptron is a single-layer neural network model that attempts to separate input data into two distinct classes using a linear decision boundary, or hyperplane. At its core, a perceptron consists of a set of input features, corresponding weights, a bias term, and an activation function—typically a step function that outputs either 0 or 1 depending on whether the weighted sum of the inputs exceeds a certain threshold.

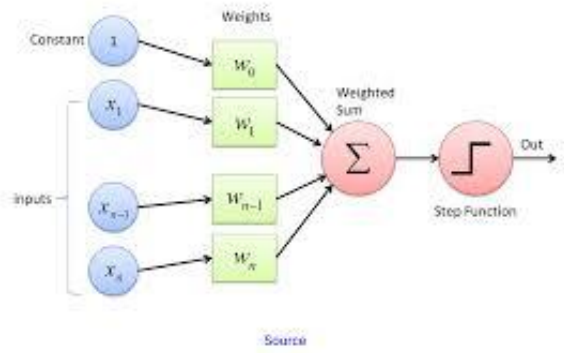

Figure 9.5.1: Perceptron Learning Algorithm

During the training phase, the perceptron processes input vectors one at a time. For each input, it computes the weighted sum by multiplying each input feature with its associated weight and adding the bias. If the output of this computation is greater than or equal to a threshold (usually zero), the perceptron outputs one class (e.g., 1); otherwise, it outputs the other class (e.g., 0). The model then compares this predicted output with the actual target label. If the prediction is correct, the weights remain unchanged. However, if the prediction is incorrect, the weights and bias are updated using the error correction rule, which is mathematically defined as:

$$w := w + \eta(t - o)x$$

$$b := b + \eta(t - o)$$

This rule ensures that the weights move in the direction that reduces the error in future predictions.

The perceptron continues to iterate over the training data, adjusting weights until it finds a decision boundary that correctly classifies all examples or until it reaches a maximum number of iterations. However, it is important to note that the perceptron learning algorithm can only converge to a solution if the training data is linearly separable—that is, if a straight line (in 2D) or hyperplane (in higher dimensions) can separate the two classes. For non-linearly separable problems, more advanced models like multi-layer perceptron's (MLPs) or deep neural networks are needed.

Despite its simplicity and limitations, the perceptron laid the groundwork for modern neural networks and continues to be an essential concept in understanding how learning rules operate within artificial neurons.

9.6. Keras and PyTorch Elements

Keras and PyTorch are two of the most popular deep learning libraries used for building and training neural networks, each with its own strengths and design philosophies. Both are open-

source and widely adopted in academia and industry, but they cater to slightly different user needs and workflows.

Keras is a high-level API initially developed as an independent project but later integrated tightly with TensorFlow as its official frontend. It offers a user-friendly and intuitive interface for building deep learning models with minimal code. Keras provides two main ways to define models: the Sequential API, which is ideal for linear stacks of layers, and the Functional API, which supports more complex architectures like multi-input/output models and shared layers. One of Keras's biggest advantages is its abstraction — it hides much of the complexity of TensorFlow's backend, making it easier for beginners and rapid prototyping. Model training is simplified using functions like model.compile(), model.fit(), and model.evaluate(), which encapsulate the training loop, loss computation, and optimization steps. Additionally, Keras integrates well with visualization tools like TensorBoard and provides built-in support for callbacks, learning rate schedulers, and model saving/loading.

On the other hand, PyTorch, developed by Facebook's AI Research lab, is known for its flexibility and transparency. It employs dynamic computation graphs, also called eager execution, which means the model's structure is defined on the fly during runtime. This makes PyTorch particularly suited for research and debugging, as it allows for dynamic changes in the model's architecture and easy inspection of intermediate outputs. In PyTorch, models are typically built by subclassing the nn.Module class and defining layers and the forward propagation logic manually. Training loops in PyTorch give users granular control over every step — from forward pass to loss calculation, backpropagation (loss.backward()), and weight updates using optimizers like SGD or Adam.

Both Keras and PyTorch support a rich set of layers, optimizers, loss functions, and tools for data loading and augmentation. While Keras emphasizes simplicity and rapid development, PyTorch is favored for its control, customization, and performance in complex, cutting-edge research. In recent years, the gap between the two has narrowed significantly, with TensorFlow introducing eager execution and PyTorch launching high-level APIs like torch.nn.Sequential and torchvision.transforms.

In summary, Keras is ideal for users who prefer quick development with minimal boilerplate, while PyTorch is preferred by those who need low-level control and dynamic behavior. Both are powerful frameworks that support end-to-end workflows for designing, training, and deploying deep learning models.

Chapter 10: Computer Vision

Computer Vision is a field of artificial intelligence (AI) that enables computers and systems to derive meaningful information from digital images, videos, and other visual inputs, and to take actions or make recommendations based on that information. It simulates the human visual system but with far greater speed, consistency, and accuracy in specific tasks. At its core, computer vision involves acquiring images through cameras or sensors, processing them using advanced algorithms, and interpreting the visual data to recognize objects, detect anomalies, track motion, or classify scenes. Techniques like image segmentation, object detection, facial recognition, and feature extraction are commonly used. Deep learning, especially convolutional neural networks (CNNs), has significantly advanced the capabilities of computer vision in recent years, enabling applications in autonomous vehicles, medical diagnostics, industrial inspection, security surveillance, and augmented reality. By bridging the gap between digital image data and intelligent decision-making, computer vision plays a crucial role in modern automation and smart systems.

10.1. AI-Roadmap to Computer Vision

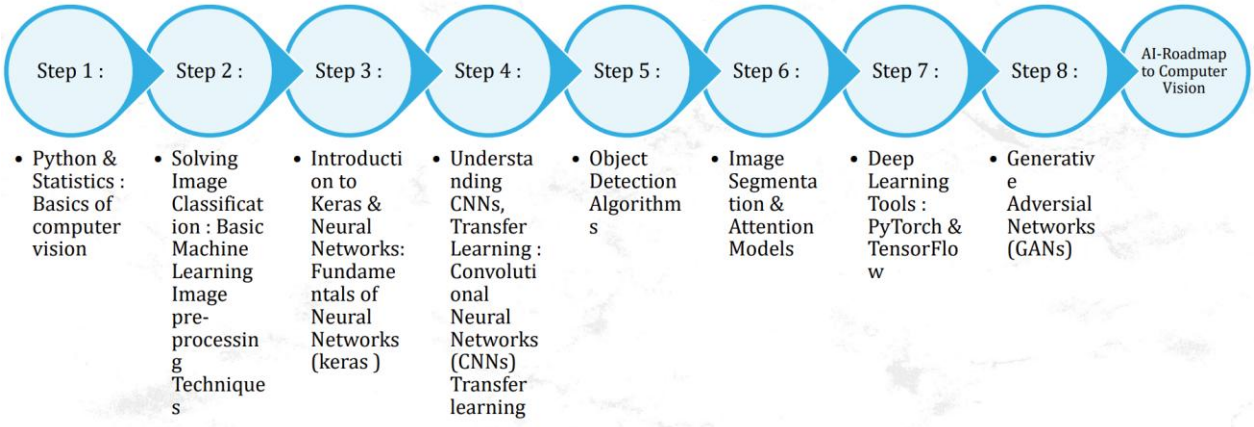

Figure 10.1: AI-Roadmap to Computer Vision

Step 1: Python & Statistics - Basics of Computer Vision

Begin by mastering Python, the primary language used in computer vision and AI. Learn basic programming concepts, libraries like NumPy, Pandas, OpenCV, and also grasp foundational statistics (mean, variance, probability) which are essential for image analysis and model evaluation.

Step 2: Solving Image Classification - Basic Machine Learning & Image Pre-processing

Explore basic ML models like Logistic Regression, SVMs, or k-NN applied to image data. Learn how to pre-process images—resizing, normalization, grayscale conversion, data

augmentation, etc., which helps improve model performance.

Step 3: Introduction to Keras & Neural Networks

Understand the fundamentals of Neural Networks using Keras, a user-friendly API built on TensorFlow. Learn how to build, train, and evaluate simple neural networks for classification tasks.

Step 4: Understanding CNNs & Transfer Learning

Deep dive into Convolutional Neural Networks (CNNs), the backbone of modern computer vision tasks.

Learn about convolution, pooling, and how CNNs extract spatial features. Study transfer learning, where pre-trained models like VGG, ResNet, or MobileNet are fine-tuned on your own datasets.

Step 5: Object Detection Algorithms

Move beyond classification and explore object detection using algorithms like YOLO (You Only Look Once), SSD (Single Shot Detector), and Faster R-CNN, which help localize and classify multiple objects in an image.

Step 6: Image Segmentation & Attention Models

Go deeper with image segmentation, which classifies each pixel (semantic segmentation like U-Net or instance segmentation like Mask R-CNN). Also, learn attention mechanisms like Self-Attention and Transformers, which allow models to focus on important image regions.

Step 7: Deep Learning Tools - PyTorch & TensorFlow

Master popular deep learning frameworks—TensorFlow and PyTorch. Learn model building, Training loops, GPU acceleration, and deployment pipelines using both.

Step 8: Generative Adversarial Networks (GANs)

Finally, learn about GANs, a powerful architecture where two networks (Generator and Discriminator) compete to create realistic images. GANs are used in tasks like image generation, super-resolution, and deepfake creation.

Following this step-by-step roadmap gives you a solid foundation in Computer Vision, equipping you with skills from image classification to advanced deep learning applications like GANs and attention models.

10.2. Importing Libraries

```
[ ]  import cv2
     import numpy as np
     from google.colab.patches import  cv2_imshow
```

Figure 10.2: Importing Libraries for Image Processing

The code snippet shown in the image is used for setting up an image processing environment in Google Colab using OpenCV and NumPy. The line import cv2 imports the OpenCV library, which is widely used for image and video processing tasks such as filtering, edge detection, object detection, and more. The line import numpy as np imports NumPy, a core library for numerical operations in Python, which is often used in image processing for handling image arrays. Finally, the line from google.colab.patches import cv2_imshow imports a special function cv2_imshow, which is required in Colab notebooks because the traditional cv2.imshow() used in local Python scripts does not work in Colab. This function allows users to display images within the notebook itself. Overall, this setup is essential for working with and visualizing images interactively in Google Colab.

10.3. Displaying Image

```
url = "Prabh.jpg"
image = cv2.imread(url)
cv2_imshow(image)
```

Figure 10.3.1: Python Code for Loading and Displaying an Image

The code is designed to load and display an image in a Google Colab environment using OpenCV. First, a filename (in this case, an image named “Prabh.jpg”) is specified. This image is then read from the working directory using OpenCV’s image reading function, which converts the image into a numerical array that represents pixel values. Instead of using the standard image display method (which doesn't work in Colab), a Colab-compatible function is used to show the image directly within the notebook. This process is commonly used to verify that an image has been correctly loaded before applying further image processing or analysis.

Figure 10.3.2: Displayed Image

The image named "Prabh.jpg" is successfully loaded and displayed. You can visually confirm that the image has been read correctly by checking its appearance. This ensures the setup for further computer vision operations is working properly.

10.4. Creating own Images using Arrays

Creating your own images using arrays involves generating pixel data manually using libraries like NumPy and then visualizing it using tools such as OpenCV or Matplotlib. In digital images, each pixel is represented by values that define its color and intensity. For example, in a color image, each pixel is defined by three values corresponding to Red, Green, and Blue (RGB), typically ranging from 0 to 255.

To create an image from scratch, you first define the size of the image using a NumPy array. For example, a blank image of size 400x400 pixels with three color channels can be created as a 3D NumPy array filled with zeros. You can then manipulate this array to draw shapes like circles, rectangles, or lines using OpenCV drawing functions, or you can manually assign values to individual pixels to create custom patterns.

This process is powerful because it gives complete control over image content, making it ideal for simulations, computer vision experiments, or understanding how images are constructed at the pixel level.

```
for i in range(0, 25):
  # randomly generate a radius size between 5 and 200, generate a random
  # color, and then pick a random point on our canvas where the circle
  # will be drawn
  radius  = np.random.randint(5, high=300)
  color   = np.random.randint(0, high=256, size = (3,)).tolist()
  pt      = np.random.randint(0, high=400, size = (2,))

  # draw our random circle
  cv2.circle(canvas, tuple(pt), radius, color, 2)

# Show our masterpiece
cv2_imshow(canvas)
```

Figure 10.4.1: Python Code for Randomly Generated Image

The given code demonstrates how to draw 25 random colored circles on that canvas using OpenCV in Python. For each iteration of the loop, the program randomly selects a radius size, a color (in RGB format), and a position on the canvas. The cv2.circle() function is then used to draw the circle at the chosen position with the generated color and size. The final output is a canvas filled with colorful, randomly placed circles. This kind of code is useful for learning how image generation and drawing operations work in computer vision and graphical programming.

Figure 10.4.2: Randomly Generated Image

The image shown is the output of the previously explained OpenCV code, where 25 randomly sized, colored, and positioned circles were drawn on a black canvas. Each circle was created using randomly generated parameters—radius, color (in RGB), and center coordinates—inside a loop.

10.5. Writing Text on Image

Writing text on an image in computer vision is a common task, especially for annotation, labeling, or watermarking purposes. In OpenCV, this is done using the cv2.putText() function, which allows you to place custom text directly onto an image at any position.

To do this, you first need an image (either loaded from a file or created using NumPy). Then, cv2.putText() takes the image as input along with several parameters:

- **Text**: The string you want to write.
- **Position**: Coordinates (x, y) on the image where the text will start.
- **Font**: OpenCV provides predefined fonts like cv2.FONT_HERSHEY_SIMPLEX.
- **Font scale**: Controls the size of the text.
- **Color**: The color of the text in BGR format (e.g., (255, 0, 0) for blue).
- **Thickness**: The thickness of the text stroke.
- **Line type** *(optional)*: Specifies the type of line used (e.g., cv2.LINE_AA for anti-aliased).

Once the text is added, you can display or save the image. This technique is useful in real-time

video applications (e.g., labeling faces or objects) or when debugging by overlaying useful information on frames.

```
url3="/content/Prabh.jpg"
image3 = cv2.imread(url3)
cv2.putText(image3, 'Prabhsimran Singh ', (30,250), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255,255, 255), 2)
cv2_imshow(image3)
```

Figure 10.5.1: Python Code for Writing Text on an Image

The given code demonstrates how to write text on an image using OpenCV in a Google Colab environment. First, an image file named "Prabh.jpg" is loaded from the specified path using cv2.imread(). Then, the cv2.putText() function is used to overlay the text "Prabhsimran Singh" onto the image. This function takes several parameters: the image to draw on, the text string, the coordinates (30, 250) which specify the position of the text, the font type (FONT_HERSHEY_SIMPLEX), the font size (0.5), the text color in BGR format (white in this case, (255, 255, 255)), and the thickness of the text (2). Finally, the modified image is displayed using cv2_imshow(), which is compatible with Colab. This approach is useful for labeling or annotating images with custom text in computer vision tasks.

Figure 10.5.2: Image with Name Getting Displayed

The name "Prabhsimran Singh" is clearly displayed at the bottom of the image in white font. This was done using the cv2.putText() function, which allows developers to write text on images at specified positions, with customizable font, size, color, and thickness. This technique is commonly used for adding labels, captions, or annotations to images in various computer vision applications such as face recognition, image tagging, and documentation.

10.6. Flipping the Image

Flipping an image means reversing it along a specific axis — either horizontally, vertically, or both —to create a mirrored version. In OpenCV, this is done using the cv2.flip() function. This function takes two arguments: the image itself and a flip code that specifies the direction of the flip.

- A flip code of 1 flips the image horizontally (left becomes right and vice versa).

- A flip code of 0 flips the image vertically (top becomes bottom).
- A flip code of -1 flips the image both horizontally and vertically, resulting in a 180-degree rotation.

Flipping is often used in data augmentation to increase the variety of images used to train computer vision models, helping improve their accuracy and generalization. It's also useful in image editing, reflection effects, and symmetry-based transformations.

```
flipped = cv2.flip(image3, 1)
print ("flipped")
cv2_imshow(flipped)
print ("original")
cv2_imshow(image3)
```

Figure 10.6.1: Python Code for Flipping an Image

The given code demonstrates how to perform and visualize a horizontal flip on an image using OpenCV in Google Colab. The cv2.flip() function is used with a flip code of 1, which mirrors the image across its vertical axis, effectively swapping the left and right sides. The flipped version is stored in the variable flipped and then displayed using cv2_imshow(). To help distinguish the two outputs, the code prints "flipped" before showing the mirrored image, and then "original" before displaying the unmodified image. This side-by-side comparison is helpful for understanding how flipping transformations alter the orientation of an image in computer vision tasks.

Figure 10.6.2.: Flipped and Original Image Getting Displayed

The image above shows a side-by-side comparison of a horizontally flipped image and its original version. The top image, labeled "flipped", is a mirrored version of the original image shown below it. This transformation was achieved using the OpenCV function cv2.flip() with a flip code of 1, which reflects the image across its vertical axis — meaning everything on the left appears on the right and vice versa.

10.7. Blurring an Image

Blurring an image using average blurring involves smoothing the image by replacing each pixel's value with the average value of its neighbouring pixels within a specified kernel (window) size. This process reduces image details and softens sharp transitions, helping to suppress noise and minor variations.

In OpenCV, average blurring is performed using the function cv2.blur(). You specify the kernel size as a tuple, such as (5, 5), which means that a 5×5 region around each pixel will be considered for averaging. The larger the kernel size, the stronger the blurring effect. This technique results in a uniform blur across the image and is often used when a general smoothing of the entire image is needed.

While average blurring is simple and efficient, it does not preserve edges well. As a result, it may lead to loss of important details in areas with sharp intensity changes. Nonetheless, it is widely used in applications where edge preservation is not critical, such as background softening, noise reduction, and preparing images for thresholding or segmentation.

```
[ ]  cv2_imshow(image3)
     blurred = cv2.blur(image3, (2, 2))
     cv2_imshow(blurred)
     blurred1 = cv2.blur(image3, (5, 5))
     cv2_imshow(blurred1)
     blurred2 = cv2.blur(image3, (15, 15))
     cv2_imshow(blurred2)
```

Figure 10.7.1.: Python code for Blurring an Image

The code shown demonstrates average blurring of an image using different kernel sizes with the OpenCV function cv2.blur(). First, the original image (image3) is displayed. Then, three blurred versions of the image are created using kernel sizes (2, 2), (5, 5), and (15, 15) respectively. Each of these kernel sizes defines the neighborhood over which the pixel values are averaged. A small kernel size like (2, 2) results in minimal blurring, while larger sizes like (5, 5) and especially (15, 15) produce increasingly stronger smoothing effects, causing more noticeable loss of detail. By displaying all versions side-by-side, the effect of kernel size on blurring intensity becomes visually apparent, showcasing how average blurring progressively softens the image as the kernel size increases.

Figure 10.7.2: Effects of Average Blurring

The image showcases the effect of average blurring applied to a photo using different kernel sizes. From top to bottom:

1. First image – This is the original image, with sharp and clear details.
2. Second image – Blurred using a small kernel (2, 2), which results in a slight softening of features while retaining most of the clarity.
3. Third image – Blurred using a medium kernel (5, 5), causing a noticeable smoothing of facial details and edges.
4. Fourth image – Blurred with a large kernel (15, 15), producing a heavily blurred version where fine details are lost, and the entire image appears soft and unfocused.

This visual progression demonstrates how increasing the kernel size intensifies the blurring effect, making average blurring a useful tool for tasks like noise reduction, artistic effects, or background abstraction in computer vision and image processing.

10.8. Face Detection

Face detection is a computer vision technique used to identify and locate human faces within digital images or videos. It serves as a foundational step for many advanced applications such as face recognition, emotion analysis, attendance systems, and security monitoring. The goal of face detection is not to recognize who the person is, but simply to determine the presence and position

of a face.

Modern face detection algorithms analyze the visual patterns and features that are commonly found in human faces, such as the eyes, nose, and mouth. Traditional methods like Haar cascades used in OpenCV rely on handcrafted features and classifiers, while more recent approaches use deep learning models like MTCNN, SSD, or YOLO, which offer higher accuracy and robustness, especially in varied lighting conditions and face orientations. Once a face is detected, a bounding box is typically drawn around it, making it ready for further processing or analysis.

Face detection is a step-by-step process used to find human faces in an image. Here's how it works in a simple and detailed way:

1. **Convert to Grayscale**: The first step is to convert the original-colored image into a grayscale image. This is because face detection does not need color information, and working with a single channel (grayscale) makes processing faster and simpler.
2. **Create Object for Classifier**: After converting the image, we need to load a pre-trained face detection model. OpenCV provides Haar cascade classifiers for this purpose. We create an object and link it to the classifier file (for example, haarcascade_frontalface_default.xml). This classifier is trained to detect patterns and features of faces.
3. **Detect Faces**: Now we apply the classifier to the grayscale image using the detectMultiScale() function. This function scans the image and detects areas that match the face pattern. It returns a list of coordinates for each detected face in the form of rectangles (x, y, width, height).
4. **Draw Rectangles Around Faces**: Once we have the list of face coordinates, we use a for loop to go through each face and draw a rectangle on the original image. This makes it easy to see where the faces are located in the image.

In the end, you get the same image with rectangles drawn around all the detected faces. This method is widely used in applications like security systems, automatic photo tagging, and facial recognition systems.

```python
def myFaceDetect( url1 , scaleFactor=1.3):
  #read the image
  img1 = cv2.imread(url1)
  # cv2_imshow(img1)
  # convert the image into gray
  img1  = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
  # cv2_imshow(imgh1)
  # create object link to selected classifier
  cc = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')
  # go for face detection
  faces = cc.detectMultiScale( img1,scaleFactor,5)
  # run for loop in faces list to draw rectangle on all the detected faces
  img1 =cv2.imread(url1)
  for ( x,y,w,h) in faces :
    img1 = cv2.rectangle (img1, (x,y), (x+w, y+h) , (255,0,0), 2)
  cv2_imshow(img1)
```

Figure 10.8.1.: Python Code for Face Detection

The provided code defines a Python function myFaceDetect() that performs face detection on an input image using OpenCV. The process begins by reading the image from the given file path (url1). Then, the image is converted from color to grayscale, which simplifies the data and speeds up the detection process since face detection algorithms don't require color information. After that, a CascadeClassifier object is created using a pre-trained Haar cascade XML file (haarcascade_frontalface_default.xml), which contains data on facial features.

The actual face detection is performed using the detectMultiScale() method, which scans the grayscale image at multiple scales to detect faces. This method returns a list of rectangles, each representing a detected face. To visualize the detection, the original image is reloaded, and a for loop iterates through all detected faces, drawing a red rectangle (with color (255, 0, 0) and thickness 2) around each face using the cv2.rectangle() function. Finally, the image with the highlighted faces is displayed using cv2_imshow(). This function demonstrates a complete face detection pipeline, useful for various computer vision applications.

Input:

Figure 10.8.2.: Input Image for Testing Face Detection

Output:

Figure 10.8.3.: Output Image of Face Detection

The image shows the result of a face detection process applied to a group photograph using OpenCV and a Haar cascade classifier. Each detected face is marked with a blue rectangle, indicating that the face detection algorithm has successfully identified multiple faces in the image, despite the diversity in orientation, lighting, and facial expressions.

This output demonstrates the power and effectiveness of automated face detection in real-world conditions.

The algorithm scans the image, detects facial features based on pre-trained data, and highlights each face region. Such techniques are commonly used in applications like group photo analysis, attendance systems, surveillance, and people counting in crowds. The image clearly shows that the method works even in outdoor environments with complex backgrounds, although a few false detections (e.g., on tree leaves) may also occur, highlighting the need for further refinement or more robust models like deep learning-based detectors for higher accuracy.

10.9. Face Recognition

Face Recognition using LBPH (Local Binary Patterns Histograms) is a widely used technique in Computer vision for identifying individuals based on their facial features. It is particularly Valued for its simplicity, efficiency, and robustness under varying lighting conditions. Unlike face detection, which only finds where faces are located in an image, face recognition aims to determine the identity of each detected face by comparing it to a known database.

The LBPH algorithm works by analyzing the texture of a grayscale face image. It examines each

pixel and compares it with its neighboring pixels. If the neighbor has a greater or equal intensity, it assigns a value of 1; otherwise, it assigns 0. These binary results form an 8-bit binary number for every central pixel, which is then converted into a decimal value. These values form what is known as local binary patterns (LBP), effectively capturing the micro-features or textures in the face.

The image is then divided into multiple grids, and for each grid, a histogram of the LBP values is created.

These histograms are concatenated to form a long vector, which becomes the feature descriptor for that particular face. During the recognition phase, this feature vector is compared with the vectors of known faces in the dataset using a simple distance measurement, such as the Euclidean distance. The closest match is taken as the identity of the person.

LBPH is relatively fast, does not require high computational power, and works well even with smaller datasets,

making it ideal for real-time face recognition applications like security systems, automated attendance, and access control. Its ability to focus on local features also makes it more tolerant to changes in lighting and facial expressions compared to some more complex algorithms.

10.9.1. Face Extraction from Multiple Images using Haar Cascade

```
face_detection = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')
facelist  =  []
imgn = cv2.imread(urln)
# detect face of Narinder Singh in image name imgn
gray = cv2.cvtColor(imgn, cv2.COLOR_BGR2GRAY)
facen = face_detection.detectMultiScale (gray, 1.3, 5)
for (x,y,w,h) in facen:
      facelist.append(gray[y:y+h,x:x+w])

imgmms = cv2.imread(urlmms)
# detect face of Manmohan Singh in image name imgmms
gray = cv2.cvtColor(imgmms, cv2.COLOR_BGR2GRAY)
facemms = face_detection.detectMultiScale (gray, 1.3, 5)
for (x,y,w,h) in facemms:
      facelist.append(gray[y:y+h,x:x+w])

imgms = cv2.imread(urlms)
# detect face of Milkha Singh in image name imgms
gray = cv2.cvtColor(imgms, cv2.COLOR_BGR2GRAY)
facems = face_detection.detectMultiScale (gray, 1.3, 5)
for (x,y,w,h) in facems:
      facelist.append(gray[y:y+h,x:x+w])

imgps = cv2.imread(urlps)
# detect face of Prabhsimran Singh in image name imgps
gray = cv2.cvtColor(imgps, cv2.COLOR_BGR2GRAY)
faceps = face_detection.detectMultiScale (gray, 1.3, 5)
for (x,y,w,h) in faceps:
      facelist.append(gray[y:y+h,x:x+w])

print ( facen, facemms, facems,  faceps, len(facelist))
```

Figure 10.9.1.1.: Python Code for Face Recognition

This code snippet demonstrates how to detect and extract faces from multiple images using OpenCV's Haar cascade face detector. The process begins by initializing the Haar cascade classifier with the pre trained XML file haarcascade_frontalface_default.xml. An empty list named facelist is created to store all the cropped face regions.

For each image (Narinder Singh, Manmohan Singh, Milkha Singh, and Prabhsimran Singh), the following steps are performed:

1. The image is read using cv2.imread().
2. It is converted into grayscale using cv2.cvtColor() because the Haar classifier works on grayscale images.
3. The detectMultiScale() function is called to detect all faces in the image, returning the coordinates (x, y, width, height) of the bounding boxes.
4. A loop iterates over the detected face coordinates, and the face region is extracted from the grayscale image using array slicing.
5. Each extracted face is appended to the facelist.

Finally, the detected face coordinates for each person are printed along with the total number of face crops collected in the facelist. This is a preparatory step typically used in face recognition systems, where cropped faces are later used for training or identification.

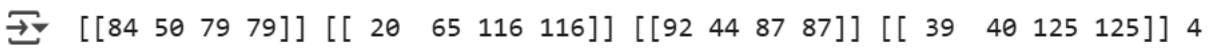

Figure 10.9.1.2: Face Coordinates for Each Person

The output shown in the image represents the bounding box coordinates of detected faces in four different images, along with the total count of extracted faces.

Each list inside the output contains four numbers:

[x, y, w, h]

where:

- x and y are the top-left coordinates of the detected face,
- w is the width of the bounding box,
- h is the height of the bounding box.

Here's a breakdown:

- [[84, 50, 79, 79]]: A face detected at (84, 50) with width 79 and height 79.
- [[20, 65, 116, 116]]: A face detected at (20, 65) with width and height of 116.
- [[92, 44, 87, 87]]: A face detected at (92, 44) with width and height of 87.
- [[39, 40, 125, 125]]: A face detected at (39, 40) with width and height of 125.

The final value 4 indicates that a total of four face regions were successfully detected and

appended to the facelist. This confirms that the face detection and cropping process worked for each of the four input images.

```
[ ]  faceidlist = [1,2,3,4]
     faceidNames = [ 'Unknown', 'Narinder Singh Kapany', 'Manmohan Singh', 'Milkha Singh', 'Prabhsimran Singh']
     print ( faceidlist)

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

Figure 10.9.1.3.: Corresponding IDs of Each Person

The output displays the coordinates of four faces detected from different images using the Haar cascade classifier. Each set of values represents the position and size of a detected face in the format [x, y, width, height], where x and y indicate the top-left corner of the face, and width and height define the size of the bounding box. The number 4 at the end confirms that a total of four faces were successfully detected and extracted. This output verifies that the face detection process is functioning correctly and that the cropped face regions are ready for further processing, such as training a face recognition model.

10.9.2. Creating a Model

```
def testLBPHmodel (url):
  img = cv2.imread(url)
  gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
  faces = face_detection.detectMultiScale(     gray,    scaleFactor = 1.2,     minNeighbors = 5       )
  font = cv2.FONT_HERSHEY_SIMPLEX

  for(x,y,w,h) in faces:

        cv2.rectangle(img, (x,y), (x+w,y+h), (0,255,0), 2)
        ID, confidence = recognizer.predict(gray[y:y+h,x:x+w])

        # Check if confidence is less them 100 ==> "0" is perfect match
        if (confidence < 100):
            id = ID
            confidence = "  {0}%".format(round(100 - confidence))
        else:
            id = "unknown"
            confidence = "  {0}%".format(round(100 - confidence))

        cv2.putText(img, str(id), (x+5,y-5), font, 1.5, (255,0,0), 2)
        cv2.putText(img, str(confidence), (x+5,y+h-5), font, 1, (255,0,0), 2)
  cv2_imshow(img)
```

Figure 10.9.2.1: Python Code for Creating a Model for Face Recognition

This code defines a function testLBPHmodel(url) that performs face recognition on an input image using a pre-trained LBPH (Local Binary Patterns Histograms) recognizer. Here's a breakdown of what the code does:

1. **Image Loading & Preprocessing**:

   The image is read from the specified path and converted to grayscale, as both face detection and recognition operate on single-channel images.

2. **Face Detection:**

   The detectMultiScale() method is used to detect faces in the grayscale image. It returns the coordinates of all detected faces.

3. **Face Recognition:**

   For each detected face, a rectangle is drawn around it using cv2.rectangle(). Then, the cropped face is passed to the LBPH face recognizer's predict() method, which returns:

   - ID: the label predicted for the face,
   - confidence: how certain the model is about the match (lower confidence means a better match).

4. **Confidence Evaluation:**

   If the confidence value is less than 100, it is considered a match. The confidence is then converted to a match percentage by subtracting it from 100. If the confidence is higher (indicating a weak match), the face is labelled as "unknown".

5. **Displaying Results:**

   The predicted ID and confidence percentage are drawn on the image using cv2.putText(), and the final image is displayed with cv2_imshow().

10.9.3. Testing

Input:

Figure 10.9.3.1.: Input Image for Testing Face Recognition Model

Output:

Figure 10.9.3.2.: Output Image of Face Recognition Model

This output demonstrates how the LBPH model not only detects the face but also tries to match it to a known identity from the training dataset. The labeling and annotation help visually confirm both the presence of a face and the result of the recognition attempt.

Input:

Figure 10.9.3.3.: Input Image for Testing Face Recognition Model

Output:

Figure 10.9.3.4.: Output Image of Face Recognition Model

The image shows the output of a face recognition system using the LBPH (Local Binary Patterns Histograms) model applied to a group photograph. Each detected face is enclosed in a colored bounding box, with a predicted label (either an ID or "unknown") and a confidence percentage displayed above the face.

Here's what the output indicates:

- The label "unknown" appears on most faces, suggesting that these individuals were not included in the training dataset of the face recognizer.
- The confidence percentage below the ID shows how confident the model is in its prediction. A lower number means a more confident match. For instance, "4 20%" implies that the face was matched with ID 4 with 80% confidence.
- The use of cv2.putText() has clearly labelled each detected face, while cv2.rectangle() has highlighted them, visually marking both identity and recognition certainty.

Overall, this output effectively demonstrates a real-time multi-face recognition system, showing both the strengths of LBPH in recognizing known faces and its limitations when encountering

unknown individuals or poor-quality face data. It's commonly used in attendance systems, surveillance, and identity verification in group scenarios.

10.10. Google Colab Link for OpenCV

https://colab.research.google.com/drive/17QfaKcxA6DTTM4QEbFjCcoWul1vQbggy?usp=sharing

10.11. Google Colab Link for Face Detection

https://colab.research.google.com/drive/1jvp-aJnDK7SAXSWNU8W8IloKNlhy7?usp=sharing

10.12. Google Colab Link for Face Recognition

https://colab.research.google.com/drive/1JUBmjjs36y0YOuICqPGgL9BqCDjrpTq?usp=sharing

Chapter 11: Introduction to Hugging Face

Hugging Face is a groundbreaking platform in the field of artificial intelligence that has revolutionized how machine learning models are built, shared, and deployed. Originally starting as a chatbot company, it quickly pivoted into becoming a central hub for Natural Language Processing (NLP) research and development. Today, Hugging Face is globally recognized for hosting one of the largest repositories of pre-trained machine learning models, covering areas such as text, image, audio, and even tabular data. The core offering is the Transformers library, a massively popular Python package that provides easy access to hundreds of powerful models like BERT, GPT, RoBERTa, T5, ViT, and more. These models can be used for a wide range of tasks such as text classification, question answering, image classification, object detection, speech recognition, and beyond.

Figure 11.1.: Logo of Hugging Face

What makes Hugging Face truly unique is its open-source spirit combined with community-driven contributions. Researchers and developers from around the world upload their models to the Hugging Face Model Hub, making it a public catalog of state-of-the-art models. Each model comes with documentation, code snippets, metrics, and configuration files, allowing anyone—even those with limited ML experience—to fine-tune and deploy them. Alongside the Model Hub, Hugging Face also provides a Datasets library, which includes thousands of datasets in multiple formats, ready to be used in training or benchmarking machine learning systems.

Another powerful feature of Hugging Face is Spaces, a free and hosted environment where developers can deploy interactive machine learning demos using tools like Gradio and Streamlit. You can write a small app and deploy it on a Hugging Face-hosted server without any need for backend or DevOps knowledge. Additionally, the company offers the Inference API, which allows real-time access to thousands of hosted models through RESTful endpoints. This is extremely useful for businesses and developers who want to use advanced models without the burden of hosting them locally. Hugging Face also supports training large-scale models using tools like Accelerate, PEFT, and Trainer, which simplify distributed training, parameter-efficient fine-tuning, and model optimization. Overall, Hugging Face represents a democratization of AI—bringing cutting-edge machine learning to everyone through transparency, community, and simplicity.

11.1. Flask

Flask is a micro web framework written in Python, widely used for building lightweight, flexible web applications and RESTful APIs. Unlike large monolithic frameworks that come with every tool pre-packaged, Flask follows a minimalist design philosophy, giving developers full control over the structure and behavior of their application. It is often described as a “micro-framework” because it does not include built-in tools like form validation, database abstraction layers, or authentication systems by default. However, this is also its strength—Flask lets developers pick and choose exactly what they want to include, making it highly modular, extensible, and customizable.

Flask is especially popular among backend developers and data scientists who want to deploy machine learning models as web services. With Flask, you can create routes (URLs) that trigger Python functions, allowing users to send input (like text or image data) and get predictions from a model in return. For example, you might write a /predict route that loads a trained model, processes the input data, and returns a classification result. Flask is designed around the concept of WSGI (Web Server Gateway Interface), which makes it scalable and production-ready when paired with tools like Gunicorn or uWSGI.

Another powerful feature of Flask is its support for Jinja2 templating, which allows developers to embed Python code inside HTML to create dynamic web pages. It also offers session management, secure cookies, and support for common HTTP methods (GET, POST, PUT, DELETE), enabling developers to build robust full-stack applications if needed. While Flask does require more boilerplate code and setup compared to some high-level frameworks, it provides total freedom in terms of design and architecture. This makes it ideal for developers who need precise control over how their application behaves, particularly in enterprise or production environments.

11.1.1. Hello App using Flask

A Hello App using Flask is one of the simplest examples of creating a web application using the Flask framework, which is a lightweight, Python-based micro web framework used for building web servers and APIs. Unlike frameworks such as Django that come with many built-in features, Flask gives developers complete control by offering only the essentials: routing, request handling, and templating. This makes it ideal for learning the fundamentals of web development and backend logic, especially for small applications and prototypes like a Hello App.

In a Flask Hello App, the primary goal is to create a webpage where the user can enter their name into a form, submit it, and receive a personalized greeting like “Hello, Prabhsimran!” displayed on the screen. The app consists of two main components: the Python backend and the HTML

front-end template. When the user visits the app, Flask serves an HTML form (created using Jinja2 templating), where they input their name. Once submitted, the form sends the data to the server using an HTTP POST request. Flask captures this data in the backend, processes it through a Python function, and sends the result back to be rendered in the HTML response.

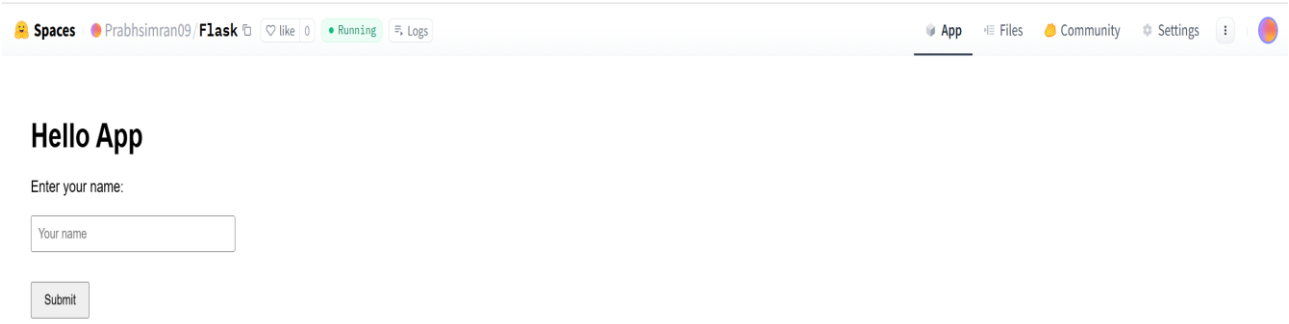

Figure 11.1.1.: Hello App using Flask Deployed on Hugging Face

The "Hello App" shown in the image is a basic web application developed using the Flask framework and deployed on Hugging Face Spaces. It features a minimalistic interface where users can enter their name into a text field and submit it via a form. This form submission sends data to the Flask backend, which processes the input and dynamically generates a personalized greeting in return. Flask, being a lightweight Python web framework, handles both routing and rendering.

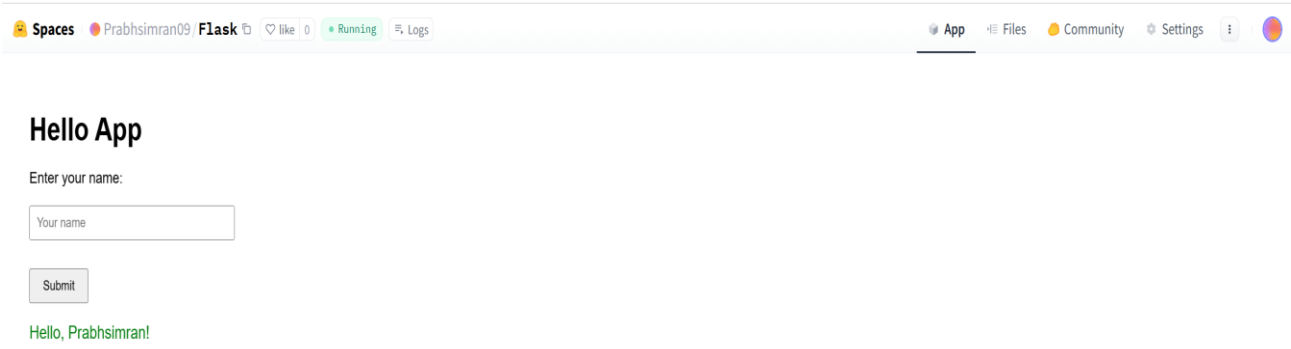

Figure 11.1.2.: Output of Hello App using Flask

When the user accesses the root URL, Flask responds with an HTML template rendered using Jinja2, Flask's built-in templating engine. Upon submitting the form, a POST request is triggered, and Flask captures the input name from the request, processes it, and displays a customized message like "Hello, Prabhsimran!" on the same page. This application illustrates the fundamental principles of Flask, including HTTP method handling, template rendering, and form processing. Deployed via Docker on Hugging Face Spaces, the app demonstrates how simple Python scripts can be turned into interactive, shareable web applications, making it an ideal project for beginners learning web development or deployment workflows.

11.1.2. Hugging Face Link for Hello App using Flask

https://huggingface.co/spaces/Prabhsimran09/Flask

Hello App - Flask

11.2. Streamlit

Streamlit is a modern open-source Python framework designed to make it effortless for data scientists and machine learning practitioners to create interactive web applications and data dashboards. Unlike traditional web frameworks that require knowledge of HTML, CSS, or JavaScript, Streamlit is built entirely in Python and uses a declarative programming approach. This means that developers simply describe what they want the app to look like using intuitive Python commands, and Streamlit handles all the rendering and interactivity behind the scenes.

Streamlit apps typically consist of linear Python scripts where each line of code corresponds to a visual element on the page. For example, st.image() displays an image, st.text_input() creates a user input field, and st.button() adds a clickable button. Every time a user interacts with a widget, the script is automatically re-executed from top to bottom, which keeps the application reactive and state-aware. This makes it especially well-suited for prototyping machine learning models, exploring data, or sharing insights with non-technical stakeholders.

One of the biggest advantages of Streamlit is its seamless integration with popular Python libraries such as pandas, NumPy, Matplotlib, Seaborn, Plotly, and TensorFlow. You can visualize data frames, plot graphs, and display model metrics with minimal effort. Streamlit also supports features like file uploads, sidebar navigation, caching for performance optimization, and even experimental support for authentication. In addition, you can deploy Streamlit apps directly to Hugging Face Spaces, Streamlit Community Cloud, or your own cloud service. Its elegant UI, fast development cycle, and minimal configuration make it a favorite among machine learning engineers and data scientists looking to share their work in a professional and interactive format.

11.2.1. Hello App using Streamlit

A Hello App using Streamlit is one of the simplest and most effective ways to demonstrate how Python scripts can be transformed into fully functional, interactive web applications. Streamlit is a Python-based open-source framework designed specifically for data scientists, machine learning engineers, and Python developers who want to build quick and intuitive user interfaces

without needing to write HTML, CSS, or JavaScript.

The main idea behind the Hello App is to capture user input, process it with a Python function, and then display the output on the same page—all with just a few lines of code. Streamlit applications run from top to bottom each time an interaction occurs, which ensures that the app remains dynamic and reactive.

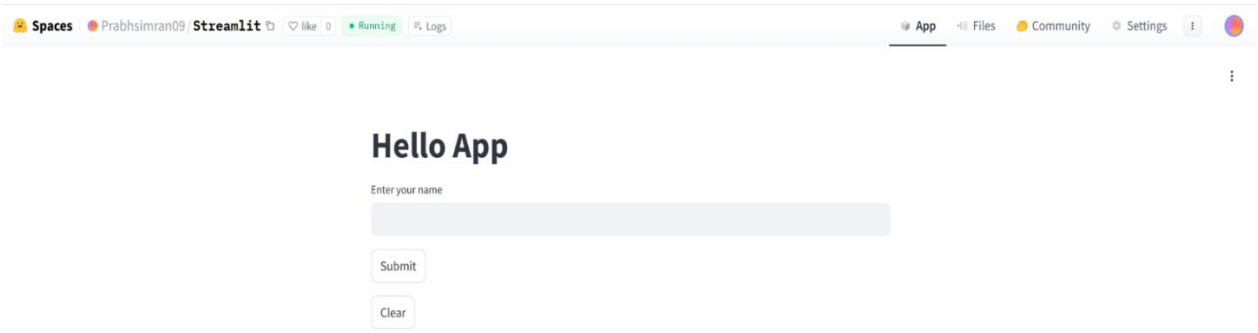

Figure 11.2.1.: Hello App using Streamlit Deployed on Hugging Face

The app layout in the image is clean and minimal, featuring a large heading ("Hello App"), a text input field labeled "Enter your name", and two buttons: "Submit" and "Clear". The Submit button likely triggers a Python function that generates a greeting based on the input name, while the Clear button resets the form, giving users a fresh input field. The interface is reactive—meaning that any interaction (like clicking a button or typing) causes the Python script to re-run from the top, which is a fundamental behavior of Streamlit apps.

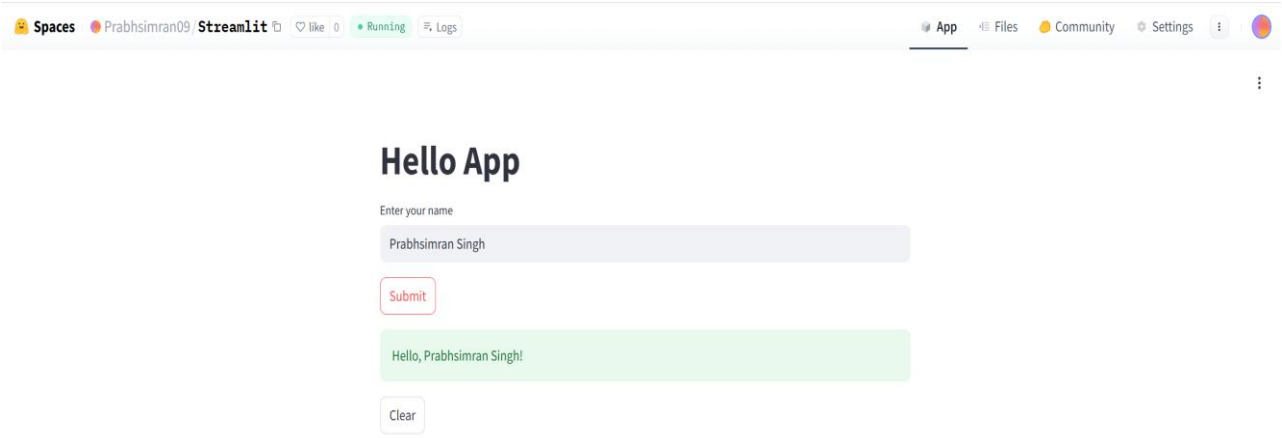

Figure 11.2.2.: Output of Hello App using Streamlit

The user has typed “Prabhsimran Singh” into the input box, clicked the “Submit” button, and the app responded with a success message: “Hello, Prabhsimran Singh!”, highlighted in a light green

notification box. This shows that the backend Python logic is functioning as expected—capturing the input, processing it, and returning the output immediately.

This simple yet powerful interaction showcases the core capability of Streamlit—building real-time web applications with just Python code and no front-end complexity. Hosted on Hugging Face Spaces, the app also benefits from cloud-based sharing, making it instantly accessible to users worldwide without local setup. Overall, this Hello App is an excellent example of how Streamlit can be used for rapid prototyping, educational purposes, or creating friendly user-facing tools with minimal effort.

11.2.2. Hugging Face Link for Hello App using Streamlit

https://huggingface.co/spaces/Prabhsimran09/Streamlit

Hello App -Streamlit

11.3. Gradio

Gradio is a highly intuitive Python library for creating interactive user interfaces around machine learning models. It's designed to make AI accessible not just to developers, but to end-users who may want to interact with a model through a web interface. Gradio allows you to wrap your ML model in a simple GUI with just a few lines of code. You define input components (like text boxes, image uploaders, sliders, or audio recorders) and output components (like labels, images, or plots), and Gradio automatically generates a responsive interface that can run locally or be shared online.

What makes Gradio particularly powerful is how easy it is to go from model to demo. For instance, you can take a trained image classification model, define an image input and a label output, and launch a web interface with gr.Interface(). Gradio supports a variety of input/output types, enabling applications in NLP, computer vision, audio processing, and beyond. Additionally, Gradio is natively supported by Hugging Face Spaces, making it extremely simple to deploy and share your apps without needing a server, domain, or complex deployment pipeline.

Another benefit of Gradio is its support for live collaboration. Each app you build can be instantly shared via a public URL, making it ideal for demonstrations, presentations, or user testing. It also

includes features for logging user inputs, which can be useful for collecting new data to improve your model. Unlike Flask or Streamlit, Gradio is less suited for full-fledged applications or dashboards, but it excels in rapid prototyping and end-user interactivity. It's the perfect tool when your main goal is to showcase your model with minimal code and maximum usability.

11.3.1. Hello App using Gradio

A Hello App using Gradio is one of the simplest examples to demonstrate how machine learning or Python-based functions can be converted into interactive web applications with minimal code. Gradio is a Python library that allows you to wrap a function inside a graphical user interface (GUI), making it accessible through a web browser without any need for HTML, CSS, or JavaScript.

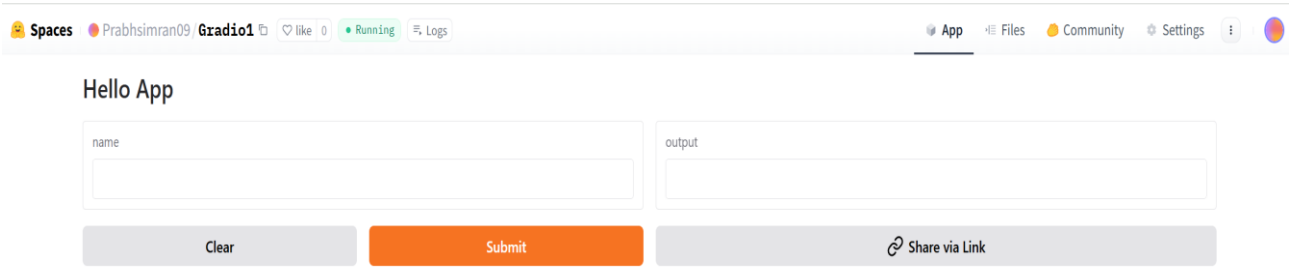

Figure 11.3.1.: Hello App using Gradio Deployed on Hugging Face

The "Hello App" typically involves a basic function that takes a user's name as input and returns a friendly greeting message, such as "Hello, Prabhsimran!" in the output.

In a Hello App, the user is presented with an input field labeled "name", where they can type their name. Upon clicking the "Submit" button, Gradio executes the backend Python function using the input provided.

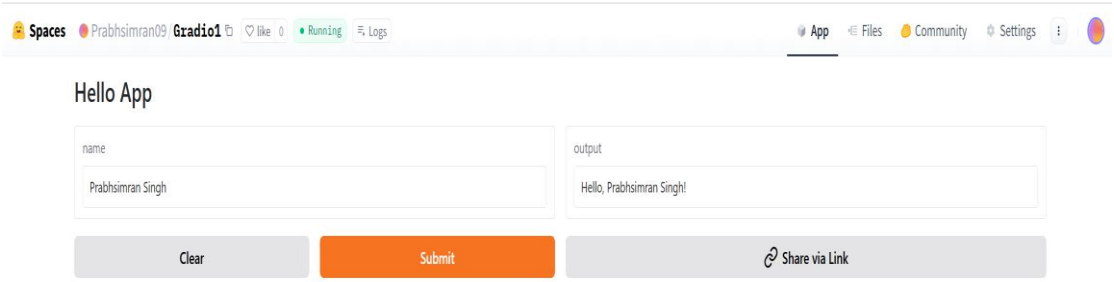

Figure 11.3.2.: Output of Hello App using Gradio

Gradio then displays the function's output—i.e., the greeting message—in a designated output field. The app typically includes a Clear button to reset the fields, and a Share via Link

option to instantly generate a public URL for easy access.

What makes Gradio especially powerful is how quickly you can deploy such an app online using platforms like Hugging Face Spaces, which provides free hosting for Gradio apps. This combination allows even non-programmers or beginners to interact with Python-based logic in a web browser. The Hello App thus serves as a gateway project for learning Gradio's components such as gr.Interface(), input/output components, event handling, and UI layout. It demonstrates the core principles of input–processing–output in an intuitive, visual way, making it ideal for teaching, demos, and rapid prototyping.

11.3.2. Hugging Face Link for Hello App using Gradio

https://huggingface.co/spaces/Prabhsimran09/Gradio1

Chapter 12: Natural Language Processing

Natural Language Processing (NLP) is a field at the intersection of linguistics, computer science, and artificial intelligence that focuses on enabling machines to understand, interpret, generate, and respond to human language in a meaningful way. It involves teaching computers to process large amounts of natural language data—whether spoken or written—and derive insights or take actions based on that data. NLP encompasses a range of tasks such as language translation, sentiment analysis, text classification, question answering, summarization, and chatbot development. At its core, NLP requires understanding both the syntax (structure) and semantics (meaning) of language. Early NLP techniques relied heavily on rule-based systems and statistical models, but modern approaches leverage deep learning, especially with models like RNNs, LSTMs, and more recently, transformers (e.g., BERT, GPT), which have revolutionized the field. NLP pipelines often involve text preprocessing steps like tokenization, stemming, lemmatization, and stopword removal, followed by feature extraction using techniques like Bag of Words, TF-IDF, or word embeddings. The ultimate goal of NLP is to bridge the gap between human communication and machine understanding, making technology more accessible, intuitive, and useful in daily life through voice assistants, recommendation systems, automated writing, and more.

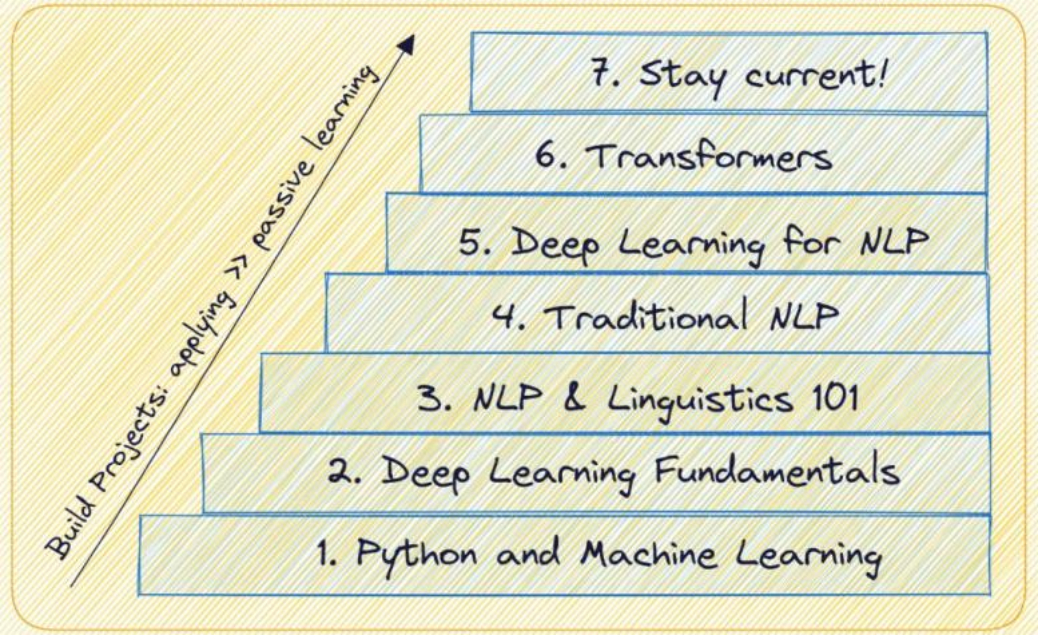

Figure 12.1: Stepwise Learning Path for NLP

The image depicts a stepwise learning path for mastering Natural Language Processing (NLP), starting from foundational concepts and progressing toward cutting-edge technologies. This

structured pyramid not only outlines what to learn, but emphasizes active learning through project-building over passive theory consumption.

1. **Python and Machine Learning:** This is the base of the pyramid and serves as the essential foundation. Python is the primary programming language used in AI and NLP. Alongside Python, understanding machine learning (ML) concepts such as supervised/unsupervised learning, model evaluation, feature engineering, and basic algorithms (like regression, decision trees, SVMs) is crucial before diving into NLP.
2. **Deep Learning Fundamentals:** Once comfortable with ML, the next step is deep learning. This includes grasping neural networks, backpropagation, activation functions, optimization techniques, and architectures like CNNs, RNNs, and LSTMs. A solid understanding of libraries like TensorFlow or PyTorch is also essential at this stage.
3. **NLP & Linguistics 101:** Here, you start exploring the actual mechanics of language. This includes syntactic and semantic analysis, part-of-speech tagging, named entity recognition, stemming, lemmatization, and an introduction to how machines understand and process human language using classical techniques.
4. **Traditional NLP:** This layer covers pre-deep learning approaches like rule-based systems, statistical NLP, TF-IDF, n-gram models, and traditional ML techniques applied to text (e.g., Naive Bayes, SVM for text classification). It helps build intuition on text processing and feature extraction.
5. **Deep Learning for NLP:** Now, deep learning techniques are applied to NLP. This includes word embeddings (Word2Vec, GloVe), sequence models (LSTMs, GRUs), attention mechanisms, and encoder-decoder frameworks for tasks like translation, text summarization, and sentiment analysis.
6. **Transformers:** This step introduces the revolutionary transformer architecture, the foundation for models like BERT, GPT, and T5. Key concepts include self-attention, multi-head attention, positional encoding, and pretraining/fine-tuning paradigms. Mastery of Hugging Face Transformers library is often part of this step.
7. **Stay Current!:** NLP is a rapidly evolving field. This top tier emphasizes the importance of keeping up with the latest research, new model releases, benchmarks, and evolving tools. Reading papers, participating in communities, and contributing to or replicating recent projects is essential.

The diagonal arrow on the left encourages project-based learning, implying that practical application (building projects) is far more effective than only consuming theoretical content. Real learning happens when you apply knowledge in real-world contexts like datasets,

competitions, or research replications.

This structured path is ideal for systematically becoming an NLP expert, ensuring both strong theoretical foundations and practical proficiency.

12.1. Natural Language Toolkit (NLTK)

NLTK (Natural Language Toolkit) is a powerful, open-source Python library designed for working with human language data, especially in the field of Natural Language Processing (NLP). It provides an extensive set of tools, datasets, and algorithms for performing a wide range of NLP tasks such as tokenization, stemming, lemmatization, part-of-speech tagging, named entity recognition (NER), parsing, and text classification. NLTK is widely used in academia for teaching and research purposes due to its simplicity and the large number of pre-built corpora (like Gutenberg, Brown, and WordNet) and lexical resources it offers.

Figure 12.1.1.: Logo of NLTK

One of the strengths of NLTK lies in its educational focus, making it ideal for beginners who want to understand the fundamentals of NLP by exploring algorithms at a low level. For example, learners can build their own Naive Bayes classifiers, perform word frequency analysis, or visualize syntax trees with minimal code. While it is excellent for learning and prototyping, NLTK is not optimized for performance in large-scale industry applications, where faster and more efficient libraries like spaCy or Hugging Face Transformers are often preferred. Nonetheless, NLTK remains a cornerstone in the NLP community for exploring the basics of language processing and building foundational knowledge.

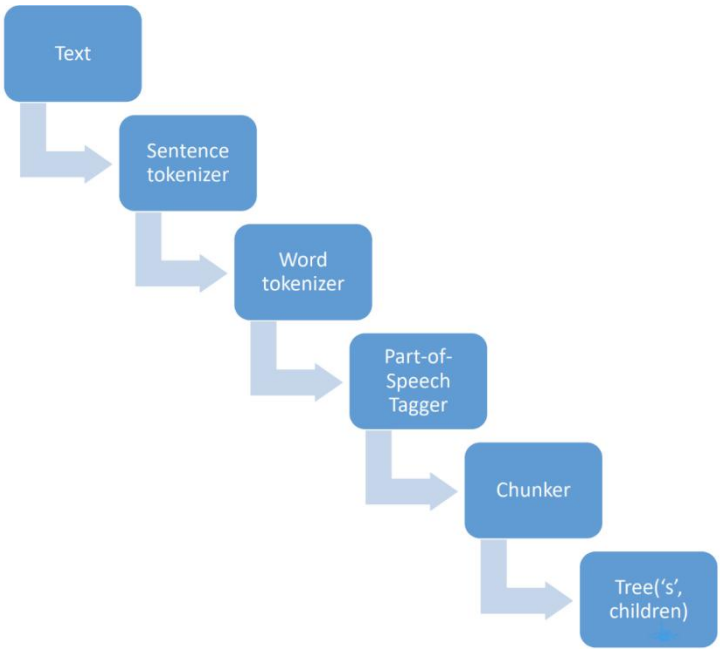

Figure 12.1.2.: NLTK Pipeline

The NLTK (Natural Language Toolkit) pipeline begins with raw text, which can be any block of written language. This text is first processed by a sentence tokenizer, which divides it into meaningful sentences, allowing sentence-level operations. These sentences are then passed through a word tokenizer to break them into individual word tokens and punctuation marks. After tokenization, each word is tagged with its part of speech (POS), such as noun, verb, or adjective, to give grammatical context. This is crucial for understanding sentence structure. The tagged tokens are then passed into a chunker, which groups them into higher-level grammatical units like noun phrases or verb phrases based on specified patterns. Finally, these chunks are structured into a parse tree, where each subtree represents a syntactic component of the sentence. This tree, often denoted as Tree('S', children), helps machines interpret and process natural language in a more human-like manner, forming the backbone of many NLP applications like text analysis, chatbots, and information extraction.

```
[4]  #import stopwords to this colab file
     import nltk
     nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
True
```

Figure 12.1.3.: Python code for Importing NLTK Library

The Python code snippet executed in Google Colab using the NLTK (Natural Language Toolkit) library is to download the stopwords dataset. Stopwords are commonly used words in a language (such as *the, is, in, and, a, to*) that usually do not carry significant meaning and are often removed

during text preprocessing in NLP tasks. In the code, the user imports the nltk module and then calls nltk.download('stopwords') to download the stopwords package. This package provides predefined lists of stopwords for various languages, which are useful for filtering out uninformative words from textual data. The output message indicates that the package has either been successfully downloaded or is already present and up-to-date in the local environment (/root/nltk_data), confirming successful access with the output True. This step is essential before using stopwords.words('english') or similar functions in NLTK-based NLP pipelines.

12.1.1. Tokenize Text Using NLTK

Tokenizing text using NLTK is one of the most fundamental steps in Natural Language Processing (NLP). It involves breaking down a piece of text into smaller units called tokens. These tokens can be words, sentences, or even characters, depending on the level of analysis. NLTK (Natural Language Toolkit) provides powerful and easy-to-use methods for both word tokenization and sentence tokenization.

1. **Word Tokenization**

   Word tokenization splits a sentence or paragraph into individual words and punctuation marks. It is useful for tasks like word frequency analysis, POS tagging, and text classification.

Example:

```
from nltk.tokenize import word_tokenize
text = "Hello, My name is Prabhsimran and I am learning NLP with NLTK!"
tokens = word_tokenize(text)
print(tokens)
```

Figure 12.1.1.1: Python Code for Word Tokenization

Output:

```
['Hello', ',', 'My', 'name', 'is', 'Prabhsimran', 'and', 'I', 'am', 'learning', 'NLP', 'with', 'NLTK', '!']
```

Figure 12.1.1.2: Individual Words and Punctuation Marks of a Sentence

This function automatically handles punctuation and contractions effectively.

2. **Sentence Tokenization**

   Sentence tokenization divides a paragraph into individual sentences, which is useful in summarization, dialogue systems, and syntactic parsing.

   Example:

```
from nltk.tokenize import sent_tokenize
paragraph = "NLTK is a leading platform for NLP. It is easy to use!"
sentences = sent_tokenize(paragraph)
print(sentences)
```

Figure 12.1.1.3.: Python Code for Sentence Tokenization

Output:

```
['NLTK is a leading platform for NLP.', 'It is easy to use!']
```

Figure 12.1.1.4: Individual Sentences of a Paragraph

NLTK uses pretrained models behind the scenes to recognize sentence boundaries accurately, accounting for common abbreviations and punctuation.

Tokenization is a key step in text preprocessing, and NLTK provides robust tools for it. Using word_tokenize() and sent_tokenize() functions, NLTK allows users to split text into individual words and sentences respectively. This process helps break down complex textual data into structured elements that are easier to analyze. Whether for information retrieval, chatbot development, or text classification, tokenization is the first step toward understanding and processing human language computationally.

12.1.2. Splitting Text

Splitting text refers to the process of dividing a block of text into smaller, manageable units for analysis or processing. This is a fundamental operation in Natural Language Processing (NLP) and is typically one of the first steps in text preprocessing. The splitting can be done at various levels depending on the application, such as splitting into sentences, words, or even characters. Python's built-in split() method can also be used to split text based on spaces or other delimiters, but it doesn't handle punctuation or complex rules like tokenizers.

Example:

```
text = "My Name is Prabhsimran Singh."
print(text.split())
```

Figure 12.1.2.1.: Python Code for Splitting Text

Output:

```
['My', 'Name', 'is', 'Prabhsimran', 'Singh.']
```

Figure 12.1.2.2.: Output of Splitting Text

Splitting text is a basic yet essential step in processing natural language. It involves dividing a larger body of text into smaller components such as sentences or words, which can then be analyzed independently. Libraries like NLTK provide specialized functions like sent_tokenize() and word_tokenize() that intelligently handle punctuation and linguistic nuances, unlike simple string methods like split(). This step plays a crucial role in preparing text for downstream NLP tasks, ensuring that language can be interpreted and processed by machines more effectively.

12.1.3. Counting Word Frequency

Word frequency is simply the number of times a word occurs in a text. Counting word frequency is a common and essential task in Natural Language Processing (NLP) that involves determining how often each word appears in a given text. It is used in text analysis, information retrieval, keyword extraction, and even training machine learning models. By counting word frequencies, we can identify important or repetitive words in a document and gain insights into its content and themes.

- **Steps to Count Word Frequency using NLTK**:

1. Tokenize the text into words.
2. Convert all words to lowercase (optional but useful to avoid case sensitivity).
3. Remove stopwords (optional to ignore common, unimportant words like “the”, “is”, “and”).
4. Use FreqDist from nltk to count word occurrences.

Example:

```
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.probability import FreqDist

# Download required resources
nltk.download('punkt')
nltk.download('stopwords')

# Sample text
text = "Data science is fun, and learning data science is useful."

# Tokenize and clean
tokens = word_tokenize(text.lower())  # Convert to lowercase
filtered_tokens = [word for word in tokens if word.isalpha()]  # Remove punctuation
stop_words = set(stopwords.words('english'))
clean_tokens = [word for word in filtered_tokens if word not in stop_words]  # Remove stopwords

# Count word frequencies
freq_dist = FreqDist(clean_tokens)
print(freq_dist.most_common())
```

Figure 12.1.3.1: Python Code for Counting Word Frequency

Output:

```
[('data', 2), ('science', 2), ('fun', 1), ('learning', 1), ('useful', 1)]
```

Figure 12.1.3.2: Words with their Frequency’s

12.1.4. Using WordNet to get synonyms words

WordNet is a large lexical database of the English language, developed at Princeton University, and is integrated into the NLTK library. It groups English words into sets of synonyms called synsets, provides short definitions, and records the various relationships between these synonym sets such as synonymy, antonymy, hypernymy, hyponymy, and more.

When working with Natural Language Processing (NLP), WordNet is particularly useful for expanding vocabulary, building word embeddings, creating semantic similarity measures, and performing tasks like word sense disambiguation or question answering.

Example:

```python
import nltk
from nltk.corpus import wordnet

# Download WordNet data (if not already downloaded)
nltk.download('wordnet')

def get_synonyms(word):
    synonyms = set()

    # Get all synsets for the word
    for synset in wordnet.synsets(word):
        for lemma in synset.lemmas():  # Get all lemma names in the synset
            synonyms.add(lemma.name())  # Add the lemma name to the set

    return sorted(synonyms)

# Example usage
word = "happy"
synonyms = get_synonyms(word)
print(f"Synonyms for '{word}':")
print(synonyms)
```

Figure 12.1.4.1: Python Code for Using WordNet to get Synonyms

Output:

```
Synonyms for 'happy':
['felicitous', 'glad', 'happy', 'well-chosen']
```

Figure 12.1.4.2: Synonyms of ‘happy’

Using WordNet to get synonymous words is a common NLP approach to enhance the semantic richness of text. WordNet organizes English words into synsets—sets of cognitive synonyms that express the same concept. Each synset provides not just synonyms but also definitions and relations to other synsets. By accessing synsets of a word through NLTK's WordNet interface and extracting their lemma names, one can compile a list of synonyms for that word. This process aids in various NLP tasks like semantic search, machine translation, paraphrase generation, and word sense disambiguation. In short, WordNet serves as a powerful linguistic tool to enrich vocabulary and capture the underlying meaning of words beyond their surface form.

12.1.5. Using WordNet to get antonyms words

WordNet not only provides synonyms through synsets and lemmas but also supports antonym relationships through lemma.antonyms(). In WordNet, antonyms are not directly available for every word; they are available only if the lexical database includes them explicitly. This feature is particularly helpful in sentiment analysis, text negation, semantic similarity, and paraphrasing.

Example:

```python
import nltk
from nltk.corpus import wordnet

# Download WordNet data if not already available
nltk.download('wordnet')

def get_antonyms(word):
    antonyms = set()

    for synset in wordnet.synsets(word):
        for lemma in synset.lemmas():
            for ant in lemma.antonyms():  # Check for antonyms
                antonyms.add(ant.name())   # Add antonym name

    return sorted(antonyms)

# Example usage
word = "happy"
antonyms = get_antonyms(word)
print(f"Antonyms for '{word}':")
print(antonyms)
```

Figure 12.1.5.1.: Python Code for using WordNet to get Antonyms

Output:

```
Antonyms for 'happy':
['unhappy']
```

Figure 12.1.5.2: Antonyms of ‘happy’

Using WordNet to find antonyms involves accessing a word’s synsets and checking each lemma for associated antonyms. In NLTK, the antonyms() method tied to a lemma provides this functionality. However, antonyms are available only for certain word types, typically adjectives and verbs, and may not exist for all entries in WordNet. The process is helpful in tasks requiring polarity detection, contrastive sentence generation, and sentiment inversion. With just a few lines of code, you can build a system that intelligently identifies words with opposite meanings, enhancing the depth of your NLP applications.

12.1.6. Word Stemming

Stemming is a fundamental technique in Natural Language Processing (NLP) used to reduce a word to its base or root form, known as the stem. The primary goal of stemming is to treat different forms of a word as a single item so that they can be analyzed together. For instance, the

words "connect", "connected", "connection", "connecting" are all reduced to the root word "connect" through stemming.

Example:

```python
from nltk.stem import PorterStemmer

# Initialize the stemmer
stemmer = PorterStemmer()

# Sample words
words = ["running", "runs", "easily", "fairness", "connected"]

# Apply stemming
stems = [stemmer.stem(word) for word in words]
print(stems)
```

Figure 12.1.6.1: Python Code for Word Stemming

Output:

```
['run', 'run', 'easili', 'fair', 'connect']
```

Figure 12.1.6.2: Stemmed Words

12.2. Google Colab Link for NLP

https://colab.research.google.com/drive/1g59naEZpQP3a9ECMoKcWjkrntqxMQ-lF?usp=sharing

NLP

12.3. Techniques for text representation used within NLP

12.3.1. Bag of Words (BoW)

Bag of Words (BoW) is one of the earliest and simplest techniques in NLP for converting text into numerical features that machine learning algorithms can process. It treats a document as a collection (or "bag") of individual words, disregarding grammar and word order, but keeping track of the number of times each word appears. To use BoW, we first build a vocabulary of all unique words in the dataset, and then each document is represented as a vector where each

dimension corresponds to a word from the vocabulary. For example, consider two sentences: *"I love NLP"* and *"I love AI"*. The combined vocabulary is ["I", "love", "NLP", "AI"], and each sentence is encoded as a vector of word counts: [1, 1, 1, 0] for the first sentence, and [1, 1, 0, 1] for the second. While BoW is simple and effective for small datasets, it lacks the ability to capture the meaning or context of words, making it less suitable for complex NLP tasks.

Example:

```
# Define some training utterances

class Category:
  BOOKS = "BOOKS"
  CLOTHING = "CLOTHING"
  ENTERTAINMENT ="ENTERTAINMENT"

train_x = ["i love the book", "this is a great book", "the fit is great", "i love the shoes", "Music is good for soul","Movies are refreshing"]
train_y = [Category.BOOKS, Category.BOOKS, Category.CLOTHING, Category.CLOTHING, Category.ENTERTAINMENT, Category.ENTERTAINMENT]
# Fit vectorizer to transform text to bag-of-words vectors

from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer(binary=True)
train_x_vectors = vectorizer.fit_transform(train_x)
# Train SVM Model

from sklearn import svm

clf_svm = svm.SVC(kernel='linear')
clf_svm.fit(train_x_vectors, train_y)
# Test new utterances on trained model
test_x1 = vectorizer.transform(['I enjoyed the story in that book'])
test_x2 = vectorizer.transform(['i love the music of the song'])
test_x3 = vectorizer.transform(['shoes are good'])
test_x4 = vectorizer.transform(['i love the shoes'])
print(clf_svm.predict(test_x1) )
print(clf_svm.predict(test_x2) )
print(clf_svm.predict(test_x3) )
print(clf_svm.predict(test_x4) )
```

Figure 12.3.1.1.: Python Code for Bag of Words (BoW)

The goal is to train a machine learning model to recognize the intent or topic of a sentence based on its word content. The process starts by defining training data (train_x), which consists of labeled sentences related to books, clothing, or entertainment. These sentences are transformed into numerical features using CountVectorizer, which converts each sentence into a binary vector indicating the presence (1) or absence (0) of each word in the vocabulary, this is the Bag-of-Words approach.

Output:

```
['BOOKS']
['CLOTHING']
['ENTERTAINMENT']
['CLOTHING']
```

Figure 12.3.1.2.: Output of Bag of Words (BoW) Code

This simple yet powerful technique demonstrates how text representation + classical ML can build functional NLP applications like chatbots, recommendation systems, or auto-tagging services.

12.3.2. Word Vector

Word vectors, also known as word embeddings, are a foundational concept in modern Natural Language Processing (NLP). They represent words as dense, continuous-valued vectors in a high-dimensional space, where the semantic relationships between words are captured through their relative positions. Unlike traditional methods like Bag of Words, which represent words as isolated tokens without meaning, word vectors assign each word a numerical representation that encodes its contextual meaning based on how frequently and in what context it appears with other words. Popular techniques for generating word vectors include Word2Vec, GloVe, and FastText. For example, in a vector space, words like "king", "queen", "man", and "woman" can exhibit meaningful relationships such as king – man + woman ≈ queen. This shows that the model has learned some understanding of gender and royalty concepts. Word vectors are particularly powerful because they reduce the dimensionality of language data while preserving relationships, making them highly useful for downstream NLP tasks like sentiment analysis, text classification, machine translation, and question answering. However, early word vectors are context-independent — meaning the word "bank" has the same vector whether it refers to a riverbank or a financial bank — which is a limitation addressed by more advanced models like BERT.

Example:

```
import spacy

nlp = spacy.load("en_core_web_md")
docs = [nlp(text) for text in train_x]
train_x_word_vectors = [x.vector for x in docs]
from sklearn import svm

clf_svm_wv = svm.SVC(kernel='linear')
clf_svm_wv.fit(train_x_word_vectors, train_y)
test_x = ["I went to the mall to buy clothes", "let's go and watch some claasic movies"]
test_docs = [nlp(text) for text in test_x]
test_x_word_vectors =  [x.vector for x in test_docs]

clf_svm_wv.predict(test_x_word_vectors)
```

Figure 12.3.2.1.: Python Code for Word Vector

This code demonstrates a text classification pipeline using word vectors from the spaCy NLP library. It uses a pre-trained English model (en_core_web_md) that provides 300-dimensional word embeddings capturing the semantic meaning of words. Each training sentence (train_x) is passed through spaCy's pipeline to extract its vector representation, which is essentially the average of its word embeddings. These sentence-level vectors (train_x_word_vectors) are then used to train an SVM classifier (svm.SVC) with a linear kernel. During testing, new sentences are similarly vectorized and passed to the trained classifier for prediction. This approach allows the model to understand semantic similarities, so it can better distinguish between meanings like “check” as a bank term versus a verification action, something BoW cannot do.

Output:

```
array(['CLOTHING', 'ENTERTAINMENT'], dtype='<U13')
```

Figure 12.3.2.2.: Output of Python Code for Word Vector

12.3.3. Regular Expressions (RegEx)

Regular Expressions (RegEx) are a powerful pattern-matching technique used in Natural Language Processing (NLP) and text processing tasks to identify, extract, or manipulate specific patterns within strings. They allow developers to define complex search criteria using a sequence of special characters that form a search pattern. For example, a RegEx like \d{3} can match any sequence of exactly three digits, while [A-Za-z]+ matches one or more alphabetic characters. In NLP, RegEx is commonly used during the text preprocessing phase to clean raw data by removing punctuation, extracting email addresses or phone numbers, identifying dates, tokenizing text, or filtering out unwanted symbols. While RegEx doesn't understand language or semantics, it is an essential rule-based method for handling repetitive or structured text formats quickly and efficiently. Despite being less intelligent than modern deep learning models, its simplicity, speed, and precision make it highly valuable in scenarios where exact string patterns matter, such as log analysis, data validation, and lightweight NLP pipelines.

Example:

```python
import re

regexp = re.compile(r"\bread\b|\bstory\b|book")

phrases = ["I liked that story.", "the car treaded up the hill", "this hat is nice"]

matches = []
for phrase in phrases:
  if re.search(regexp, phrase):
    matches.append(phrase)

print(matches)
```

Figure 12.3.3.1.: Python Code for RegEx

This code uses Python's Regular Expressions (RegEx) to search for specific words—"bread", "story", or "book"—within a list of phrases. The pattern \bbread\b|\bstory\b|book matches these words as whole words (\b indicates word boundaries), ensuring it doesn't match them as part of other words. The code then loops through each phrase and uses re.search() to check if the pattern exists. If a match is found, the phrase is added to the matches list. Finally, it prints all phrases containing the target words. This is a common NLP technique to filter or extract sentences based on keyword patterns.

Output:

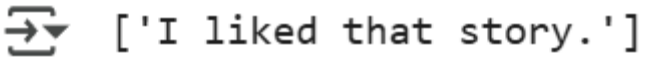

Figure 12.3.3.2.: Output of RegEx Code

The pattern used (\bbread\b|\bstory\b|\bbook\b) is designed to detect any of the words "bread", "story", or "book" as complete, standalone words (thanks to the \b word boundaries). Among the provided phrases, only "I liked that story." contained one of these keywords, "story", so it was added to the matches list and printed. This demonstrates how regex can be used to filter meaningful sentences based on specific keywords in NLP tasks.

12.4. Google Colab link for NLP Techniques

https://colab.research.google.com/drive/128_FC1C-_BAIYFnrxwH_cx9scrXiIr_9?usp=sharing

NLP Techniques

12.5. Recurrent Neural Networks (RNN)

Recurrent Neural Networks (RNNs) are a class of artificial neural networks specifically designed for processing sequential data, where the order of input data plays a crucial role in understanding its meaning. Unlike traditional feedforward neural networks, which assume that all inputs are independent of each other, RNNs leverage the temporal or sequential nature of data by incorporating a memory mechanism. This memory allows them to retain information about past inputs, making RNNs suitable for a wide range of tasks such as natural language processing (NLP), time-series prediction, speech recognition, and music generation.

At the heart of an RNN is the idea of sharing parameters across time steps and maintaining a hidden state that gets updated as each new input is processed. When a sequence is fed into an RNN, it processes one element (e.g., a word or time-step value) at a time. At each time step, the RNN takes the current input along with the hidden state from the previous time step and computes a new hidden state. This updated state contains information from both the current and past inputs, thereby enabling the network to "remember" past context as it moves through the sequence. This is particularly useful in language modelling where the meaning of a word often depends on the words that precede it.

The diagram below illustrates the architecture and working principle of a Recurrent Neural Network (RNN), which is designed to process sequential data by maintaining an internal memory across time steps. On the left side, the image shows a single RNN unit that receives an input xxx and passes information to its internal state through a feedback mechanism, also known as a recurrent loop. This feedback loop allows the network to retain information from previous inputs, enabling it to "remember" past context while processing new data. This is the core feature that distinguishes RNNs from traditional feedforward neural networks, which process each input independently.

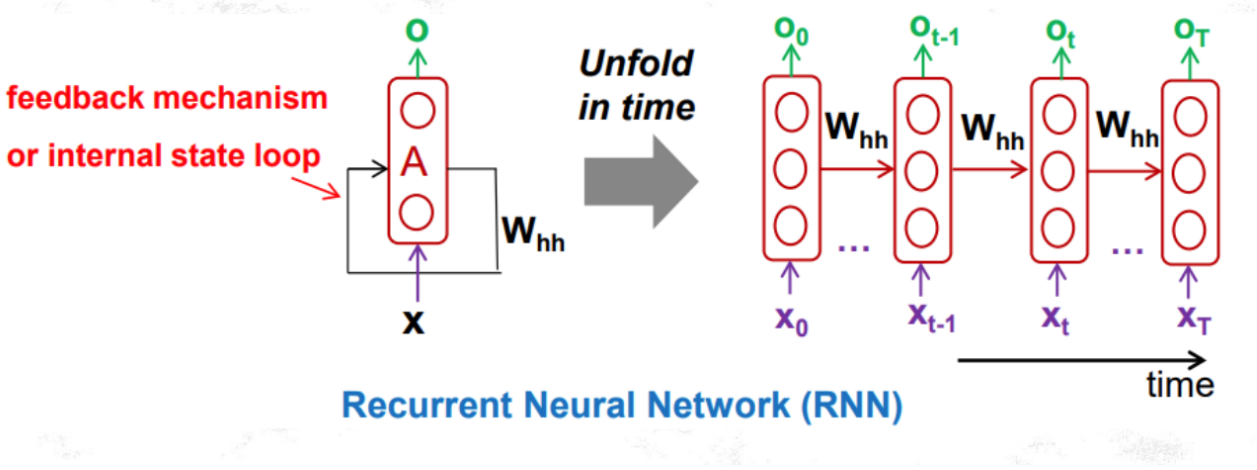

Figure 12.5.1: Architecture of Recurrent Neural Network (RNN)

The right side of the diagram shows the unfolding of the RNN through time. This unrolling represents how the network handles an entire sequence. At each time step t, the RNN receives a new input xt, combines it with the hidden state from the previous step and computes a new hidden state ht. This process continues for every element in the sequence allowing the network to accumulate knowledge about the sequence as it progresses. The same weights are shared across all time steps, which reduces the number of parameters and ensures consistency in how the network learns temporal dependencies.

Each step also generates an output ot, which can be used for tasks like sequence prediction, classification, or translation. Because the hidden state is carried forward, the RNN can learn relationships between earlier and later inputs—making it particularly useful in natural language processing tasks such as language modeling, sentiment analysis, and speech recognition. However, standard RNNs struggle with long-term dependencies due to vanishing or exploding gradients during training, which has led to the development of more advanced models like LSTM and GRU. Still, this diagram provides a foundational understanding of how RNNs work: by processing sequences one step at a time, remembering past inputs, and learning temporal patterns through internal state updates.

The diagram in Figure provides a simplified representation of how a Recurrent Neural Network (RNN) works in the context of Named Entity Recognition (NER). It demonstrates how the RNN processes a sentence — here, “Prabhsimran lives in Chandigarh” — one word at a time to assign a label to each word, such as person, location, or other. The process begins at the bottom, where each word is encoded as a one-hot vector — a binary representation indicating the presence of a specific word in a fixed vocabulary. These one-hot vectors are input to the RNN.

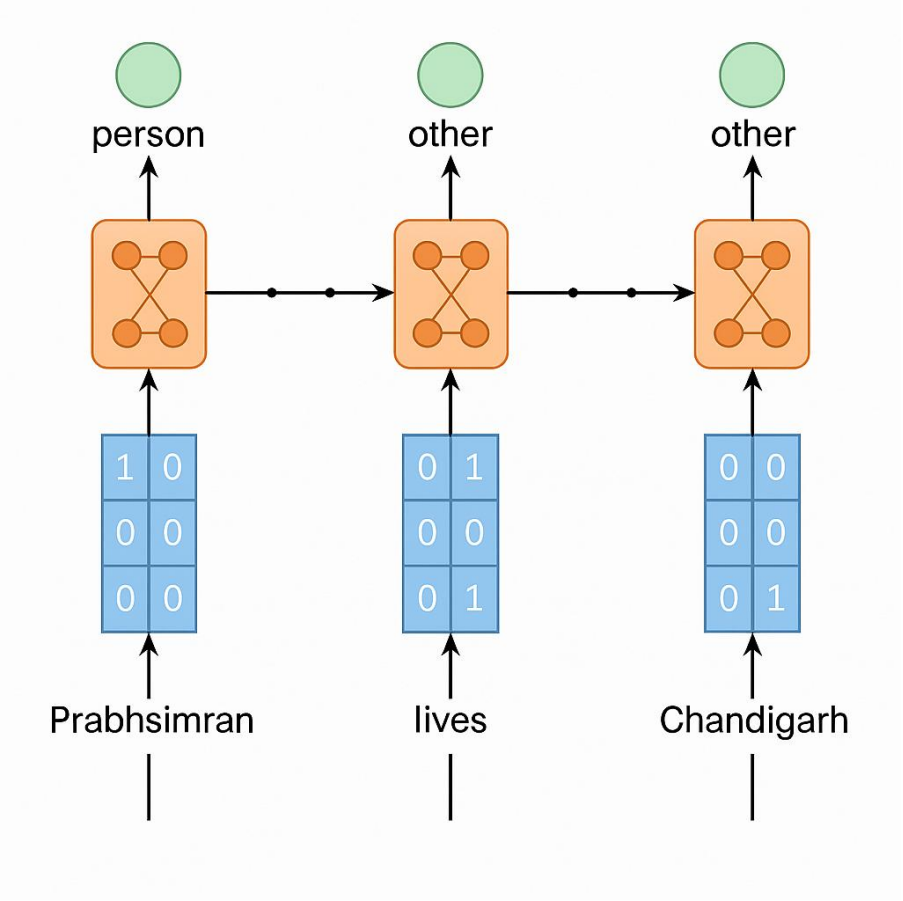

Figure 12.5.2.: Working of RNN

The orange blocks in the diagram represent the RNN cells. Each cell is responsible for processing its respective input word while maintaining a hidden state that gets passed forward to the next cell. This hidden state acts as memory, allowing the model to retain context from earlier words in the sequence. For example, when the RNN processes "Prabhsimran", it learns that it could be a proper noun, and when the next word "lives" follows, the RNN can use that context to better classify "Prabhsimran" as a person.

Each RNN cell outputs a label prediction (shown as green circles above each cell) based on the hidden state and current input. These predictions are the final NER tags for the input words. In this example, "Prabhsimran" is correctly identified as a person, while "lives" and "Chandigarh" are classified as other — though in a real-world model, "Chandigarh" would ideally be tagged as a location, depending on training data and model complexity.

12.5.1. Computational Graphs

The diagram in Figure illustrates the sequential nature of a Recurrent Neural Network (RNN), highlighting how it processes a sequence of inputs over time by maintaining a hidden state that evolves at each time step. The purple blocks labeled x1, x2, x3, x_1, x_2, x_3 represent the input sequence, such as words in a sentence or time-series data points. The RNN begins with an initial state A0, which can be a zero vector or a learned parameter, and updates this state as it receives each new input. The red function blocks labeled ‘gθ’ denote the RNN cell's computation using parameters θ, which are shared across all time steps to ensure consistency and reduce the number of learnable parameters.

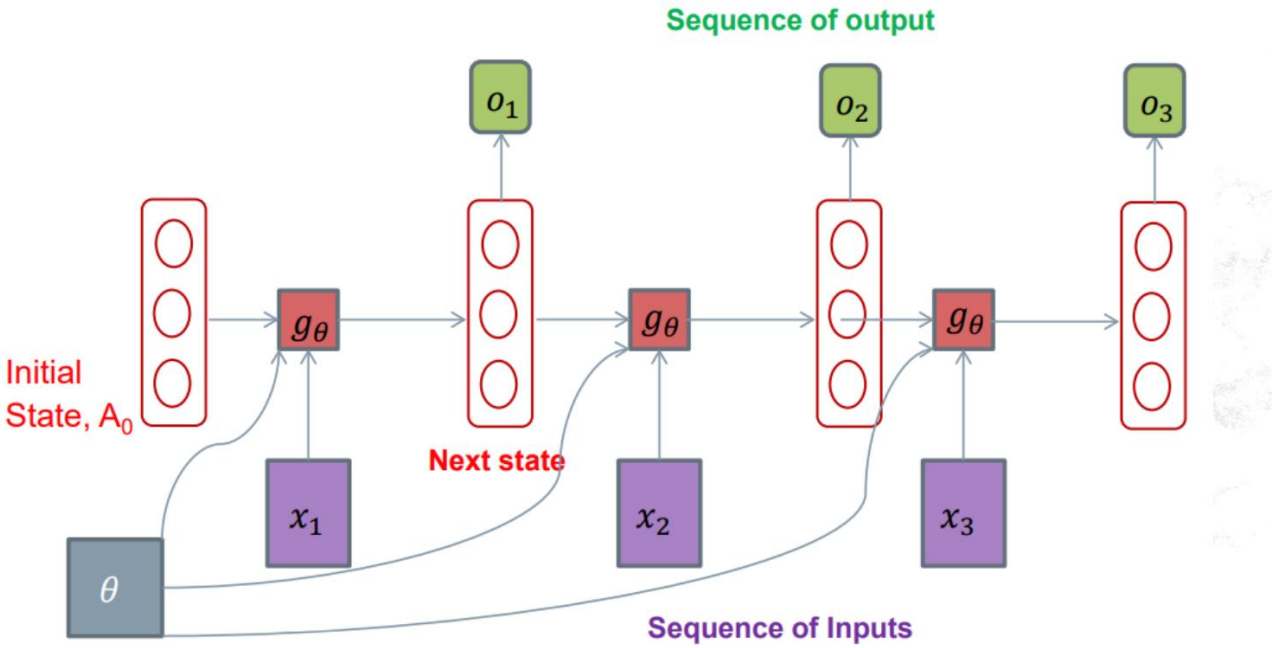

Figure 12.5.1.1.: Computational Graph

At each time step, the current input xt and the previous hidden state are combined using the function gθ to produce a new hidden state, which serves as memory and context for future inputs. This updated state is then passed forward to the next time step. Simultaneously, each hidden state is also used to generate an output ot (green blocks at the top), making the RNN capable of producing a sequence of outputs — such as labels, predictions, or generated text — as it reads the sequence. This dynamic structure enables the RNN to capture temporal dependencies and contextual relationships in sequential data, which is essential for tasks like language modeling, speech recognition, and time-series forecasting. The unfolding of the network in time shown in the diagram emphasizes how the same network cell is reused to process each element in the sequence while preserving the temporal context.

12.5.2. Different Computational Graphs

The diagram in Figure illustrates the five main types of sequence processing models using Recurrent Neural Networks (RNNs), categorized based on the number of inputs and outputs. These patterns are fundamental in modeling different NLP and time-series tasks where sequences are involved.

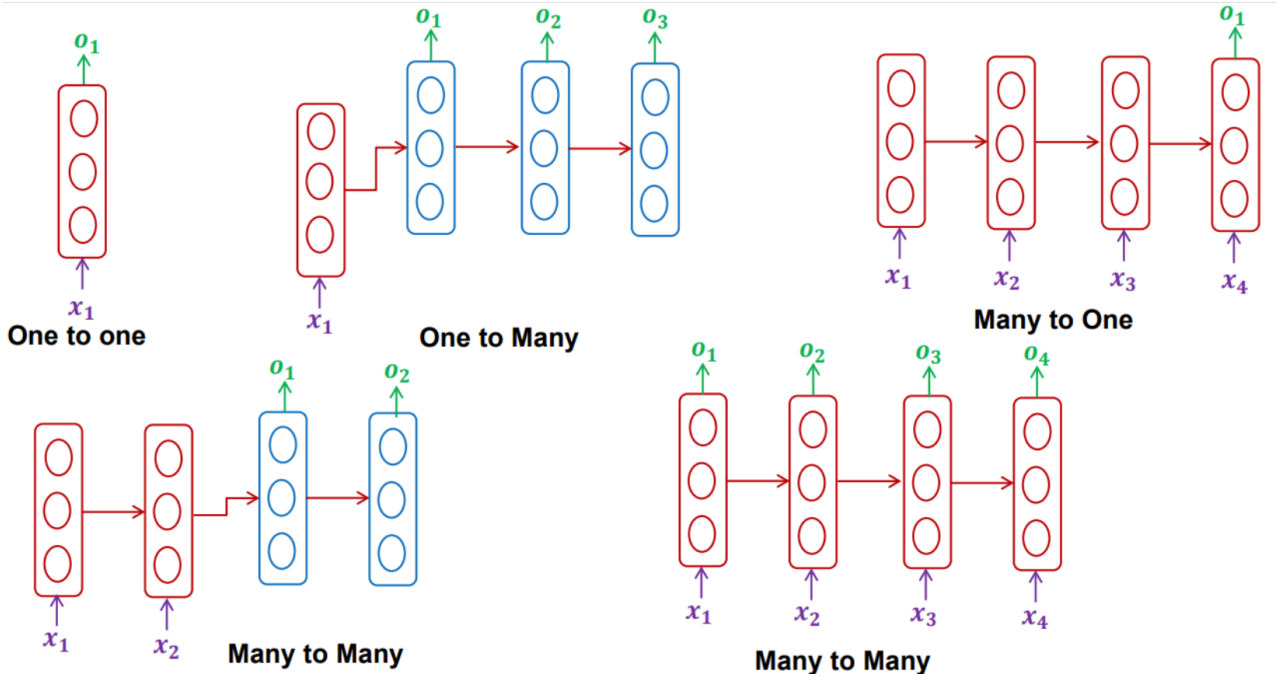

Figure 12.5.2.1: Different Computational Graphs

1. **One-to-One (Top-left):** This is the simplest model where a single input corresponds to a single output. It's equivalent to a basic feedforward neural network and is commonly used in tasks like image classification where the input is a single instance and so is the output.
2. **One-to-Many (Top-middle):** Here, a single input leads to a sequence of outputs. This setup is useful in tasks like music generation or image captioning, where one input (e.g., a musical key or an image feature) generates a sequence (notes or words).
3. **Many-to-One (Top-right):** This model processes a sequence of inputs and produces a single output. It's widely used in sentiment analysis, where the full sentence (sequence of words) is used to classify sentiment (positive/negative).
4. **Many-to-Many (Equal Length) (Bottom-right):** In this type, each input in a sequence has a corresponding output. It's commonly used in part-of-speech tagging or named entity recognition, where every word in a sentence (input) must be labeled with a tag (output).

5. **Many-to-Many (Encoder-Decoder) (Bottom-left):** This is the most flexible and widely used architecture in tasks like machine translation. The model first reads a sequence of inputs (e.g., an English sentence), processes it into a fixed-length context vector (encoder), and then generates a sequence of outputs (e.g., a sentence in French) via a decoder.

Each of these configurations enables RNNs to tackle a different class of problems by adapting the input-output structure to the task's nature. These architectures have become the backbone of modern applications like language modelling, translation, summarization, and conversational AI.

12.5.3. Applications of RNN's

1. Natural Language Processing (NLP)

a) Language Modelling

RNNs are used to predict the next word or character in a sequence by learning patterns in language. For example, after training on a large text corpus, an RNN might predict that the word following "the cat sat on the" is "mat".

b) Text Generation

After learning language structures, RNNs can generate new, coherent text by sampling one word at a time and feeding its output back as the next input. This is used in applications like chatbot responses or auto-writing tools.

c) Machine Translation

In sequence-to-sequence models, an encoder RNN processes the source sentence, and a decoder RNN generates the translated sentence. Though Transformers now dominate this field, RNNs laid the foundation for early translation tools (e.g., English to French).

d) Sentiment Analysis

RNNs can process entire sequences of words in reviews or tweets and predict whether the overall sentiment is positive, negative, or neutral by understanding context across the sentence.

e) Named Entity Recognition (NER)

RNNs label each word in a sentence (e.g., "John" as a person, "India" as a location), enabling downstream tasks like information extraction, question answering, and voice assistants.

2. Speech Recognition and Processing

RNNs excel in speech-to-text systems by converting audio sequences into written words. Since speech is inherently sequential and contextual (intonation and pauses matter), RNNs are ideal for learning patterns in spoken language. Long Short-Term Memory (LSTM) and Bidirectional RNNs (Bi-RNNs) are often used for this task.

3. Time Series Prediction

RNNs are heavily used in financial markets, weather forecasting, and demand planning, where future values depend on historical sequences.

a) Stock Price Prediction

Given past stock prices, RNNs can predict future trends or prices.

b) Weather Forecasting

By feeding the model with historical weather parameters (temperature, humidity), RNNs can predict short-term or even seasonal weather conditions.

c) Sales and Inventory Forecasting

Retailers use RNNs to forecast future product demand based on previous sales data, seasonality, and promotions.

4. Video Analysis and Classification

RNNs process video frame sequences for tasks like:

- **Activity recognition** (e.g., walking, jumping)
- **Scene understanding**
- **Video captioning** (generating text that describes video content)

Here, each video frame is processed sequentially, and the RNN learns temporal dependencies between them.

5. Music and Audio Generation

RNNs are used to create music by learning from sequences of musical notes or MIDI events.

- **Music Composition:** Models like LSTM can generate melodies or accompaniment that sound like they were composed by humans.
- **Voice synthesis:** In text-to-speech systems, RNNs help convert sequences of phonemes or characters into realistic-sounding audio.

6. Healthcare and Medical Monitoring

a) Patient Monitoring

RNNs can process time-series data from wearable sensors (heart rate, ECG, etc.) to detect abnormalities or predict medical events like seizures or arrhythmias.

b) Electronic Health Record (EHR) Analysis

Patient histories are naturally sequential. RNNs can predict disease progression, treatment outcomes, or readmission risk based on temporal health data.

7. Handwriting and Gesture Recognition

RNNs can interpret sequential pen movements or touch inputs to:

- Recognize handwritten text in applications like note-taking apps.
- Interpret gestures from touchscreens or sensors for human-computer interaction.

8. Autonomous Systems and Control

RNNs are used in:

- Autonomous driving, for predicting pedestrian movements and vehicle trajectories.
- Robotics, where RNNs learn control policies over time from sensor inputs.

9. Recommendation Systems

RNNs can model a user's sequence of clicks, views, or purchases to make dynamic recommendations. For example, if a user watches a series of horror movies, the system might recommend more intense or related content based on sequential behavior.

Chapter 13: Projects

13.1. QuadPredict

The website QuadPredict is a platform that focuses on predictive analytics using artificial intelligence and machine learning. The site is designed to provide users with various prediction tools across different domains, leveraging AI-based models for data-driven decision-making.

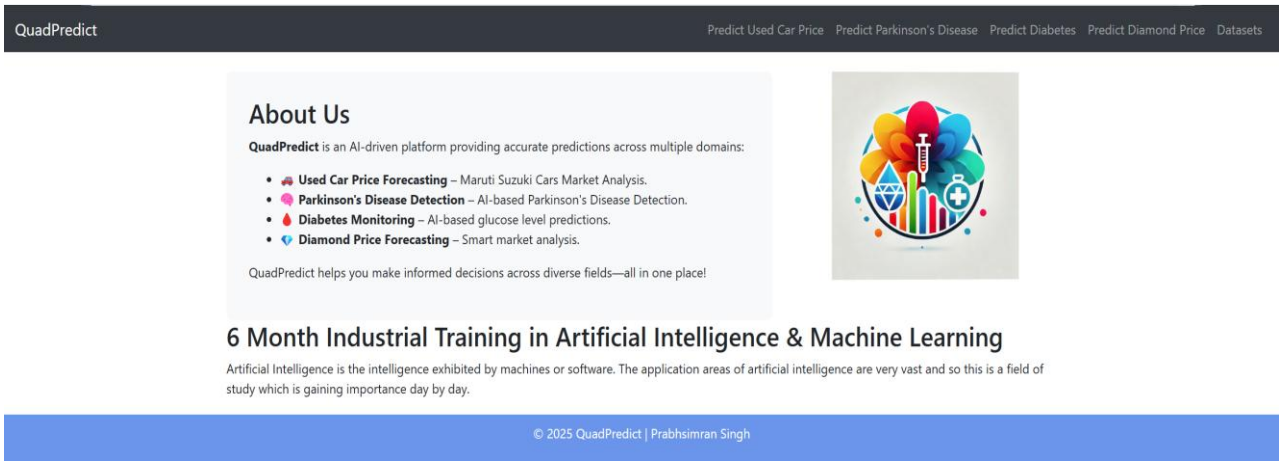

Figure 13.1: QuadPredict Homepage

It features tools for forecasting used car prices, specifically focusing on Maruti Suzuki models, detecting Parkinson's Disease through AI-based analysis, monitoring diabetes via glucose level predictions, and forecasting diamond prices through market analysis. Additionally, the site provides access to relevant datasets, supporting users in making informed decisions across these diverse fields.

13.1.1. Used Car Price Prediction (Maruti Suzuki Models)

One of the primary tools on the website is a machine learning model designed to predict the prices of used Maruti Suzuki cars. The model is trained on real-world datasets containing details such as car model, year of manufacturing, fuel type, mileage, and other relevant attributes. By inputting the required details, users can estimate the expected resale value of a car. This tool is particularly useful for buyers and sellers in the used car market, enabling them to make informed financial decisions.

Histograms are useful for understanding data distributions, patterns, and outliers, which are crucial for preprocessing and model training.

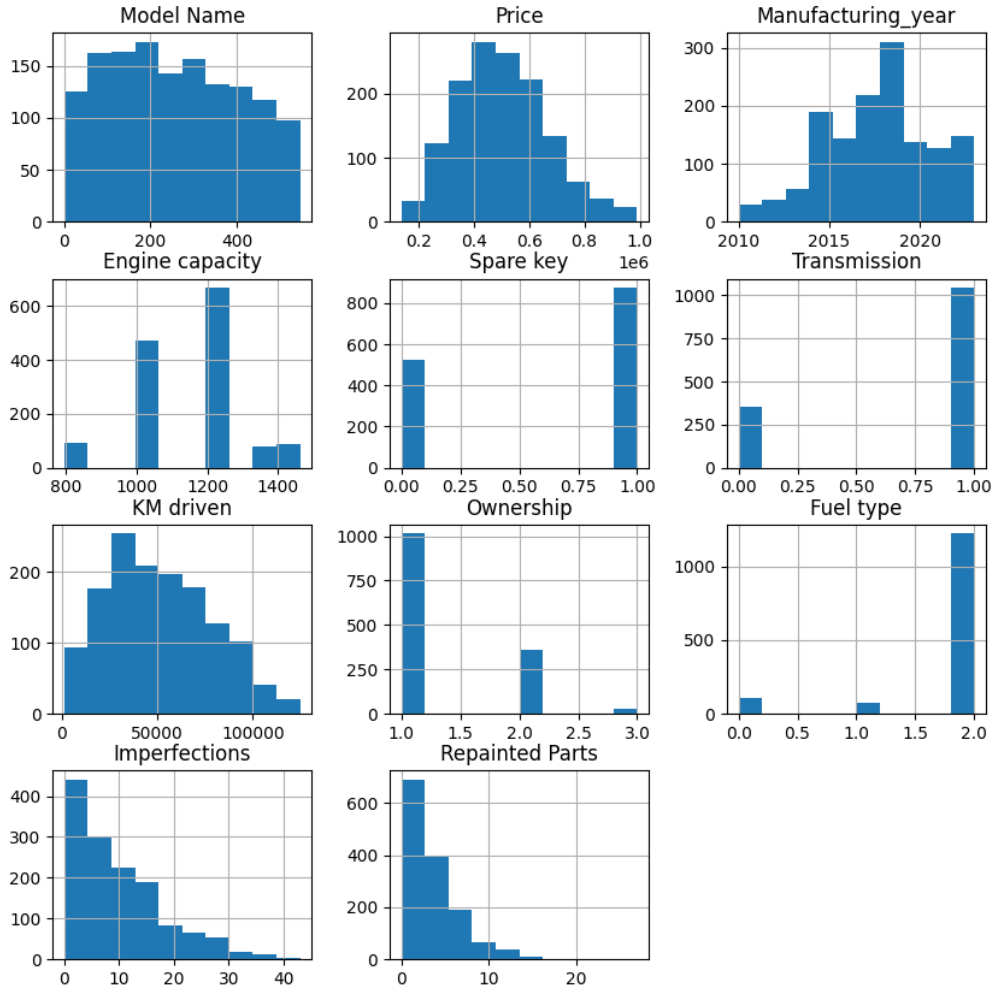

Figure 13.1.1.1: Histograms for Used Car Price Prediction Model

The Model Name histogram suggests that certain car models appear more frequently. The Price distribution is slightly right-skewed, indicating that most cars fall within a lower price range, with fewer expensive cars. The Manufacturing Year histogram shows that newer cars, especially after 2015, dominate the dataset. Engine Capacity data suggests that specific capacities, such as 1000cc and 1200cc, are more common. The Spare Key and Transmission histograms likely represent categorical features, showing the availability of spare keys and the proportion of manual versus automatic cars. KM Driven follows a spread where most cars have moderate mileage, while Ownership data reveals that the majority of vehicles are first-hand owners. The Fuel Type distribution indicates one dominant fuel category, possibly petrol or diesel. Imperfections and Repainted Parts histograms show that most cars have minimal physical wear and repainting, but some have significant defects. These histograms help in feature engineering, data cleaning, and model training by identifying key trends and anomalies. Proper handling of categorical and numerical variables, such as encoding transmission type and normalizing price or mileage, can improve the performance of predictive models. Ultimately, these visualizations provide valuable insights for car resale value estimation, helping in decision-making for buyers,

sellers, and machine learning applications.

The heatmap uses a color gradient, where lighter colors indicate stronger positive correlations, and darker colors indicate stronger negative correlations.

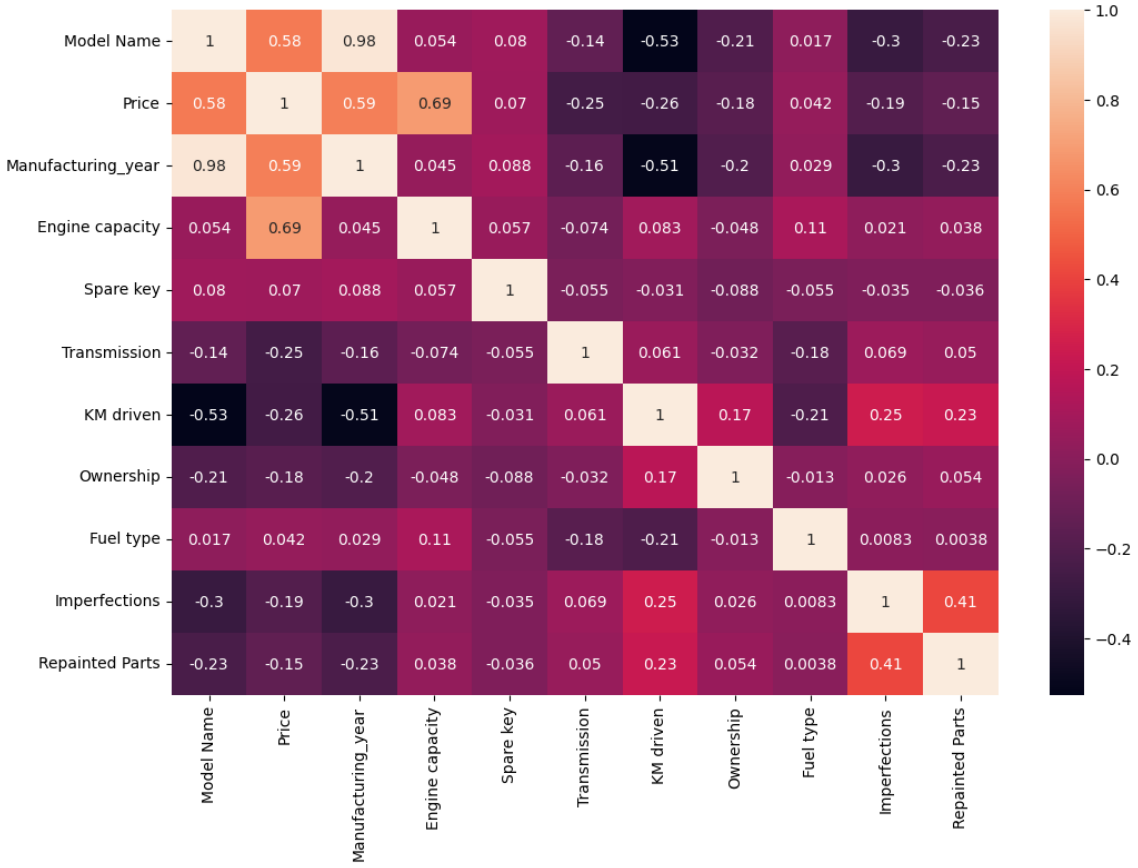

Figure 13.1.1.2: Heatmap for Used Car Price Prediction Model

The diagonal values are all 1, as each variable is perfectly correlated with itself. From the heatmap, we observe that Price has a positive correlation with Engine Capacity (0.69) and Manufacturing Year (0.59), indicating that newer cars and those with larger engines tend to have higher prices. Model Name is highly correlated with Manufacturing Year (0.98), suggesting that newer car models are frequently recorded in the dataset. KM Driven has a negative correlation with Manufacturing Year (-0.51) and Price (-0.26), meaning older cars tend to have higher mileage and lower resale value. Imperfections and Repainted Parts show a strong correlation (0.41), implying that cars with more imperfections are more likely to have undergone repainting. Transmission (-0.25) and Fuel Type (0.04) have weaker correlations with price, suggesting they have a minimal impact on price prediction. The heatmap is valuable for feature selection in machine learning models, as highly correlated variables can introduce redundancy, while weakly correlated features may have limited predictive power. Understanding these relationships helps in building more efficient predictive models by focusing on the most influential variables.

13.1.1.1. Comparison between the Algorithms based upon Metrics

	MAE	MSE	RMSE	R^2
Linear Regression	572.14	723141.18	850.38	0.90
Polynomial Regression	414.002	439062.221	662.618	0.937
Ridge Regression	572.549	723101.903	850.354	0.895
Lasso Regression	573.2064	725982.5895	852.0461	0.8951
SVR	360	397672.88	630.61	0.94
Decision Tree	411.4654	451255.7369	671.7557	0.9348
Random Forest	215.7772	131873.4050	363.1438	0.9809

Table 13.1.1.1: Comparison Between Algorithms for Used Car Price Prediction

After analyzing the performance of different regression models based on MAE, MSE, RMSE, and R2 Score, the key takeaways are:

- **Best Model: Random Forest Regression**

  Achieves the lowest error values (MAE: 215.77, RMSE: 363.14). Highest R2 Score (0.9809), indicating it explains 98% of the variance in the data. Best suited for high-accuracy predictions.

Example:

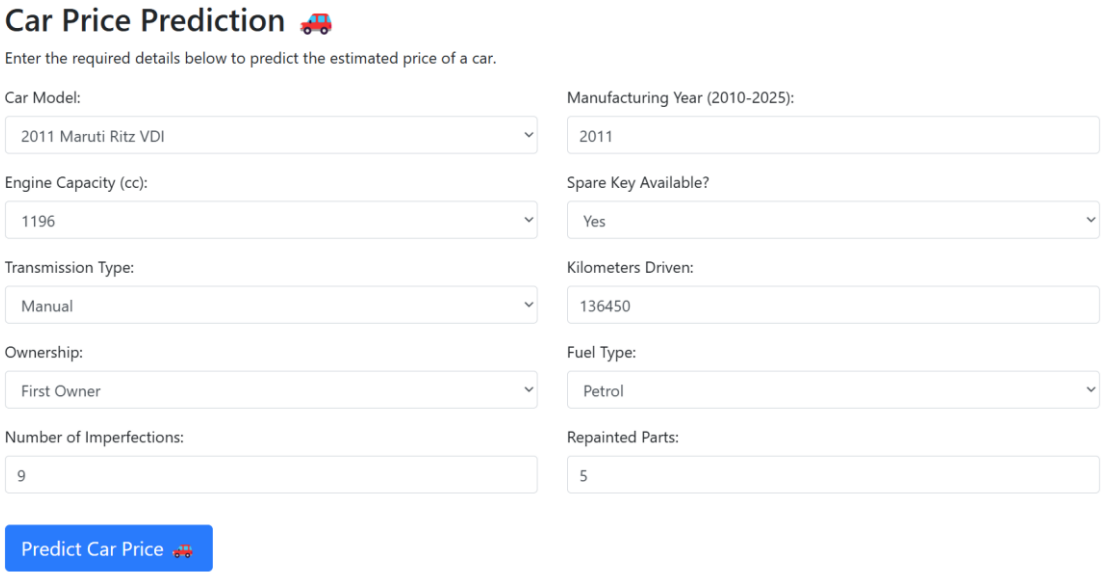

Figure 13.1.1.3: Example of Used Car Price Prediction Model

Output:

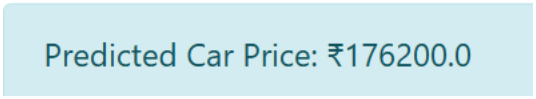

Figure 13.1.1.4: Output of Used Car Price Prediction Model

13.1.1.2. Google Colab Link for Used Car Price Prediction

https://colab.research.google.com/drive/1BhRCpMZsbhLPkBVeSGQa1Ehv76xzYhqu?usp=sharing

13.1.2. Parkinson's Disease Detection

Another significant feature of the website is an AI-based Parkinson's Disease Detection system. Parkinson's Disease is a neurodegenerative disorder that affects movement and speech. The detection model on the site likely utilizes medical datasets containing voice recordings or movement-related metrics to predict whether a person may have Parkinson's. Such AI-driven early detection systems are crucial in the medical field, as they can assist in faster diagnosis and intervention, improving patient outcomes.

The dataset contains detailed patient information, including demographics, lifestyle factors, medical history, clinical measurements, cognitive and functional assessments, symptoms, and diagnosis information, primarily focusing on Parkinson's Disease. Each patient is assigned a unique PatientID, ranging from 3058 to 5162. Demographic details include Age, ranging from 50 to 90 years, Gender (0 for Male, 1 for Female), and Ethnicity, categorized as Caucasian (0), African American (1), Asian (2), or Other (3). The Education Level is recorded as None (0), High School (1), Bachelor's (2), or Higher (3).

Lifestyle factors include BMI, ranging from 15 to 40, Smoking status (0 for No, 1 for Yes), and Alcohol Consumption, measured in weekly units from 0 to 20. Physical Activity is recorded in weekly hours (0 to 10), while Diet Quality and Sleep Quality are scored between 0-10 and 4-10, respectively. Medical history includes a Family History of Parkinson's Disease (0 for No, 1 for Yes), Traumatic Brain Injury history (0 for No, 1 for Yes), and the presence of conditions such as Hypertension, Diabetes, Depression, and Stroke, all recorded as binary indicators (0 for No, 1 for Yes).

The dataset also includes clinical measurements, such as Systolic Blood Pressure (90-180

mmHg) and Diastolic Blood Pressure (60-120 mmHg). Cholesterol levels are divided into Total Cholesterol (150-300 mg/dL), LDL Cholesterol (50-200 mg/dL), HDL Cholesterol (20-100 mg/dL), and Triglycerides (50-400 mg/dL). Cognitive and functional assessments include the UPDRS score (0-199), where higher scores indicate greater disease severity, MoCA score (0-30), where lower scores suggest cognitive impairment, and Functional Assessment (0-10), where lower scores indicate higher impairment.

The symptoms section records the presence of key Parkinson's Disease symptoms, such as Tremor, Rigidity, Bradykinesia, Postural Instability, Speech Problems, Sleep Disorders, and Constipation, all coded as 0 for No and 1 for Yes. The diagnosis information confirms whether a patient has Parkinson's Disease, recorded as 0 for No and 1 for Yes. Lastly, the dataset contains a confidential field, "DoctorInCharge," which holds the masked identifier "DrXXXConfid" for all patients. This dataset is structured to aid in the study of Parkinson's Disease risk factors, symptoms, and diagnosis, supporting both clinical research and predictive modelling.

Figure 13.1.2.1: Histograms of Parkinson's Disease Prediction

Histograms are used to analyze the frequency distribution of continuous and categorical variables, providing insights into data trends, skewness, and potential outliers. The dataset includes demographic details, such as Age, which ranges from 50 to 90 and appears relatively evenly distributed, and Gender, which is represented as a binary variable (0 for Male, 1 for Female). Ethnicity and Education Level histograms show categorical distributions, with some

groups having significantly higher representation than others.

Lifestyle factors include BMI, Smoking status, Alcohol Consumption, Physical Activity, Diet Quality, and Sleep Quality. The BMI distribution appears to be skewed, with more patients having moderate BMI values. Smoking and Family History of Parkinson's Disease are binary features, with histograms indicating a significantly larger number of non-smokers and patients without a family history of the disease. The distributions of weekly alcohol consumption, physical activity, diet quality, and sleep quality show a more spread-out pattern, reflecting varying health habits among patients.

Medical history variables such as Hypertension, Diabetes, Depression, Stroke, and Traumatic Brain Injury are represented as binary variables (0 for No, 1 for Yes), with histograms showing that most patients do not have these conditions. Clinical measurements, including Systolic and Diastolic Blood Pressure, Total Cholesterol, LDL, HDL, and Triglycerides, show more diverse distributions, with some data appearing to follow a normal or uniform distribution.

The cognitive and functional assessments, including UPDRS (Unified Parkinson's Disease Rating Scale), MoCA (Montreal Cognitive Assessment), and Functional Assessment, display varying levels of impairment among patients. UPDRS scores range from 0 to 199, indicating different severities of Parkinson's Disease, while MoCA scores range from 0 to 30, showing different levels of cognitive function.

The symptom histograms, such as Tremor, Rigidity, Bradykinesia, Postural Instability, Speech Problems, Sleep Disorders, and Constipation, reveal that most patients either exhibit or do not exhibit these symptoms, as shown by the high frequency of values at 0 or 1. Finally, the Diagnosis histogram, which indicates whether a patient has Parkinson's Disease (0 for No, 1 for Yes), suggests an imbalanced dataset with a higher count of one category, potentially indicating the need for data balancing in machine learning applications.

Overall, these histograms help in understanding data distributions, identifying imbalances, and selecting important features for predictive modeling in Parkinson's Disease research.

A correlation heatmap visually depicts the relationships between numerical variables using a color gradient, where values close to 1 indicate strong positive correlations, values near -1 indicate strong negative correlations, and values around 0 suggest weak or no correlation.

In this heatmap, the diagonal elements are all equal to 1, as each variable is perfectly correlated with itself. The off-diagonal elements display pairwise correlation coefficients between different features. The color intensity varies, with darker shades representing weaker correlations and lighter shades (closer to red) indicating stronger relationships.

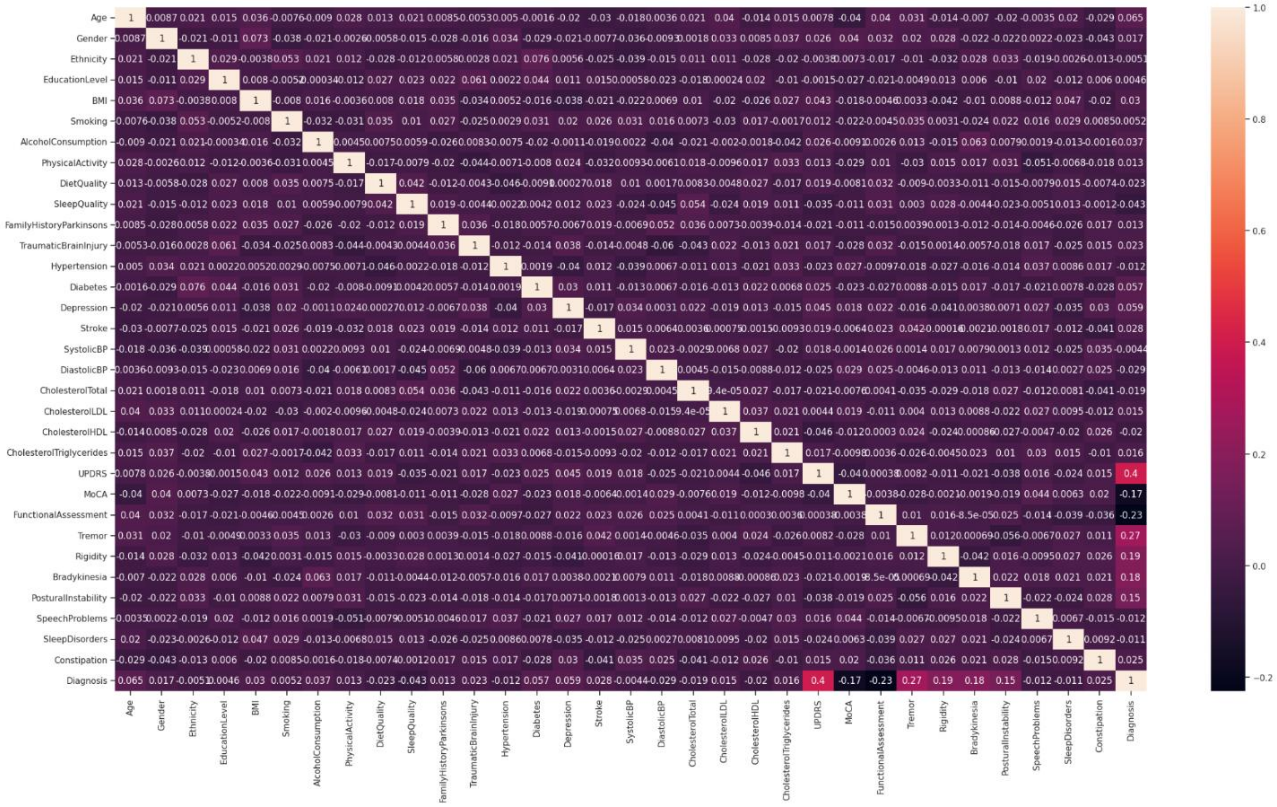

Figure 13.1.2.2: Heatmap for Parkinson's Disease Prediction

- Demographic and lifestyle factors, such as Age, Gender, Ethnicity, BMI, Smoking, Alcohol Consumption, Physical Activity, Diet Quality, and Sleep Quality, generally show weak correlations with each other and with medical history or clinical measurements.
- Medical history variables, including Hypertension, Diabetes, Depression, Stroke, and Traumatic Brain Injury, have some weak to moderate correlations among themselves.
- Clinical measurements, such as Systolic Blood Pressure, Diastolic Blood Pressure, Cholesterol levels (Total, LDL, HDL, Triglycerides), and UPDRS scores, show varying degrees of interdependence.
- The UPDRS score, a key indicator of Parkinson's Disease severity, shows a positive correlation with Diagnosis ($r \approx 0.4$), meaning higher UPDRS values are associated with Parkinson's Disease.
- The MoCA score, which assesses cognitive function, exhibits a negative correlation with Diagnosis ($r \approx -0.17$), suggesting that lower cognitive scores are linked to Parkinson's Disease.
- Motor symptoms, including Tremor, Rigidity, Bradykinesia, and Postural Instability, show positive correlations with Diagnosis, implying that these symptoms are more common in Parkinson's patients.

- Functional Assessment has a negative correlation with Diagnosis ($r \approx -0.23$), indicating that lower functional ability is associated with Parkinson's Disease.
- Some clinical features such as cholesterol levels and blood pressure appear to have little to no correlation with Parkinson's Disease diagnosis.

This heatmap provides useful insights for feature selection and exploratory data analysis in Parkinson's Disease research. It helps in identifying key relationships that can be leveraged for predictive modeling and medical diagnosis.

13.1.2.1. Comparison between Algorithms Based Upon Classification Report

	Precision (0)	Precision (1)	Recall (0)	Recall (1)	F1-Score (0)	F1-Score (1)	Final Accuracy
Logistic Regression	0.717	0.823	0.664	0.857	0.690	0.840	0.789
SVC	0.726	0.844	0.711	0.853	0.719	0.848	0.803
LDA	0.733	0.847	0.718	0.857	0.725	0.852	0.808
GaussianNB	0.687	0.830	0.691	0.827	0.689	0.829	0.779
Decision Tree	0.853	0.923	0.859	0.919	0.856	0.921	0.898
Random Forest	0.920	0.919	0.846	0.960	0.881	0.939	0.919
KNN	0.612	0.767	0.550	0.809	0.580	0.787	0.717
XGBoost	0.929	0.936	0.879	0.963	0.903	0.949	0.933

Table 13.1.2.1: Comparison Between Algorithms for Parkinson's Disease

Precision, recall, F1-score, and accuracy are key performance metrics used to evaluate classification models. Each of these metrics provides a different perspective on model performance, particularly in cases of class imbalance.

From the table provided, XGBoost and Random Forest classifiers perform best across all metrics, with XGBoost achieving the highest accuracy (0.933). Decision Tree also performs well, while simpler models like KNN and Logistic Regression have lower scores. The trade-offs between precision and recall are evident in different models, where some favor minimizing false positives while others focus on capturing more true positives.

Example:

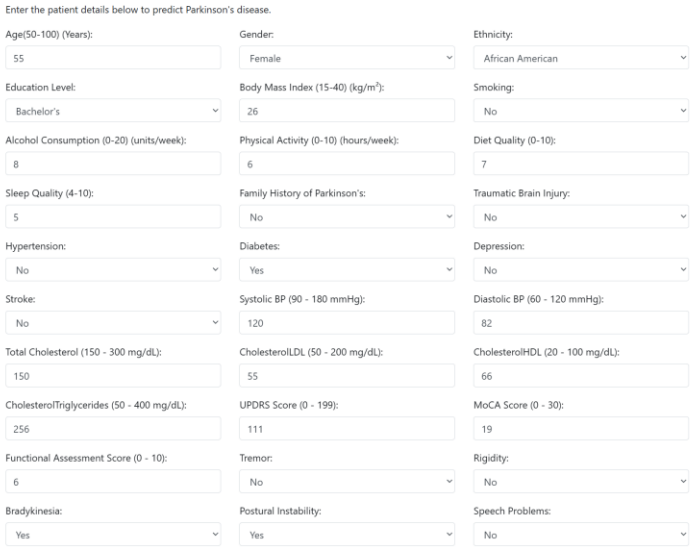

Figure 13.1.2.3: Example of Parkinson's Disease Prediction Model

Output:

Figure 13.1.2.4: Output of Parkinson's Disease Prediction Model

13.1.2.2. PPMI Dataset

The Parkinson's Progression Markers Initiative (PPMI) is a large-scale, longitudinal research study sponsored by the Michael J. Fox Foundation. Its primary objective is to identify biomarkers and progression patterns for Parkinson's Disease (PD) and related disorders through comprehensive, standardized data collection over time. The initiative involves collaboration among global research institutions, clinicians, and data scientists, making it one of the most authoritative and high-quality sources for Parkinson's-related data in the world.

Figure 13.1.2.2.1.: PPMI Logo

In this project, I obtained access to the PPMI dataset through a formal data access request and compliance with its data use agreement. The dataset served as the primary benchmark for evaluating the diagnostic performance of my AI model — Quad-Predict, which is designed to assess the likelihood of a subject having Parkinson's Disease based on clinical and non-clinical variables.

The version of the PPMI dataset utilized for this evaluation included comprehensive clinical records from over 15,000 individual patients. Each patient profile is richly annotated with:

- Demographic details (age, sex, ethnicity)
- Motor and non-motor clinical assessments
- Neuropsychological tests
- Genetic screening results
- Medication and treatment response history

The 15,000 patients in the dataset represent a broad spectrum of 17 neurological categories, carefully curated to encompass both Parkinsonian and non-Parkinsonian conditions. These include:

- Idiopathic Parkinson's Disease (the classical form of PD)
- Atypical Parkinsonian Syndromes such as:
    - Multiple System Atrophy (MSA)
    - Progressive Supranuclear Palsy (PSP)
    - Corticobasal Syndrome (CBS)
- Genetic and juvenile-onset forms of PD
- Secondary parkinsonism, including:
    - Neuroleptic-induced parkinsonism
    - Vascular parkinsonism
- Prodromal PD conditions, where early motor or non-motor symptoms are present but full PD criteria are not yet met
- Control groups and non-PD disorders such as:
    - Alzheimer's Disease
    - Frontotemporal dementia
    - Essential tremor
    - Spinocerebellar ataxia
    - Patients with no neurological disorders

The diversity and granularity of this dataset make it exceptionally valuable for validating clinical AI systems. It ensures that the model is not only trained and tested on classical cases of PD but

is also exposed to edge cases, early-onset variants, and clinically confounding disorders, which often pose diagnostic challenges in real-world neurology.

This makes the PPMI dataset an ideal foundation for a rigorous and clinically relevant evaluation of Quad-Predict's performance. The goal was not only to assess the model's accuracy in identifying PD but also its specificity in avoiding false positives, especially in cases with overlapping symptoms or mimicking features.

13.1.2.3. Parkinson's Disease Probability Estimation on PPMI Patient Data

Patient No.	Disease	Parkinson Disease (Y/N)	Parkinson Disease Probability using Quad-Predict
100001	Idiopathic PD	Y	58%
113043	Alzheimer's disease	N	8%
3476	Corticobasal syndrome	N	7%
151050	Dementia with Lewy bodies	N	43%
100445	Essential tremor	N	15%
149511	Juvenile autosomal recessive parkinsonism	Y	78%
3425	Motor neuron disease with parkinsonism	Y	55%
103914	Multiple system atrophy	Y	60%
40587	Neuroleptic-induced parkinsonism	Y	52%
145939	Progressive supranuclear palsy	Y	49%
3072	Psychogenic parkinsonism	Y	34%
3902	Vascular parkinsonism	Y	88%
100232	No PD nor other neurological disorder	N	6%
225423	Spinocerebellar Ataxia (SCA)	N	23%
10874	Prodromal non-motor PD	Y	54%
12499	Prodromal motor PD	Y	67%
100119	Prodromal Synucleinopathy	Y	45%
109921	Other neurological disorder(s)	N	10%

Table 13.1.2.3.: Parkinson's Disease Probability Estimation on PPMI Patient Data

- Observations:

1. Patient 100001 – Idiopathic Parkinson's Disease

This patient has a classical diagnosis of idiopathic Parkinson's Disease, the most common and well-studied form of PD. The AI model assigned a moderate probability of 58%, which supports the clinical diagnosis. This suggests that the model successfully identified typical PD features such as bradykinesia, rigidity, and possibly tremor, though the confidence score indicates that

some clinical characteristics might have been subtle or underrepresented in the input data.

2. Patient 113043 – Alzheimer's Disease

This patient was diagnosed with Alzheimer's disease, a neurodegenerative disorder that primarily affects memory and cognitive function. The AI model correctly assigned a very low probability of 8%, indicating high specificity in distinguishing Alzheimer's from Parkinson's Disease, despite possible minor motor abnormalities that occasionally appear in dementia patients.

3. Patient 3476 – Corticobasal Syndrome (CBS)

Diagnosed with CBS, this patient has a condition that can mimic Parkinsonism with symptoms like rigidity and dystonia. However, the AI model gave a low score of 7%, demonstrating its ability to differentiate atypical parkinsonian syndromes from idiopathic PD by likely recognizing asymmetry, apraxia, and cortical features uncommon in PD.

4. Patient 151050 – Dementia with Lewy Bodies (DLB)

Though DLB shares pathological features with PD, including alpha-synuclein deposition, this patient's diagnosis is not idiopathic PD. The AI assigned a probability of 43%, reflecting this overlap. The score suggests the model detected motor symptoms or REM sleep behavior disorder that are common in both conditions, but it correctly did not classify it with high PD certainty.

5. Patient 100445 – Essential Tremor

This patient's diagnosis was essential tremor, a non-Parkinsonian movement disorder. The AI assigned a low probability of 15%, appropriately differentiating it from PD. The slight elevation may result from tremor similarities, though essential tremor typically lacks bradykinesia and rigidity, which helps the model avoid misclassification.

6. Patient 149511 – Juvenile Autosomal Recessive Parkinsonism

This patient was diagnosed with an early-onset genetic form of Parkinsonism. The AI predicted a high probability of 78%, showing strong agreement with the diagnosis. The model was able to generalize its understanding of PD features to an age group typically underrepresented in idiopathic PD studies, showing robust recognition of genetic parkinsonism traits.

7. Patient 3425 – Motor Neuron Disease with Parkinsonism

This patient had overlapping features of PD and motor neuron disease. The AI prediction of 55% suggests moderate detection of parkinsonian traits, though the presence of upper motor neuron

signs likely reduced certainty. This highlights the model's ability to recognize complex clinical pictures while avoiding overclassification.

8. Patient 103914 – Multiple System Atrophy (MSA)

Diagnosed with MSA, this patient likely presented with autonomic dysfunction and parkinsonian symptoms. The AI assigned a 60% probability, indicating recognition of parkinsonian motor patterns despite differences in treatment response and progression when compared to PD. This suggests the model's sensitivity to motor signs common to both PD and MSA.

9. Patient 40587 – Neuroleptic-Induced Parkinsonism

This patient's parkinsonian symptoms were drug-induced. The AI's score of 52% shows it detected PD-like motor features, though the absence of progressive neurodegeneration likely limited the probability. This moderate score demonstrates the model's ability to flag symptom presence while remaining cautious about etiology.

10. Patient 145939 – Progressive Supranuclear Palsy (PSP)

With a diagnosis of PSP, this patient likely exhibited vertical gaze palsy and rapid progression, which can initially resemble PD. The AI's 49% score reflects this overlap. It's consistent with diagnostic uncertainty that exists even among neurologists, reinforcing the model's appropriate caution.

11. Patient 3072 – Psychogenic Parkinsonism

This patient's symptoms stemmed from a functional (non-organic) neurological disorder. The AI gave a relatively low score of 34%, reflecting some detection of motor symptoms but also recognition of atypical patterns inconsistent with neurodegeneration. This shows that the model can identify inconsistencies seen in psychogenic movement disorders.

12. Patient 3902 – Vascular Parkinsonism

This patient was diagnosed with vascular parkinsonism, a secondary form of PD resulting from cerebrovascular disease. The AI assigned a high probability of 88%, strongly identifying parkinsonian symptoms like gait freezing and rigidity. While not idiopathic PD, the model responded to prominent motor features, which could merit additional subclassification in future versions.

13. Patient 100232 – No Neurological Disorder

This patient served as a control and had no clinical evidence of PD or any neurological disease. The AI correctly assigned a very low probability of 6%, demonstrating excellent specificity and its ability to confidently exclude non-PD individuals from being misclassified.

14. Patient 225423 – Spinocerebellar Ataxia (SCA)

Diagnosed with SCA, this patient likely presented with coordination issues and gait disturbances. The AI assigned a 23% probability, which is slightly elevated due to overlapping motor symptoms. However, the score remained low enough to differentiate SCA from PD, showing good but improvable phenotypic distinction.

15. Patient 10874 – Prodromal Non-Motor PD

This patient was in a prodromal stage of PD, exhibiting non-motor symptoms such as constipation, hyposmia, or REM sleep behavior disorder. The AI assigned a moderate score of 54%, indicating early detection capability and sensitivity to pre-motor features that may precede classical PD symptoms.

16. Patient 12499 – Prodromal Motor PD

This patient had early-stage motor symptoms consistent with developing PD. The AI model gave a 67% probability, showing strong confidence in the presence of PD even at a subtle, preclinical stage. This highlights the model's strength in detecting early bradykinesia and movement asymmetries.

17. Patient 100119 – Prodromal Synucleinopathy

This patient had early signs of a synucleinopathy (such as DLB, MSA, or PD). The model gave a score of 45%, reflecting moderate uncertainty — which is reasonable given the transitional nature of such disorders where PD symptoms may emerge over time. This output shows the model's nuanced sensitivity to the biological continuum of neurodegeneration.

18. Patient 109921 – Other Neurological Disorder(s)

The final patient in this subset had a non-specific neurological condition unrelated to PD. The AI assigned a very low score of 10%, supporting its ability to recognize and exclude non-PD disorders even in the presence of potential overlapping symptoms. This reflects strong generalization and accurate negative prediction.

13.1.2.4. Google Colab Link for Parkinson's Disease Dataset:
https://colab.research.google.com/drive/1FCKNtlzAQ2lMy7iW4tGhAuyE5fAij?usp=sharing

Parkinson's

13.1.3. Diabetes Monitoring and Prediction

The platform also offers a diabetes monitoring tool that helps predict blood glucose levels based on user inputs. Diabetes is a chronic disease that requires continuous monitoring to avoid complications. The AI model behind this feature analyzes various health parameters such as past glucose levels, insulin intake, and dietary habits to provide an estimate of future glucose levels. This tool can be beneficial for diabetic patients looking to maintain stable blood sugar levels and avoid unexpected spikes or drops.

Age is an important factor in predicting diabetes risk, as the likelihood of developing diabetes increases with age due to reduced physical activity, hormonal changes, and a higher chance of developing related health conditions. Gender also plays a role, with women who have had gestational diabetes facing a higher risk of type 2 diabetes, while some studies suggest that men may have a slightly higher overall risk. Body Mass Index (BMI) is another key indicator, as higher BMI levels, especially excess fat around the waist, contribute to insulin resistance and impaired blood sugar regulation. Hypertension, or high blood pressure, is closely linked to diabetes, with both conditions sharing common risk factors and exacerbating each other's effects on cardiovascular health. Similarly, heart disease, including coronary artery disease and heart failure, is associated with an increased risk of diabetes due to overlapping risk factors such as obesity, hypertension, and high cholesterol. Smoking history is another modifiable risk factor, as smoking has been found to increase insulin resistance and impair glucose metabolism, making quitting smoking an essential step in reducing diabetes risk. HbA1c level, which reflects average blood glucose over the past 2-3 months, is a crucial measure of long-term blood sugar control, with higher levels indicating poorer glycemic control and an increased risk of diabetes and its complications. Blood glucose level, which measures the amount of glucose in the blood at a given time, is a key indicator of glucose regulation, and persistently elevated levels can signal a heightened risk of developing diabetes. Regular monitoring and management of these factors are

essential for diabetes prevention and overall health.

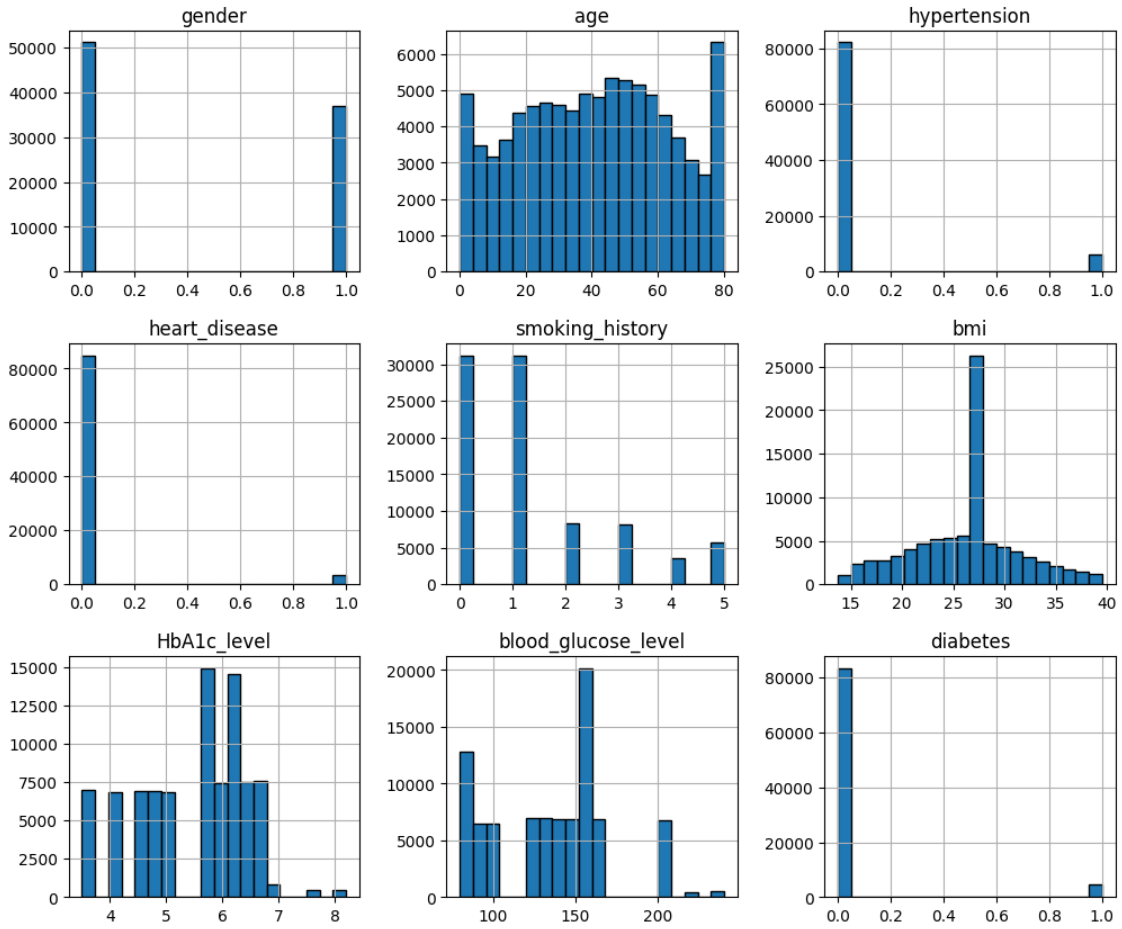

Figure 13.1.3.1: Histograms for Diabetes Prediction Dataset

The image contains histograms representing various features of a heart disease dataset. Each histogram visually displays the distribution of values for a particular feature.

- **Gender**: This appears to be a binary variable, with two distinct bars indicating the number of male and female participants in the dataset.
- **Age**: The age distribution is roughly uniform, with a higher frequency of individuals in the middle age range (20–60 years).
- **Hypertension**: Most individuals do not have hypertension, as shown by a dominant bar at 0 (absence of hypertension).
- **Heart Disease**: The majority of people in the dataset do not have heart disease, indicated by a large number of observations at 0.
- **Smoking History**: The histogram suggests multiple categories, with a significant number of non-smokers and varying distributions for different smoking statuses.
- **BMI (Body Mass Index)**: The BMI values follow a normal distribution, with most individuals having a BMI between 20 and 35.

- **HbA1c Level**: The HbA1c levels show multiple peaks, indicating varying blood sugar control among individuals.
- **Blood Glucose Level**: The glucose levels show a right-skewed distribution, with some individuals having significantly high values.
- **Diabetes**: The dataset has far more individuals without diabetes, as seen by the large concentration of values at 0.

These histograms help in understanding the spread and characteristics of the data, which can be useful for analyzing health trends and predicting heart disease risks.

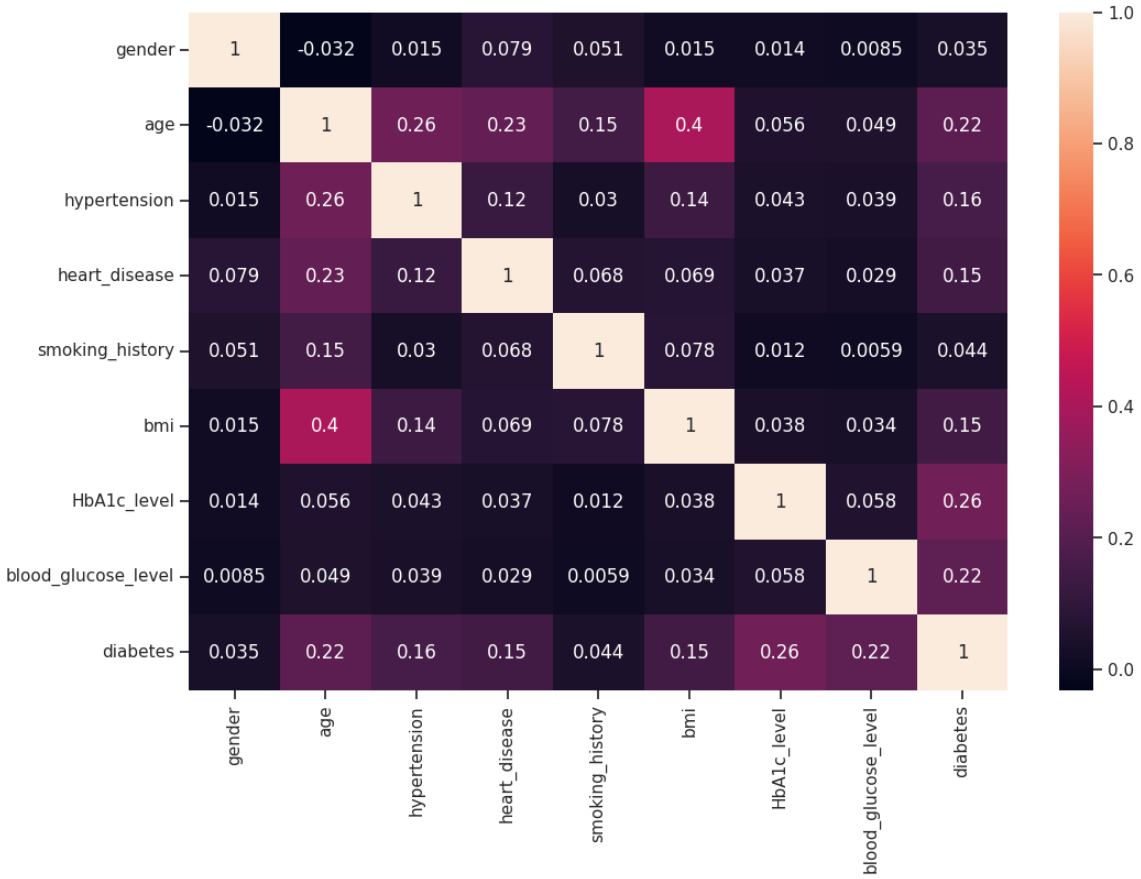

Figure 13.1.3.2: Heatmap for Diabetes Prediction Dataset

The heatmap displays the correlation coefficients between different variables. Correlation measures the extent to which two variables change together. A positive correlation (values closer to 1) indicates that as one variable increases, the other tends to increase as well. A negative correlation (values closer to -1) suggests that as one variable increases, the other tends to decrease. A correlation close to 0 implies a weak or no linear relationship between the variables. In this specific heatmap, the diagonal values are all 1, as each variable is perfectly correlated with itself. The off-diagonal values show the correlations between pairs of different variables. For example, age and bmi have a correlation of 0.4, indicating a moderate positive relationship, while gender and age have a correlation of -0.032, suggesting a very weak negative relationship.

13.1.3.1. Comparison between Algorithms Based Upon Classification Report

Metrics	Class	Logistic Regression	Support Vector Classifier	Linear Discriminant Analysis	GaussianNB	Decision Tree Classifier	Random forest Classifier	KNeighbours Classifiers	Gradient Boosting
Precision	0 (No Diabetes)	0.96	0.96	0.96	0.97	0.98	0.97	0.96	0.95
Precision	1 (No Diabetes)	0.58	0.88	0.64	0.26	0.52	0.92	0.71	1.00
Recall	0 (No Diabetes)	0.99	1.00	0.99	0.93	0.97	1.00	0.99	1.00
Recall	1 (No Diabetes)	0.16	0.27	0.31	0.42	0.57	0.48	0.28	0.00
F1-Score	0 (No Diabetes)	0.97	0.98	0.98	09.5	0.97	0.98	0.98	0.97
F1-Score	1 (No Diabetes)	0.26	0.41	0.42	0.32	0.54	0.63	0.40	0.00
Macro Avg.	All Classes	0.77/0.58/0.61	0.92/0.63/0.70	0.80/0.65/0.70	0.61/0.68/0.64	0.75/0.77/0.76	0.94/0.74/0.81	0.83/0.64/0.69	0.97/0.50/0.49
Weighted Avg.	All Classes	0.94/0.95/0.94	0.96/0.96/0.95	0.95/0.95/0.95	0.93/0.91/0.92	0.95/0.95/0.95	0.97/0.97/0.97	0.95/0.96/0.95	0.95/0.95/0.92
Accuracy	-	0.95	0.96	0.95	0.91	0.95	0.97	0.96	0.95

Table 13.1.3.1: Comparison Between Algorithms for Diabetes Prediction

- Random Forest Classifier has the highest accuracy (97%), with strong precision, recall, and F1-score.
- Gradient Boosting achieves 100% precision for class 1 but suffers from 0% recall, meaning it fails to detect actual diabetes cases.
- K-Nearest Neighbors (KNN) performs well overall, but its recall for diabetes cases (1) is only 28%, making it unreliable for detecting diabetes.
- Logistic Regression & Support Vector Classifier (SVC) both have high accuracy (~95-96%), but their recall for detecting diabetes cases is relatively low.

Best Algorithm Choice:

- Random Forest Classifier is the best model, with the highest accuracy (97%), balanced precision, recall, and F1-score.
- Support Vector Classifier (SVC) and Logistic Regression can be considered alternatives, but their recall on diabetes cases is lower.
- Gradient Boosting is unreliable despite high accuracy, as it completely fails to identify diabetes cases (recall = 0%).
- KNN struggles with recall, meaning it misclassifies diabetes cases too often.

Example:

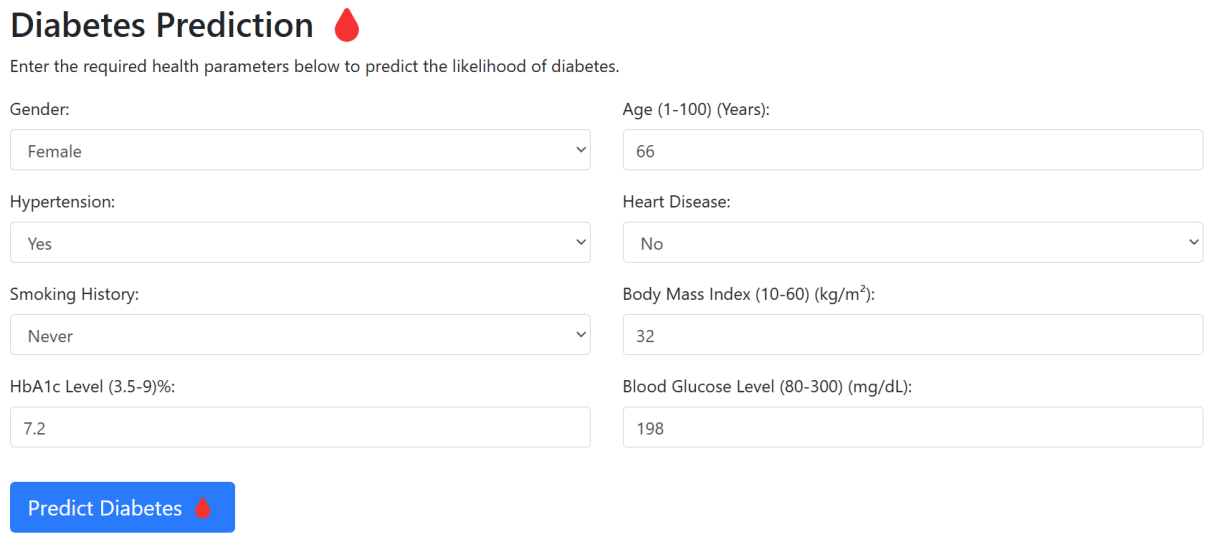

Figure 13.1.3.3: Diabetes Prediction Model

Output:

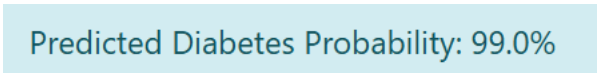

Figure 13.1.3.4: Result of Diabetes Prediction Model

13.1.3.2. Google Colab Link for Diabetes Dataset

https://colab.research.google.com/drive/1N9egmh2nJEe__2_Vym1Mz7HiK3fUk5Qy?usp=sharing

13.1.4. Diamond Price Prediction

Another unique offering on the site is a diamond price prediction tool. Diamonds are valued based on multiple factors such as carat, cut, clarity, and color. The AI model in this section helps estimate the price of diamonds based on these attributes. Such a tool can be useful for both buyers and sellers in the jewellery market, providing insights into price trends and fair market value.

This dataset contains information on nearly 54,000 diamonds, including their prices and various attributes. The key attribute, carat, represents the diamond's weight, measured in metric carats, with one carat equalling 0.20 grams. The cut of the diamond, ranging from Fair to Ideal,

determines its brilliance, with better cuts reflecting more light. The color is graded from J (worst) to D (best), where colorless diamonds are the rarest and most valuable. Clarity describes the presence of inclusions or blemishes, graded from I1 (worst) to IF (best), with flawless diamonds being extremely rare. The depth of a diamond, expressed as a percentage, is the ratio of its total height to its average diameter, while the table measures the width of the diamond's top surface relative to its widest point, affecting its brilliance. The price, which is the target variable in the dataset, ranges from $326 to $18,826. Additionally, the dataset includes the x, y, and z dimensions, representing the diamond's length, width, and depth in millimetres. These attributes collectively help in evaluating and pricing diamonds based on their physical and optical characteristics.

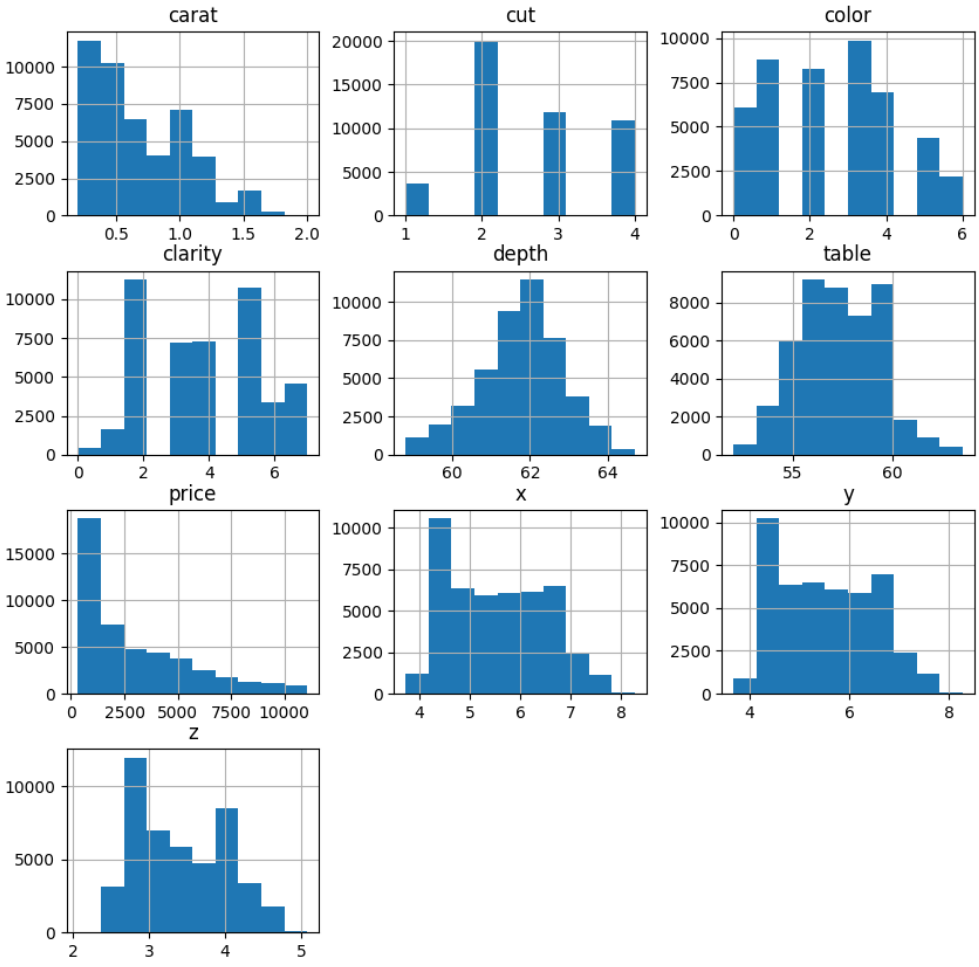

Figure 13.1.4.1: Histograms for Diamond Price Prediction Dataset

Each histogram visualizes the frequency distribution of a specific feature, providing insights into the data's characteristics. For instance, the 'carat' histogram shows that most diamonds have a carat weight between 0 and 1, with the frequency decreasing as the carat weight increases. The 'cut' histogram reveals that 'Ideal' (represented by a smaller numerical value) is the most frequent cut grade. Similarly, the 'color' histogram indicates a higher frequency for certain color grades.

The histograms for 'depth', 'table', 'x', 'y', and 'z' show the distribution of diamond dimensions. The 'price' histogram is skewed to the right, suggesting that most diamonds fall into a lower price range, with fewer diamonds at higher prices. Examining these histograms helps understand the distribution and range of values for each feature in the dataset.

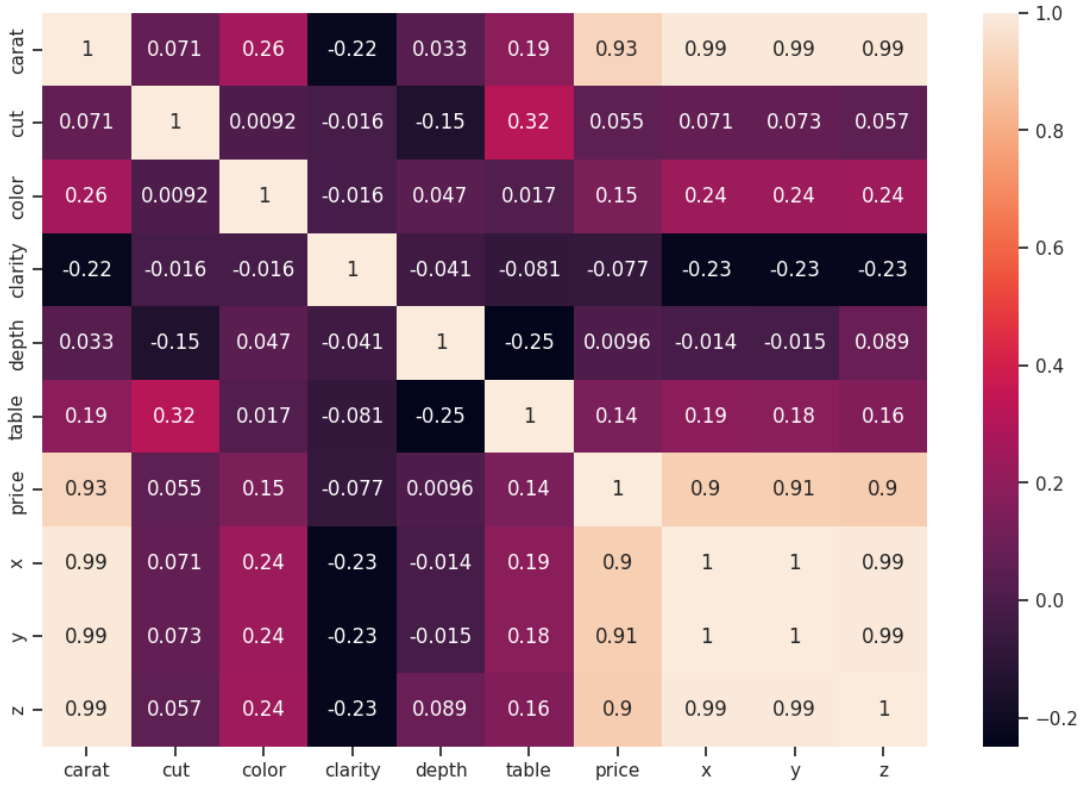

Figure 13.1.4.2: Heatmap for Diamond Price Prediction Dataset

The heatmap displays the correlation coefficients between different features of a diamond dataset. Correlation measures the linear relationship between two variables, ranging from -1 (perfect negative correlation) to +1 (perfect positive correlation),
with 0 indicating no linear correlation. The diagonal elements are all 1, as each feature is perfectly correlated with itself. Off-diagonal elements show the correlation between pairs of features. For instance, 'carat' and 'price' have a strong positive correlation of 0.93, indicating that as carat weight increases, the price tends to increase significantly. Conversely, 'clarity' and 'price' have a negative correlation of -0.077, suggesting a weak negative relationship, meaning higher clarity might slightly decrease the price, though this relationship is not very strong. 'Cut' and 'price' also show a positive correlation (0.055), implying that better cuts are associated with slightly higher prices. The correlations between dimensions 'x', 'y', 'z', and 'carat' are very high (around 0.99), which is expected as these dimensions are directly related to the diamond's size, which is also reflected in its carat weight. Understanding these correlations can be useful in predicting diamond prices or identifying features that strongly influence each other.

13.1.4.1. Comparison between the Algorithms based upon Metrics

	MAE	MSE	RMSE	R^2
Linear Regression	572.14	723141.18	850.38	0.90
Polynomial Regression	414.002	439062.221	662.618	0.937
Ridge Regression	572.549	723101.903	850.354	0.895
Lasso Regression	573.2064	725982.5895	852.0461	0.8951
SVR	360	397672.88	630.61	0.94
Decision Tree	411.4654	451255.7369	671.7557	0.9348
Random Forest	215.7772	131873.4050	363.1438	0.9809

Table 13.1.4.1: Comparison Between Algorithms for Diamond Price Prediction

After analyzing the performance of different regression models based on MAE, MSE, RMSE, and R2 Score, the key takeaways are:

- Best Model: **Random Forest Regression**

Achieves the lowest error values (MAE: 215.77, RMSE: 363.14). Highest R2 Score (0.9809), indicating it explains 98% of the variance in the data. Best suited for high-accuracy predictions.

Example:

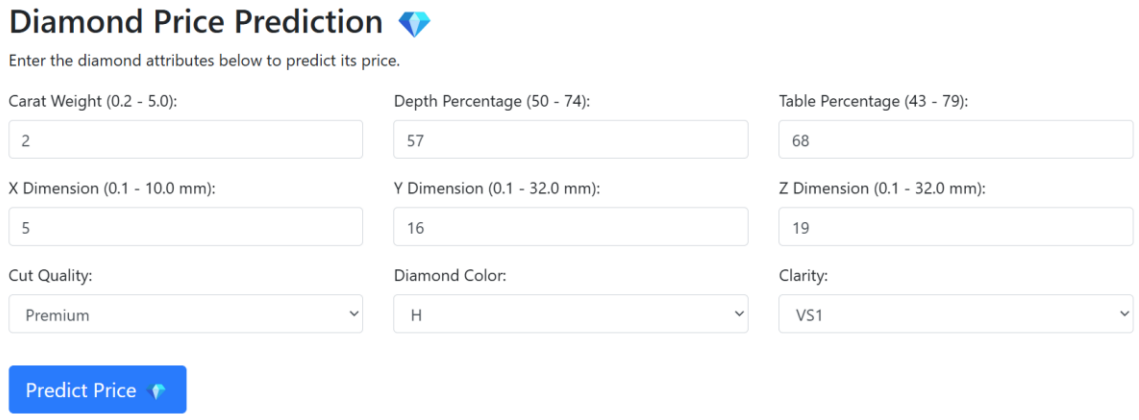

Figure 13.1.4.3: Diamond Price Prediction Model

Output:

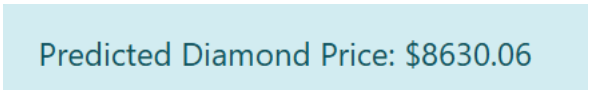

Figure 13.1.4.4: Result of Diamond Price Prediction Model

13.1.4.2. Google Colab Link for Diamond Price Prediction Dataset:

https://colab.research.google.com/drive/185IyhuaC23D8UqVXUuB1qlbHCCcuN0L?usp=sharing

Diamond

13.1.5. Datasets for Analysis

The website also provides access to relevant datasets used in these AI models. These datasets can be valuable for students, researchers, and data scientists who want to experiment with machine learning models or improve existing predictions. By offering datasets, the platform encourages learning and exploration in the field of artificial intelligence.

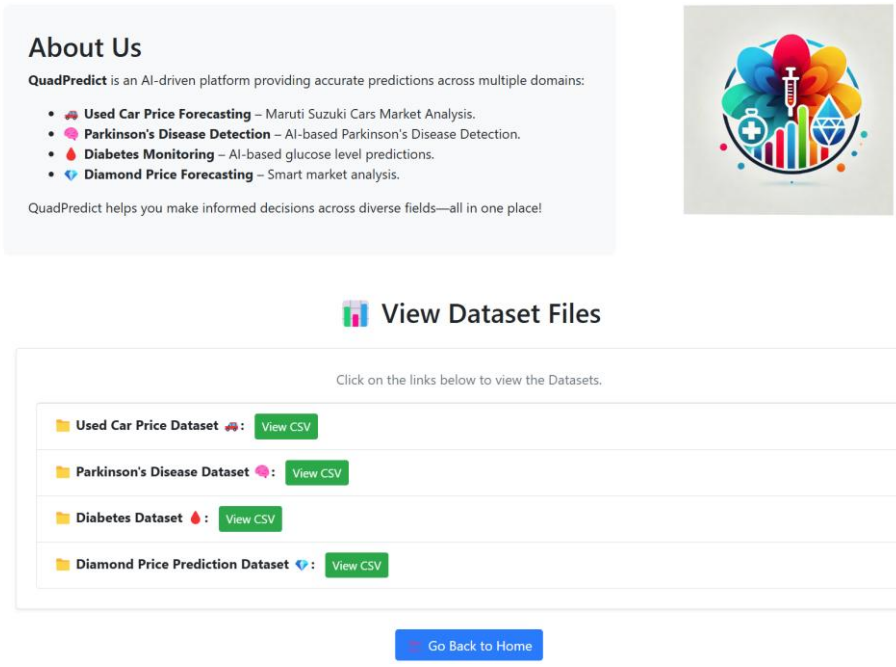

Figure 13.1.5: Dataset of Prediction Models

13.1.6. QuadPredict Website Link

https://prabhsimran-project.glitch.me/

QuadPredict

13.2. Handwritten Digit Recognition

Handwriting recognition is a crucial application of artificial intelligence (AI) and machine learning (ML). It enables computers to interpret and process handwritten text, facilitating automation in numerous fields such as banking, healthcare, and education. The MNIST (Modified National Institute of Standards and Technology) dataset is a widely used benchmark dataset for training and evaluating handwriting recognition models.

The MNIST dataset comprises 70,000 grayscale images of handwritten digits (0–9), each with a resolution of 28x28 pixels. It is structured as follows:

- Training Set: 60,000 images
- Test Set: 10,000 images

Each image is labelled with its corresponding digit, making it ideal for supervised learning tasks. Due to its standardized format, MNIST serves as an excellent dataset for benchmarking image classification models.

Figure 13.2.1: MNIST Dataset

- **Steps:**
  1. **Load the data**
     - The **training** dataset consists of 60000 28x28px images of hand-written digits from 0 to 9.
     - The **test** dataset consists of 10000 28x28px images.

```
[63] mnist_dataset = tf.keras.datasets.mnist
     (x_train, y_train), (x_test, y_test) = mnist_dataset.load_data()
```

```
[64] print('x_train:', x_train.shape)
     print('y_train:', y_train.shape)
     print('x_test:', x_test.shape)
     print('y_test:', y_test.shape)
```

```
x_train: (60000, 28, 28)
y_train: (60000,)
x_test: (10000, 28, 28)
y_test: (10000,)
```

Figure 13.2.2: Loading the Data

  2. **Data Preprocessing**

     Data preprocessing is essential for improving the performance and accuracy of machine learning models. The key steps include:

     - **Normalization**: Scaling pixel values to the range [0,1] by dividing by 255 to enhance numerical stability.
     - **Reshaping**: Adjusting image dimensions if required for compatibility with various ML models.
     - **Data Augmentation**: Applying transformations such as rotation, zoom, and translation to improve generalization and model robustness.

  3. **Building the Model**

- We will use Sequential Keras model.
- Then we will have two pairs of Convolution2D and MaxPooling2D layers. The MaxPooling layer acts as a sort of downsampling using max values in a region instead of averaging.
- After that we will use Flatten layer to convert multidimensional parameters to vector.
- The las layer will be a Dense layer with 10 Softmax outputs. The output represents the network guess. The 0-th output represents a probability that the

input digit is 0, the 1-st output represents a probability that the input digit is 1 and so on...

4. Compiling the Model

```
[78] adam_optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)

     model.compile(
         optimizer=adam_optimizer,
         loss=tf.keras.losses.sparse_categorical_crossentropy,
         metrics=['accuracy']
     )
```

Figure 13.2.3: Compiling the Model

5. Training the Model

```
[79] log_dir=".logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
     tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)

     training_history = model.fit(
         x_train_normalized,
         y_train,
         epochs=10,
         validation_data=(x_test_normalized, y_test),
         callbacks=[tensorboard_callback]
     )
```

Figure 13.2.4: Training the Model

Example:

Figure 13.2.5: Input to the Model

```
[96] loaded_model = load_model("/content/digits_recognition_cnn.h5")
WARNING:absl:Compiled the loaded model, but the compiled metrics have yet to be built. `model

[97] img1  = Image.open("5.png").convert("L")

[38] data = np.asarray( img1 )

[98] data
ndarray (28, 28) show data
5

[99] gray1 = 255-data

[100] print(gray1.shape)
(28, 28)

[101] # reshape
      gray1_with_channels = gray1.reshape(    1, 28,28, 1 )

[102] res = loaded_model.predict([gray1_with_channels])
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 108ms/step

[103] print ( res)
[[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]]

[104] print ( "Predicted result : ", res.argmax() ,  " Probability :", res.max()*100 ,"%" )
Predicted result :  5  Probability : 100.0 %
```

Figure 13.2.6: Result of the Model Prediction

13.2.1. Google Colab Link for Handwritten Digit Recognition

https://colab.research.google.com/drive/1MKahyrUW_gegSLbJuRy1MjVBuOzaZQH9?usp=sharing

Digit Recognition

13.3. Face Detection using Gradio

Face detection using Gradio and Haar cascade classifier combines the power of classical computer vision with an easy-to-use graphical interface, making it possible to deploy and interact with face detection models directly in a web browser. The goal of face detection is to locate and highlight human faces within an image. The Haar cascade classifier, provided by OpenCV, is a machine learning-based approach where a cascade function is trained from many positive and negative images to detect objects—in this case, faces.

The Haar cascade face detection process works by scanning the image with multiple rectangular features (like edges, lines, and changes in brightness), each resembling parts of a human face (e.g., eyes, nose bridge, mouth). These features are applied to a grayscale image through a sliding window approach. If enough features match in a given region, the classifier determines that a face is present and returns its bounding box coordinates. Haar cascades are very fast and were among the earliest techniques used for real-time face detection, including in webcams and surveillance systems.

When integrated with Gradio, the process becomes interactive and user-friendly. Gradio provides a simple Python interface where users can upload an image through the browser, and the backend processes it using OpenCV's Haar classifier. The result is then displayed with rectangles drawn around detected faces. The Gradio interface can be built with just a few lines of code using gr.Interface(), where the face detection function is linked to image input and output components. This setup allows even non-programmers to experiment with face detection models, test different images, and understand the functionality visually.

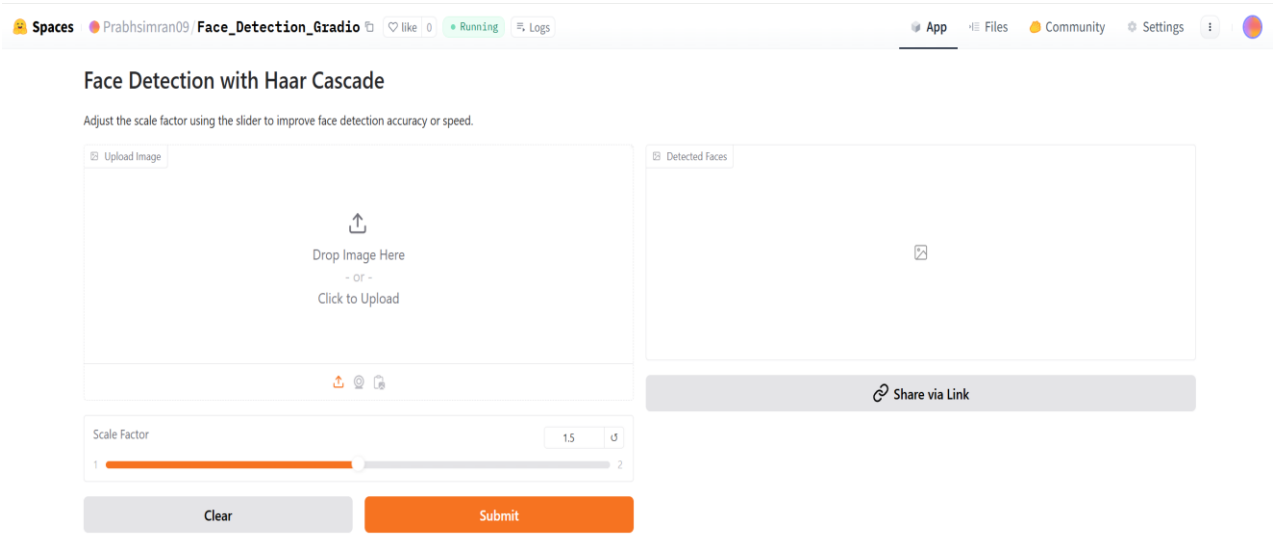

Figure 13.3.1: Face Detection Detection App Deployed on Hugging Face

This application allows users to upload an image and automatically detects faces within it using OpenCV's Haar cascade classifier. The interface is user-friendly and interactive, providing two main sections: the left side for image input and parameter control, and the right side for displaying the output with detected faces.

The app utilizes a Haar cascade classifier, which is a traditional object detection algorithm based on machine learning. It works by identifying facial features through trained classifiers using edge and line features known as Haar-like features. These features are evaluated across the image using a sliding window, and when enough features match the trained face pattern, a face is detected. This method is fast and suitable for real-time applications, especially when computational resources are limited.

What makes this app unique is its integration with Gradio, which provides a graphical interface to the face detection pipeline. Users can upload images directly, adjust a scale factor slider to fine-tune the detection sensitivity, and instantly view the output image with rectangles drawn around detected faces. The scale factor controls how much the image size is reduced at each image scale in the detection process—smaller values mean higher accuracy but slower performance, while larger values improve speed at the cost of potential accuracy.

The Submit button triggers the detection process, while the Clear button resets the interface, and the app also includes an option to share the interface via a link, making it accessible to others without any installation. Deployed on Hugging Face Spaces, this app is cloud-hosted and publicly accessible, showcasing how classical computer vision techniques can be combined with modern interactive tools to build educational and practical web applications. It is ideal for students, developers, and researchers who want to explore face detection interactively and understand how

parameter tuning affects the outcome.

The image below in displays the output of a Gradio-based face detection web app titled “Face Detection with Haar Cascade”, hosted on Hugging Face Spaces. This application enables users to upload an image—such as a group photo—and automatically detects human faces using OpenCV’s classical Haar Cascade Classifier. The app then visually marks each detected face with a green rectangle, as seen on the right side of the interface, indicating successful face localization within the image.

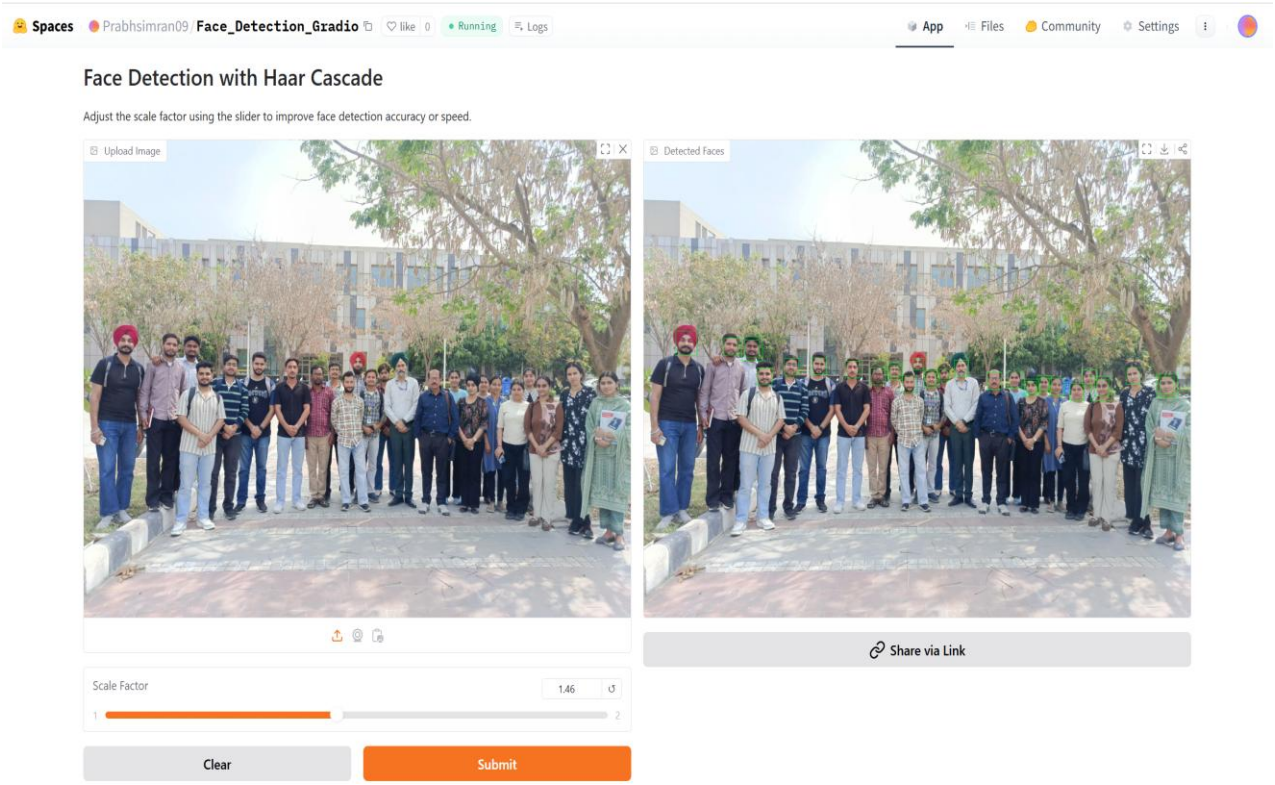

Figure 13.3.2: Face Detection App Demo

Gradio allows users to easily interact with the model by uploading an image and adjusting the scale factor slider. This slider controls the scale at which the image is reduced during the detection process. A lower scale factor improves detection accuracy by scanning the image more thoroughly, while a higher scale factor speeds up processing but may miss smaller or partially obscured faces. In this instance, a scale factor of 1.46 is applied, balancing both speed and accuracy to detect multiple faces in a crowded scene.

The clear separation of input (left) and output (right) in the UI, along with an intuitive Submit button and image preview, makes this app highly user-friendly. It demonstrates the practical power of combining traditional computer vision techniques (like Haar cascades) with modern deployment tools (Gradio + Hugging Face Spaces).

13.3.1. Hugging Face Link for Face Detection using Gradio

https://huggingface.co/spaces/Prabhsimran09/Face_Detection_Gradio

Face Detection

13.4. Fastfood Classifier using Streamlit

The project “Fast Food Classifier” is a web-based machine learning application built using Streamlit and deployed on Hugging Face Spaces. The goal of the application is to allow users to upload an image of a fast-food item—such as a burger, pizza, cupcake, fries, or waffle—and receive a prediction about what type of food it is. This prediction is made using a trained image classification model, typically based on deep learning architectures like CNNs (Convolutional Neural Networks) or transfer learning models such as MobileNet, ResNet, or VGG.

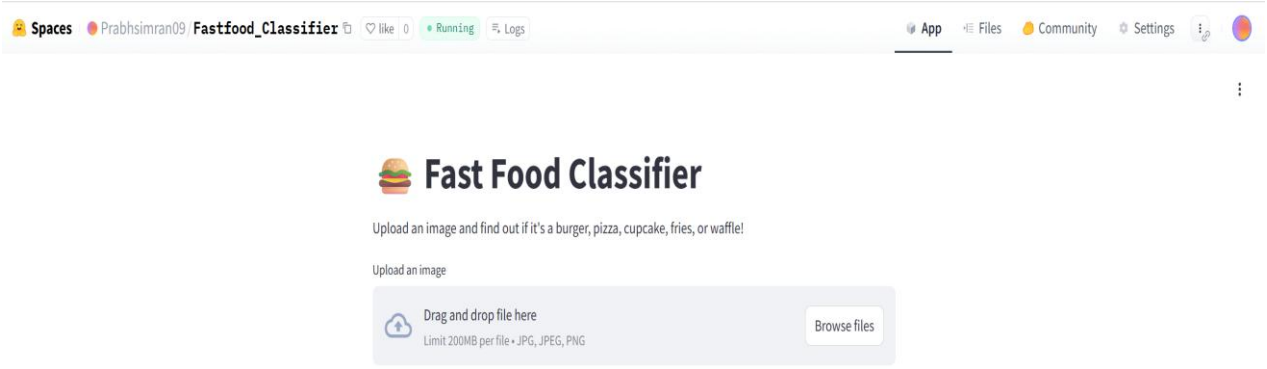

Figure 13.4.1: Fast Food Classifier App Deployed on Hugging Face

Streamlit provides the interface for this app, making it extremely simple and intuitive for users to interact with. Upon visiting the app, users are presented with a file uploader where they can drag and drop or browse an image file (JPG, PNG, etc.). Once an image is uploaded, the app displays it and uses the backend model to classify it into one of the predefined categories. The result, usually shown as the predicted label along with the confidence score, is displayed in real time below the image or as a success message.

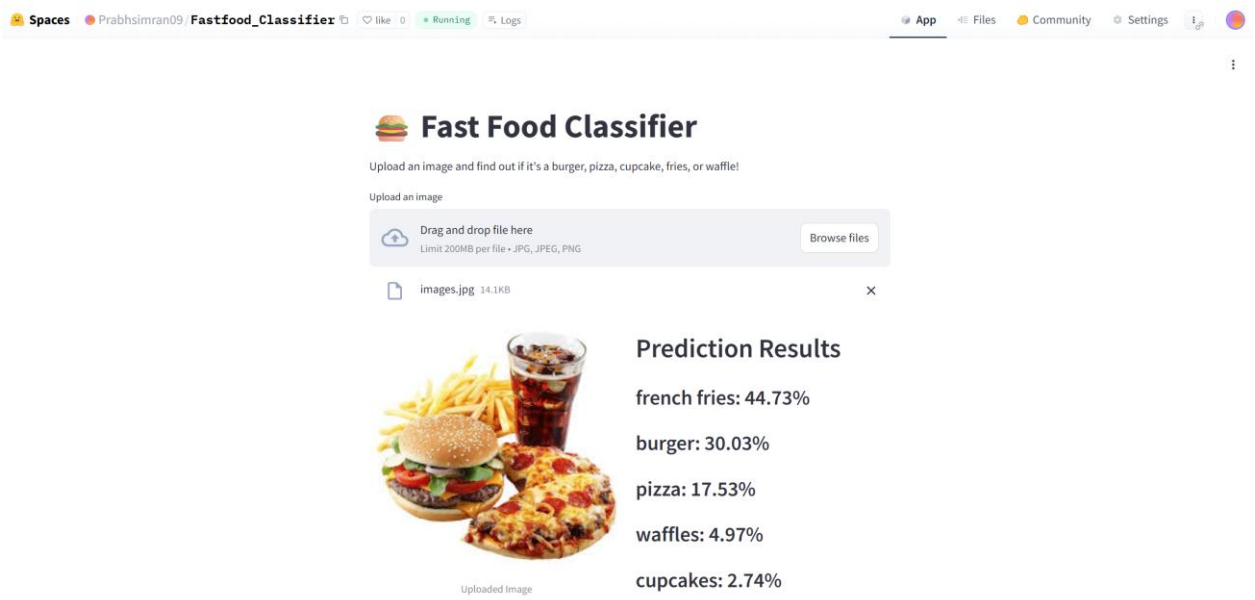

Figure 13.4.2: Fast Food Classifier App Demo

In this particular output, the uploaded image contains a mix of fast-food items such as fries, burger, and pizza. The classifier returns a ranked list of predictions:

- French fries: 44.73%
- Burger: 30.03%
- Pizza: 17.53%
- Waffles: 4.97%
- Cupcakes: 2.74%

This suggests that the model has identified French fries as the most dominant food item in the image with the highest confidence, followed by a burger and pizza.

This project demonstrates how machine learning and web frameworks can be combined to create fun, educational, and functional applications. It's ideal for showcasing image classification concepts, promoting interactive ML demos, or even building the foundation for food recognition tools used in health tracking, restaurant automation, or digital menus.

13.4.1. Hugging Face Link for Fast-food Classifier using Streamlit

https://huggingface.co/spaces/Prabhsimran09/Fastfood_Classifier

Fastfood Classifier

13.5. YOLOS Object Detection

YOLOS (You Only Look One-level Series) is a transformer-based object detection model introduced by Microsoft Research. It represents a significant shift in how object detection is approached, moving away from traditional CNN-based architectures (like YOLOv4 or Faster R-CNN) to a pure Transformer-based framework, inspired by the Vision Transformer (ViT). YOLOS is designed to detect objects in images by learning object representations directly from image patches, without using handcrafted components like anchors, feature pyramids, or region proposal networks.

Unlike traditional object detection models that rely heavily on convolutional layers to extract spatial hierarchies, YOLOS uses a Vision Transformer (ViT) backbone, which treats an image as a sequence of patches. Each patch is flattened and linearly embedded, and positional embeddings are added to retain spatial information. These patch embeddings are passed through standard Transformer encoder layers.

YOLOS introduces learnable object queries, similar to DETR (DEtection TRansformer). These queries attend to the encoded image representations and output predictions such as bounding box coordinates and class labels. The output of each query is interpreted as a potential object in the image. A key distinction of YOLOS is its minimalist design—there are no multi-scale features, no additional heads for refinement, and no convolutional layers in the detector. The simplicity of this architecture makes it elegant and fully end-to-end trainable.

YOLOS is trained using a set-based loss function that combines classification loss (e.g., cross-entropy) and bounding box regression loss (e.g., L1 loss and GIoU loss). The model is supervised by assigning each object query to a ground-truth box using Hungarian matching, ensuring that each prediction is uniquely matched to one object.

During inference, the model outputs predictions for a fixed number of object queries (e.g., 100). Each prediction includes a bounding box and class confidence score. Post-processing includes applying a confidence threshold and non-maximum suppression (NMS) to filter out redundant boxes.

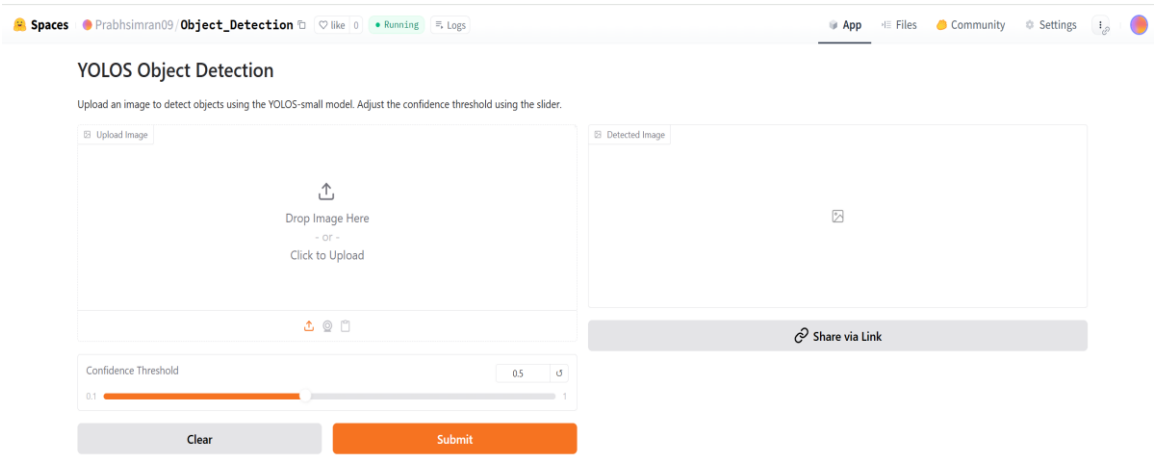

Figure 13.5.1: YOLOS Object Detection App Deployed on Hugging Face

The application interface is divided into two main sections: one for uploading an image and adjusting the confidence threshold, and the other for displaying the output image with bounding boxes around detected objects.

At the core of this project lies the YOLOS-small model, a lightweight version of Microsoft's YOLOS architecture, which is a transformer-based object detection model built on the Vision Transformer (ViT) paradigm. Unlike conventional object detection models that rely on convolutional layers, YOLOS leverages self-attention mechanisms to capture both global and local contextual information from the image. The input image is split into fixed-size patches and passed through transformer layers to produce high-level features. These features are then decoded using object query embeddings to generate bounding boxes and class labels for objects detected in the scene.

This application allows users to control the confidence threshold using a slider, which determines the minimum confidence score an object must have to be displayed. Lowering the threshold results in more detections (including less confident ones), while increasing it filters out uncertain predictions. Once the user uploads an image and clicks Submit, the YOLOS model processes the image, performs inference, and the result is rendered with labelled bounding boxes around each detected object.

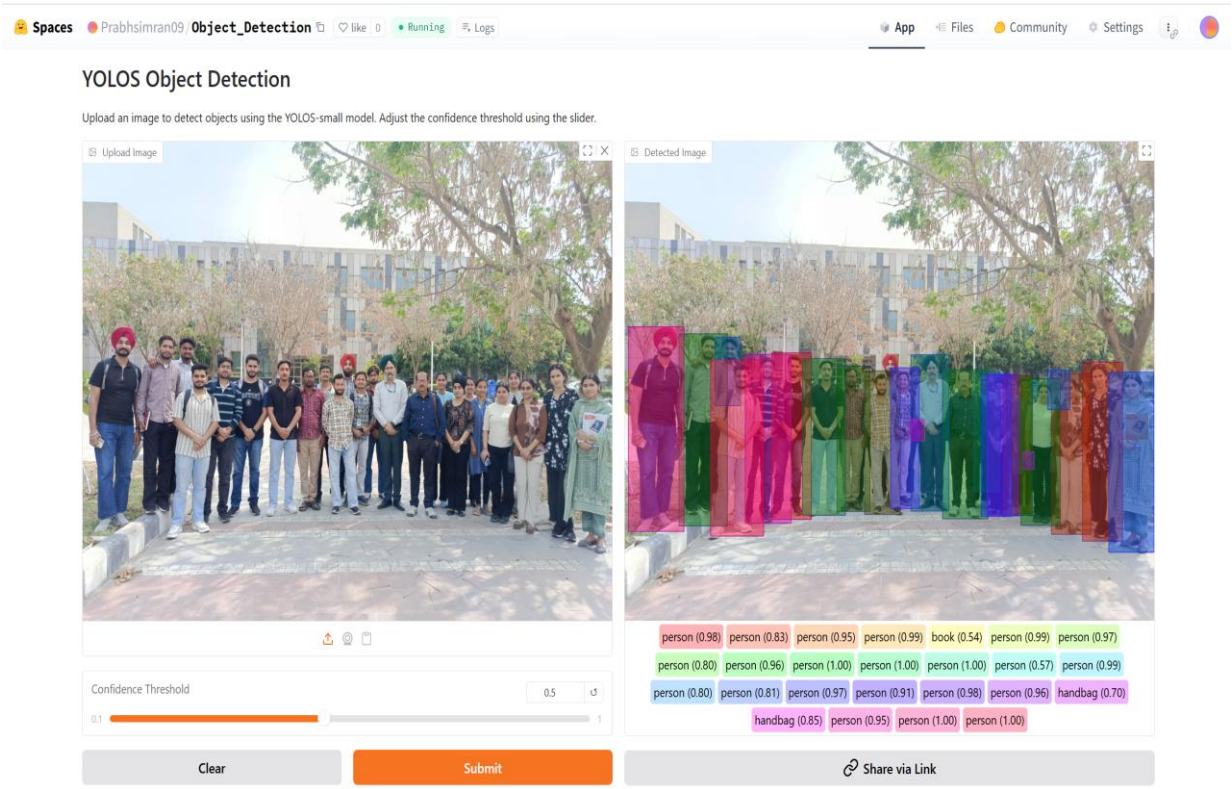

Figure 13.5.2: YOLOS Object Detection App Demo-1

In the given example, the uploaded image features a group of people. The YOLOS model has successfully identified multiple instances of the "person" class with high confidence scores (close to 1.00), along with other detected objects such as “book” and “handbag”. The application allows the user to adjust a confidence threshold using a slider; this threshold determines which detections are displayed based on the model’s certainty. Lowering the threshold displays more results (including lower-confidence predictions), while increasing it filters the results for higher accuracy.

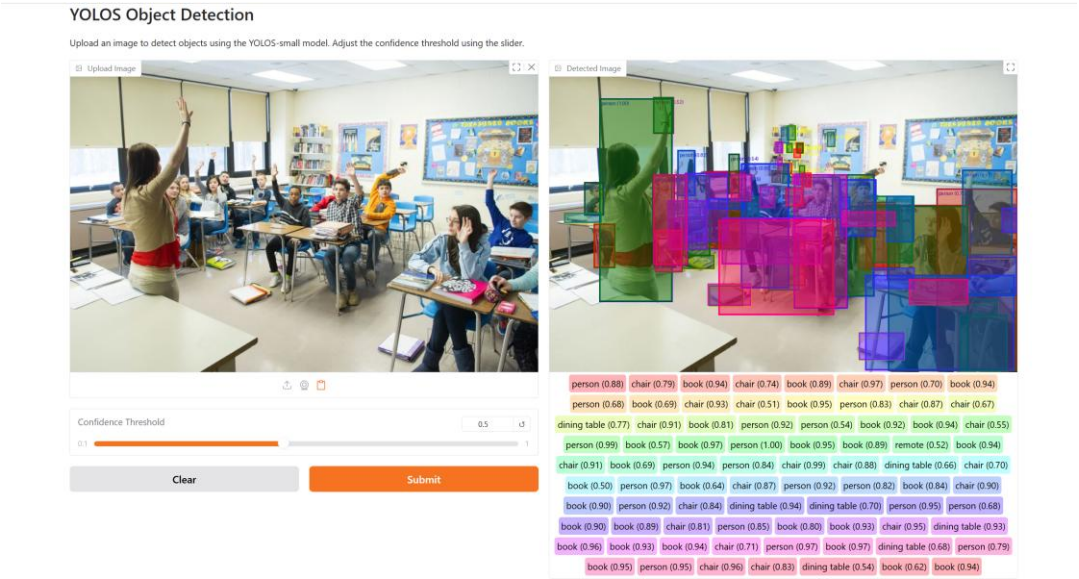

Figure 13.5.3: YOLOS Object Detection App Demo-2

In this case, the YOLOS model has successfully detected and labeled multiple objects such as person, book, chair, remote, and dining table. Each detection is accompanied by a confidence value, showing how certain the model is about its prediction. For example, the teacher and students are detected as "person" with confidence scores close to 1.00, and several books and chairs are identified throughout the scene with varying levels of certainty.

This project demonstrates the practical deployment of cutting-edge vision transformers for object detection in a web-accessible format. With Streamlit managing the frontend and Hugging Face Spaces enabling cloud hosting, this app becomes a powerful tool for educational, research, or demonstration purposes. Users can upload any image, tune detection sensitivity, and instantly visualize how transformer-based models like YOLOS interpret scenes and detect objects.

Overall, this project showcases the synergy of modern deep learning models (YOLOS) with streamlined deployment tools (Streamlit + Hugging Face) to deliver a powerful and accessible object detection system.

13.5.1. Hugging Face Link for YOLOS Object Detection

https://huggingface.co/spaces/Prabhsimran09/Object_Detection

Object Detection

13.6. Face Recognition Based Attendance System

A face recognition-based attendance system is an advanced application of artificial intelligence and computer vision that automates the process of marking attendance using facial features. Unlike traditional methods that rely on manual entry, ID cards, or biometric fingerprints, this system leverages deep learning models to detect, extract, and compare facial embeddings to identify individuals accurately. When a user stands in front of a camera, the system captures their image, processes it using a face detection model (such as InsightFace or MTCNN), and then generates a unique facial embedding. This embedding is compared against a database of pre-registered employees or students. If a match is found above a defined similarity threshold, the person is identified, and their attendance is logged along with the date and timestamp in a secure database.

This solution not only increases efficiency by reducing the time and effort required for manual attendance but also minimizes the chances of proxy attendance or fraudulent entries.

Additionally, it enhances safety and hygiene—particularly in post-pandemic scenarios—by offering a completely contactless mechanism. Most systems incorporate real-time feedback, display welcome messages, and store logs for administrative auditing. Face recognition attendance systems are widely applicable in schools, universities, offices, and secured environments where tracking personnel presence is critical. When integrated with platforms like Streamlit or deployed on the web through Docker and Hugging Face Spaces, these systems become scalable, accessible, and highly interactive, allowing institutions to embrace smart, AI-driven automation seamlessly.

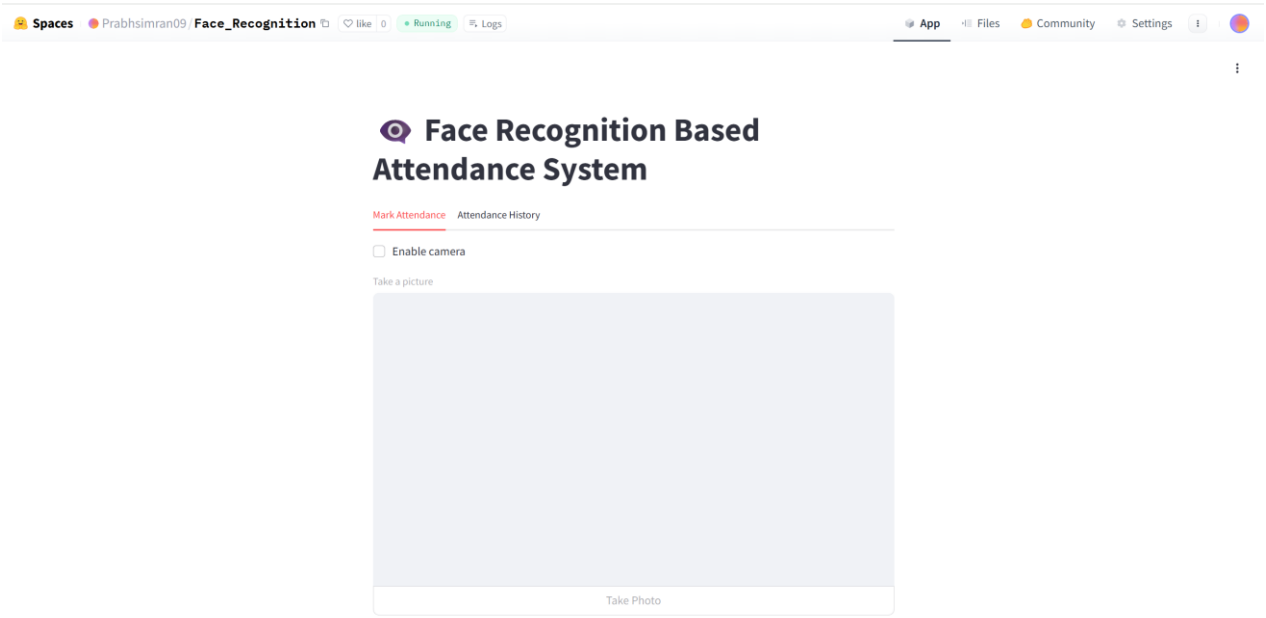

Figure 13.6.1: Face Recognition Based Attendance System App Deployed on Hugging Face

The "Face Recognition Based Attendance System" is a real-time AI-powered web application that enables contactless, automatic attendance logging through facial verification. Designed using computer vision and deep learning techniques, this system utilizes a camera interface to capture a user's image and then compares their facial features with a database of enrolled individuals. If the face is recognized with sufficient confidence, the system automatically logs the user’s name, date, and time into an attendance record, ensuring both accuracy and security. The web interface, built using tools like Streamlit and deployed on Hugging Face Spaces, allows for user-friendly interaction with options to enable the camera, take a photo, and view attendance history through intuitive tabs. This smart attendance system replaces traditional methods like roll calls or ID card scans, offering a hygienic, scalable, and efficient solution for institutions, workplaces, and events that require reliable identity verification and record keeping.

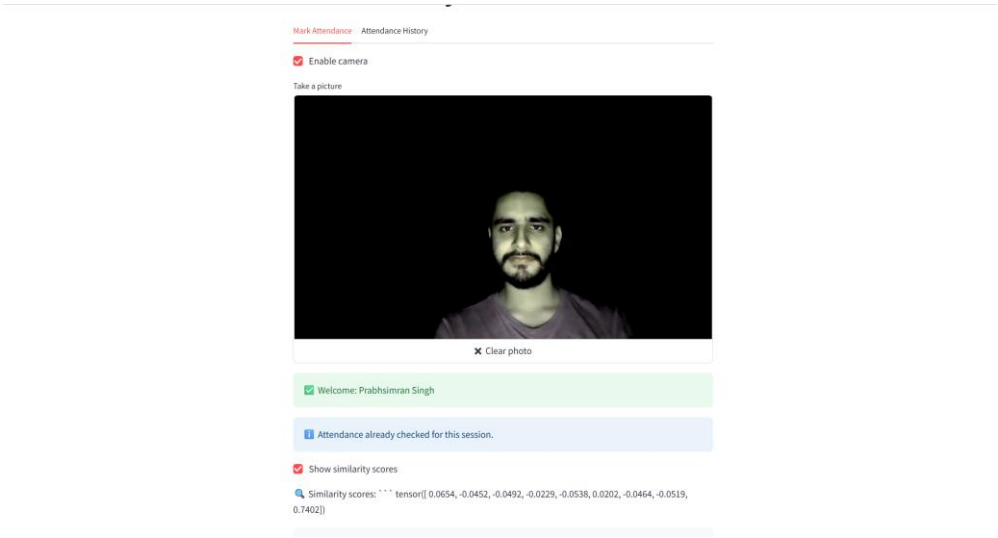

Figure 13.6.2: Face Recognition Based Attendance App Demo

The displayed interface showcases a fully functional AI-based face recognition attendance system in action. In this instance, the user has enabled the webcam and successfully captured their photo through the built-in camera module of the application. Upon submitting the image, the system detects the user's face and compares it with the pre-stored images in the database using advanced facial embedding techniques powered by InsightFace. The system identifies the person as "Prabhsimran Singh" and greets them with a welcome message. Since the attendance for this user has already been recorded for the current date, the system intelligently avoids duplicate entries by notifying that the attendance has already been checked for this session. Additionally, the option to display similarity scores is enabled, providing a transparent view of how closely the captured image matches each stored identity, based on cosine similarity values. This not only validates the recognition process but also builds user trust. Overall, the system offers a seamless, contactless, and intelligent attendance marking experience suitable for educational institutions and workplaces.

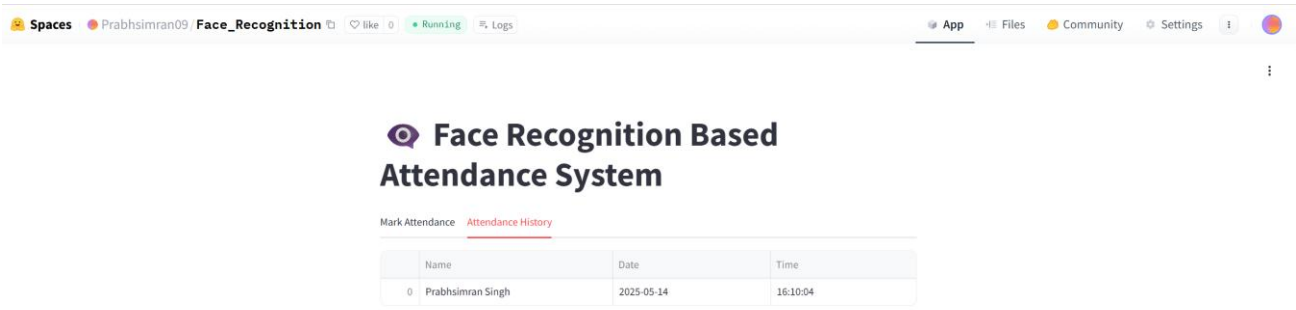

Figure 13.6.3: Face Recognition Based Attendance System’s Attendance History

This section displays a table that logs all recognized entries, including the individual's name, the date of attendance, and the exact time at which the face was verified and marked. In this case, the attendance record shows that "Prabhsimran Singh" successfully marked his presence on 2025-05-14 at 16:10:04. This historical log is automatically maintained in a CSV file and dynamically updated each time a unique attendance is marked. It serves as an organized and transparent attendance record that administrators or users can view anytime. The user-friendly layout ensures ease of access and helps verify the proper functioning and reliability of the system in real time.

13.6.1. Hugging Face Link for Face Recognition Based Attendance System

https://huggingface.co/spaces/Prabhsimran09/Face_Recognition

Face Recognition

13.7. Sentiment Analysis using Streamlit

Sentiment Analysis is a branch of Natural Language Processing (NLP) that focuses on identifying and extracting subjective information from textual data. Its primary goal is to determine the emotional tone behind a body of text, which could range from positive and negative to neutral sentiments. This technique is crucial for understanding public opinion, customer feedback, and social media conversations, helping businesses and organizations make data-driven decisions. The process typically begins with text preprocessing, where the raw input is cleaned and transformed through steps like tokenization, stop-word removal, and lemmatization to prepare it for analysis. Feature extraction methods like Bag of Words (BoW), TF-IDF, or modern word embeddings such as Word2Vec and BERT are then used to convert the text into numerical representations that machine learning algorithms can understand.

Sentiment classification can be achieved through various approaches. Rule-based systems use predefined lists of positive and negative words to determine sentiment scores, while machine learning models like Naive Bayes, Support Vector Machines (SVM), and Logistic Regression rely on training data to learn sentiment patterns. Deep learning models, particularly Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) networks, are capable of capturing contextual relationships in text for more accurate analysis. More recently, transformer-based models like BERT and RoBERTa have become popular for their superior language

understanding capabilities, often achieving state-of-the-art performance in sentiment analysis tasks.

There are different types of sentiment analysis, including fine-grained analysis (e.g., rating sentiments on a scale), aspect-based analysis (evaluating sentiments on specific features like "battery life" or "camera"), and emotion detection (identifying emotions like joy, anger, or sadness). Despite its many advantages, sentiment analysis faces challenges such as detecting sarcasm, understanding context, handling negations, and analyzing text in multiple languages. Tools like VADER, TextBlob, spaCy, and transformer models from Hugging Face are commonly used for implementing sentiment analysis. Ultimately, this technique finds wide application in areas like product review mining, brand monitoring, customer service, and even political analysis, offering valuable insights from large volumes of unstructured text.

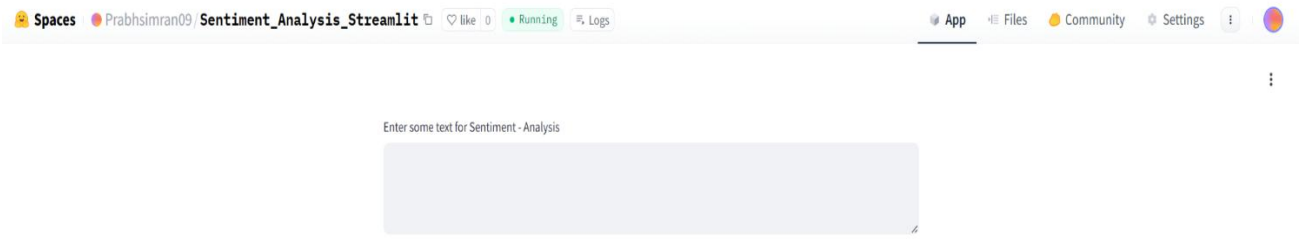

Figure 13.7.1: Senitment Analysis App Deployed on Hugging Face

This interface allows users to enter a block of text into a textbox, after which the system likely performs sentiment analysis on the input and returns whether the sentiment is positive or negative.

This application demonstrates the core concept of Sentiment Analysis, a subfield of Natural Language Processing (NLP) that involves computationally identifying and categorizing opinions expressed in a piece of text. The purpose is to determine the writer's attitude toward a particular topic, product, or event. At the heart of such systems lies a language model or sentiment classifier, which processes the input text and predicts the emotional tone—such as joy, anger, sadness, positivity, negativity, or neutrality.

The app uses Hugging Face Transformers' default sentiment analysis pipeline, which internally loads a pretrained transformer mode. These models work by analyzing the words, phrases, and context of the input sentence. For example, in "I love this movie!", words like "love" carry strong positive polarity, and the model learns to associate such expressions with positive sentiment.

The backend, powered by Streamlit, provides a simple Python-based interface for deploying such NLP models with interactive UI components like text inputs and output display. The user enters

a sentence, which is then processed and analyzed by the model hosted in the backend. The output sentiment is displayed either directly or as a probability score across categories.

This application is valuable in real-world scenarios such as:

- Brand monitoring on social media,
- Customer feedback analysis,
- Product review mining,
- Market research.

Example 1 – Positive Sentiment:

Input:

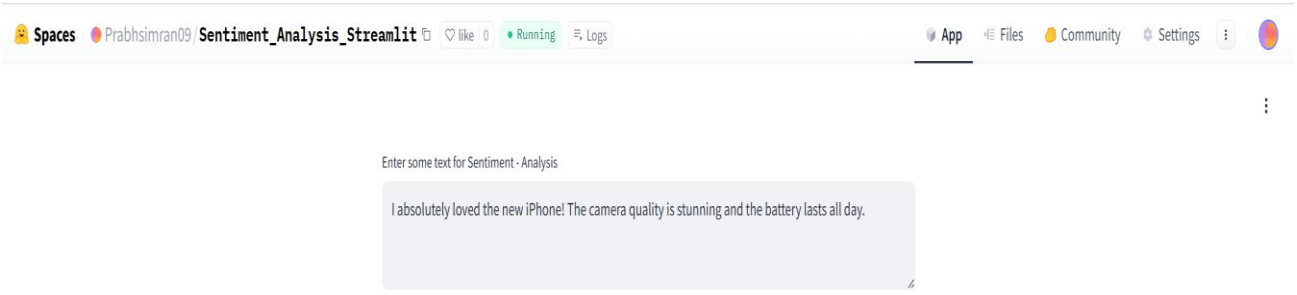

Figure 13.7.2: Example of Positive Sentiment

This is a clearly positive statement, filled with emotionally strong words like “loved,” “stunning,” and “lasts all day.”

Output:

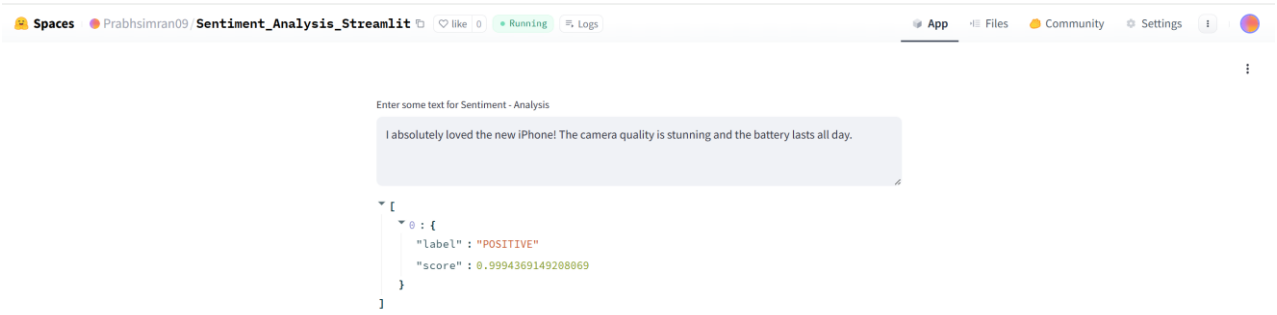

Figure 13.7.3: Resulting Output of Positive Sentiment

It correctly identifies the sentiment as POSITIVE with a high confidence score (typically above 0.99). The model that is being used is trained to understand such expressions and can accurately classify them by analyzing word tone, grammar, and context. This demonstrates that your app is functioning well and correctly applying a transformer-based sentiment analysis pipeline in a user-friendly interface.

Example 1 – Negative Sentiment:

Input:

Figure 13.7.4: Example of Negative Sentiment

This is a clearly negative statement, filled with emotionally strong words like “disappointment,” and “terrible.”

Output:

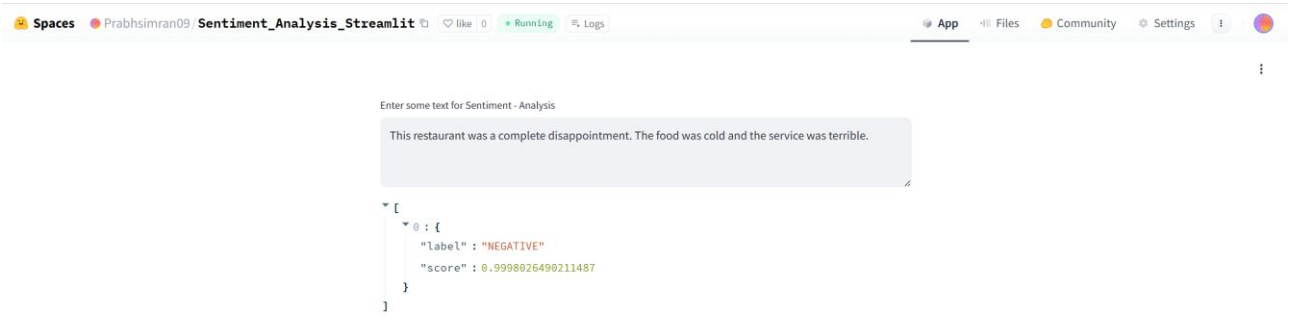

Figure 13.7.5: Resulting Output of Negative Sentiment

It correctly identifies the sentiment as NEGATIVE with a high confidence score (typically above 0.99). The model that is being used is trained to understand such expressions and can accurately classify them by analyzing word tone, grammar, and context. This demonstrates that your app is functioning well and correctly applying a transformer-based sentiment analysis pipeline in a user-friendly interface.

13.7.1. Hugging Face Link for Sentiment Analysis using Streamlit

https://huggingface.co/spaces/Prabhsimran09/Sentiment_Analysis_Streamlit

Sentiment Analysis

13.8. Fake News Detection

Fake News Detection is a process that involves identifying whether a piece of information—typically in the form of a news article, headline, or social media post—is true, false, or misleading. With the rapid spread of digital media and the increasing ease of publishing content online, fake news has become a major global concern. It can mislead public opinion, create panic, influence elections, or harm reputations and public trust. Detecting fake news requires analyzing both the linguistic content and the contextual cues surrounding a piece of information.

At the core of fake news detection lies Natural Language Processing (NLP) and machine learning. These technologies allow computers to read and understand human language. Models are trained on datasets containing both real and fake news samples. These models learn to recognize patterns in vocabulary, sentence structure, tone, and factual inconsistencies. For instance, fake news may use exaggerated language, clickbait headlines, or include contradictory or unverifiable claims. Advanced models, especially transformer-based language models like Zephyr, GPT, or BERT, are capable of understanding context, fact-checking against known information, and generating reasoned responses.

The detection process usually involves several steps. First, the system pre-processes the text, cleaning and tokenizing it. Then, the model analyzes the semantic content and evaluates its credibility based on prior knowledge or training. In some cases, external knowledge bases or fact-checking tools are integrated to compare claims with verified data. The model then classifies the input into categories such as "Likely Real," "Likely Fake," or "Uncertain," often with a brief rationale.

Modern fake news detectors are increasingly conversational and interactive, like the one built using Gradio and the Zephyr-7B model. These allow users to input news and receive not just a label but also an explanation of the reasoning behind the classification. This transparency helps users develop critical thinking and understand how AI evaluates trustworthiness in information. Overall, fake news detection is a vital application of AI in promoting information literacy and safeguarding public discourse.

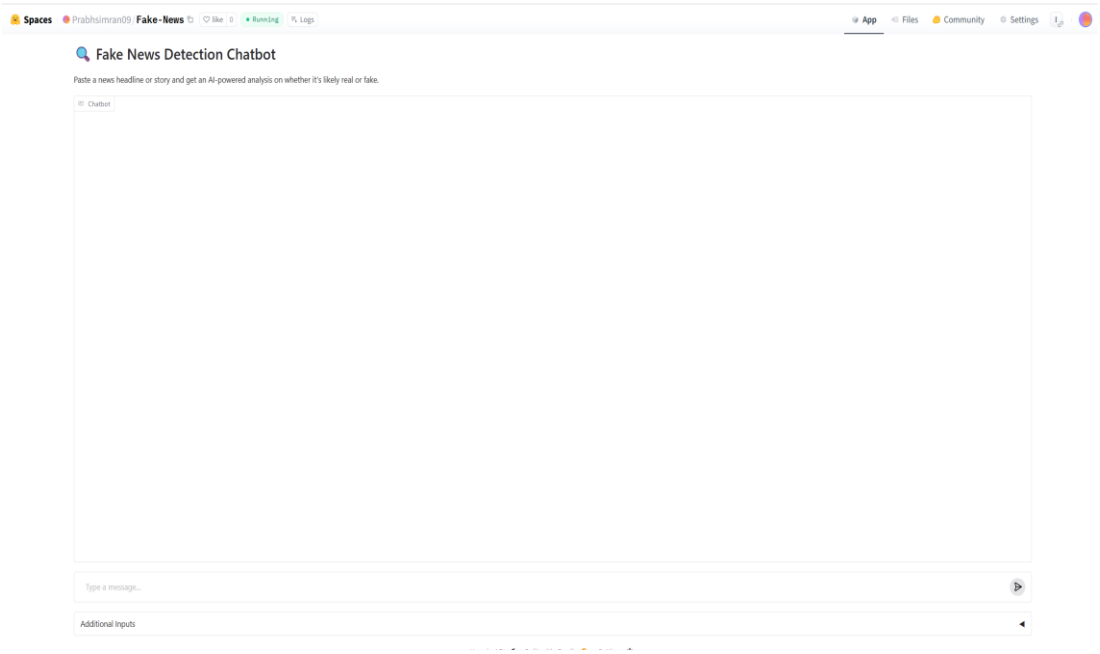

Figure 13.8.1: Fake News Detection Chatbot Deployed on Hugging Face

Fake news detection is an application of Natural Language Processing (NLP) and deep learning, aimed at evaluating the credibility of text-based content. The model powering this app—Zephyr-7B—is a powerful transformer-based language model trained to understand and generate human-like text. When a user enters a headline, the model considers the linguistic style, tone, content structure, and known facts (learned during training) to make a judgment.

The app runs on a chat interface built using Gradio. The backend logic sends the user's message, along with conversation history and a system instruction, to the Zephyr model via the Hugging Face InferenceClient. The system prompt defines the AI's behavior—here, as a fake news expert. As the user submits a message, the model processes it and streams back a response token-by-token, making it appear as though a human expert is responding in real time.

In technical terms, the model performs classification with explanation, where it not only labels the news as "Likely Fake" or "Likely Real" but also justifies its reasoning based on learned patterns. For example, fake news might include implausible claims, lack of named sources, or exaggerated language—traits the model picks up from its training data.

Figure 13.8.2: Additional Inputs of Fake News Chatbot

The image shows the additional input controls for a fake news detection chatbot powered by a large language model. These inputs allow users to customize the behavior of the AI and fine-tune its responses in real time. At the top is the System Message, which sets the role and context for the AI assistant. In this case, the system message instructs the model to behave as a fake news expert who classifies user-provided headlines as "Likely Fake," "Likely Real," or "Uncertain", and gives a brief explanation for the classification. This prompt acts as a behavioral guide for the AI model, ensuring that its replies are focused and relevant to the task of misinformation detection.

Beneath the system message are three sliders that directly affect how the model generates responses:

- **Max New Tokens** determines the maximum length of the response. A value of 512 allows moderately long responses while keeping them concise enough to avoid unnecessary verbosity.
- **Temperature** controls the randomness of the output. A value of 0.7 strikes a balance between creativity and reliability, allowing the model to generate diverse yet sensible replies. Lower values make the output more deterministic, while higher values can increase variation but may reduce accuracy.
- **Top-p (Nucleus Sampling)** is another parameter that controls output variability. With a value of 0.95, the model considers the smallest possible set of tokens whose cumulative probability exceeds 95%, ensuring that the most likely words are used while still allowing for nuanced and informative responses.

These controls give users the flexibility to adjust how cautious or expressive the model should be, making the app suitable for a range of use cases—from strict fact-checking to exploratory analysis. Overall, this design demonstrates the integration of human-in-the-loop control in AI systems, where users can guide large language models to deliver more relevant, accurate, and interpretable results.

13.8.1. Examples

- Example 1 – Likely Real:

Figure 13.8.1.1: Example 1 of Likely Real News

In the example provided—"India to Lead in Shaping 6G Technology Regulations"—the model identified the claim as 'Likely Real' based on contextual and historical cues. It analyzed India's past involvement in 5G development, government initiatives for 6G, and national ambitions to lead in technological innovation. By matching these facts with the language and tone of the headline, the model concluded that the statement fits a realistic and credible pattern. This reasoning ability is enabled by the model's pretraining, which equips it with an internal representation of how true and false statements typically appear linguistically and contextually.

- Example 2 – Likely Real:

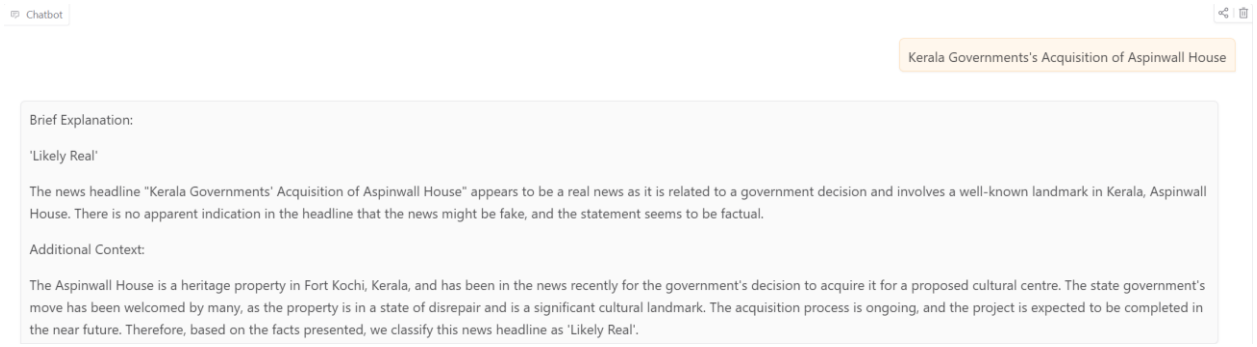

Figure 13.8.1.2: Example 2 of Likely Real News

The output shown in the image demonstrates the functioning of a fake news detection AI chatbot, where the system has classified the news headline "Kerala Government's Acquisition of Aspinwall House" as 'Likely Real'. This classification is not based on superficial word-matching but on deep contextual understanding powered by a large language model. The underlying theory of this system lies in the field of Natural Language Processing (NLP) and transformer-based models, which are trained on diverse corpora of real-world information to develop a nuanced understanding of human language, facts, and social behaviour.

When a user submits a news headline, the model evaluates its plausibility by analyzing linguistic structure, factual relevance, and alignment with known real-world events. In this case, the AI recognizes that the statement references a government action, involves a known cultural heritage site (Aspinwall House in Fort Kochi), and reflects ongoing news developments. Since such patterns are common in legitimate news reporting—and there is no hyperbolic or illogical content—it classifies the claim as believable. The model also supplements this with an explanation, helping users understand the rationale behind its judgment. This is possible because the AI has been trained to reason with contextual and historical knowledge rather than just keywords.

- Example 3 – Likely Fake:

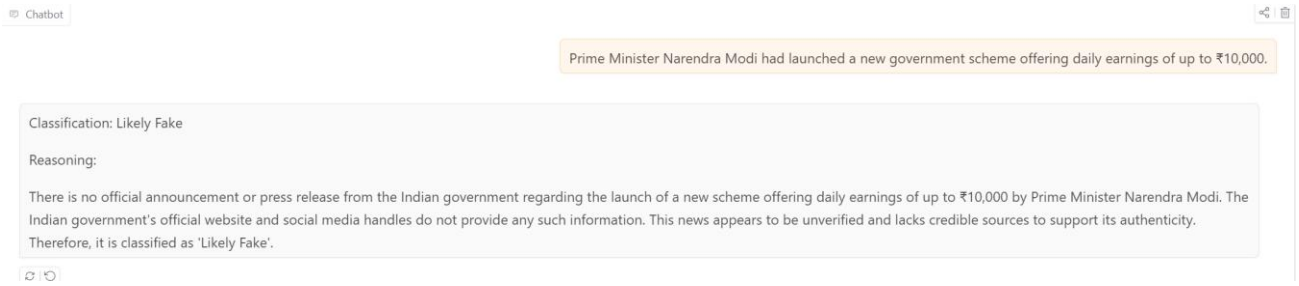

Figure 13.8.1.3: Example 1 of Likely Fake News

The image displays the output of a fake news detection chatbot classifying the headline “Prime Minister Narendra Modi had launched a new government scheme offering daily earnings of up to ₹10,000” as “Likely Fake”, along with a clear explanation. This output demonstrates how the AI model uses logical reasoning and contextual understanding to detect misinformation. The underlying theory behind this detection relies on Natural Language Processing (NLP) combined with knowledge-grounded transformer models, which are capable of assessing the credibility of statements by comparing them with known facts, official information sources, and common linguistic patterns of fake content.

In this case, the model detects the absence of official press releases or announcements from government platforms, which is a key factor in evaluating the truthfulness of a claim involving public policy. Additionally, the model is aware that large monetary claims like “₹10,000 daily” often signal potential scams or misinformation. Rather than blindly flagging headlines, the AI reasons through lack of evidence and credibility by simulating the logic a human fact-checker might use. This form of intelligent assessment is made possible because the model has been trained on vast datasets containing real and fake news, enabling it to learn subtle cues like exaggeration, lack of attribution, and implausible context.

Thus, fake news detection here is not just keyword matching but context-sensitive classification, where the model interprets the intent, checks the factual foundation, and generates a reasoned output. This empowers users to not only identify misinformation but also understand the "why" behind the verdict, promoting critical thinking and digital awareness.

- Example 4 – Likely False:

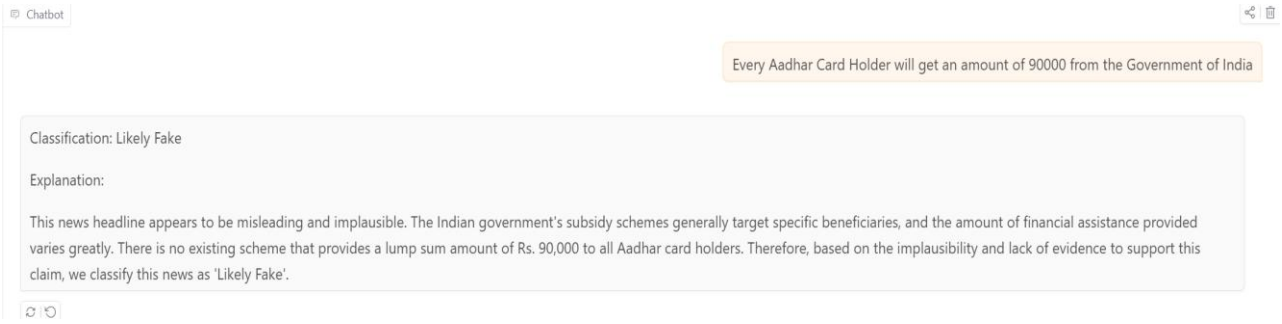

Figure 13.8.1.4: Example 2 of Likely Fake News

The image shows a response from the Fake News Detection Chatbot classifying the claim "Every Aadhaar Card Holder will get an amount of ₹90,000 from the Government of India" as 'Likely Fake', along with a detailed explanation. The theory behind this classification lies in the chatbot's ability to evaluate the plausibility of a statement using Natural Language Processing (NLP) and contextual reasoning powered by a large language model. Instead of blindly flagging or accepting headlines, the model draws from its training on credible government policies, economic norms, and social schemes to assess whether a claim is realistic and factually grounded.

In this example, the model recognizes that such a large universal payout is inconsistent with how government subsidy programs typically function in India. Financial assistance in India is generally targeted, means-tested, and varies across schemes—never offered as a flat ₹90,000 payout to all Aadhaar holders. The AI evaluates the linguistic cues (e.g., over-generalization, exaggerated figures), the absence of evidence from known credible sources (such as official portals or press releases), and the implausibility based on economic feasibility. Combining these factors, the model determines that the claim lacks credibility and labels it as "Likely Fake."

This theoretical framework emphasizes reasoned classification rather than superficial keyword detection. The AI simulates how a fact-checker would investigate the authenticity of a statement—by weighing realism, available evidence, official validation, and context. Thus, the chatbot empowers users to critically assess viral news and become more informed digital citizens in the fight against misinformation.

- Example 5 – Uncertain:

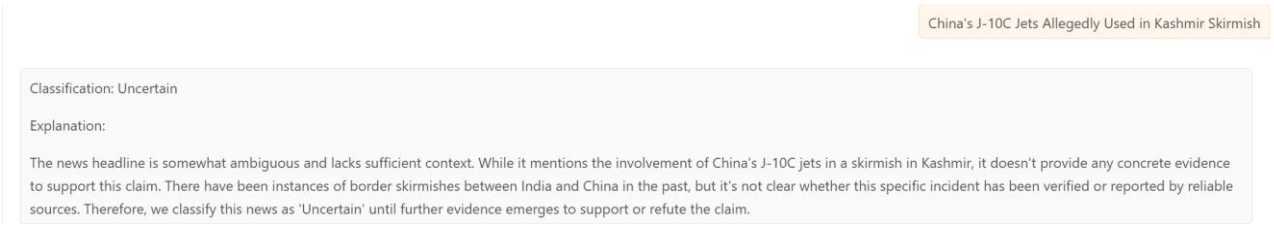

Figure 13.8.1.5: Example 1 of Uncertain News

The image displays an example of a news headline— “China’s J-10C Jets Allegedly Used in Kashmir Skirmish”—being classified as ‘Uncertain’ by the Fake News Detection Chatbot. This classification showcases the chatbot’s ability to recognize not only clearly real or fake news but also cases where the available evidence is inconclusive. The theory behind this decision-making process relies on contextual reasoning, evidence validation, and ambiguity detection, all powered by a transformer-based large language model trained in Natural Language Processing (NLP).

In this scenario, the model understands that the headline mentions a significant geopolitical development involving two countries with a sensitive history. However, it also recognizes that the statement uses speculative language (“allegedly”) and fails to cite credible sources or confirm the event’s occurrence. This triggers the model’s uncertainty mechanism, where it assesses the lack of verifiable facts, vague context, and absence of confirmation from trusted media outlets or government statements. The AI is trained to avoid making definitive classifications when evidence is weak, unverified, or speculative in nature.

This intelligent hesitation is an essential part of explainable and ethical AI, preventing the chatbot from reinforcing or spreading unconfirmed narratives. By labelling such content as "Uncertain," the system encourages users to seek more reliable information before drawing conclusions. This theoretical framework highlights the model’s ability to not only evaluate factual accuracy but also measure the confidence level of available information, making it a sophisticated tool in combating misinformation and promoting informed decision-making.

- Example 6 – Uncertain:

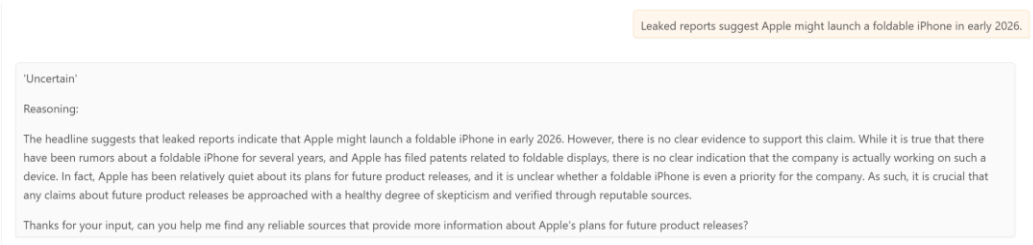

Figure 13.8.1.6: Example 2 of Uncertain News

The headline claims that “Apple might launch a foldable iPhone in early 2026,” which is categorized as "uncertain" because it lacks verifiable evidence. The reasoning highlights that while there have been ongoing rumors and patents related to foldable displays by Apple, the company has not made any official announcements. Apple is known for its secrecy regarding future products, and just because patents exist doesn't confirm a product is being developed or scheduled for release. Therefore, the theory emphasizes the importance of treating such speculative news with skepticism and only accepting it as plausible — not factual — until credible sources or the company itself provide confirmation. This critical approach helps avoid misinformation and encourages readers to validate claims through reliable, official channels.

The Fake News Detection App is an AI-powered chatbot designed to classify news headlines or short text snippets as “Likely Real,” “Likely Fake,” or “Uncertain.” Built using Gradio and powered by a large language model such as Zephyr-7B from Hugging Face, the app simulates the behavior of a human fact-checker by analyzing linguistic patterns, factual accuracy, and context. Users input a news statement, and the model evaluates it using a predefined system message that guides its behavior as a fake news analyst. It uses advanced Natural Language Processing (NLP) techniques to reason about the input and generate a detailed explanation for its classification. The chatbot also includes adjustable parameters such as token limit, temperature, and top-p sampling, allowing users to control the length, creativity, and confidence of the AI's responses. By offering clear, explainable judgments and interactive conversation, the app promotes media literacy and helps users critically evaluate the reliability of online information.

13.8.2. Hugging Face Link for Fake News Detection

https://huggingface.co/spaces/Prabhsimran09/Fake-News

Fake News Detection

13.9. Deepfake Detection

13.9.1. Introduction

Deepfake detection is a critical area of research and application in artificial intelligence, particularly in computer vision and digital forensics. Deepfakes refer to media—mostly videos, images, or audio recordings—that are manipulated or entirely synthesized using deep learning techniques to make them appear real. These are often generated by models like Generative Adversarial Networks (GANs) or deep autoencoders. While these technologies have beneficial applications in fields like film-making, education, and accessibility, they are increasingly misused to spread misinformation, impersonate individuals, and deceive the public. Hence, robust deepfake detection systems are essential to safeguard the authenticity and integrity of digital content.

The process of detecting deepfakes begins with the understanding that forged media, despite their realistic appearance, often carry subtle inconsistencies that machines can be trained to detect. These inconsistencies can be in facial movements, lighting patterns, blinking frequency, head orientation, or texture mismatches caused by the generative model. Traditional detection methods focused on identifying such handcrafted features, like irregular blinking or unnatural facial movements, but these methods often fail against more sophisticated deepfakes. Consequently, modern approaches employ deep learning, where models learn discriminative features automatically from large datasets.

Among the most commonly used architectures are Convolutional Neural Networks (CNNs), which excel at learning spatial patterns in images and video frames. Advanced models like XceptionNet, EfficientNet, and ResNeXt have shown promising results in identifying facial artifacts in deepfake videos. For video analysis, combining CNNs with Recurrent Neural Networks (RNNs), particularly Long Short-Term Memory (LSTM) networks, helps capture the temporal relationships across video frames—this is essential because deepfakes often suffer from temporal inconsistencies in facial dynamics that are hard for the human eye to spot. These combined CNN+LSTM pipelines first extract visual features from each frame and then analyze how those features evolve over time to detect unnatural transitions.

Another advancement is the use of Vision Transformers (ViTs), which rely on self-attention mechanisms to analyze image regions in relation to one another. ViTs can better capture long-range dependencies and global context in the input data, making them increasingly popular for detecting subtle deepfake traces. On the other hand, when it comes to audio deepfakes—synthetic speech generated to impersonate someone—models analyze spectrograms or raw waveform features using CNNs, RNNs, or even transformer-based audio models like Wav2Vec or RawNet.

These models look for telltale signs like unnatural pitch modulations, frequency distortions, or inconsistencies in speech cadence.

A standard deepfake detection pipeline generally involves several steps. First, the input media is preprocessed, where faces are detected and aligned using tools like MTCNN or InsightFace. Then, deep learning models are used to extract features and predict whether the input is real or fake. For video, predictions may be made for each frame and then aggregated using voting or sequence models. Some systems incorporate explainability tools like Grad-CAM, which highlight the areas of the image that contributed most to the model's decision, helping users understand why a sample was classified as fake.

Datasets play a vital role in training and evaluating deepfake detection systems. Popular datasets include FaceForensics++, which contains a variety of manipulation techniques; the Deepfake Detection Challenge (DFDC) dataset from Facebook; Celeb-DF, known for its high-quality celebrity deepfakes; and FakeAVCeleb, which includes both video and audio deepfakes. These datasets help build models that are generalizable across different manipulation methods.

Despite impressive progress, deepfake detection still faces several challenges. The rapid advancement of generative models like StyleGAN3 and diffusion models means that fake content is becoming increasingly photorealistic and harder to detect. Furthermore, detection models often overfit to the training data and struggle to generalize to unseen manipulation techniques or identities. Robustness to compression, noise, and adversarial attacks remains a significant concern. Another challenge is multimodal deepfakes, where both the audio and video are manipulated to create highly convincing fakes. Additionally, explainability is often limited, and it is difficult to interpret why a model classified a sample as fake, which is crucial in legal or journalistic contexts.

In summary, deepfake detection combines deep learning, signal processing, and forensic techniques to identify synthetically manipulated media. As generative techniques evolve, detection must also become more sophisticated, leveraging better architectures, larger and more diverse datasets, and explainable AI methods. It is an ongoing arms race between creation and detection, and ensuring the trustworthiness of digital content depends heavily on the continuous advancement of deepfake detection technologies.

13.9.2. Architecture of Deepfake Detection Pipeline

The diagram below in Figure illustrates a deepfake detection pipeline specifically designed for analyzing videos. This architecture is a hybrid of Convolutional Neural Networks (CNNs) for spatial feature extraction and Recurrent Neural Networks (RNNs) for temporal sequence

modeling. The overall goal is to determine how much of the video content has been manipulated and to output a fake video probability percentage.

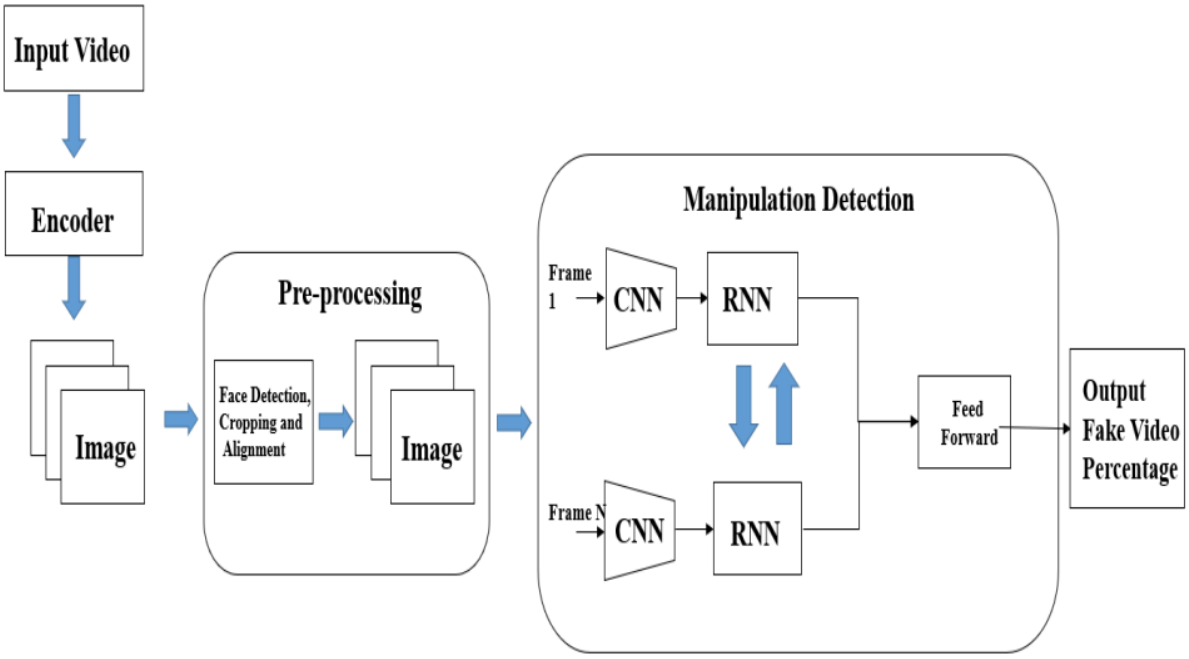

Figure 13.9.2: Architecture of Deepfake Detection Pipeline

1. Input Video

The process begins with an input video that needs to be analyzed for signs of manipulation. This video can originate from any source—social media, video calls, surveillance footage, or uploaded content—and serves as the raw data. The video itself is a sequence of frames containing one or more subjects, and deepfake manipulation, if present, is usually focused on the facial region. The primary objective at this stage is to prepare the video for frame-wise inspection, which will later allow a deep learning model to analyze its content in detail.

2. Frame Extraction (Encoder)

Once the video is received, the next step is to extract individual frames using an encoder. This encoder decodes the video into image frames, usually at a fixed frame rate (e.g., every 5th or 10th frame), depending on the processing requirements and model design. Each of these frames captures the facial expressions, lighting conditions, and other visual information necessary for deepfake detection. The goal is to reduce the video into a series of representative still images that can be examined independently and collectively for anomalies.

3. Preprocessing

Preprocessing is a vital step where the focus shifts entirely to the face, as that's typically the manipulated region in a deepfake. This stage includes face detection, cropping, and alignment. Face detection models like MTCNN or InsightFace are used to locate the face within each

frame. Once detected, the facial region is cropped out to eliminate background noise. These cropped faces are then aligned based on facial landmarks (eyes, nose, etc.) to ensure they are normalized in size and orientation. This consistent formatting is essential for the CNN to effectively learn and detect manipulation features, regardless of variations in angle or lighting.

4. Manipulation Detection – CNN Block

After preprocessing, the aligned face images are passed through a Convolutional Neural Network (CNN). The CNN plays a crucial role in learning spatial features—such as texture mismatches, abnormal shading, blending artifacts, or edge distortions—that may indicate manipulation. Deepfakes often leave behind tiny inconsistencies that are invisible to the human eye but can be detected by a CNN trained on real vs. fake data. The CNN converts each image into a set of feature maps or embeddings, which represent the critical information needed for classification.

5. Manipulation Detection – RNN Block

The features extracted from each frame by the CNN are then passed into a Recurrent Neural Network (RNN), such as an LSTM or GRU. This component is key to capturing temporal dependencies—how facial features move and evolve over time. Deepfakes might maintain good spatial quality in individual frames but often falter when it comes to smooth, natural transitions across frames. RNNs analyze these temporal patterns, such as eye blinking rates, mouth movements, or expression flow, to identify unnatural behavior. By processing sequences of frame embeddings, the RNN enhances the model's ability to detect subtle, time-based anomalies that a CNN alone would miss.

6. Feedforward Network

After the RNN has processed the temporal features, the resulting data is passed into a feedforward neural network (a fully connected layer). This layer functions as a decision-making unit that aggregates all the insights gained from spatial and temporal analysis. It combines the features into a single vector and passes it through activation functions (like sigmoid or softmax) to generate a probability score. This step essentially translates complex feature representations into a simple, interpretable output for the user or system.

7. Output Prediction

The final output of the model is a numerical probability or percentage that indicates the likelihood of the video being fake. For example, a result of 92% fake suggests that the model is highly confident that the input video contains manipulated content. This value can be used directly or subjected to a threshold (e.g., above 50% = fake) to produce a binary classification (Real vs. Fake). This result can then be visualized for decision-makers, used to trigger alerts in

content moderation systems, or serve as evidence in forensic investigations.

13.9.3. Deployment of the Deepfake Detection System

The proposed deepfake detection system was deployed as an interactive web application to make it accessible for real-time testing and user interaction. The application is hosted on Hugging Face Spaces using the Gradio interface, which allows users to upload images or videos and receive deepfake classification results along with Grad-CAM visualizations for explainability.

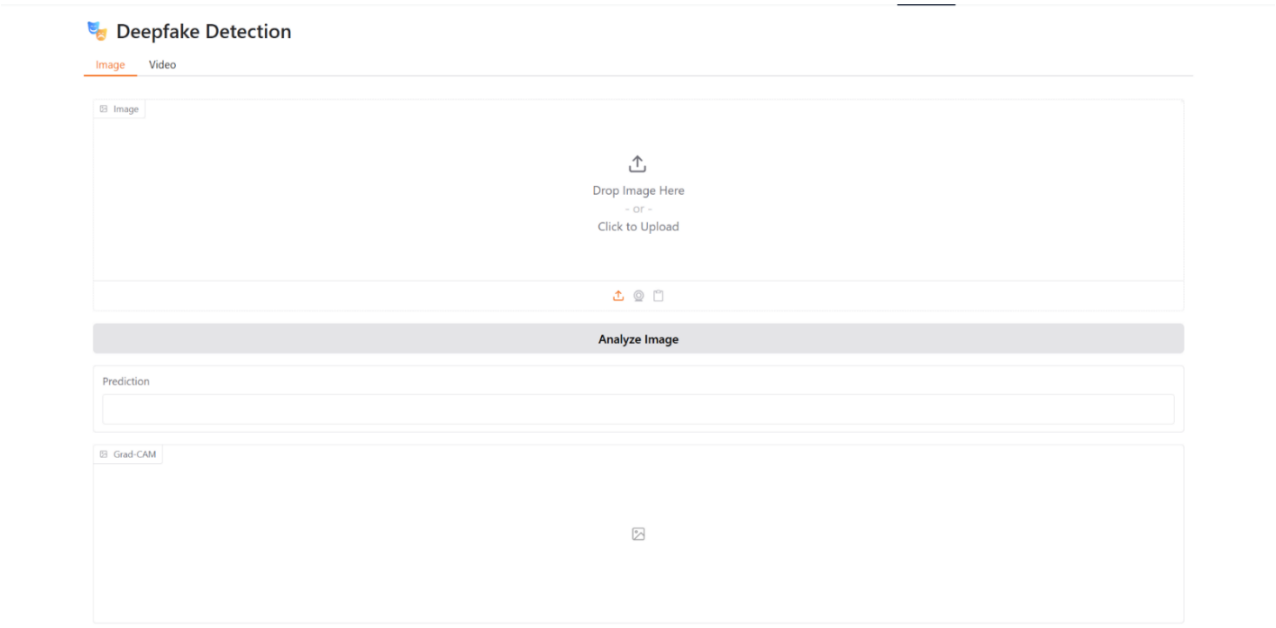

Figure 13.9.3.1: Deepfake Detection (Image Tab) App Deployed on Hugging Face

The interface provides two primary tabs: Image and Video, allowing users to test static images or entire video clips for deepfake characteristics. In the image tab, users can upload a facial image either by dragging and dropping or by clicking to browse from their device. Upon uploading, the user can click on "Analyze Image", which triggers the model to process the image and return the prediction results.

The system then utilizes a deep learning pipeline that typically includes preprocessing (face detection, cropping, and alignment), feature extraction through a Convolutional Neural Network (CNN), and optional sequence modelling using Recurrent Neural Networks (RNN) for video inputs. The final prediction is rendered in the "Prediction" section, which shows whether the input is real or fake, along with a confidence score or probability. Additionally, for enhanced transparency and interpretability, the interface also displays a Grad-CAM (Gradient-weighted Class Activation Mapping) visualization. Grad-CAM helps highlight the specific regions of the image that most influenced the model's decision, thereby making the detection process more explainable for users.

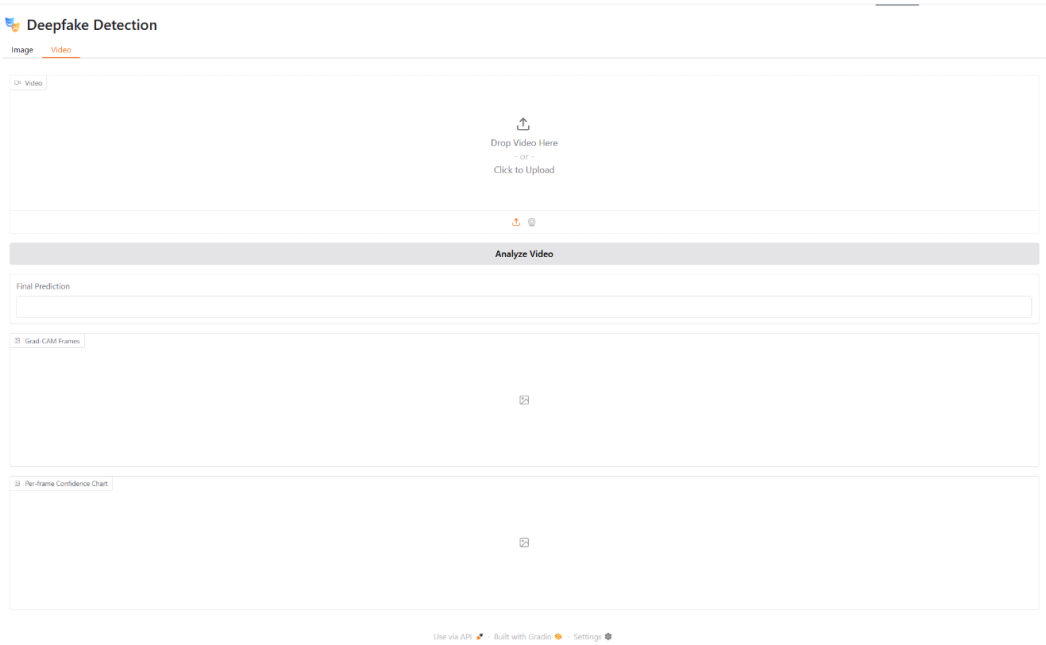

Figure 13.9.3.2: Deepfake Detection (Video Tab) App Deployed on Hugging Face

The video tab of the deepfake detection web application, as displayed in the image, enables users to upload and analyze entire video files for possible synthetic manipulation. This feature is particularly useful for detecting deepfakes in dynamic content, where facial expressions, movements, and temporal inconsistencies are more apparent than in still images. The interface offers a clean and intuitive layout where users can drag and drop a video or select one from their device by clicking the upload area. Once a video is uploaded, the user initiates the detection process by clicking “Analyze Video”.

Internally, the system extracts multiple frames from the uploaded video and applies a series of steps including face detection, cropping, and alignment for each frame. Each aligned frame is passed through a Convolutional Neural Network (CNN) to extract spatial features and then through a Recurrent Neural Network (RNN), such as LSTM, to model the temporal sequence of facial movements. This combination ensures both visual and behavioural inconsistencies are detected effectively. The model then aggregates predictions across all frames to produce a final verdict on the entire video.

The output is presented in three forms. First, the "Final Prediction" box shows the overall probability or percentage that the video is fake. This is a high-level result meant for quick interpretation. Second, the "Grad-CAM Frames" section visualizes the areas of each frame that the model focused on to make its prediction, helping users understand what features led to the classification. Third, the "Per-frame Confidence Chart" graphically displays the fake/real confidence scores for each frame, highlighting variations across the video timeline and offering insight into which moments contributed most to the final result.

This interface transforms a complex deep learning pipeline into a user-friendly tool that supports real-time, interpretable video deepfake analysis. It has practical applications in digital forensics, journalism, law enforcement, and any domain where verifying the authenticity of video content is essential. The design is ideal for non-technical users while still being detailed enough for research and educational purposes.

13.9.3.1. Grad-CAM (Gradient-weighted Class Activation Mapping)

Grad-CAM is a powerful visualization technique used to interpret and explain the decisions made by deep learning models, especially Convolutional Neural Networks (CNNs). In the context of deepfake detection, Grad-CAM helps users understand which parts of the input image or video frame influenced the model's prediction the most.

When a frame from the input video is passed through the CNN, the network extracts various feature maps representing different levels of visual detail. Grad-CAM works by computing the gradients of the output class (real or fake) with respect to the final convolutional layers. These gradients are then used to generate a heatmap that highlights the important regions in the image that contributed to the prediction.

The provided diagram offers a clear and structured visualization of how Grad-CAM works. Grad-CAM is an explainable AI (XAI) technique used in deep learning, particularly with Convolutional Neural Networks (CNNs), to visualize which parts of an input image influenced the model's decision. This is extremely useful in tasks like classification, object detection, and especially in sensitive applications like deepfake detection, medical diagnosis, and autonomous driving, where interpretability is critical.

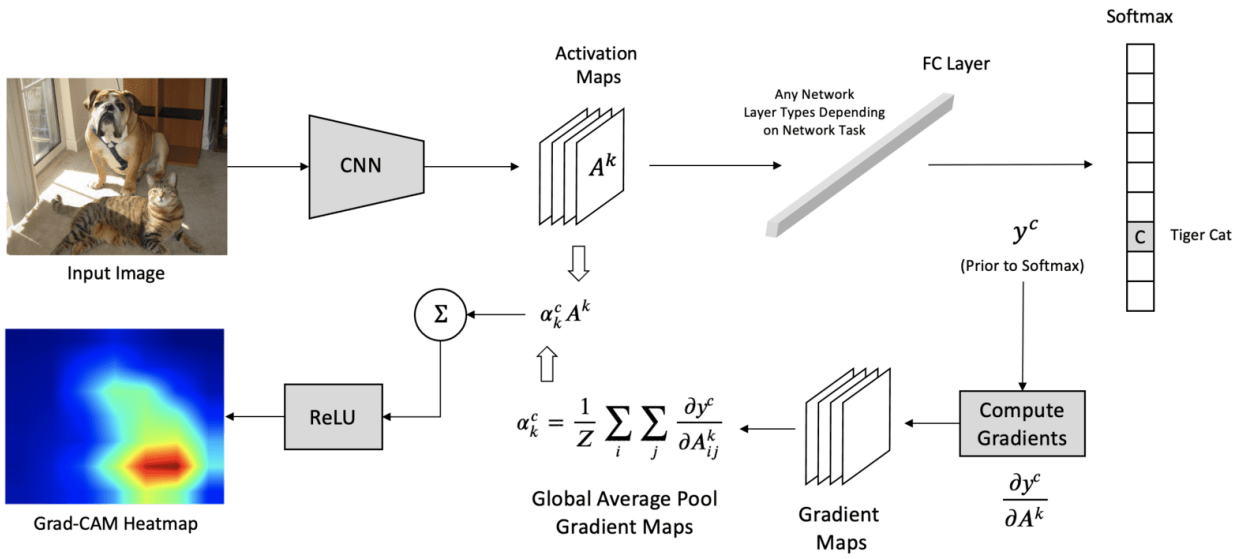

Figure 13.9.3.1.1: Working of Grad-CAM

1. Input Image and CNN Feature Extraction

The process starts with an input image—such as the image of a dog and a cat in the top-left of the diagram. This image is fed into a Convolutional Neural Network (CNN), which processes it through multiple convolutional layers to extract activation maps (feature maps). These maps represent different learned features like edges, textures, or object parts.

2. Forward Pass and Classification Output

The output of the convolutional layers is then passed through fully connected layers (FC Layer) and ultimately reaches a softmax classifier, which gives the final class prediction (e.g., “Tiger Cat” in this example).

3. Backpropagation and Gradient Calculation

To identify which parts of the image were most influential in predicting class c, Grad-CAM computes the gradients with respect to each activation map from the final convolutional layer. These gradients tell us how much each feature in the activation map contributed to the class prediction.

4. Global Average Pooling (GAP)

The gradients are globally averaged over spatial dimensions (i and j) to obtain the importance weights for each feature map.

5. Weighted Sum of Feature Maps

These importance weights are then used to compute a weighted sum of the activation maps.
The ReLU (Rectified Linear Unit) is applied to retain only the features that have a positive influence on the class prediction.

6. Grad-CAM Heatmap Generation

The result of this weighted sum is a heatmap, which is then upsampled to the size of the input image and overlaid on top of it. This Grad-CAM heatmap visually highlights the regions of the image that were most critical in determining the final class label (in this case, the "Tiger Cat").
In the lower-left corner of the diagram, the heatmap shows red/orange regions where the CNN focused most when predicting the target class, and blue regions that were less influential or ignored.

In the web application, after analyzing the video, the Grad-CAM results are shown for multiple keyframes. If the video is classified as fake, the heatmap typically highlights blended facial regions, eyes, mouth, or edges of the face where manipulation artifacts are often found. This makes the system explainable and trustworthy, as it allows users to visually confirm where the model found suspicious patterns.

13.9.3.2. Per-frame Confidence Chart

The Per-frame Confidence Chart is a visual representation of the model's prediction confidence across all frames of a video. For every extracted frame, the model calculates a confidence score—usually a probability value between 0 and 1—indicating how likely that specific frame is manipulated (fake) or genuine (real).

This chart typically appears as a line graph with:

- **X-axis:** representing the frame number or timestamp.
- **Y-axis:** showing the confidence value (e.g., 0 to 1 or 0% to 100%).

It provides insights into the temporal dynamics of the model's decisions. For example:

- A consistently high fake confidence across most frames supports a strong fake classification.
- A mixed pattern may suggest partial manipulation or inconsistencies in detection quality.
- Sudden spikes or drops may indicate frames with prominent artifacts or real facial expressions.

This chart is especially useful for videos where manipulation is not present throughout but only in certain segments. It helps users pinpoint the exact moments where the model detected suspicious content and gives a more nuanced understanding than a single prediction score.

13.9.4. Testing of the Deepfake Detection System

- **Test Case: 1**
  - **Input (Uploaded Media)**
    - **Type:** Image
    - The image captures a memorable moment from our class visit to the Indian Institute of Technology (IIT) Ropar, one of India's premier engineering institutions. The visit was organized as an academic enrichment activity aimed at providing students with exposure to cutting-edge research, campus infrastructure, and the innovation ecosystem at IIT.

Figure 13.9.4.1: Input Image-1 for Deepfake Detection

- **Output (Model Prediction Result)**

Figure 13.9.4.2: Output of Deepfake Detection App for Image-1

The model has classified the image as **real** with a moderate confidence score of **0.58.**

The **Grad-CAM heatmap** provides insight into how the model arrived at its decision. In the visual, warmer colors (red/yellow) highlight regions that had a stronger influence on the model's prediction. In this case, the Grad-CAM focuses on the **upper and side areas** of the image—possibly evaluating the facial and background consistency.

- **Test Case: 2**
  - **Input (Uploaded Media)**
    - **Type:** Image
    - The image displays a clear, vertically oriented portrait of a young man standing outdoors. He is dressed in a navy-blue button-up jacket over a checkered shirt and blue jeans. The background suggests a natural setting with a cemented edge, greenery, and a staircase partially visible. The lighting appears natural, and the subject is centered in the frame with a calm facial expression, directly facing the camera.

Figure 13.9.4.3: Input Image-2 for Deepfake Detection

  - **Output (Model Prediction Result)**

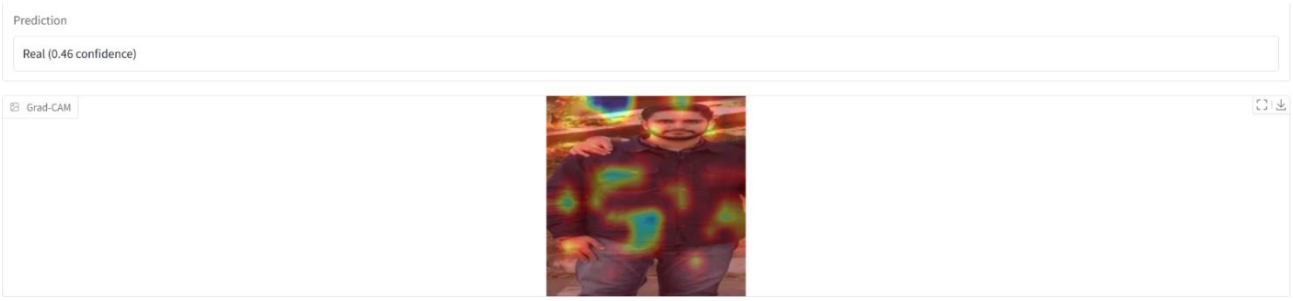

Figure 13.9.4.4: Output of Deepfake Detection App for Image-2

The model predicted the image as **real**, though with a **relatively low confidence of 46%**, which places the prediction close to the decision boundary. This moderate certainty may result from several factors:

- The presence of an external hand on the subject's shoulder could introduce features not typically associated with standard portrait images.
- The uniform texture of clothing and the complex background may make it harder for the model to focus on distinct facial cues.
- Possible compression or resolution variations affecting feature extraction.

The Grad-CAM heatmap indicates that the model focused on multiple regions, including the face, chest, and background areas, which may have diluted the influence of facial authenticity in favour of broader contextual cues. Ideally, for confident predictions, the focus should be mostly on the face.

- **Test Case: 3**
  - **Input (Uploaded Media)**
    - **Type:** Image

This image captures the Indian cricket team celebrating a historic Test series victory. The players and support staff are seen posing together with the trophy, exuding joy and pride. Dressed in official team jerseys sponsored by BYJU'S, the group showcases unity, determination, and national spirit. The moment reflects a significant triumph in Indian cricket history.

Figure 13.9.4.5: Input Image-3 for Deepfake Detection

- **Output (Model Prediction Result)**

Figure 13.9.4.6: Output of Deepfake Detection App for Image-3

The model classified the image as **real**, though with a **low confidence score of 46%**. This suggests uncertainty in the model's prediction, which could be due to:

- The presence of many faces and overlapping elements
- Vibrant colors and complex textures from jerseys, background, and lighting
- Possible trophy reflections or compression artifacts that may confuse the classifier

The Grad-CAM heatmap focuses primarily on the central players holding the trophy,

indicating that these regions were most influential in the model's decision. However, since attention is spread across several areas, the model may have struggled to prioritize one consistent region, reducing overall confidence.

- **Test Case: 4**
  - **Input (Uploaded Media)**
    - **Type:** Image
    - The image shows a professionally composed, full-body portrait of a man standing outdoors. He is wearing a dark navy jacket over a buttoned shirt and blue jeans. The background consists of a slightly elevated grassy area with a paved sidewalk and natural sunlight, suggesting a casual and calm outdoor setting. The subject is directly facing the camera with a composed, neutral expression and relaxed posture. The image quality is high, with sharp details and well-balanced lighting, making it ideal for use in profile documentation, facial recognition applications, or visual demonstrations in machine learning tasks like deepfake detection.

Figure 13.9.4.7: Input Image-4 for Deepfake Detection

  - **Output (Model Prediction Result)**

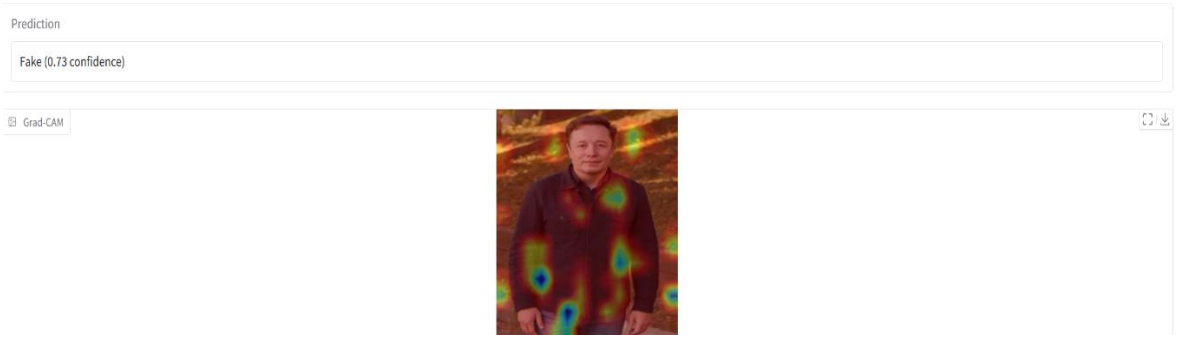

Figure 13.9.4.8: Output of Deepfake Detection App for Image-4

The model has classified the image as **fake** with **high confidence (73%)**. Several factors could contribute to this outcome:

- The image may have synthetic or digitally altered properties (e.g., AI-generated,

deepfake, or edited portrait).

- Subtle inconsistencies in texture, lighting gradients, or sharpness can mislead the model, even if the image appears real to the human eye.
- The Grad-CAM heatmap shows attention across multiple regions—especially the chest, hands, and face—indicating the model identified visual patterns in these areas that matched features commonly found in manipulated images.

Notably, Grad-CAM does not confirm manipulation itself but highlights the regions that influenced the model's prediction the most. In this case, the bright-coloured heat zones suggest strong activation from areas possibly associated with synthetic cues (e.g., overly smooth textures or anomalous shading).

- **Test Case: 5**
  - **Input (Uploaded Media)**
    - **Type:** Image
    - The image features a well-composed portrait of two men standing side by side in an outdoor setting. Both individuals are dressed identically in navy blue jackets and jeans, facing forward with calm, neutral expressions. The background includes a sunlit grassy area, softly blurred to focus attention on the subjects. The lighting is natural and evenly distributed, contributing to a high-quality, professional appearance.

Figure 13.9.4.9: Input Image-5 for Deepfake Detection

- **Output (Model Prediction Result)**

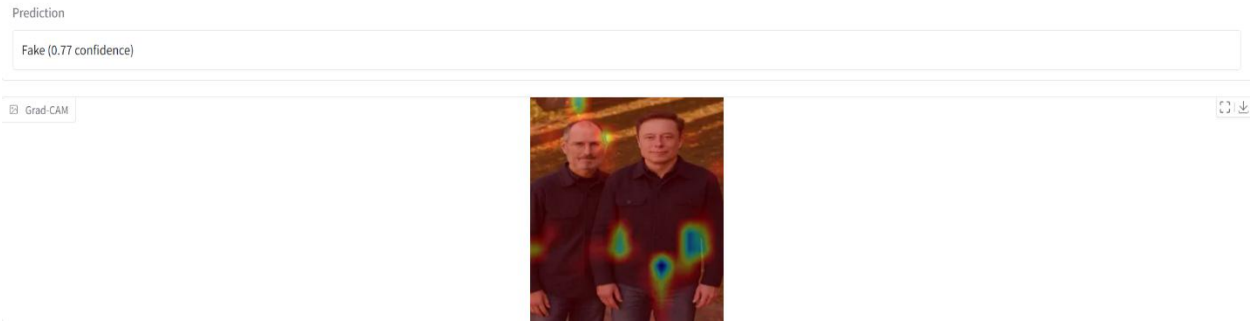

Figure 13.9.4.10: Output of Deepfake Detection App for Image-5

The model has identified the image as **fake with high confidence (0.77)**, indicating strong evidence of synthetic manipulation or AI-generated visual patterns. Several observations support this conclusion:

- The striking visual symmetry—in outfit, pose, and lighting—may resemble patterns often found in AI-generated or manipulated content.
- Skin texture uniformity, lighting smoothness, and unnatural facial harmony can mislead the model into associating the image with deepfake traits.
- The Grad-CAM activation across the torso and background rather than just the face may suggest the model is detecting subtle artifacts or inconsistencies in non-facial features, which are often less convincingly synthesized by generative models.

This suggests the model is not just relying on the face for decision-making but analyzing the entire visual composition, including the hands, clothing texture, and overall symmetry—factors often influenced during synthetic image generation.

- **Test Case: 6**
- **Input (Uploaded Media)**
  - **Type:** Video
  - The uploaded video features a public figure (former U.S. President Barack Obama) in what is likely an AI-generated or altered speech scenario.

- **Output (Model Prediction Result)**

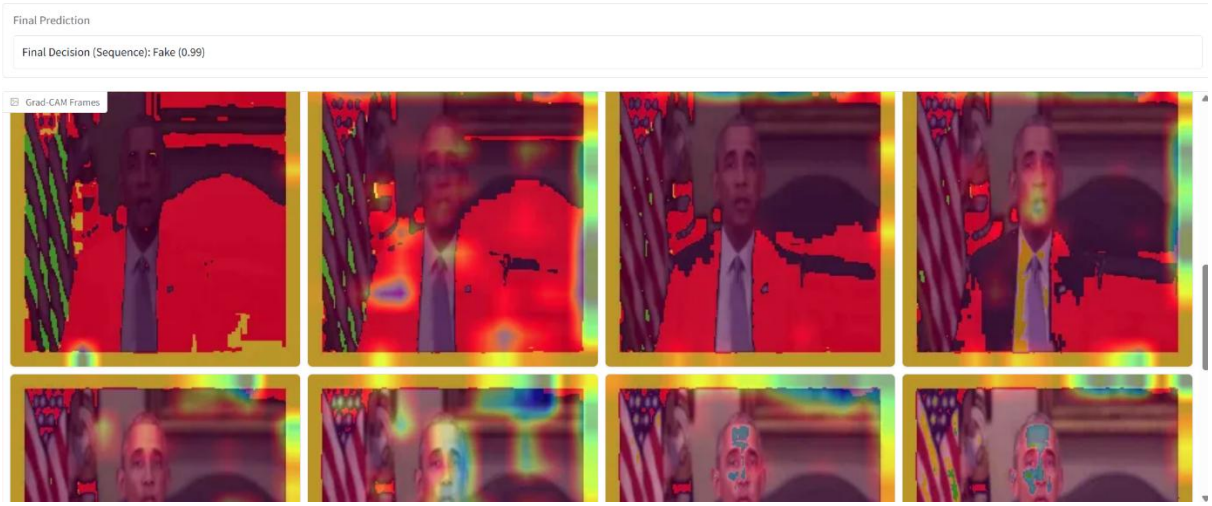

Figure 13.9.4.11: Output of Deepfake Detection App for Video-1

The deepfake detection model **correctly identifies the video as fake with extremely high confidence (99%)**, confirming the system's capability to detect even highly realistic synthetic media. The prediction is based on a combination of:

- Temporal inconsistencies across video frames (e.g., unnatural eye movement, mouth sync)
- Subtle spatial anomalies like inconsistent lighting, blurred edges, or mismatched facial textures
- The Grad-CAM heatmaps shown for each frame highlight regions where the model detected irregularities. In this case, the model's attention (marked in red/yellow) is focused on:
- The face, particularly around the mouth and eyes
- The shoulder and suit region, likely due to rendering artifacts or background interference
- Certain background areas where pixel blending may appear artificial

These activations suggest that the model has learned to spot not only facial manipulation but also global frame-level irregularities often present in GAN-generated or altered videos.

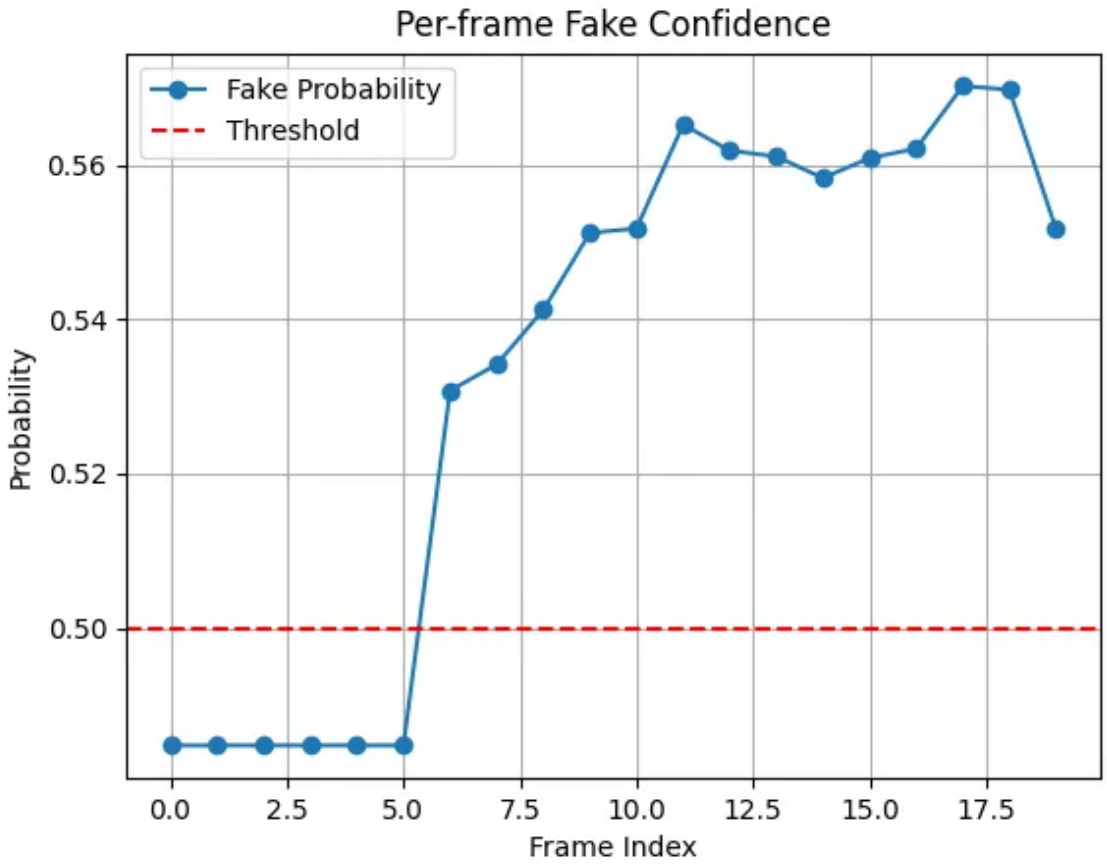

Figure 13.9.4.12: Per-Frame Confidence Graph for Video-1

The per-frame fake confidence graph provides a detailed, frame-by-frame analysis of how the deepfake detection model evaluates the likelihood of manipulation across a video. Each point on the graph corresponds to a frame extracted from the video, with the x-axis representing the frame index and the y-axis indicating the model's confidence that a given frame is fake. A red dashed line at the 0.5 threshold is used to differentiate between real and fake classifications—any value above this line indicates a prediction of "fake," while values below it is considered "real."

In the given graph, the first six frames (indices 0 to 5) maintain a consistent confidence level around 0.48, staying below the threshold, which suggests that the model initially perceives the video content as authentic. However, starting from frame 6, there is a sharp upward shift in fake confidence, and from frame 7 onwards, all frames are classified as fake, with probabilities climbing to values between 0.54 and 0.58. This trend implies a possible transition in the video—where initially real footage becomes synthetically manipulated—or reflects subtle temporal inconsistencies that the model associates with deepfakes, such as facial blending errors, lip-sync mismatches, or unnatural motion.

Such frame-level analysis is crucial for understanding not only whether a video is fake but also when and how strongly the signs of manipulation appear. This adds interpretability and

transparency to the model's decision-making process, especially in scenarios where only parts of a video are altered. The graph thus serves as a valuable tool in temporal deepfake detection, helping to pinpoint specific manipulated segments with visual clarity.

- **Test Case: 7**
  - **Input (Uploaded Media)**
    - **Type:** Video
    - The uploaded video titled "real.mp4" features former U.S. President Donald Trump speaking in what appears to be a natural, unscripted setting. The video showcases typical characteristics of a real, unmanipulated video, including consistent lip-sync, facial expressions, eye movement, and natural lighting. There are no immediate signs of visual or audio artifacts that would suggest tampering or synthetic generation.

  - **Output (Model Prediction Result)**

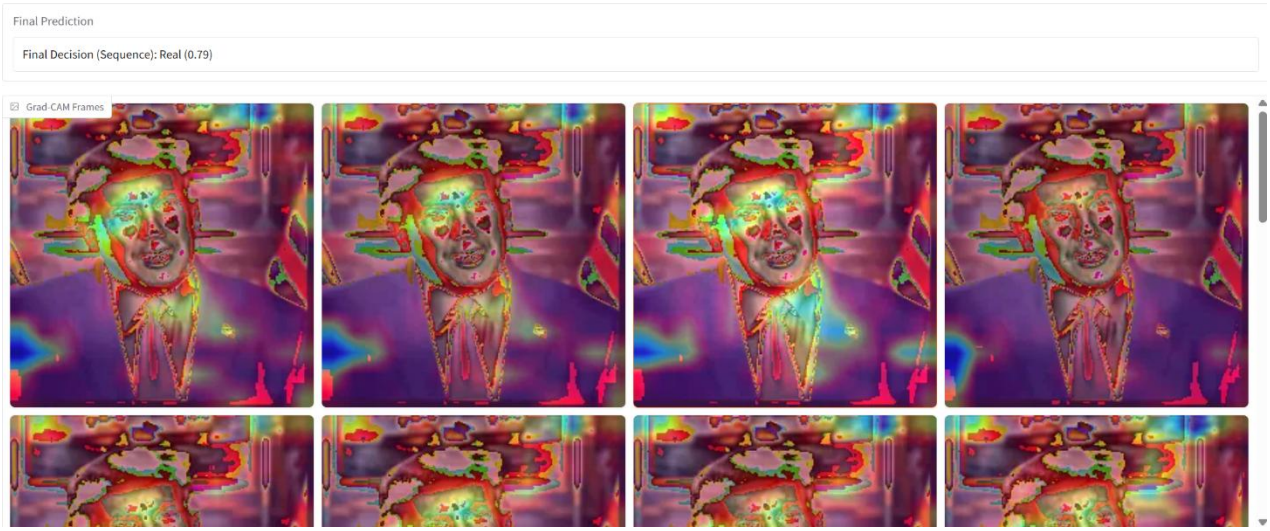

Figure 13.9.4.13: Output of Deepfake Detection App for Video-2

The deepfake detection model correctly identifies the video as **real** with a high confidence score of **79%**, confirming the system's capability to distinguish authentic video content from manipulated or synthetic media. The prediction is based on a combination of:

- Temporal consistency across video frames, such as natural eye movement, smooth lip synchronization, and coherent facial expressions
- Spatial coherence in textures and lighting, with no visible distortions, edge artifacts, or blending inconsistencies that are commonly found in deepfakes
- The Grad-CAM heatmaps shown for each frame highlight regions where the model focused its attention during evaluation. In this case, the model's attention (marked in red and yellow) is concentrated on:

- The face, particularly around the eyes, nose, and mouth, which are key indicators of authenticity in facial motion
- The suit and upper body, which exhibit consistent lighting and texture, supporting the real classification
- The background elements, which appear undistorted and naturally integrated into the scene, unlike synthetic backdrops

These activations suggest that the model has confidently recognized human-consistent facial features and environmental integrity, and has not detected signs of manipulation such as pixel mismatches, flickering, or visual discontinuities. This reinforces the model's strength in real-world verification, particularly in high-stakes media involving public figures.

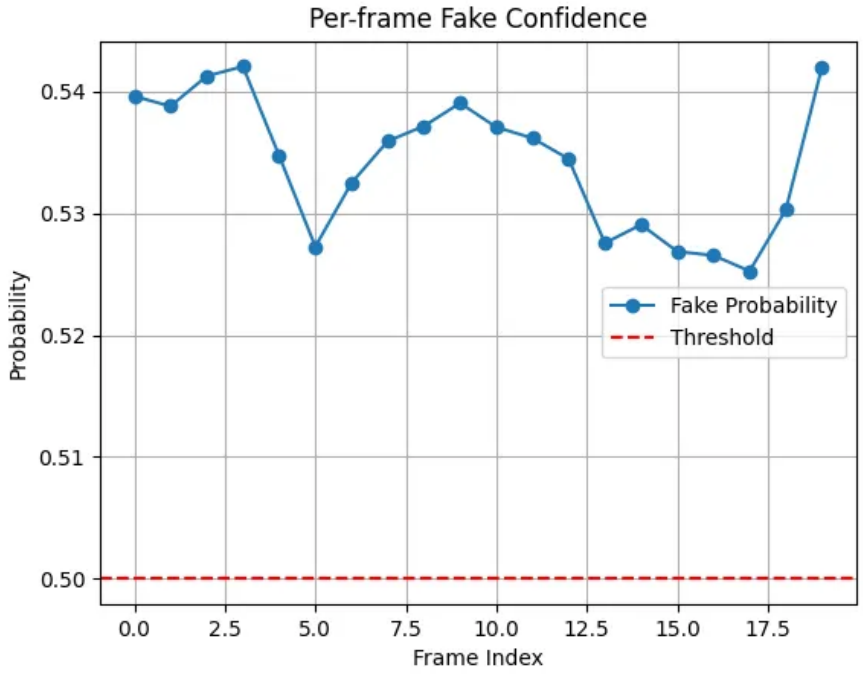

Figure 13.9.4.14: Per-Frame Confidence Graph for Video-2

The deepfake detection model analyzes the video at a frame level and predicts each frame as fake with consistently moderate confidence, fluctuating around the 0.53 to 0.545 range. The red dashed line represents the 0.50 decision threshold, above which frames are classified as fake. Since all frame-level probabilities are above the threshold, the model considers the entire video suspicious despite not showing extremely high confidence. This implies the model detects subtle and persistent indicators of manipulation across the entire sequence.

The prediction is based on a combination of:

- Mild temporal inconsistencies across video frames, such as slight irregularities in facial expressions or motion smoothness
- Subtle spatial anomalies like minor inconsistencies in texture, lighting gradients, or

blending at facial boundaries

The per-frame graph indicates that while no single frame spikes to high fake confidence (e.g., >0.60), the collective evidence from all frames being just over the threshold suggests the model detects signs of low-level tampering or synthetic enhancement.

This type of result typically reflects videos that may have been partially modified or generated using high-quality AI models that reduce but don't eliminate detectable artifacts. It also showcases the model's sensitivity to consistent, small deviations from natural video patterns, even when the manipulation is hard for humans to spot.

- **Test Case: 8**
  - **Input (Uploaded Media)**
    - **Type:** Video
    - The uploaded video titled "deepfake.mp4" features former U.S. President Donald Trump in what appears to be a formal speaking scenario. Although the video visually resembles real footage at first glance, it is a deepfake, meaning the facial expressions, speech synchronization, or visual features have been synthetically altered using AI-based generation techniques such as GANs or face-swapping models.

  - **Output (Model Prediction Result)**

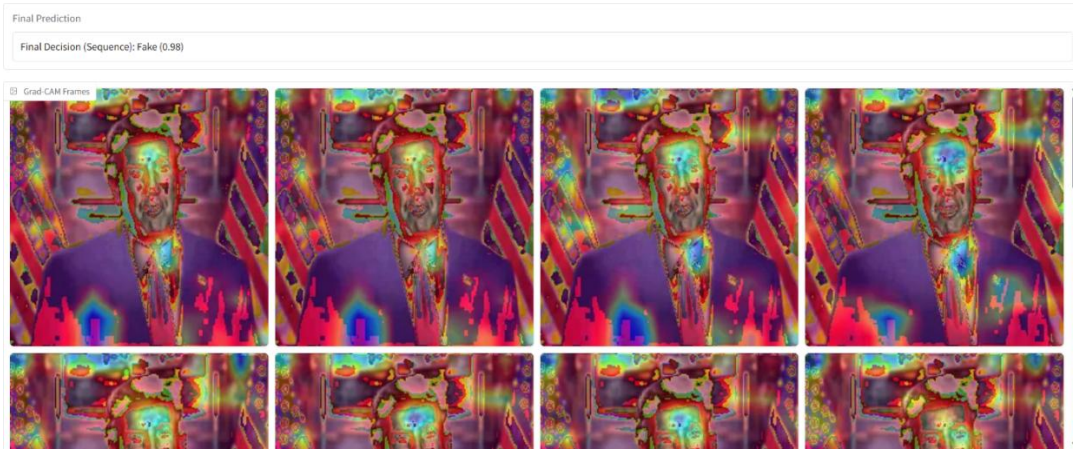

Figure 13.9.4.15: Output of Deepfake Detection App for Video-3

The deepfake detection model correctly identifies the video as **fake** with extremely high confidence (**98%**), confirming the system's capability to detect even highly realistic synthetic media. The prediction is based on a combination of:

- Temporal inconsistencies across video frames, such as unnatural blinking, irregular lip-sync, and subtle motion jittering that breaks the flow of natural expressions
- Subtle spatial anomalies, including inconsistent lighting across facial regions, texture irregularities, or blending errors around the edges of the face, jacket, or background

The Grad-CAM heatmaps displayed for each frame highlight the regions where the model focused its attention during classification. In this case, the model's attention (shown in red and yellow) is primarily focused on:

- The face, especially around the eyes, nose, and mouth, where manipulation artifacts such as expression misalignment or texture smoothing typically appear
- The upper torso and suit region, possibly due to background blending issues or digital clothing overlays
- Portions of the background, which may exhibit unnatural patterns or color transitions introduced during compositing

These activations suggest that the model has learned to recognize not only facial manipulation but also broader frame-level irregularities commonly found in GAN-generated or AI-altered videos. This result demonstrates the model's high sensitivity and accuracy in flagging deepfake content, even when it visually appears realistic to human viewers.

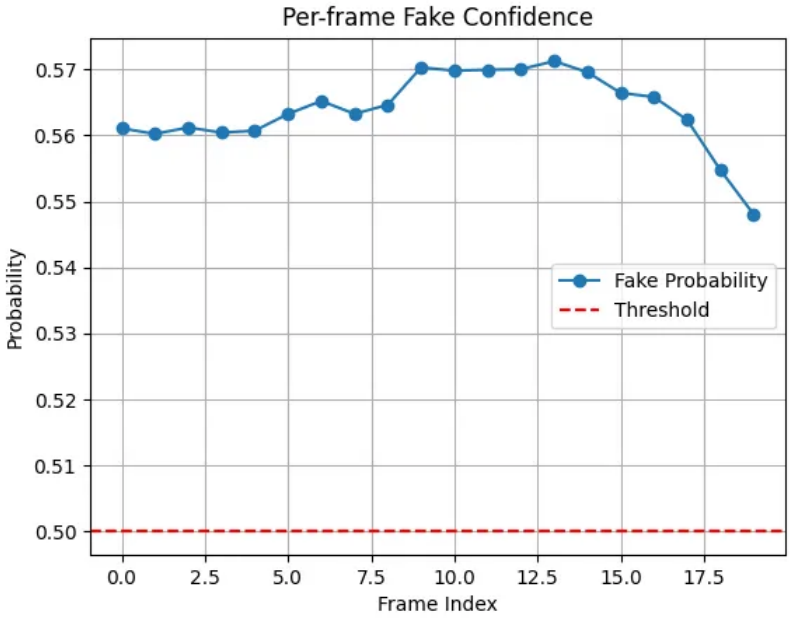

Figure 13.9.4.16: Per-Frame Confidence Graph for Video-3

The deepfake detection model evaluates each frame of the video and predicts them as **fake** with **consistently high confidence**, maintaining probabilities well above the 0.50 threshold. As shown in the graph, the fake probability starts at around **0.56** and gradually rises to a peak of about 0.572, before slightly dropping toward the final frames. Despite this minor dip, all frames remain confidently above the red threshold line (0.50), indicating a strong belief from the model that the content is manipulated.

This prediction is based on a combination of:

Temporal consistency in detection, where nearly all frames exhibit features the model associates with deepfake content. This could include subtle mismatches in lip movement, eye blinking rhythm, or facial transition smoothness across frames.

Spatial cues such as texture anomalies, edge blending errors, or unnatural lighting patterns—particularly around facial landmarks like the eyes, mouth, and jawline.

The consistently elevated prediction scores across nearly all frames suggest the manipulation is present throughout the video, not just isolated to a few segments. This type of graph indicates a highly probable full-sequence deepfake, where synthetic generation techniques have been applied uniformly.

Such stable detection performance reinforces the model's effectiveness in identifying well-generated but still detectable synthetic media, and highlights the value of per-frame probability graphs for detailed forensic analysis.

13.9.5. Hugging Face Link for Deepfake Detection

https://huggingface.co/spaces/Prabhsimran09/Deepfake

Deepfake Detection

13.10. TexifySymPy Solver: AI-Driven OCR and Symbolic Math Engine

TexifySympySolver is a powerful AI-driven web application that seamlessly integrates Optical Character Recognition (OCR) using Texify with symbolic computation using SymPy. This system is designed to recognize handwritten or printed mathematical expressions, convert them into LaTeX, and symbolically solve them with step-by-step breakdowns. It empowers learners, educators, and researchers to interact with mathematical problems visually or via code-based inputs.

13.10.1. Texify

Texify is an open-source deep learning-based Optical Character Recognition (OCR) system specifically designed for recognizing and converting mathematical expressions from images into LaTeX format. Unlike traditional OCR tools that primarily focus on plain text, Texify is tailored to the unique visual structure and complexity of mathematical notation, including fractions, superscripts, subscripts, radicals, integrals, summations, and multi-line expressions. It uses a Transformer-based encoder-decoder architecture, similar to those used in image captioning tasks, where the image is first passed through a Convolutional Neural Network (CNN) (often a ResNet backbone) to extract visual features. These features are then processed by a Transformer decoder, which sequentially generates LaTeX tokens corresponding to the recognized mathematical structure.

Texify's strength lies in its ability to handle handwritten, printed, and mixed-style equations from scanned documents, whiteboards, or notebooks, making it incredibly valuable for educational tools, math solvers, and digital archiving. The model is typically trained on large-scale datasets such as Im2LaTeX-100K or CROHME (Competition on Recognition of Online Handwritten Mathematical Expressions), which provide ground truth LaTeX labels for diverse mathematical expressions. The output of Texify is a LaTeX string, which can then be parsed by tools like SymPy for symbolic manipulation or rendered directly using MathJax for clean display.

Because of its high accuracy and open-source nature, Texify is widely used in academic, research, and developer communities as a robust LaTeX OCR engine. In the TexifySympySolver project, Texify plays the crucial role of bridging the visual and symbolic world—allowing users to snap an image of a math problem and immediately translate it into a symbolic format ready for computation.

13.10.2. SymPy

SymPy (short for *Symbolic Python*) is a powerful, open-source Python library for symbolic mathematics. Unlike numerical libraries such as NumPy, which work with approximated

floating-point numbers, SymPy performs exact symbolic computations — similar to what a human might do when solving math problems by hand. Built entirely in pure Python, SymPy requires no external dependencies and is extremely portable, making it ideal for integration into educational apps, math solvers, web tools, and research systems.

SymPy provides a comprehensive suite of capabilities including algebraic simplification, equation solving, differentiation, integration, limits, series expansion, matrix operations, logic simplification, and calculus, among others. For example, a quadratic equation like $x^2 + 5\,x + 6 = 0$ can be solved symbolically with solve(...), producing exact roots rather than numerical approximations. SymPy also includes a powerful LaTeX rendering engine, which makes it easy to display mathematical expressions in a readable and professional format using latex().

One of SymPy's standout features is its human-readable API. Expressions like $\frac{d}{dx}\sin x^2$ or $\int_{-oo}^{oo} e^{-x^2}\,dx$ are both intuitive and expressive, enabling users to quickly write and manipulate complex mathematical formulas. Under the hood, SymPy represents every mathematical expression as a tree of symbolic objects (e.g., Add, Mul, Pow, etc.), which allows for deep introspection, transformation, and simplification.

In the TexifySympySolver project, SymPy plays the role of the mathematical engine. Once an expression is recognized (via OCR or manual input), SymPy takes over to interpret the expression, simplify it, solve it symbolically, and optionally render it in LaTeX for clean presentation. It also ensures that both exact solutions and approximate numerical values are accessible to users, bridging the gap between pure math theory and practical computation.

13.10.3. System Workflow and Functional Architecture

The TexifySymPy Solver is a dual-mode artificial intelligence system that integrates optical character recognition (OCR) with symbolic mathematical computation to provide accurate and detailed solutions to a wide range of mathematical expressions. The workflow begins with two possible inputs: an image or direct manual input. On the left path of the flowchart, users can upload an image containing a handwritten or printed mathematical expression. This image is processed through Texify OCR, a deep learning-based model trained specifically to recognize mathematical notation. Texify scans the image and converts the content into LaTeX code, which is a standardized typesetting language used to represent mathematical formulas. This LaTeX string captures the precise structure and symbols of the original input and forms the foundation for symbolic computation.

Simultaneously, the right path of the flowchart supports manual input, where users can enter

equations or expressions using SymPy Pythonic syntax. This bypasses the need for OCR and is particularly useful for users who prefer keyboard entry or are familiar with Python's mathematical libraries. These manual expressions are sent directly for evaluation, utilizing Python's eval() function within a safe sandbox environment that grants access to only math-related operations and variables like x, y, diff, integrate, and solve.

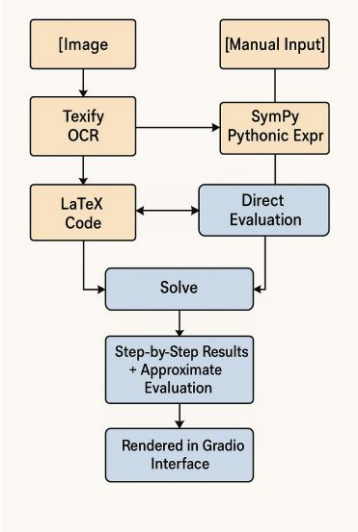

Figure 13.10.3: Architecture of TexifySymPy Solver App Deployed on Hugging Face

Once either the OCR-generated LaTeX code or the manually entered Pythonic expression is available, the next step is to interpret and parse the expression. For LaTeX input, the system uses SymPy's LaTeX parser to convert the string into a symbolic expression that can be manipulated algebraically. If the expression includes integrals with limits, the bounds are extracted using regular expressions to ensure accurate evaluation. Similarly, direct expressions are evaluated without transformation if already in Pythonic form.

Following parsing, the system determines the type of mathematical operation required. If the expression is an integral, the integrate() function is applied; if it's a derivative, diff() is used. Equations with = symbols invoke solve(), and general expressions are simplified or expanded as needed using simplify() and expand() respectively. This process is centralized under the "Solve" block in the diagram, where all symbolic computations converge.

Once solved, the results are not simply returned as raw outputs. Instead, the system generates step-by-step explanations, breaking down how the solution was derived. This pedagogical feature enhances understanding, particularly for students. It also includes approximate numerical evaluations using SymPy's evalf() method for cases where symbolic results are not intuitive. These results, formatted in LaTeX for readability, are then passed to the final stage.

The final output is rendered in a clean, user-friendly Gradio interface. This web-based platform

presents the original expression, its parsed form, the solution steps, and the final result, all in a visually intuitive manner. Users can edit LaTeX, re-run solutions, or test additional problems using pre-filled examples. The integration of Gradio makes the system accessible on browsers without requiring local installations, making it a powerful educational tool.

13.10.4. Deployment of TexifySymPy Solver

The TexifySymPy Solver presents a modern, interactive web-based interface that seamlessly bridges the gap between handwritten mathematical content and symbolic computational solving. The interface is built using Gradio, a Python library that simplifies the deployment of machine learning models through clean, user-friendly web applications. The layout consists of two primary tabs—Image OCR Solver and Manual Equation Solver—clearly indicating dual functionality: one for image-based input processed through OCR, and the other for direct symbolic equation input. This bifurcation enhances usability, catering to both casual users who prefer writing equations and advanced users familiar with symbolic programming using SymPy.

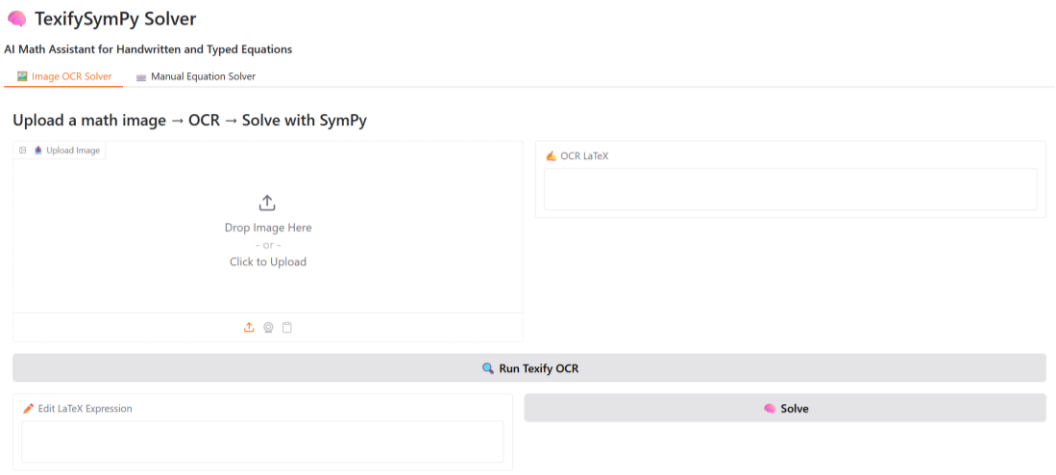

Figure 13.10.4.1: TexifySymPy Solver App (Image OCR Solver Tab) Deployed on Hugging Face

In the Image OCR Solver tab, the user is prompted to upload an image containing handwritten or printed mathematical content. This image is then passed through Texify, a specialized OCR engine trained on mathematical notation. The model converts the contents of the image into LaTeX, a precise typesetting language for mathematical expressions. The recognized LaTeX code is automatically populated in the "OCR LaTeX" textbox, allowing the user to view the interpreted expression. For quality control or further editing, this LaTeX output can be modified in a dedicated "Edit LaTeX Expression" field. The ability to view and edit the OCR result ensures robustness and transparency, particularly when dealing with complex or messy handwritten inputs.

Once the expression has been finalized in LaTeX form, users can click the “Solve” button, which triggers the symbolic solving engine powered by SymPy. This Python library is known for its capabilities in symbolic mathematics, supporting operations such as simplification, differentiation, integration, and equation solving. The backend logic converts the LaTeX string into a SymPy-compatible symbolic expression using the parse_latex() function. Based on the parsed structure—whether it contains equations, integrals, derivatives, or algebraic expressions—the solver intelligently applies the corresponding SymPy method (e.g., solve(), diff(), integrate(), or simplify()).

The solver not only returns the final result but also provides step-by-step explanations of the mathematical process involved. These steps are rendered using LaTeX syntax, visually displaying each transformation or operation in a mathematically expressive format. This educational feature is particularly valuable for students seeking to understand not just the answer but also the logical progression of steps. The interface ensures that both the symbolic and approximate numerical results are displayed, allowing users to cross-verify and gain conceptual clarity.

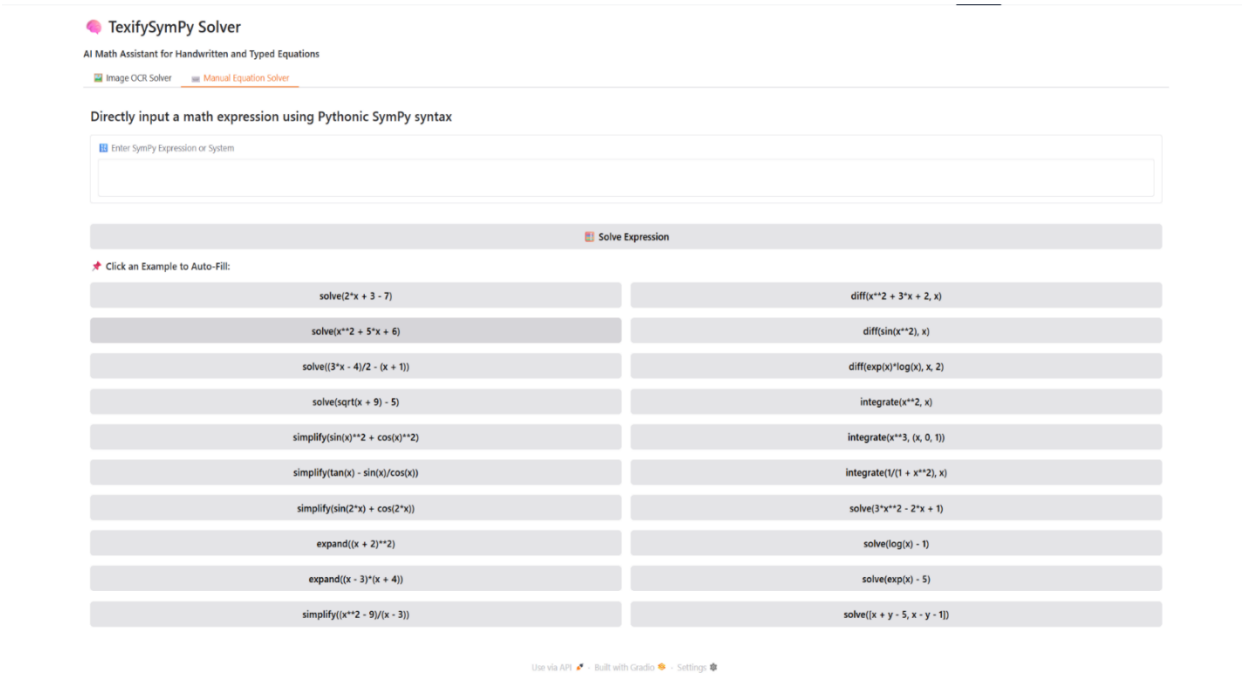

Figure 13.10.4.2: TexifySymPy Solver App (Manual Equation Solver Tab) Deployed on Hugging Face

The Manual Equation Solver tab in the TexifySymPy Solver interface is designed to provide a streamlined and user-friendly experience for solving mathematical expressions directly using SymPy’s Pythonic syntax. This mode targets users who are comfortable with symbolic mathematics in Python and prefer typing equations manually over image-based OCR. The layout centers around a single input box labeled “Enter SymPy Expression or System”, which allows users to type any valid mathematical command supported by the SymPy library. These

expressions may include algebraic equations, calculus operations such as differentiation and integration, trigonometric simplifications, and even systems of equations.

What makes this interface especially accessible is the inclusion of a section labeled “Click an Example to Auto-Fill”, which lists a wide variety of pre-configured example expressions. These examples act as both a reference and a quick-start tool, allowing users to populate the input box with complex expressions by simply clicking a button. The examples cover an impressive range of mathematical problems — from basic algebraic solutions like solve($2x + 3 - 7$) and quadratic equations, to more advanced symbolic tasks such as diff(exp(x)*log(x), x, 2) and integrate(1/(1 + x**2), x). This variety ensures that users can explore and understand the full scope of what SymPy can handle, while also making it easier to test the system’s capabilities without requiring deep programming knowledge.

Once an expression is entered, the “Solve Expression” button triggers the symbolic computation engine. This engine evaluates the expression within a secure, sandboxed environment where only mathematical operations are permitted. For instance, the solve() function can be used to compute the roots of equations, diff() can calculate derivatives, integrate() handles both definite and indefinite integrals, and simplify() reduces complex expressions to simpler forms. Importantly, the system also supports solving simultaneous equations, such as solve([x + y - 5, x - y - 1]), making it suitable for both single-variable and multivariable problem sets.

The results are rendered in LaTeX format within the interface, making the mathematical output visually clear and aesthetically aligned with traditional textbook notation. This tab complements the OCR-based image solver by giving full control to users who want to input well-formed SymPy syntax directly. Whether for learning, testing, or verification purposes, the manual input system enhances the flexibility and power of the overall application. It transforms the TexifySymPy Solver into not just an OCR-powered tool, but a comprehensive math assistant capable of addressing both visual and textual problem-solving needs.

13.10.5. Testing of TexifySymPy Solver using Manual Equation Solver

- Example 1:

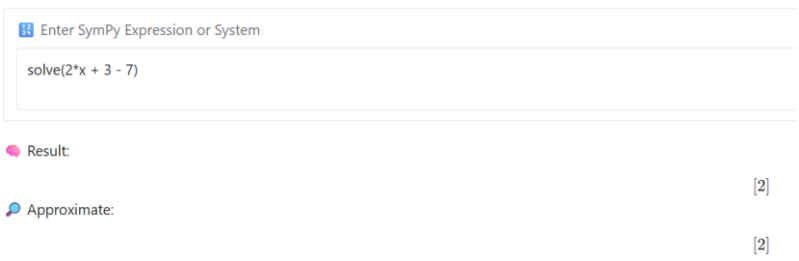

Figure 13.10.5.1: Example-1 of Manual Equation Solver

The provided image demonstrates a successful test of the manual input functionality within the TexifySymPy Solver interface, specifically through the use of a direct symbolic expression. In this case, the user inputs a simple algebraic equation, solve($2x + 3 - 7$), into the SymPy expression field. This equation represents a linear equation of the form $ax + b = c$, which is straightforward to solve using symbolic computation. Upon clicking the Solve Expression button, the system correctly interprets the input using the SymPy library's solve() function, resulting in the solution x = 2.

- Example 2:

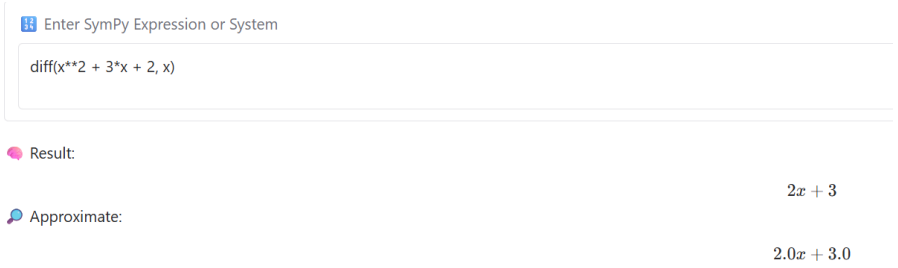

Figure 13.10.5.2: Example-2 of Manual Equation Solver

The image illustrates the successful execution of a differentiation operation using the manual input feature of the TexifySymPy Solver. The expression entered by the user is $\frac{d}{dx}(x^2 + 3x = 2)$ which requests the first derivative of a quadratic polynomial with respect to the variable x. This is a foundational calculus operation where the solver, powered by the SymPy symbolic mathematics library, computes the derivative analytically.

Upon submission, the solver correctly processes the expression and returns the exact symbolic result as 2x + 3. This matches the expected outcome derived from the standard rules of differentiation—where the derivative of x^2 is 2x, and the derivative of 3x is 3, while the constant term 2 vanishes. In addition to the symbolic result, the system also provides an approximate numeric representation as 2.0x + 3.0, which, while numerically identical in this context, demonstrates the solver's capability to handle both exact and floating-point forms.

- Example 3:

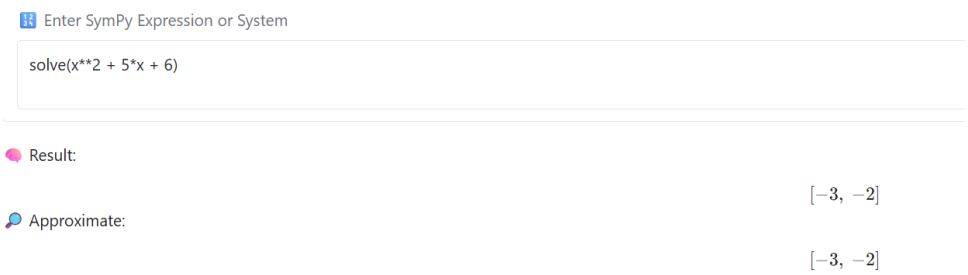

Figure 13.10.5.3: Example-3 of Manual Equation Solver

The image displays a successful test case of the TexifySymPy Solver's manual expression input, where the user enters a quadratic equation in Pythonic SymPy syntax: solve($x^2 + 5x + 6$). The solver uses SymPy's solve() function to analyze the expression and calculate its exact roots.
The symbolic result returned is [-3, -2], which correctly corresponds to the factors of the equation (x+3) (x+2) = 0. These roots are also displayed as the approximate output, confirming the solver's consistency in both symbolic and numeric interpretation. Since the roots are rational numbers, the approximate output is numerically identical to the exact result.

- Example 4:

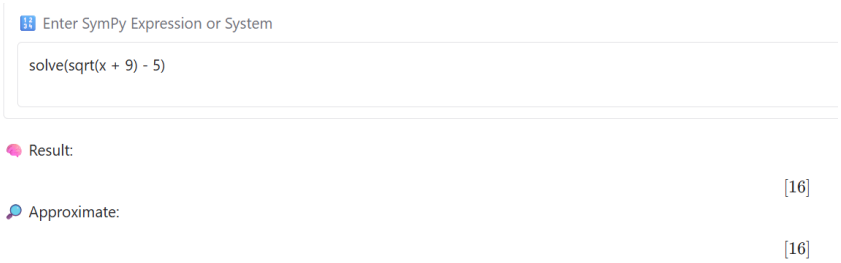

Figure 13.10.5.4: Example-4 of Manual Equation Solver

The image demonstrates the use of the manual SymPy input interface in the TexifySymPy Solver to solve a nonlinear equation involving a square root. The user inputs the expression solve($\sqrt{x+9} - 5$). This is a typical algebraic equation that requires isolating the square root and then squaring both sides to eliminate the radical.
The solver processes the expression using SymPy's symbolic solve() function and correctly returns the result [16], indicating that when x=16, the equation is satisfied. The system also displays the approximate value, which matches the exact result, confirming that the solution is a precise integer. The problem-solving workflow is executed flawlessly: the solver interprets the syntax, simplifies the equation, and returns the valid solution.

- Example 5:

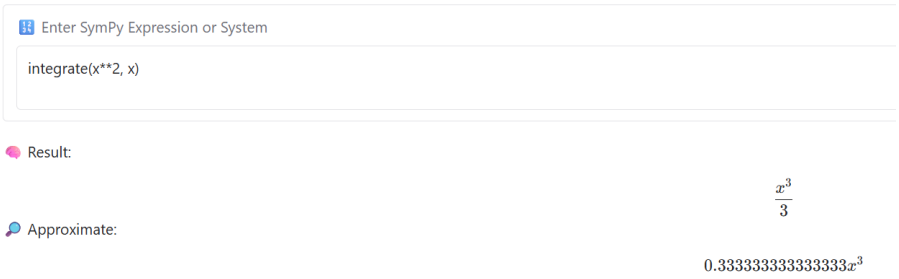

Figure 13.10.5.5: Example-5 of Manual Equation Solver

The image shows the execution of an indefinite integration task using the manual expression interface in the TexifySymPy Solver. The user has entered the input integrate(x**2, x), which

instructs the system to compute the integral of the function x**2 with respect to the variable x. Upon processing the expression, the solver correctly returns the symbolic result $\frac{x^3}{3}$, reflecting the application of the power rule for integration. The solver also provides the approximate numeric equivalent, which is displayed as $0.333333x^3$. While the numeric form may be useful in practical estimations, the symbolic form retains mathematical precision, which is crucial for formal problem-solving and educational clarity.

- Example 6:

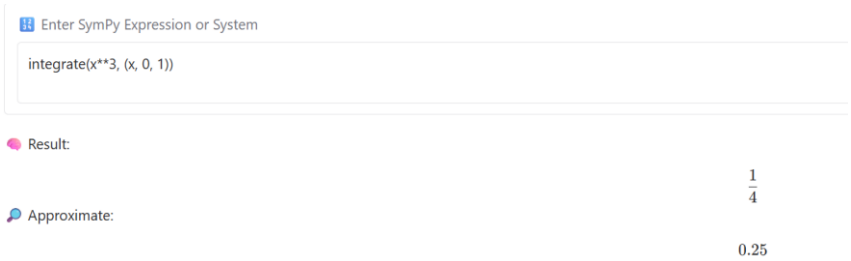

Figure 13.10.5.6: Example-6 of Manual Equation Solver

The image illustrates the successful computation of a definite integral using the manual input feature of the TexifySymPy Solver. The expression provided by the user is integrate(x**3, (x, 0, 1)), which asks the system to compute the integral of the function x**3 over the interval from x=0 to x=1. This type of input represents a fundamental operation in calculus where the area under a curve is evaluated within specific bounds.

Upon evaluation, the solver correctly returns the symbolic result $\frac{1}{4}$, which is the exact value of the definite integral. Alongside this, it also displays the approximate numeric result as 0.25, confirming the accuracy of the computation. The system's ability to return both symbolic and decimal formats offers clarity and reinforces the correctness of the solution.

- Example 7:

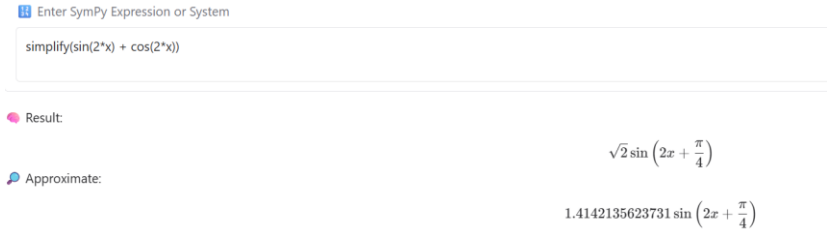

Figure 13.10.5.7: Example-7 of Manual Equation Solver

The image shows the use of the simplify function in the manual expression input section of the TexifySymPy Solver, where the user has entered the trigonometric expression simplify(sin(2*x) + cos(2*x)). This is a classic identity-based simplification problem that combines sine and cosine

functions with the same frequency. The purpose of such a simplification is to express the combination in a more compact or standard trigonometric form.

- Example 8:

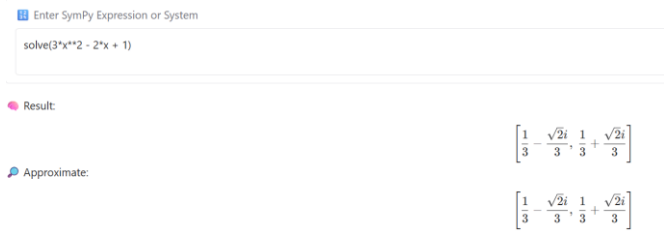

Figure 13.10.5.8: Example-8 of Manual Equation Solver

The image shows the solution of a quadratic equation with complex roots using the manual input interface of the TexifySymPy Solver. The user inputs solve(3*x**2 - 2*x + 1), which represents the standard quadratic equation $3x^2 - 2x + 1 = 0$. This equation has two complex (imaginary) roots.

- Example 9:

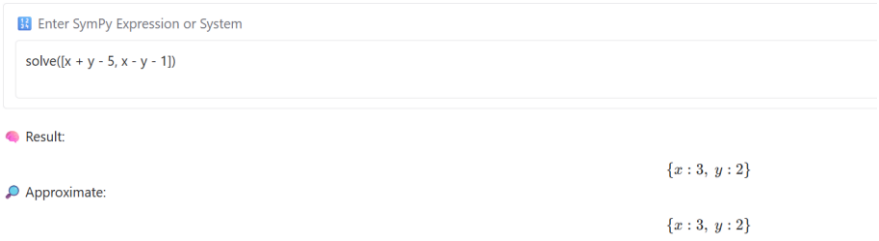

Figure 13.10.5.9: Example-9 of Manual Equation Solver

The image illustrates the use of the TexifySymPy Solver's manual input interface to solve a system of linear equations involving two variables. The user inputs the command solve([x + y - 5, x - y - 1]), which represents a pair of linear equations:

$$x + y - 5$$

$$x - y - 1$$

This system is solvable using basic algebraic techniques such as substitution or elimination. The solver internally applies symbolic methods to derive the values of the variables that simultaneously satisfy both equations. The symbolic result returned by the solver is {x: 3, y: 2}, indicating that x=3 y=2 is the unique solution to the system. The approximate result matches exactly, since both values are integers and no further decimal evaluation is needed.

- Example 10:

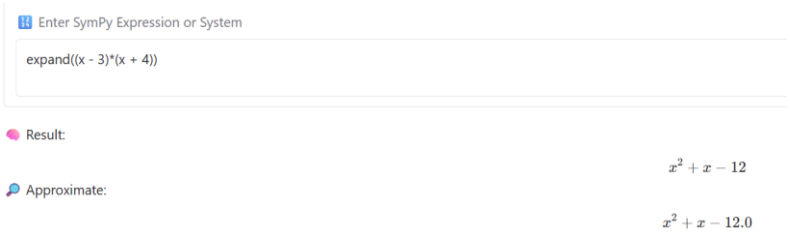

Figure 13.10.5.10: Example-10 of Manual Equation Solver

The image demonstrates the use of the expand() function within the manual SymPy input interface of the TexifySymPy Solver, where the user inputs the algebraic expression expand((x - 3)*(x + 4)). This input requests the system to perform polynomial expansion by multiplying the two binomial expressions. Such expansion is a fundamental concept in algebra, often used to simplify expressions and prepare them for further operations like factoring or solving.

Upon processing, the solver returns the symbolic result $x^2 + x - 12$.

Additionally, the solver provides the approximate result, where the constant term is rendered in floating-point format. This dual representation showcases the solver's ability to handle both exact symbolic and decimal computations, ensuring clarity for users who may prefer numerical output.

13.10.6. Testing of TexifySymPy Solver using Image OCR Solver

- Example 1:

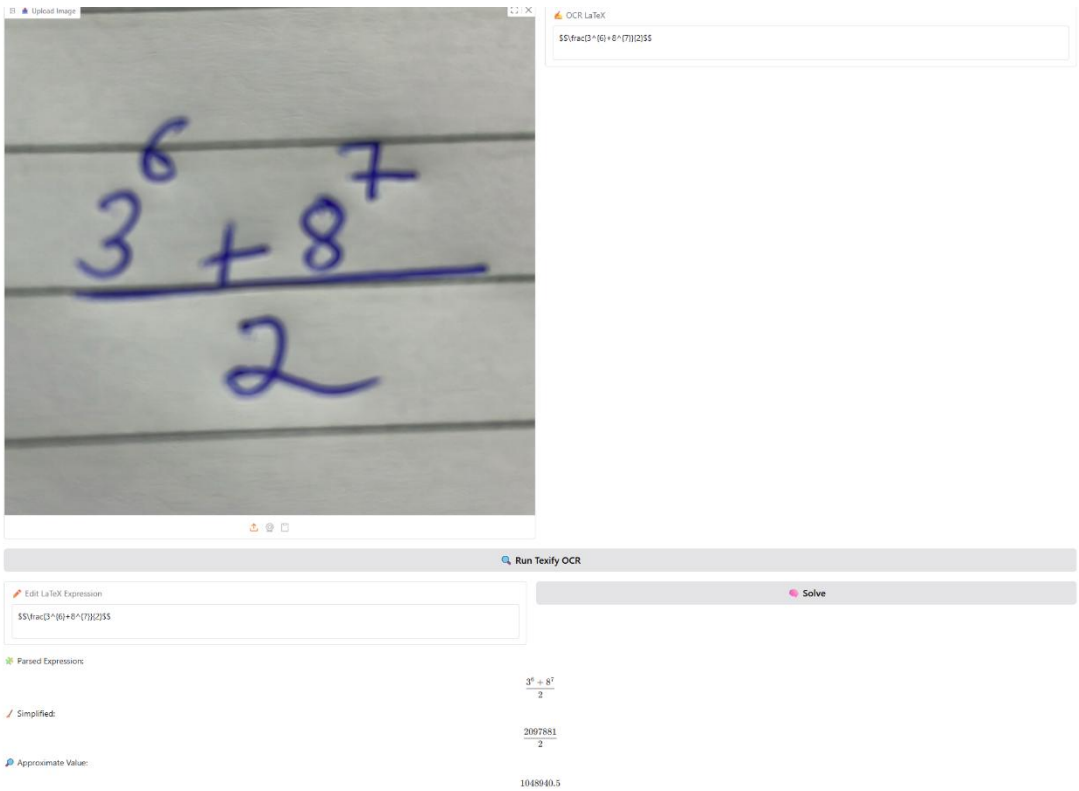

Figure 13.10.6.1: Example-1 of Image OCR Solver

The mathematical expression shown in the image represents the evaluation of an arithmetic operation involving exponents. It is structured as:

$$\frac{3^6 + 8^7}{2}$$

The full expression can be written in LaTeX as:

$$\frac{3^{6} + 8^{7}}{2}$$

This expression involves both exponentiation and division. The term 3^6 means raising 3 to the power of 6, which equals 729, and 8^7 means raising 8 to the power of 7, which equals 2,097,152. Adding these together yields 2,097,881. Dividing this sum by 2 gives the final result of 1,048,940.5.

- Example 2:

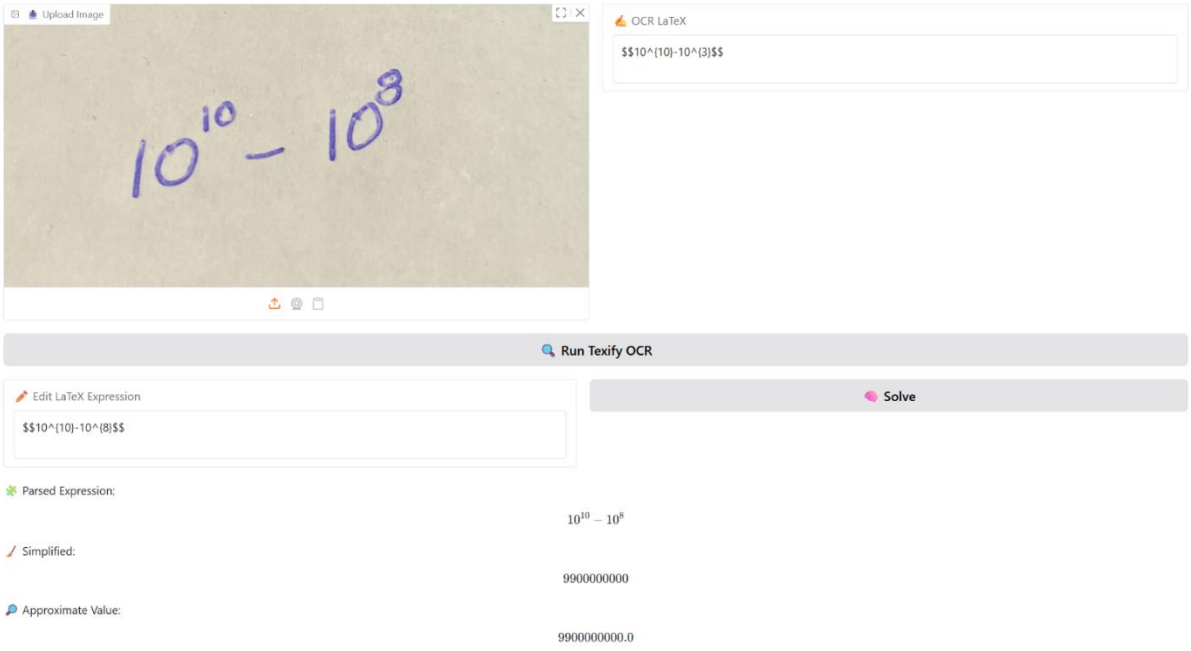

Figure 13.10.6.2: Example-2 of Image OCR Solver

The expression shown in the image represents a subtraction of two exponential numbers. Specifically, the problem is:

$$10^{10} - 10^{8}$$

But OCR detects it as:

$$10^{10} - 10^{3}$$

So the Expression is then corrected in the Edit LaTex Expression Block.

This is a basic operation involving powers of 10. In mathematics, exponentiation is a shorthand for repeated multiplication. For example, 10^8 means multiplying 10 by itself 8 times, which gives 100,000,000, while 10^10 is 10 multiplied by itself 10 times, resulting in

10,000,000,000.After Subtraction the answer comes out to be equal to 9,900,000,000.

This operation demonstrates how exponents can quickly scale numbers and how subtraction works between large exponential values. Such operations are commonly seen in scientific notation, engineering calculations, and computational estimations where dealing with powers of 10 helps manage very large or very small numbers efficiently. The simplification in this case leads to an exact numeric result of 9,900,000,000.

- Example 3:

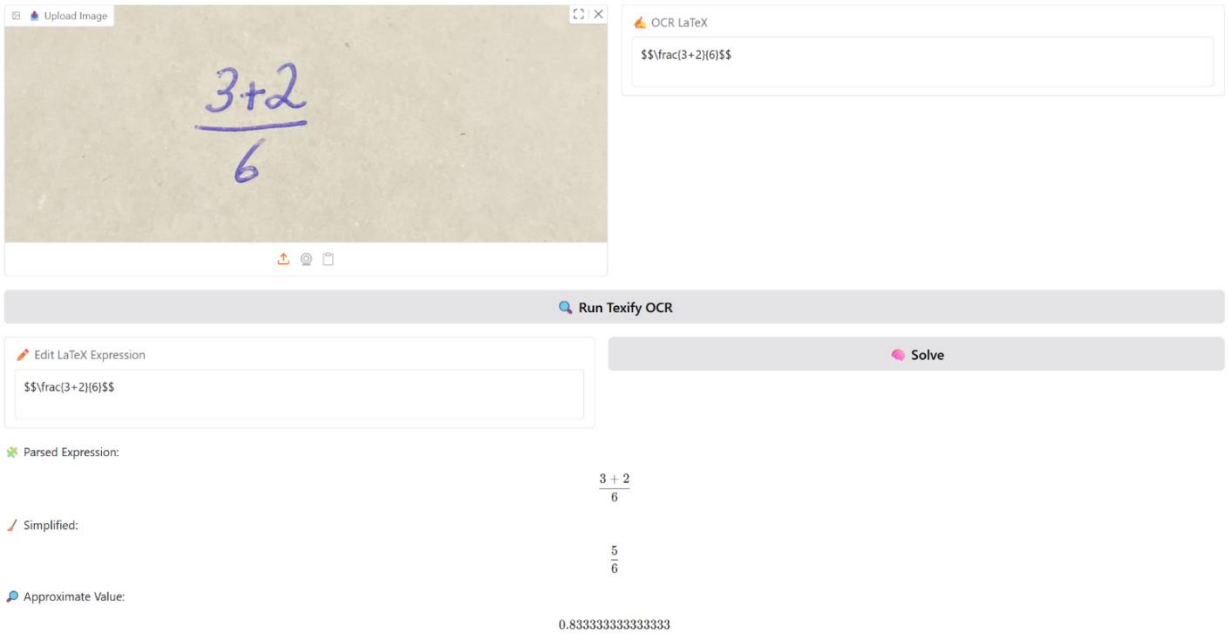

Figure 13.10.6.3: Example-3 of Image OCR Solver

The expression shown in the image is a simple arithmetic fraction:

$$\frac{3+2}{6}$$

This fraction involves the addition of two numbers in the numerator (3 and 2), and the result is then divided by the number in the denominator (6). This type of operation demonstrates the proper use of the order of operations — first evaluating the expression in the numerator before performing the division.

- Example 4:

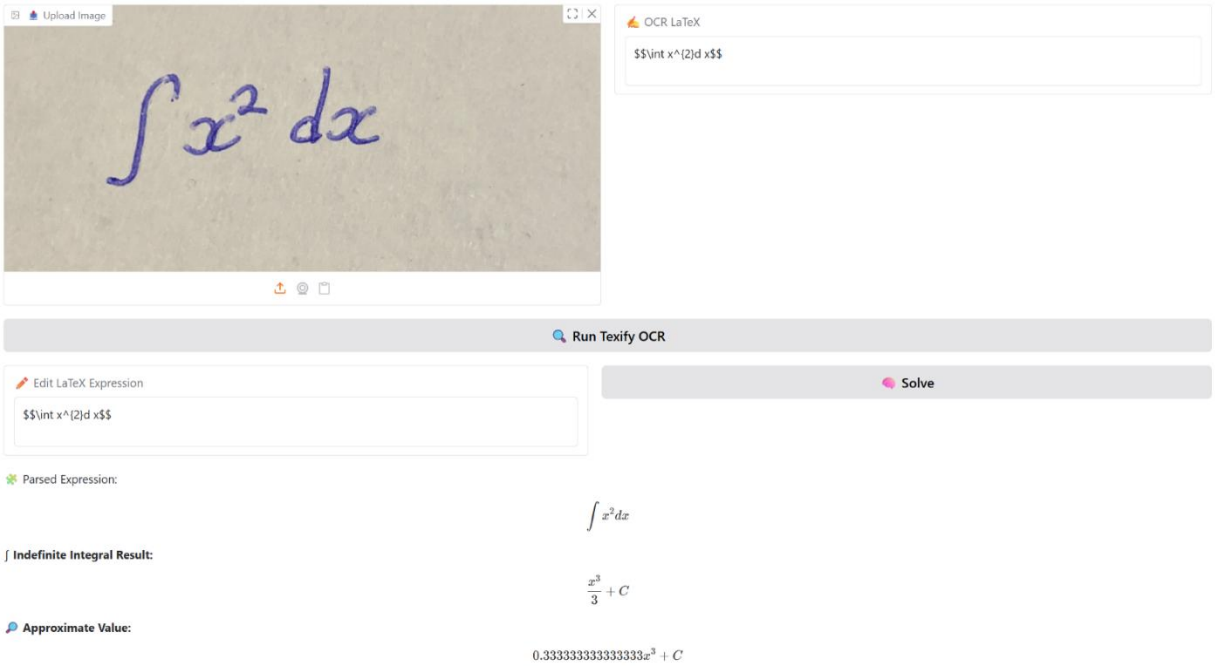

Figure 13.10.6.4: Example-4 of Image OCR Solver

The image displays the mathematical expression:

$$\int x^2\,dx$$

This is an **indefinite integral** problem, which means we are finding the antiderivative of the function x^2 with respect to x. In calculus, integration is the reverse process of differentiation, and an indefinite integral represents a general form of a function whose derivative is the integrand.

The symbolic engine also provides a decimal approximation for general interpretation:

$$0.333333333333333x^3 + C$$

This shows how integration simplifies a power function into another polynomial with adjusted degree and coefficient. The result reflects the area under the curve x^2 with an unknown constant shift, a foundational concept in calculus.

➢ Example 5:

Figure 13.10.6.5: Example-5 of Image OCR Solver

The image represents the derivative of a polynomial expression:

$$\frac{d}{dx}(x^2 + 3x + 2)$$

This is a standard problem in differential calculus, where we are asked to differentiate a quadratic function with respect to the variable xxx. The goal of differentiation is to find the **rate of change** or the **slope** of a function at any given point.

The derivative of the function $x^2 + 3x + 2$with respect to x is:

2x+3.

➢ Example 6:

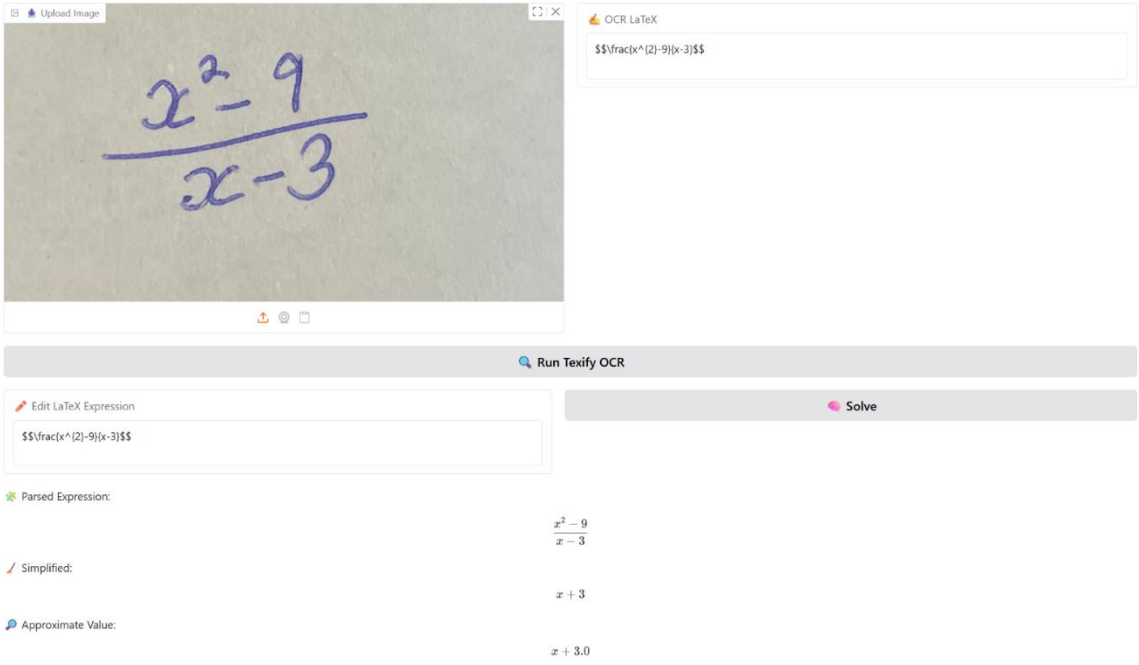

Figure 13.10.6.6: Example-6 of Image OCR Solver

The expression shown in the image is:

$$\frac{x^2 - 9}{x - 3}$$

This is a rational algebraic expression, where the numerator is a quadratic polynomial x^2−9 and the denominator is a linear polynomial x−3. This expression can be simplified using the method of factoringand answer comes out to be x+3.

This simplification shows how algebraic identities (like the difference of squares) help reduce expressions to simpler forms.

- Example 7:

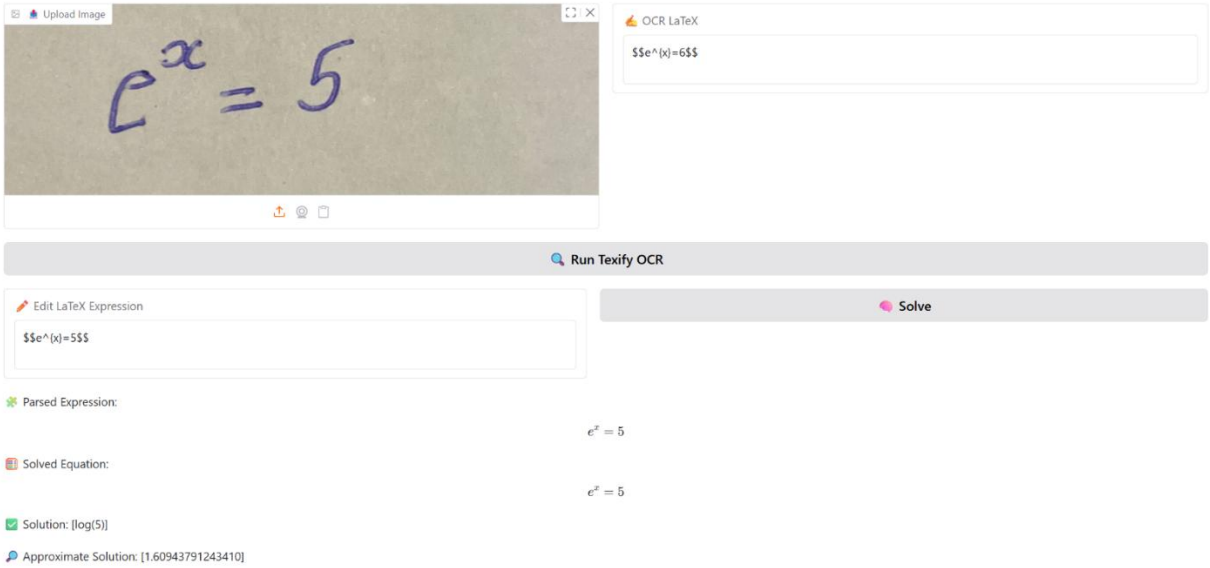

Figure 13.10.6.7: Example-7 of Image OCR Solver

The given mathematical expression in the image is:

$$e^x = 5$$

But OCR detected it as:

$$e^x = 6$$

Then the expression is corrected in the Edit LaTex Expression Block.

This is an **exponential equation**, where the base is Euler's number e≈2.718, and the exponent is the unknown variable x. Exponential equations like this are commonly encountered in fields such as growth modeling, compound interest, and natural processes governed by rates of change.

x≈1.6094

This value represents the exponent to which the number e must be raised to obtain 5.

- Example 8:

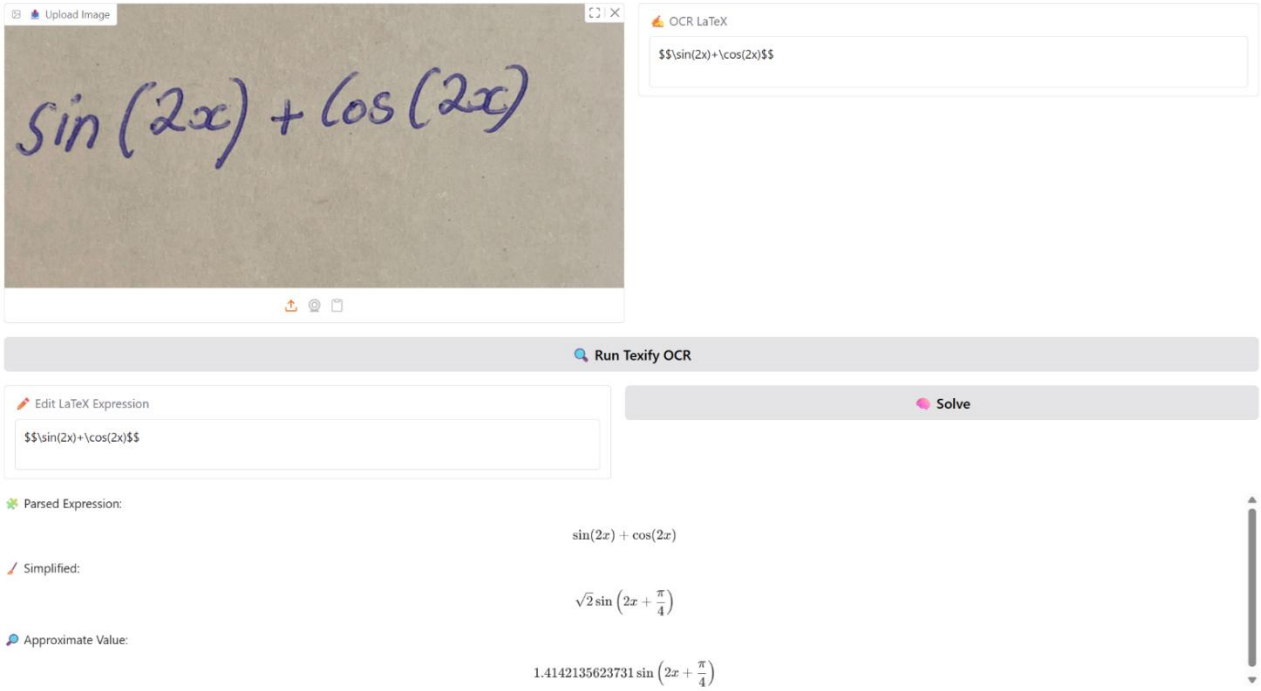

Figure 13.10.6.8: Example-8 of Image OCR Solver

The expression in the image is:

$$(\sin(2x) + \cos(2x))$$

This is a **sum of trigonometric functions** involving the same argument, 2x. In trigonometry, such expressions can be simplified using a known identity for the sum of sine and cosine with the same angle.

The Result comes out to be:

$$\sqrt{2}\ \sin(2x + \frac{\pi}{4})$$

➢ Example 9:

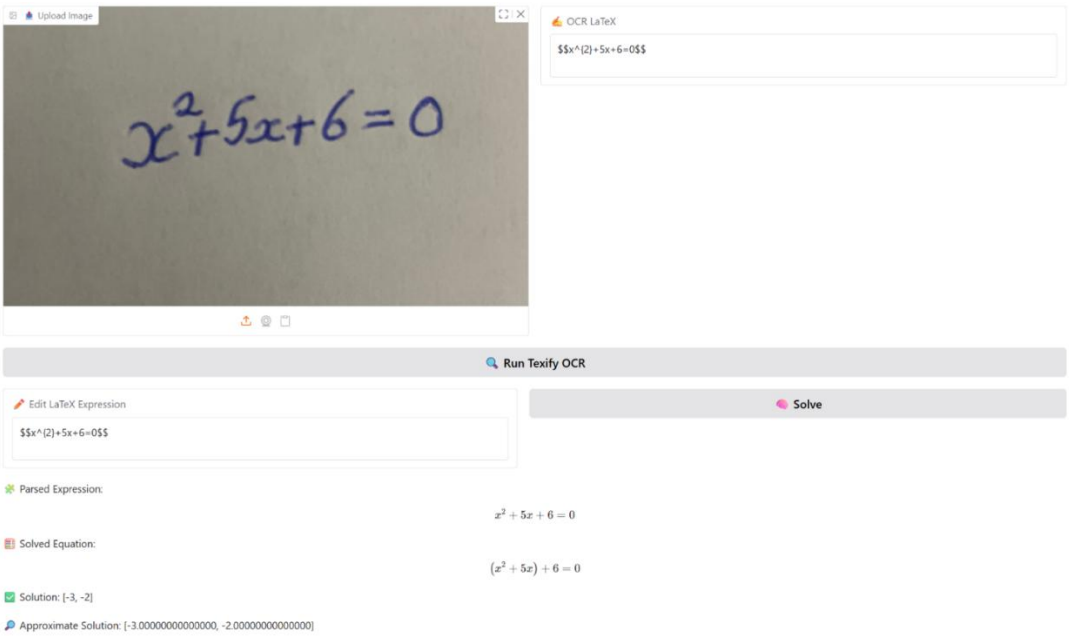

Figure 13.10.6.9: Example-9 of Image OCR Solver

The given image shows the quadratic equation:

$$x^2 + 5x + 6 = 0$$

This is a standard form of a **quadratic equation**, which is written as:

$$ax^2 + bx + c = 0$$

In this equation:

- a=1
- b=5
- c=6

The solutions are:

x= −2 and x= −3.

These are the **real and distinct roots** of the quadratic equation.

➢ Example 10:

Figure 13.10.6.10: Example-10 of Image OCR Solver

The equation shown in the image is a **cubic polynomial equation**:

$$x^3 - 6x^2 + 11x - 6 = 0$$

This equation is of the form:

$$ax^3 + bx^2 + cx + d = 0$$

where:

- a=1
- b= −6
- c=11
- d= −6

The roots of the equation are:

x=1, x=2, x=3

These are **real and distinct roots** of the cubic polynomial.

➢ Example 11:

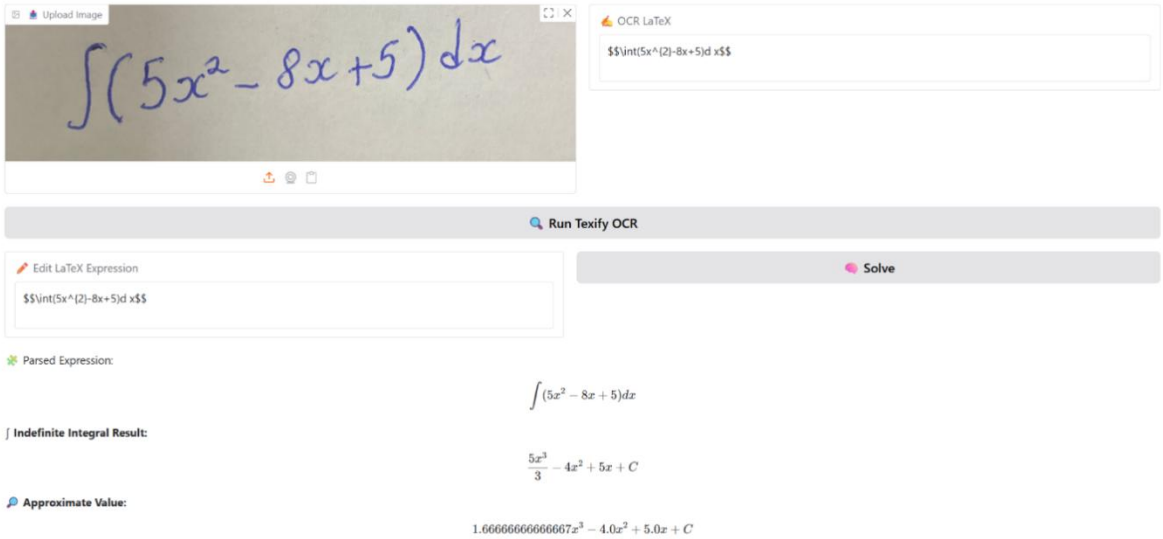

Figure 13.10.6.11: Example-11 of Image OCR Solver

The mathematical expression shown in the image is an **indefinite integral**:

$$\int (5x^2 - 8x + 5)\, dx$$

This integral involves a **polynomial expression**, which is one of the most straightforward types of functions to integrate.

This expression represents the family of all functions whose derivative gives the original polynomial $5x^2 - 8x + 5$The integration constant C accounts for the infinite number of vertical shifts in the graph of the antiderivative.

The Result comes out to be:

$$\frac{5x^3}{3} - 4x^2 + 5 + C$$

The TexifySymPy Solver is an AI-powered math assistant deployed on Hugging Face Spaces that allows users to solve mathematical problems from both handwritten images and typed expressions. The app integrates Texify, a LaTeX OCR model, with SymPy, a symbolic mathematics engine, to provide detailed solutions for a wide range of mathematical expressions. Users can upload handwritten or printed math images, which the app processes to extract LaTeX code using the Texify model. This LaTeX is then parsed and interpreted by SymPy to perform symbolic computations such as simplification, differentiation, integration, and equation solving. The app also supports manual input through a dedicated tab, where users can type SymPy-compatible expressions and receive both exact symbolic answers and approximate numerical results. Built with Gradio, the user interface is clean and intuitive, enabling real-time feedback and explanations. From basic algebraic expressions and trigonometric identities to calculus

problems involving definite and indefinite integrals or derivatives, the app handles it all. Overall, TexifySymPy Solver serves as an intelligent and accessible platform for students, teachers, and enthusiasts to explore and understand mathematical concepts through both visual and textual inputs.

13.10.7. Hugging Face Link for TexifySymPy Solver

https://huggingface.co/spaces/Prabhsimran09/Mathematical_Equation_Solver

Conclusion

The industrial training program at NIELIT Ropar on Artificial Intelligence and Machine Learning (AIML) has been an enriching and invaluable experience, providing in-depth knowledge and practical skills in various cutting-edge technologies. The program's structured approach, combining theoretical insights with hands-on exercises, has significantly enhanced our technical expertise and problem-solving abilities, preparing us for real-world applications in AI/ML.

A major focus of the training was Pandas, which is a cornerstone of data manipulation and analysis in the world of AI and data science. We gained a deep understanding of Pandas' powerful data structures, including DataFrames and Series, which allowed us to efficiently process, clean, and analyze large datasets. This foundational knowledge was crucial for tackling real-world problems and extracting meaningful insights from raw data.

In the realm of Machine Learning, we explored a wide array of algorithms, learning how to apply techniques such as regression, classification, and clustering to solve complex problems. We gained hands-on experience with model evaluation, tuning, and optimization, further strengthening our ability to implement practical solutions. By working on real datasets, we learned how to overcome common challenges such as data inconsistencies, model overfitting, and underfitting, and how to enhance model performance through feature engineering and hyperparameter adjustments.

The Deep Learning component of the training introduced us to neural networks and their applications in fields such as image recognition and natural language processing. We learned how to build, train, and evaluate deep learning models using popular libraries such as TensorFlow and Keras. Through this experience, we developed a strong understanding of how deep learning can be used to solve complex, high-dimensional problems, from image classification to advanced predictive tasks.

One of the standout projects was the QuadPredict platform, where we applied our knowledge of machine learning to build a system that predicts the likelihood of Parkinson’s disease using medical input data. This project deepened our understanding of model deployment and emphasized the importance of AI in healthcare diagnostics. We also explored web integration using HTML and Flask, making our model accessible through an intuitive user interface.

Another key project was the Deepfake Detection System, where we leveraged CNN-LSTM architectures, MTCNN-based face detection, and Grad-CAM explainability to detect manipulated faces in videos and images. This real-world application honed our skills in computer vision, sequence modelling, and ethical AI practices.

Equally impactful was the TexifySymPy Solver, a handwritten math recognition system that combined OCR and symbolic computation using SymPy. This project demonstrated the power of integrating computer vision with symbolic AI to solve handwritten mathematical expressions, which has wide applications in education and automation.

Additionally, the training exposed us to core computer vision concepts such as face detection and face recognition. These applications were deployed using platforms such as Hugging Face, Gradio and Streamlit, enhancing our understanding of AI deployment workflows.

Throughout the training, we not only learned technical concepts but also developed essential skills such as data preprocessing, statistical analysis, and data visualization using tools like Matplotlib. We gained practical experience in deploying AI models and integrating them into user-facing applications, ensuring our solutions are scalable, efficient, and impactful.

Looking forward, we are eager to continue expanding our knowledge in advanced AI techniques, such as reinforcement learning, generative models, and real-time inferencing. Our exposure to deep learning, machine learning, computer vision, and web deployment has laid a solid foundation, and we are now better equipped to contribute to the development of innovative AI solutions across diverse sectors.

In conclusion, the industrial training program at NIELIT Ropar has been a transformative experience, enriching our technical expertise, honing our problem-solving abilities, and preparing us for the challenges and opportunities in the rapidly evolving field of AI and machine learning. We are excited to continue exploring these technologies and contributing to their advancement in both academic and industry contexts.

References

1. Abadi, M., Barham, P., Chen, J. et al. (2016) ‘TensorFlow: A System for Large-Scale Machine Learning’, *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp.265–283.
2. Aggarwal, C.C. (2015) *Data Mining: The Textbook*, Springer, Cham.
3. Chollet, F. (2015) ‘Keras: Deep Learning Library for Theano and TensorFlow’.).
4. Deng, J., Guo, J., Xue, N. and Zafeiriou, S. (2019) ‘ArcFace: Additive Angular Margin Loss for Deep Face Recognition’, *IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp.4690–4699.
5. Géron, A. (2019) *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd ed., O’Reilly Media, Sebastopol, CA.
6. Goodfellow, I., Bengio, Y. and Courville, A. (2016) *Deep Learning*, MIT Press, Cambridge, MA.
7. Han, J., Kamber, M. and Pei, J. (2011) *Data Mining: Concepts and Techniques*, 3rd ed., Morgan Kaufmann Publishers, Burlington, MA.
8. Hugging Face (2024) ‘Gradio: Build Machine Learning Web Apps’.
9. Hunter, J.D. (2007) ‘Matplotlib: A 2D Graphics Environment’, *Computing in Science & Engineering*, Vol.9, No.3, pp.90–95.
10. LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P. (1998) ‘Gradient-Based Learning Applied to Document Recognition’, *Proceedings of the IEEE*, Vol.86, No.11, pp.2278–2324.
11. LeCun, Y. (1998) ‘The MNIST Database of Handwritten Digits’.
12. McKinney, W. (2018) *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*, 2nd ed., O’Reilly Media, Sebastopol, CA.
13. Oliphant, T.E. (2006) *A Guide to NumPy*, Trelgol Publishing, USA.
14. OpenCV.org (2024) ‘Open Source Computer Vision Library’.
15. Pedregosa, F., Varoquaux, G., Gramfort, A. et al. (2011) ‘Scikit-learn: Machine Learning in Python’, *Journal of Machine Learning Research*, Vol.12, pp.2825–2830.
16. Schmidhuber, J. (2015) ‘Deep Learning in Neural Networks: An Overview’, *Neural Networks*, Vol.61, pp.85–117.
17. Simonyan, K., Vedaldi, A. and Zisserman, A. (2014) ‘Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps’, arXiv preprint arXiv:1312.6034.

18. SymPy Development Team (2023) ‘SymPy: Python Library for Symbolic Mathematics’.
19. Texify.ai (2024) ‘Texify Handwritten Math Recognition’.
20. UCI Machine Learning Repository (1988) ‘Iris Dataset’.
21. Van Rossum, G. and Drake, F.L. (2009) *Python 3 Reference Manual*, CreateSpace, Scotts Valley, CA.
22. Vaswani, A., Shazeer, N., Parmar, N. et al. (2017) ‘Attention is All You Need’, *Advances in Neural Information Processing Systems*, Vol.30, pp.5998–6008.
23. W3Schools (2024) ‘HTML Tutorial’.
24. Zhang, K., Zhang, Z., Li, Z. and Qiao, Y. (2016) ‘Joint Face Detection and Alignment Using Multi-task Cascaded Convolutional Networks’, *IEEE Signal Processing Letters*, Vol.23, No.10, pp.1499–1503.