Intel Powered Foundation Course in Machine Learning

To my son,

without whom I should have finished this book two years earlier

Contents

List of Figures

List of Tables

Preface

Step into a realm of innovation with the Intel Unnati Certificate Programme, an avant-garde initiative meticulously crafted to delve into the intricacies of advanced machine learning. This course is not just an educational endeavor; it's a gateway to a world where theoretical understanding seamlessly converges with hands-on practical application, providing you with a distinctive edge in an ever-evolving tech landscape.

In a rapidly changing technological landscape, the Intel Unnati Certificate Programme is your ticket to mastering the latest advancements in machine learning, all powered by cutting-edge Intel technologies. Immerse yourself in a transformative learning experience that transcends conventional boundaries, and emerge not just as a participant but as a pioneer in the future of advanced machine learning. The journey begins here, where intellect meets innovation.

Acknowledgments

A lot of people helped me when I was preparing this Course material.

Coordinator (Training)

Intel-Unnati Programme

Saintgits College of Engineering

About the Intel-Unnati Programme

Welcome to the dynamic universe of Machine Learning with Intel Unnati, a pioneering initiative that fuses cutting-edge technology with artificial intelligence. Machine Learning, situated at the crossroads of computer science and AI, has undergone a metamorphosis, and Intel Unnati is at the forefront of this evolution, offering a unique perspective on theory and application.

In the realm of Intel Unnati's Machine Learning, you will embark on a journey exploring essential concepts like supervised learning, where models glean insights from labeled data, and unsupervised learning, unraveling patterns within unlabeled data. Our courses prioritize hands-on experience, guiding learners through the practical implementation of algorithms and models. Additionally, delve into the realm of reinforcement learning, a paradigm where agents learn through dynamic interactions with their environments.

Intel Unnati's Machine Learning community thrives on ingenuity, pushing the boundaries of what's achievable. With a blend of theoretical depth and real-world applicability, we aim to empower you with the skills to unravel intricate problems and contribute to the ever-evolving landscape of intelligent systems. Welcome to the Intel Unnati Machine Learning experience – a convergence of curiosity, computation, and actionable knowledge.

1 Introduction to Machine Learning

Welcome to the "Introduction to Machine Learning", where we aim to provide a thorough understanding of the principles, applications, and algorithms in machine learning (ML). In this module, we will explore different types of ML and delve into representative algorithms, shedding light on their applications in various domains.

1.1 Machine Learning (ML)

Machine learning (ML) is a type of artificial intelligence (AI) focused on building computer systems that learn from data. The broad range of techniques ML encompasses enables software applications to improve their performance over time.

Machine learning algorithms are trained to find relationships and patterns in data. They use historical data as input to make predictions, classify information, cluster data points, reduce dimensionality and even help generate new content, as demonstrated by new ML-fueled applications such as ChatGPT, Dall-E 2 and GitHub Copilot.

1.1.1 Why is machine learning important?

Machine learning has played a progressively central role in human society since its beginnings in the mid-20th century, when AI pioneers like Walter Pitts, Warren McCulloch, Alan Turing and John von Neumann laid the groundwork for computation. The training of machines to learn from data and improve over time has enabled organizations to automate routine tasks that were previously done by humans – in principle, freeing us up for more creative and strategic work.

Machine learning also performs manual tasks that are beyond our ability to execute at scale – for example, processing the huge quantities of data generated today by digital devices. Machine learning's ability to extract patterns and insights from vast data sets has become a competitive differentiator in fields ranging from finance and retail to healthcare and scientific discovery. Many of today's leading companies, including Facebook, Google and Uber, make machine learning a central part of their operations.

As the volume of data generated by modern societies continues to proliferate, machine learning will likely become even more vital to humans and essential to machine intelligence itself. The technology not only helps us make sense of the data we create, but synergistically the abundance of data we create further strengthens ML's data-driven learning capabilities.

What will come of this continuous learning loop? Machine learning is a pathway to artificial intelligence, which in turn fuels advancements in ML that likewise improve AI and progressively blur the boundaries between machine intelligence and human intellect.

1.1.2 Machine learning examples in industry

Machine learning has been widely adopted across industries. Here are some of the sectors using machine learning to meet their market requirements:

Financial services. Risk assessment, algorithmic trading, customer service and personalized banking are areas where financial services companies apply machine learning. Capital One, for example, deployed ML for credit card defense, which the company places in the broader category of anomaly detection.

Pharmaceuticals. Drug makers use ML for drug discovery, in clinical trials and in drug manufacturing. Eli Lilly has built AI and ML models, for example, to find the best sites for clinical trials and boost the diversity of participants. The models have sharply reduced clinical trial timelines, according to the company.

Manufacturing. Predictive maintenance use cases are prevalent in the manufacturing industry, where an equipment breakdown can lead to expensive production delays. In addition, the computer vision aspect of machine learning can inspect items coming off a production line to ensure quality control.

Insurance. Recommendation engines can suggest options for clients based on their needs and how other customers have benefited from specific insurance products. Machine learning is also useful in underwriting and claims processing.

Retail. In addition to recommendation systems, retailers use computer vision for personalization, inventory management and planning the styles and colors of a given fashion line. Demand forecasting is another key use case.

1.2 Different types of machine learning

Classical machine learning is often categorized by how an algorithm learns to become more accurate in its predictions. There are four basic types of machine learning: supervised learning, unsupervised learning, semi-supervised learning and reinforcement learning.

The type of algorithm data scientists choose depends on the nature of the data. Many of the algorithms and techniques aren't limited to just one of the primary ML types listed here. They're often adapted to multiple types, depending on the problem to be solved and the data set. For instance, deep learning algorithms such as convolutional neural networks and recurrent neural networks are used in supervised, unsupervised and reinforcement learning tasks, based on the specific problem and availability of data.

1.2.1 Supervised Learning

In supervised learning, data scientists supply algorithms with labeled training data and define the variables they want the algorithm to assess for correlations. Both the input and output of the algorithm are specified in supervised learning. Initially, most machine learning algorithms worked with supervised learning, but unsupervised approaches are becoming popular.

Supervised Learning

Definition: Supervised learning involves training a model on a labeled dataset, where each input is paired with the corresponding output. The model learns to map inputs to outputs, enabling predictions on new, unseen data. *Popular Algorithms:*- Linear Regression, logistic regression, support vector machines etc.,

1.2.2 Unsupervised Learning

Unsupervised machine learning algorithms don't require data to be labeled. They sift through unlabeled data to look for patterns that can be used to group data points into subsets. Most types of deep learning, including neural networks, are unsupervised algorithms.

Unsupervised Learning

Definition:Unsupervised learning deals with unlabeled data, aiming to discover patterns, structures, or relationships within the data without explicit guidance. *Popular Algorithms:*- K-means clustering, Principal Component Analysis etc.,

1.2.3 Reinforcement Learning

Reinforcement learning works by programming an algorithm with a distinct goal and a prescribed set of rules for accomplishing that goal. A data scientist will also program the algorithm to seek positive rewards for performing an action that's beneficial to achieving its ultimate goal and to avoid punishments for performing an action that moves it farther away from its goal.

Reinforcement Learning

Definition:Reinforcement learning involves an agent interacting with an environment, learning to make decisions by receiving feedback in the form of rewards or penalties. *Popular Algorithms:*- Q-learning, Deep Q Networks (DQN), Policy gradient method etc.,

1.3 Choose and build the right machine learning model

Developing the right machine learning model to solve a problem can be complex. It requires diligence, experimentation and creativity, as detailed in a seven-step plan on how to build an ML model, a summary of which follows.

1. **Understand the business problem and define success criteria.** The goal is to convert the group's knowledge of the business

problem and project objectives into a suitable problem definition for machine learning. Questions should include why the project requires machine learning, what type of algorithm is the best fit for the problem, whether there are requirements for transparency and bias reduction, and what the expected inputs and outputs are.

2. **Understand and identify data needs.** Determine what data is necessary to build the model and whether it's in shape for model ingestion. Questions should include how much data is needed, how the collected data will be split into test and training sets, and if a pre-trained ML model can be used.
3. **Collect and prepare the data for model training.** Actions include cleaning and labeling the data; replacing incorrect or missing data; enhancing and augmenting data; reducing noise and removing ambiguity; anonymizing personal data; and splitting the data into training, test and validation sets.
4. **Determine the model's features and train it.** Select the right algorithms and techniques. Set and adjust hyperparameters, train and validate the model, and then optimize it. Depending on the nature of the business problem, machine learning algorithms can incorporate natural language understanding capabilities, such as recurrent neural networks or transformers that are designed for NLP tasks. Additionally, boosting algorithms can be used to optimize decision tree models.
5. **Evaluate the model's performance and establish benchmarks.** The work here encompasses confusion matrix calculations, business key performance indicators, machine learning metrics, model quality measurements and determining whether the model can meet business goals.
6. **Deploy the model and monitor its performance in production.** This part of the process is known as operationalizing the model and is typically handled collaboratively by data science and machine learning engineers. Continually measure the model for performance, develop a benchmark against which to measure future iterations of the model and iterate to improve overall performance. Deployment environments can be in the cloud, at the edge or on the premises.
7. **Continuously refine and adjust the model in production.** Even after the ML model is in production and continuously monitored, the job continues. Business requirements, technology capabilities and real-world data change in unexpected ways, potentially giving rise to new demands and requirements.

1.4 Advantages and disadvantages of machine learning

Machine learning's ability to identify trends and predict outcomes with higher accuracy than methods that rely strictly on conventional statistics – or human intelligence – provides a competitive advantage to businesses that deploy ML effectively. Machine learning can benefit businesses in several ways:

- Analyzing historical data to retain customers.
- Launching recommender systems to grow revenue.
- Improving planning and forecasting.
- Assessing patterns to detect fraud.
- Boosting efficiency and cutting costs.

But machine learning also comes with disadvantages. First and foremost, it can be expensive. Machine learning projects are typically driven by data scientists, who command high salaries. These projects also require software infrastructure that can be expensive. And businesses can encounter many more challenges.

There's the problem of machine learning bias. Algorithms trained on data sets that exclude certain populations or contain errors can lead to inaccurate models of the world that, at best, fail and, at worst, are discriminatory. When an enterprise bases core business processes on biased models, it can suffer regulatory and reputational harm.

1.5 Future of machine learning

Fueled by the massive amount of research by companies, universities and governments around the globe, machine learning is a rapidly moving target. Breakthroughs in AI and ML seem to happen daily, rendering accepted practices obsolete almost as soon as they're accepted. One thing that can be said with certainty about the future of machine learning is that it will continue to play a central role in the 21st century, transforming how work gets done and the way we live.

In the field of NLP, improved algorithms and infrastructure will give rise to more fluent conversational AI, more versatile ML models capable of adapting to new tasks and customized language models fine-tuned to business needs.

The fast-evolving field of computer vision is expected to have a profound effect on many domains, from healthcare where it will play an increasingly important role in diagnosis and monitoring as the technology improves, to environmental science where it could be used to analyze and monitor habitats, to software engineering where it's a core component of augmented and virtual reality technologies.

In the near term, machine learning platforms are among enterprise technology's most competitive realms. Major vendors like Amazon, Google, Microsoft, IBM and OpenAI are racing to sign customers up for automated machine learning platform services that cover the spectrum of ML activities, including data collection, data preparation, data classification, model building, training and application deployment.

Amid the enthusiasm, companies will face many of the same challenges presented by previous cutting-edge, fast-evolving technologies. New challenges include adapting legacy infrastructure to machine learning systems, mitigating ML bias and figuring out how to best use these awesome new powers of AI to generate profits for enterprises, in spite of the costs.

1.6 FAQs

1. **What is Machine Learning?**
    - *Machine Learning* is a subset of artificial intelligence that involves the development of algorithms allowing computers to learn patterns and make decisions without being explicitly programmed.
2. **What are the three main types of Machine Learning?**
    - The three main types of *Machine Learning* are *Supervised Learning*, *Unsupervised Learning*, and *Reinforcement Learning.*
3. **Provide an example of Supervised Learning.**
    - Predicting house prices based on features like square footage and number of bedrooms.
4. **How does Unsupervised Learning differ from Supervised Learning?**
    - *Unsupervised Learning* deals with unlabeled data, discovering patterns without predefined outputs, while *Supervised Learning* involves labeled data with known outputs for training.
5. **What is the primary concept in Reinforcement Learning?**

- *Reinforcement Learning* involves an agent learning by interacting with an environment and receiving feedback in the form of rewards or penalties.

6. **Name a famous algorithm used in Reinforcement Learning for game playing.**
   - *Q-Learning* is a notable algorithm used in *Reinforcement Learning* for game playing.
7. **What is the advantage of using Principal Component Analysis (PCA)?**
   - *PCA* reduces dimensionality while retaining most of the variability in the data, aiding in data analysis and visualization.
8. **How does Machine Learning contribute to predictive analytics in healthcare?**
   - *Machine Learning* can predict patient outcomes or disease occurrences based on historical patient data and medical records.
9. **What is the main idea behind Support Vector Machines (SVM)?**
   - *SVM* finds the hyperplane that best separates data into different classes, making it a powerful classifier.
10. **Can you provide an example of Unsupervised Learning in real-world applications?**
    - An example is clustering customers based on purchasing behavior for targeted marketing.
11. **What is the significance of Deep Learning in image recognition?**
    - *Deep Learning*, especially *Convolutional Neural Networks (CNNs)*, excels in identifying objects in images, forming the backbone of modern image recognition systems.
12. **Discuss one disadvantage of using unsupervised learning algorithms.**
    - One disadvantage is the difficulty in evaluating the performance of unsupervised models due to the absence of labeled data for comparison.
13. **How does Reinforcement Learning apply to autonomous vehicles?**
    - *Reinforcement Learning* can be used to train autonomous vehicles by rewarding safe driving behavior and penalizing risky actions.

14. **What role does Machine Learning play in natural language processing?**
    - *Machine Learning* algorithms, like *Recurrent Neural Networks (RNNs)*, contribute to tasks such as language translation and sentiment analysis in natural language processing.
15. **State one potential disadvantage of using deep neural networks in Machine Learning.**
    - One potential disadvantage is the computational complexity and resource-intensive nature of training deep neural networks.
16. **How does Supervised Learning contribute to email spam detection?**
    - *Supervised Learning* algorithms, such as *Logistic Regression*, can classify emails as spam or not spam based on predefined categories in labeled data.
17. **What is the future outlook for artificial intelligence (AI)?**
    - The future of *AI* involves advancements in explainable AI, ethical considerations, and increased integration of *AI* technologies into various industries, leading to smarter and more adaptive systems.
18. **Can you name a popular machine learning algorithm used for text classification?**
    - *Support Vector Machines (SVM)* is often used for text classification, categorizing documents into predefined categories.
19. **What is a potential challenge in deploying machine learning models in real-world scenarios?**
    - One challenge is the need for large and diverse datasets for effective training, as models may struggle with limited or biased data.
20. **How can Machine Learning contribute to predictive maintenance in industries?**
    - *Machine Learning* can predict equipment failures based on sensor data, allowing for proactive maintenance and minimizing downtime.

2

Python for Machine Learning

2.1 History of Python Programming:

Python, conceived by Guido van Rossum in the late 1980s, was officially released as Python 0.9.0 in February 1991. The language aimed to prioritize code readability and ease of use, distinguishing itself with a design philosophy that emphasized clarity and simplicity. Python's name, inspired by Monty Python's Flying Circus, reflects its creator's humor.

Throughout the 1990s, Python underwent significant developments. The release of Python 2.0 in 2000 introduced list comprehensions and garbage collection, enhancing the language's expressiveness and memory management. Python 3.0, released in 2008, marked a major shift with a focus on eliminating inconsistencies and improving code readability.

Over the years, Python has become one of the most popular programming languages, known for its versatility and extensive standard library. It gained traction in web development, scientific computing, and data analysis. Today, Python is a language of choice for a wide range of applications, from web development and automation to artificial intelligence and machine learning.

2.2 Python as the Best Language for Machine Learning

Python's dominance in the field of machine learning is justified by several key factors:

1. **Extensive Libraries:** Python boasts powerful libraries for machine learning, such as TensorFlow, PyTorch, and scikit-learn. These libraries provide pre-built functions and tools that significantly accelerate the development of machine learning models.
2. **Community Support:** Python has a vibrant and active community that contributes to the development of machine learning tools

and frameworks. This ensures continuous improvements, updates, and a wealth of resources for developers.

3. **Ease of Learning:** Python's syntax is clear, concise, and readable, making it accessible for beginners. Its simplicity accelerates the learning curve, allowing developers to quickly grasp machine learning concepts and focus on problem-solving.
4. **Versatility:** Python's versatility enables seamless integration with other technologies and tools, facilitating data manipulation, visualization, and model deployment. It is not confined to machine learning but can be utilized across the entire data science pipeline.
5. **Adoption by Industry Giants:** Leading tech companies, including Google, Facebook, and Microsoft, use Python extensively for machine learning applications. This widespread industry adoption reflects Python's reliability and effectiveness in real-world scenarios.
6. **Open Source Nature:** Python is an open-source language, fostering collaboration and innovation. The open-source community has contributed to the development of a vast ecosystem of machine learning tools and frameworks that continue to evolve.

2.3 Concept of `Libraries` in Python Programming

In Python programming, a library is a collection of pre-written code or modules that can be imported and used in your own programs. Libraries provide a set of functions and methods that can be utilized to perform specific tasks, saving developers time and effort by avoiding the need to write code from scratch for common functionalities.

2.3.1 Key Aspects of Libraries in Python:

1. **Modularity:**
   - Libraries promote modularity by breaking down complex functionalities into smaller, manageable modules. Each module within a library is designed to handle a specific aspect of a task.
2. **Reuse of Code:**
   - Libraries enable code reuse. Instead of duplicating code for common operations, developers can import relevant libraries

and leverage the existing functionality. This enhances code efficiency and reduces the chances of errors.

3. **Functionality Expansion:**
   - Python libraries expand the functionality of the language. Whether it's handling data (NumPy, Pandas), building web applications (Django, Flask), or implementing machine learning models (TensorFlow, scikit-learn), libraries provide a wide range of capabilities beyond the built-in Python functions.
4. **Ease of Development:**
   - Using libraries simplifies development. Developers can focus on solving specific problems or building applications without having to worry about low-level implementations. This leads to faster development cycles and more robust applications.
5. **Community Contributions:**
   - Python has a large and active community that contributes to the development of libraries. This collaborative effort results in a rich ecosystem of libraries covering diverse domains, from scientific computing to web development and machine learning.
6. **Installation and Management:**
   - Libraries can be easily installed and managed using package managers like `pip` (Python Package Installer). This simplifies the process of keeping libraries up-to-date and ensures compatibility with different Python projects.
7. **Standard Libraries vs. External Libraries:**
   - Python comes with a set of standard libraries that are included with the language installation. These libraries cover a wide range of tasks, such as file I/O, regular expressions, and networking. Additionally, developers can install external libraries based on project requirements.
8. **Importing Libraries:**
   - To use a library in Python, you typically start by importing it into your script or program using the `import` statement. For example:

```
import math
```

     This allows you to use functions and constants from the `math` library in your code.

2.4 Importance of Libraries in Machine Learning:

Libraries play a pivotal role in the field of Machine Learning, streamlining the development process, providing essential tools, and accelerating the implementation of complex algorithms. Here's why libraries are crucial in the context of Machine Learning:

1. **Efficiency and Time Savings:**
   - Machine Learning libraries provide pre-implemented algorithms, functions, and tools. This eliminates the need for developers to code these functionalities from scratch, saving a significant amount of time and effort.
2. **Accessibility of Algorithms:**
   - Libraries make cutting-edge machine learning algorithms easily accessible to developers, even those without a deep understanding of the underlying mathematical intricacies. This accessibility democratizes machine learning, allowing a broader range of professionals to harness its power.
3. **Standardization of Implementations:**
   - Libraries establish standardized implementations of algorithms. This ensures consistency across different projects and facilitates collaboration within the machine learning community. Standardization also makes it easier to compare and reproduce results.
4. **Scalability and Performance Optimization:**
   - Machine Learning libraries are often optimized for performance, taking advantage of parallel processing, vectorization, and other optimization techniques. This scalability is crucial when working with large datasets or training complex models.
5. **Diverse Functionality:**
   - Machine Learning libraries offer a wide range of functionalities beyond basic algorithms. They include tools for data preprocessing, feature engineering, model evaluation, and visualization. This comprehensive support streamlines the end-to-end machine learning workflow.
6. **Community Contributions and Updates:**

    - Active communities surround popular machine learning libraries, contributing to their improvement and extension. Regular updates, bug fixes, and the addition of new features ensure that practitioners have access to the latest advancements in the field.

7. **Flexibility in Model Deployment:**
    - Libraries facilitate the deployment of machine learning models into real-world applications. Integration with deployment platforms and frameworks allows developers to transition from model development to deployment seamlessly.
8. **Support for Various Domains:**
    - Machine Learning libraries cater to diverse domains, such as natural language processing, computer vision, reinforcement learning, and more. This versatility allows developers to apply machine learning techniques across a broad spectrum of use cases.
9. **Ease of Experimentation:**
    - Libraries provide a platform for experimenting with different models, hyperparameters, and datasets. This flexibility is crucial for researchers and practitioners to iterate quickly and fine-tune models for optimal performance.
10. **Educational Value:**
    - Machine Learning libraries serve as valuable educational tools, allowing students and researchers to experiment with algorithms and gain hands-on experience. This contributes to the growth of knowledge and expertise in the field.

Popular machine learning libraries, such as `TensorFlow`, `PyTorch`, `scikit-learn`, and `Keras`, have become integral to the success and widespread adoption of machine learning. They encapsulate best practices, foster collaboration, and empower developers to tackle increasingly complex challenges in the realm of artificial intelligence.

2.5 Introduction to Essential Python Libraries for Machine Learning

In a machine learning environment, Python leverages powerful libraries to handle various aspects of data representation, fundamental analysis, numerical computation, and visualization. Here's a practical overview of the key libraries that form the backbone of machine learning workflows:

2.5.1 1. Data Representation: `NumPy`

- **Purpose:** `NumPy` is fundamental for handling numerical data in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.
- **Practical Use:** In machine learning, `NumPy` is essential for representing datasets as arrays, performing mathematical operations on features, and facilitating seamless integration with other machine learning libraries.

2.5.2 2. Fundamental Analysis: `Pandas`

- **Purpose:** `Pandas` is designed for data manipulation and analysis. It introduces data structures like DataFrames and Series, making it efficient to handle and analyze structured data.
- **Practical Use:** In a machine learning context, `Pandas` is invaluable for data preprocessing tasks, such as cleaning, filtering, and transforming datasets. It enables easy exploration and understanding of the data before model training.

2.5.3 3. Numerical Computation: `SciPy`

- **Purpose:** `SciPy` builds on NumPy and provides additional functionality for scientific and technical computing. It includes modules for optimization, integration, interpolation, eigenvalue problems, and more.
- **Practical Use:** In machine learning, `SciPy` complements `NumPy` by offering advanced mathematical and statistical functions. For instance, optimization algorithms from `SciPy` can be employed to fine-tune machine learning models.

2.5.4 4. Visualization: `Matplotlib` and `Seaborn`

- **Purpose:**
  - `Matplotlib` is a versatile 2D plotting library, offering a wide range of visualization options.
  - `Seaborn` is built on top of `Matplotlib` and provides a high-level interface for statistical graphics.
- **Practical Use:** Visualization is crucial for understanding data patterns and

model performance. `Matplotlib` and `Seaborn` enable the creation of informative plots, charts, and graphs to aid in data exploration and presentation of results.

2.5.5 5. Machine Learning: `scikit-learn`

- **Purpose:** `Scikit-learn` is a machine learning library that provides simple and efficient tools for data analysis and modeling. It features various algorithms for classification, regression, clustering, and dimensionality reduction, along with tools for model selection and evaluation.
- **Practical Use:** In machine learning workflows, `scikit-learn` is a go-to library for implementing and applying machine learning algorithms. It simplifies the process of building, training, and evaluating models, making it suitable for both beginners and experienced practitioners.

2.5.6 Practical Perspective:

In a typical machine learning workflow:

- **Data Loading and Representation:** Use `NumPy` arrays to efficiently load and represent datasets.
- **Exploratory Data Analysis (EDA):** Employ `Pandas` for data manipulation, cleaning, and EDA to gain insights into the dataset.
- **Numerical Computations:** For advanced numerical operations, `SciPy` provides tools for optimization, statistical analysis, and more.
- **Visualization:** `Matplotlib` and `Seaborn` help visualize data distributions, relationships, and model performance, aiding in decision-making and communication of results.
- **Machine Learning Modeling:** `Scikit-learn` simplifies the implementation and application of machine learning algorithms.

These libraries work seamlessly together, forming the foundation for effective and efficient machine learning development. Familiarity with these tools is essential for any practitioner looking to navigate the complexities of data analysis and model building in the Python ecosystem.

2.6 Essential `NumPy` Functions

In this section, we'll explore some of the most important `NumPy` functions that are crucial for data manipulation and handling in the context of a Machine Learning course.

2.6.1 1. Creating NumPy Arrays:

- **Function:** `np.array()`
- **Example:**

```
import numpy as np

# Create a 1D array
array_1d = np.array([1, 2, 3, 4, 5])

# Create a 2D array
array_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

2.6.2 2. Array Shape and Dimensions:

- **Functions:** `shape`, `ndim`, `size`
- **Example:**

```
import numpy as np

# Get the shape of an array
shape_2d = np.array([[1, 2, 3], [4, 5, 6]]).shape

# Get the number of dimensions
dimensions_2d = np.array([[1, 2, 3], [4, 5, 6]]).ndim

# Get the total number of elements
size_2d = np.array([[1, 2, 3], [4, 5, 6]]).size
```

2.6.3 3. Indexing and Slicing:

- **Example:**

```
import numpy as np

# Indexing a 1D array
element = np.array([1, 2, 3, 4, 5])[2]

# Slicing a 1D array
sliced_array = np.array([1, 2, 3, 4, 5])[1:4]

# Indexing a 2D array
element_2d = np.array([[1, 2, 3], [4, 5, 6]])[1, 2]

# Slicing a 2D array
sliced_array_2d = np.array([[1, 2, 3], [4, 5, 6]])[:, 1:3]
```

2.6.4 4. Array Reshaping:

- **Function:** `reshape()`
- **Example:**

```
import numpy as np

# Reshape a 1D array into a 2D array
reshaped_array = np.array([1, 2, 3, 4, 5, 6]).reshape(2, 3)
```

2.6.5 5. Mathematical Operations:

- **Example:**

```
import numpy as np

# Element-wise addition
sum_array = np.array([1, 2, 3]) + np.array([4, 5, 6])

# Element-wise multiplication
```

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

# Dot product of two arrays
dot_product = np.dot(np.array([1, 2, 3]), np.array([4, 5, 6]))
```

2.6.6 6. Statistical Operations:

- **Functions:** `mean()`, `median()`, `std()`
- **Example:**

```
import numpy as np

# Calculate mean of an array
mean_value = np.mean(np.array([1, 2, 3, 4, 5]))

# Calculate median of an array
median_value = np.median(np.array([1, 2, 3, 4, 5]))

# Calculate standard deviation of an array
std_deviation = np.std(np.array([1, 2, 3, 4, 5]))
```

2.6.7 7. Higher-Dimensional Array Operations:

- **Example:**

```
import numpy as np

# Create a 3D array
array_3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

# Sum along a specific axis
sum_axis_0 = np.sum(array_3d, axis=0)  # Sum along the first axis
sum_axis_1 = np.sum(array_3d, axis=1)  # Sum along the second axis
sum_axis_2 = np.sum(array_3d, axis=2)  # Sum along the third axis
```

2.6.8 8. Advanced Indexing:

- **Example:**

```python
import numpy as np

# Create a 2D array
array_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Fancy indexing - selecting specific elements
selected_elements = array_2d[[0, 2], [1, 2]]  # Select elements at (0, 1) and (2, 2)

# Boolean indexing - selecting elements based on a condition
condition = array_2d > 5
elements_greater_than_5 = array_2d[condition]
```

2.7 Essential `Pandas` Functions

In this section, we'll explore some of the most important `Pandas` functions that are crucial for data manipulation and handling in the context of a Machine Learning course.

2.7.1 1. Loading Data:

- **Function:** `pd.read_csv()`, `pd.read_excel()`, `pd.read_sql()`
- **Example:**

```python
import pandas as pd

# Load data from a CSV file
df_csv = pd.read_csv('data.csv')

# Load data from an Excel file
df_excel = pd.read_excel('data.xlsx')

# Load data from a SQL database
sql_query = 'SELECT * FROM table_name;'
df_sql = pd.read_sql(sql_query, connection)
```

2.7.2 2. Exploratory Data Analysis (EDA):

- **Functions:** `head()`, `info()`, `describe()`
- **Example:**

```
import pandas as pd

# Display the first few rows of the DataFrame
df_head = df_csv.head()

# Display the summary information of the DataFrame
df_info = df_csv.info()

# Generate descriptive statistics of the DataFrame
df_describe = df_csv.describe()
```

2.7.3 3. Data Preprocessing:

- **Functions:** `drop()`, `fillna()`, `replace()`
- **Example:**

```
import pandas as pd

# Drop missing values
df_no_na = df_csv.dropna()

# Fill missing values with a specific value
df_fill_na = df_csv.fillna(0)

# Replace values in the DataFrame
df_replace = df_csv.replace({'column_name': {'old_value': 'new_value'}})
```

2.7.4 4. Slicing and Indexing:

- **Example:**

```
import pandas as pd

# Select a column
column_data = df_csv['column_name']

# Select multiple columns
multiple_columns_data = df_csv[['column_1', 'column_2']]

# Select rows based on a condition
condition_data = df_csv[df_csv['column_name'] > 5]
```

2.7.5 5. Merging DataFrames:

- **Function:** `merge()`
- **Example:**

```
import pandas as pd

# Merge two DataFrames based on a common column
merged_df = pd.merge(df1, df2, on='common_column')
```

2.7.6 6. Joining DataFrames:

- **Function:** `join()`
- **Example:**

```
import pandas as pd

# Join two DataFrames based on an index
joined_df = df1.join(df2, how='inner')
```

2.7.7 7. Cross-Tabulation:

- **Function:** `pd.crosstab()`
- **Example:**

```
import pandas as pd

# Create a cross-tabulation of two categorical variables
cross_tab = pd.crosstab(df_csv['Category'], df_csv['Label'])
```

2.7.8 8. Value Counts:

- **Function:** `value_counts()`
- **Example:**

```
import pandas as pd

# Count unique values in a column
value_counts_column = df_csv['Column'].value_counts()
```

2.7.9 9. Visualization:

- **Functions:** `plot()`, `hist()`, `boxplot()`
- **Example:**

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

# Plot a line chart
df_csv['Column'].plot()

# Plot a histogram
df_csv['Numeric_Column'].hist()

# Create a boxplot
sns.boxplot(x='Category', y='Numeric_Column', data=df_csv)
plt.show()
```

2.7.10 Practical Perspective

Understanding Pandas functions for loading, preprocessing, slicing, merging, joining, cross-tabulation, value counts, and visualization is crucial for effective machine learning workflows. These operations provide the flexibility to handle diverse datasets, clean and preprocess data, and gain insights through visualizations.

In a real-world machine learning scenario, you'll often use value counts to understand the distribution of categorical variables and leverage visualization techniques to explore data patterns. Pandas, along with visualization libraries like `Matplotlib` and `Seaborn`, facilitates these tasks, making it a powerful tool for data exploration and model development.

2.8 Essential `SciPy` Functions

In this section, we'll explore some of the key `SciPy` functions that are crucial for fundamental mathematical operations in the context of a Machine Learning course, including linear algebra, calculus, optimization, descriptive statistics, inferential statistics, and hypothesis testing.

2.8.1 1. Linear Algebra:

- **Module:** `scipy.linalg`
- **Functions:** `inv()`, `det()`, `eig()`
- **Example:**

```
import numpy as np
from scipy.linalg import inv, det, eig

# Create a square matrix
A = np.array([[4, 2], [3, 1]])

# Calculate the inverse of a matrix
A_inv = inv(A)

# Calculate the determinant of a matrix
A_det = det(A)
```

```
# Calculate the eigenvalues and eigenvectors of a matrix
eigenvalues, eigenvectors = eig(A)
```

2.8.2 2. Calculus:

- **Module:** `scipy.optimize`
- **Functions:** `minimize()`, `fsolve()`
- **Example:**

```
from scipy.optimize import minimize, fsolve

# Define a simple objective function
def objective_function(x):
    return x**2 + 5*x + 6

# Minimize the objective function
result_minimize = minimize(objective_function, x0=0)

# Solve a system of nonlinear equations
def equations_system(x):
    return [x[0] + x[1] - 2, x[0] - x[1] - 1]

result_fsolve = fsolve(equations_system, x0=[0, 0])
```

2.8.3 3. Optimization:

- **Module:** `scipy.optimize`
- **Functions:** `minimize()`, `linprog()`
- **Example:**

```
from scipy.optimize import minimize, linprog

# Define a linear objective function for optimization
c = [2, 3]  # Coefficients of the objective function
A_eq = [[1, 2]]  # Coefficients of the equality constraint
```

```
b_eq = [5]  # RHS value of the equality constraint

# Linear programming optimization
result_linprog = linprog(c, A_eq=A_eq, b_eq=b_eq)

# Nonlinear optimization using the minimize function
result_minimize_opt = minimize(objective_function, x0=0)
```

2.8.4 4. Descriptive Statistics:

- **Module:** `scipy.stats`
- **Functions:** `describe()`
- **Example:**

```
from scipy.stats import describe

# Generate a random dataset
data = np.random.randn(100)

# Compute descriptive statistics
stats_result = describe(data)
```

2.8.5 5. Inferential Statistics and Hypothesis Testing:

- **Module:** `scipy.stats`
- **Functions:** `ttest_ind()`, `wilcoxon()`, `chi2_contingency()`
- **Example:**

```
from scipy.stats import ttest_ind, wilcoxon, chi2_contingency

# Generate two random samples
sample1 = np.random.normal(0, 1, 100)
sample2 = np.random.normal(1, 1, 100)

# Independent two-sample t-test
t_stat, p_value = ttest_ind(sample1, sample2)
```

```
# Wilcoxon signed-rank test for paired samples
wilcoxon_stat, wilcoxon_p_value = wilcoxon(sample1, sample2)

# Chi-squared test for independence
contingency_table = np.array([[30, 10], [20, 40]])
chi2_stat, chi2_p_value, _, _ = chi2_contingency(contingency_table)
```

2.8.6 Practical Perspective:

Understanding `SciPy` functions for descriptive and inferential statistics, as well as hypothesis testing, is essential for analyzing and drawing conclusions from data in machine learning. Descriptive statistics provide summaries of data distributions, while inferential statistics and hypothesis testing help make inferences about populations based on sample data.

In a real-world machine learning scenario, you might use hypothesis testing to compare sample means, assess the significance of differences, and validate assumptions underlying machine learning models.

2.9 Essential `Matplotlib` Functions

In this section, we'll explore some of the key functions in `Matplotlib`, a widely-used data visualization library, essential for creating informative plots and charts in the context of a Machine Learning course.

2.9.1 1. Basic Plots:

- **Module:** `matplotlib.pyplot`
- **Functions:** `plot()`, `scatter()`, `bar()`
- **Example:**

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

# Create a simple line plot
```

```
x = np.linspace(0, 10, 100)
y = np.sin(x)
plt.plot(x, y)
plt.title('Sine Wave')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.show()

# Create a scatter plot
x = np.random.rand(50)
y = np.random.rand(50)
plt.scatter(x, y, c='blue', marker='o')
plt.title('Scatter Plot')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.show()

# Create a bar chart
categories = ['Category A', 'Category B', 'Category C']
values = [30, 45, 20]
plt.bar(categories, values, color='green')
plt.title('Bar Chart')
plt.xlabel('Categories')
plt.ylabel('Values')
plt.show()
```

2.9.2 2. Histograms and Density Plots:

- **Module:** `matplotlib.pyplot`
- **Functions:** `hist()`, `hist2d()`, `contour()`
- **Example:**

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

# Create a histogram
data = np.random.randn(1000)
plt.hist(data, bins=30, color='purple', alpha=0.7)
plt.title('Histogram')
plt.xlabel('Values')
```

```python
plt.ylabel('Frequency')
plt.show()

# Create a 2D histogram
x = np.random.randn(1000)
y = np.random.randn(1000)
plt.hist2d(x, y, bins=30, cmap='Blues')
plt.colorbar()
plt.title('2D Histogram')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.show()

# Create a contour plot
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2))
plt.contour(X, Y, Z, cmap='viridis')
plt.title('Contour Plot')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.show()
```

2.9.3 3. Box Plots and Violin Plots:

- **Module:** `matplotlib.pyplot`
- **Functions:** `boxplot()`, `violinplot()`
- **Example:**

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

# Create a box plot
data = [np.random.normal(0, std, 100) for std in range(1, 4)]
plt.boxplot(data, vert=True, patch_artist=True)
plt.title('Box Plot')
plt.xlabel('Data Sets')
plt.ylabel('Values')
plt.show()
```

```python
# Create a violin plot
data = [np.random.normal(0, std, 100) for std in range(1, 4)]
plt.violinplot(data, showmedians=True)
plt.title('Violin Plot')
plt.xlabel('Data Sets')
plt.ylabel('Values')
plt.show()
```

2.10 Essential `Seaborn` Functions

In this section, we'll explore some of the key functions in `Seaborn`, a statistical data visualization library built on `Matplotlib`, essential for creating visually appealing and insightful plots in the context of a Machine Learning course.

2.10.1 1. Statistical Plots:

- **Module:** `seaborn`
- **Functions:** `sns.scatterplot()`, `sns.lineplot()`, `sns.barplot()`
- **Example:**

```python
import seaborn as sns
import numpy as np

# Create a scatter plot
x = np.linspace(0, 10, 100)
y = np.sin(x)
sns.scatterplot(x, y, color='blue', marker='o')
sns.title('Scatter Plot')
sns.xlabel('X-axis')
sns.ylabel('Y-axis')
sns.show()

# Create a line plot
sns.lineplot(x, y, color='green')
sns.title('Line Plot')
sns.xlabel('X-axis')
sns.ylabel('Y-axis')
sns.show()

# Create a bar plot
categories = ['Category A', 'Category B', 'Category C']
values = [30, 45, 20]
sns.barplot(categories, values, color='purple')
sns.title('Bar Plot')
sns.xlabel('Categories')
sns.ylabel('Values')
sns.show()
```

2.10.2 2. Distribution Plots:

- **Module:** `seaborn`
- **Functions:** `sns.histplot()`, `sns.kdeplot()`, `sns.rugplot()`
- **Example:**

```
import seaborn as sns
import numpy as np

# Create a histogram
data = np.random.randn(1000)
sns.histplot(data, bins=30, color='orange', kde=True)
sns.title('Histogram')
sns.xlabel('Values')
sns.ylabel('Frequency')
sns.show()

# Create a kernel density estimation (KDE) plot
sns.kdeplot(data, color='red')
sns.title('KDE Plot')
sns.xlabel('Values')
sns.ylabel('Density')
sns.show()

# Create a rug plot
sns.rugplot(data, height=0.2, color='green')
sns.title('Rug Plot')
sns.xlabel('Values')
sns.show()
```

2.10.3 3. Categorical Plots:

- **Module:** `seaborn`
- **Functions:** `sns.boxplot()`, `sns.violinplot()`, `sns.swarmplot()`
- **Example:**

```
import seaborn as sns
import numpy as np

# Create a box plot
data = [np.random.normal(0, std, 100) for std in range(1, 4)]
sns.boxplot(data=data, palette='pastel')
sns.title('Box Plot')
sns.xlabel('Data Sets')
sns.ylabel('Values')
sns.show()

# Create a violin plot
sns.violinplot(data=data, inner='quartile', palette='pastel')
sns.title('Violin Plot')
sns.xlabel('Data Sets')
sns.ylabel('Values')
sns.show()

# Create a swarm plot
sns.swarmplot(data=data, color='purple', size=3)
sns.title('Swarm Plot')
sns.xlabel('Data Sets')
sns.ylabel('Values')
sns.show()
```

2.10.4 Practical Perspective:

`Matplotlib` and `Seaborn` are two `Python` libraries that simplifies the process of creating aesthetically pleasing and informative visualizations. These functions allow you to explore relationships in your data, convey patterns, and present results effectively.

2.11 Essential scikit-learn Functions

In this section, we'll explore some of the key functions in `scikit-learn`, a powerful machine learning library, essential for various tasks including data preprocessing, model selection, training, and evaluation.

2.11.1 1. Data Preprocessing:

- **Module:** `sklearn.preprocessing`
- **Functions:** `StandardScaler`, `MinMaxScaler`, `LabelEncoder`
- **Example:**

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler, LabelEncoder
from sklearn.model_selection import train_test_split

# Load your dataset
X, y = load_dataset()

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

# Standardize features by removing the mean and scaling to unit variance
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Normalize features by scaling each feature to a specified range
minmax_scaler = MinMaxScaler()
X_train_normalized = minmax_scaler.fit_transform(X_train)
X_test_normalized = minmax_scaler.transform(X_test)

# Encode categorical labels into numerical format
label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train)
y_test_encoded = label_encoder.transform(y_test)
```

2.11.2 2. Model Selection:

- **Module:** `sklearn.model_selection`

- **Functions:** `train_test_split`, `StratifiedKFold`, `GridSearchCV`
- **Example:**

```
from sklearn.model_selection import train_test_split, StratifiedKFold, GridSearchCV
from sklearn.svm import SVC

# Load your dataset
X, y = load_dataset()

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

# Use stratified k-fold cross-validation for better representation of classes
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Define a support vector machine (SVM) classifier
svm_classifier = SVC()

# Perform grid search for hyperparameter tuning
param_grid = {'C': [0.1, 1, 10], 'kernel': ['linear', 'rbf']}
grid_search = GridSearchCV(svm_classifier, param_grid, cv=cv)
grid_search.fit(X_train, y_train)
best_params = grid_search.best_params_
```

2.11.3 3. Model Training:

- **Module:** Various (`sklearn.svm`, `sklearn.ensemble`, etc.)
- **Functions:** `fit()`
- **Example:**

```
from sklearn.svm import SVC

# Load your dataset
X, y = load_dataset()

# Define a support vector machine (SVM) classifier
svm_classifier = SVC(C=1, kernel='rbf')

# Train the SVM classifier
svm_classifier.fit(X, y)
```

2.11.4 4. Model Evaluation:

- **Module:** `sklearn.metrics`
- **Functions:** `accuracy_score`, `confusion_matrix`, `classification_report`
- **Example:**

```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

# Make predictions on the test set
y_pred = svm_classifier.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
confusion_mat = confusion_matrix(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)
```

2.11.5 Practical Perspective:

`scikit-learn` provides a comprehensive set of functions for various stages of the machine learning workflow. From data preprocessing to model selection, training, and evaluation, `scikit-learn` simplifies the implementation of machine learning pipelines.

3

Supervised Learning

"In the vast landscape of Machine Learning, Supervised Learning is the wise mentor guiding algorithms on a journey akin to human learning. Imagine a world where machines not only observe but truly comprehend, much like how we, as humans, learn from examples."

Welcome to Module 3, where we dive into the realm of Supervised Learning, the virtuoso of machine cognition. Just as a teacher imparts knowledge to a student with labeled guidance, supervised learning equips algorithms with the ability to decipher patterns and make intelligent decisions.

Picture this: a teacher showing a child pictures of different animals, explaining each one. The child learns by associating the visuals with the names – supervised, guided, and nurtured to recognize the world. Similarly, in the realm of machines, Supervised Learning involves presenting algorithms with labeled examples, allowing them to generalize and predict with remarkable accuracy.

Join us in unraveling the magic behind this method, where machines graduate from being mere observers to insightful predictors. Let's explore the fundamentals, where algorithms don the role of eager students, absorbing knowledge from labeled datasets, and emerge as experts capable of tackling real-world challenges.

We decode the intricacies of Supervised Learning, bridging the gap between human understanding and artificial intelligence. Get ready to witness the power of learning by example, where algorithms, much like their human counterparts, evolve into intelligent decision-makers through the artistry of labeled data.

"In the grand symphony of machine cognition, Supervised Learning orchestrates harmony between data and predictions. Let the learning commence!"

3.1 Understanding Supervised Learning

In the realm of machine learning, Supervised Learning is a paradigm where algorithms learn from labeled training data to make predictions or decisions.

The learning process involves mapping input data to corresponding output labels, guided by the examples provided during the training phase. The primary goal is for the algorithm to generalize its learning to accurately predict outcomes for new, unseen data.

Key Elements of Supervised Learning:

- **Input Data:** The information used for making predictions.
- **Output Labels:** The desired predictions or outcomes associated with the input data.
- **Labeled Dataset:** A collection of input-output pairs used for training.
- **Model:** The algorithm that learns the mapping from input to output.
- **Training:** The iterative process where the model adjusts its parameters to minimize prediction errors.

3.2 Key Elements of Supervised Learning Illustrated with the Iris Dataset

The Iris dataset consists of measurements of sepal length, sepal width, petal length, and petal width for three species of iris flowers: setosa, versicolor, and virginica. A glimps of this dataset is shown below:

Sepal Length	Sepal Width	Petal Length	Petal Width	Iris Species
5.1	3.5	1.4	0.2	Setosa
4.9	3.0	1.4	0.2	Setosa
6.7	3.1	4.7	1.5	Versicolor
5.8	2.7	4.1	1.0	Versicolor
6.3	3.3	6.0	2.5	Virginica
7.2	3.6	6.1	2.5	Virginica

3.2.1 1. Input Data

In the Iris dataset, the input data comprises the measurements of sepal length, sepal width, petal length, and petal width for each iris flower. For example, the sepal length, sepal width, petal length, and petal width values for an iris flower might be [5.1, 3.5, 1.4, 0.2].

3.2.2 2. Output Labels

The output labels in the Iris dataset correspond to the species of iris flowers: setosa, versicolor, or virginica. For instance, the output label for the first iris flower in the dataset is “Setosa.”

3.2.3 3. Labeled Dataset

The Iris dataset is a labeled dataset, where each row contains both input features (sepal length, sepal width, petal length, petal width) and the corresponding output label (iris species). The labeled dataset is used for training the Supervised Learning model.

3.2.4 4. Model

The model in this case could be a classification algorithm, such as a decision tree or a support vector machine. The model is trained on the input features and output labels from the Iris dataset to learn the relationship between the measurements and the species of iris flowers.

3.2.5 5. Training

During the training phase, the model processes the labeled dataset and adjusts its internal parameters to make accurate predictions. It learns to associate specific patterns in the input data with the correct output labels, allowing it to generalize and make predictions on new, unseen data.

3.3 Real-World Use Cases of Supervised Learning

Supervised Learning finds application in a myriad of real-world problems, bringing intelligence and automation to various domains. Here are some notable use cases:

1. Image Classification

- **Problem:** Automatically categorizing images into predefined classes.
- **Application:** Medical image diagnosis, facial recognition, autonomous vehicles.

2. Speech Recognition

- **Problem:** Converting spoken language into text.
- **Application:** Virtual assistants, voice commands in smart devices, transcription services.

3. Fraud Detection

- **Problem:** Identifying fraudulent activities in financial transactions.
- **Application:** Credit card fraud detection, anomaly detection in transactions.

4. Sentiment Analysis

- **Problem:** Determining the sentiment expressed in text (positive, negative, neutral).
- **Application:** Social media monitoring, customer feedback analysis, brand reputation management.

5. Predictive Maintenance

- **Problem:** Anticipating equipment failures or maintenance needs.
- **Application:** Manufacturing, aviation, energy sector for optimizing maintenance schedules.

6. Health Diagnosis

- **Problem:** Predicting diseases based on patient data.
- **Application:** Early detection of diseases, personalized medicine.

7. Financial Forecasting

- **Problem:** Predicting stock prices, market trends, or financial metrics.
- **Application:** Stock trading algorithms, investment strategies, risk assessment.

3.4 Understanding Machine Learning Algorithms

Machine Learning Algorithms form the backbone of predictive modeling and decision-making in the field of machine learning. At their core, these algorithms are sets of rules and statistical techniques that enable systems to learn patterns and relationships from data. The remarkable aspect of machine learning is the ability of algorithms to make predictions or decisions without being explicitly programmed for each specific scenario.

3.4.1 What are Machine Learning Algorithms?

Machine learning algorithms are computational procedures designed to learn from data and make predictions or decisions. They generalize patterns from the provided examples, allowing them to perform effectively on new, unseen instances. The flexibility and adaptability of these algorithms make them powerful tools for various applications.

3.4.2 The Role of Models in Machine Learning

A machine learning model is the tangible embodiment of a machine learning algorithm. It encapsulates the learned parameters and the ability to make predictions on new data. Models, essentially, are the products of the training process where the algorithm refines its understanding of patterns present in the labeled dataset. The success of a model is gauged by its capacity to generalize and provide accurate predictions on data it hasn't encountered during training.

3.5 Popular Supervised Learning Algorithms

In the realm of Supervised Learning, numerous algorithms have been developed to address a variety of problems. Each algorithm possesses unique characteristics, strengths, and use cases. Let's explore some popular Supervised Learning algorithms:

3.6 1. Regression Models

Understanding Regression Process

Regression analysis is a powerful statistical method used in machine learning to model the relationship between a dependent variable and one or more independent variables. The primary goal of regression is to predict the value of the dependent variable based on the values of the independent variables. In this section, we'll explore the regression process from both mathematical and statistical viewpoints, delving into the key concepts, equations, and assumptions.

3.6.1 Mathematical Viewpoint

3.6.1.1 Simple Linear Regression

In the case of simple linear regression, where there is only one independent variable, the relationship between the independent variable X and the dependent variable Y can be represented by the equation:

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

- Y: Dependent variable (the variable we are trying to predict)
- X: Independent variable (the variable used for prediction)
- β_0: Y-intercept (constant term)
- β_1: Slope of the regression line
- ϵ: Error term (captures unobserved factors affecting Y)

The objective in simple linear regression is to estimate the values of β_0 and β_1 that minimize the sum of squared differences between the observed and predicted values of Y.

3.6.1.2 Multiple Linear Regression

In the case of multiple linear regression, where there are multiple independent variables $(X_1, X_2, \dots, X_n)$, the equation extends to:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_n X_n + \epsilon$$

3.6.2 Statistical Viewpoint

3.6.2.1 Assumptions of Regression

1. **Linearity:** The relationship between the independent and dependent variables is assumed to be linear.
2. **Independence:** Observations are assumed to be independent of each other.
3. **Homoscedasticity:** Residuals (differences between observed and predicted values) should have constant variance.
4. **Normality of Residuals:** Residuals are assumed to be normally distributed.
5. **No Perfect Multicollinearity:** Independent variables are not perfectly correlated.

3.6.3 Example: House Price Prediction

Suppose we have a small dataset with information on house prices (Y) and square footage (X):

Square Footage (X)	House Price (Y)
1500	250,000
2000	300,000
1700	270,000
2200	350,000
1800	280,000

3.6.3.1 Simple Linear Regression Equation:

$$\text{House Price} = \beta_0 + \beta_1 \times \text{Square Footage} + \epsilon$$

3.6.3.2 Calculations:

1. **Mean of X and Y:**

   - $\bar{X} = \frac{1500+2000+1700+2200+1800}{5} = 1840$
   - $\bar{Y} = \frac{250{,}000+300{,}000+270{,}000+350{,}000+280{,}000}{5} = 290,000$

2. **Calculating β_1 (Slope):**

$$\beta_1 = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sum_{i=1}^{n}(X_i - \bar{X})^2}$$

   After performing the calculations, let's assume we find $\beta_1 \approx 75$.

3. **Calculating β_0 (Y-intercept):**

$$\beta_0 = \bar{Y} - \beta_1 \times \bar{X}$$

   After performing the calculations, let's assume we find $ _0 135,000 $.

4. **Final Regression Equation:**

$$\text{House Price} = 135,000 + 75 \times \text{Square Footage} + \epsilon$$

3.6.3.3 Making Predictions:

Now, using the calculated values of β_0 and β_1, we can make predictions for new data. For example, if we have a house with 1900 square footage, the predicted house price ($\hat{Y}$) would be:

$$\hat{Y} = 135,000 + 75 \times 1900$$

After performing the calculations, let's assume we find $\hat{Y} \approx 297,750$.

This small demonstration illustrates the process of simple linear regression, including the calculation of regression coefficients and making predictions. In a real-world scenario, more sophisticated tools and statistical software would be used for these calculations.

3.6.4 Modern Machine Learning Approach with scikit-learn

Now, let's explore the same regression process using the scikit-learn library in Python. Scikit-learn provides a convenient and efficient way to implement machine learning algorithms, including regression.

3.6.4.1 Step 1: Importing Libraries and Loading Data

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn import metrics

# Load the dataset (assuming 'data' is already available from the previous example)
X = data['SquareFootage'].values.reshape(-1, 1)
y = data['HousePrice'].values
```

3.6.4.2 Step 2: Splitting the Data into Training and Testing Sets

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

3.6.4.3 Step 3: Creating and Training the Model

```
model = LinearRegression()
model.fit(X_train, y_train)
```

3.6.4.4 Step 4: Making Predictions

```
y_pred = model.predict(X_test)
```

3.6.4.5 Step 5: Evaluating the Model

```
print('Mean Absolute Error:', metrics.mean_absolute_error(y_test, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(y_test, y_pred))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(y_test, y_pred)))
```

3.6.5 Comparison and Considerations

Traditional vs. Machine Learning Approach

Traditional Approach: - Requires manual calculation of coefficients. - Assumes a linear relationship between variables. - Relies on strict assumptions about data distribution.

Machine Learning Approach: - Utilizes machine learning libraries for efficient implementation. - Handles complex relationships and patterns in data. - More flexible and robust to deviations from strict assumptions.

3.6.5.1 When to Use Regression Algorithms over Traditional Mathematical Approaches?

The choice between using a regression algorithm and a traditional mathematical approach depends on several factors and the nature of the data. Here are situations where using a regression algorithm is often more useful than a traditional mathematical approach:

1. **Complex Relationships:**

- *Regression Algorithm:* Well-suited for capturing complex and nonlinear relationships between variables.
- *Traditional Approach:* Assumes a linear relationship, may struggle with intricate patterns.

2. **Large Datasets:**
   - *Regression Algorithm:* Efficiently handles large datasets, making it suitable for big data scenarios.
   - *Traditional Approach:* Manual calculations can become impractical with a large amount of data.
3. **Predictive Accuracy:**
   - *Regression Algorithm:* Focuses on predictive accuracy, making it valuable when the primary goal is accurate predictions.
   - *Traditional Approach:* May prioritize interpretability over predictive accuracy.
4. **Automated Feature Engineering:**
   - *Regression Algorithm:* Can automatically handle feature engineering, extracting relevant patterns from data.
   - *Traditional Approach:* May require manual feature engineering, which can be time-consuming.
5. **Dynamic Environments:**
   - *Regression Algorithm:* Adapts to changing environments, learns from new data, suitable for dynamic and evolving scenarios.
   - *Traditional Approach:* May struggle to adapt to changes and updates in data.
6. **Handling Multivariate Relationships:**
   - *Regression Algorithm:* Naturally extends to multiple independent variables, making it suitable for multivariate scenarios.
   - *Traditional Approach:* May become more complex when dealing with multiple independent variables.
7. **Robustness to Assumption Violations:**
   - *Regression Algorithm:* More robust to deviations from assumptions like normality, homoscedasticity, and linearity.
   - *Traditional Approach:* Sensitive to assumption violations, which may limit its applicability in real-world scenarios.
8. **Efficiency and Automation:**
   - *Regression Algorithm:* Utilizes machine learning libraries for efficient implementation, reducing the need for manual calculations.

- *Traditional Approach:* May involve manual calculations, which could be time-consuming and error-prone.

Regression algorithms are particularly beneficial when dealing with complex relationships, large datasets, and scenarios where predictive accuracy is crucial. They provide a more flexible and automated approach compared to traditional methods, making them suitable for a wide range of real-world applications.

3.6.6 Tasks

Task 1: Create a synthetic data using random numbers which includes the radius and price of Pizza with proper logic. Create a regression model to predict the pizza price while its' radius is given.

Solution:

3.6.6.1 Step 1: Importing Necessary Libraries

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

3.6.6.2 Step 2: Generating Synthetic Pizza Price Data

```
np.random.seed(42)

# Generating random radius values
radius = np.random.uniform(5, 15, 500)

# Generating synthetic prices based on a linear relationship with some noise
price = 5 * radius + np.random.normal(0, 2, 500)

# Creating a DataFrame
pizza_data = pd.DataFrame({'Radius': radius, 'Price': price})
```

3.6.6.3 Step 3: Saving the Dataset to a CSV File

```
pizza_data.to_csv('pizza_price.csv', index=False)
print(pizza_data.head())
```

3.6.6.4 Step 4: Splitting the Data into Training and Testing Sets

```
X = pizza_data[['Radius']]
y = pizza_data['Price']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

3.6.6.5 Step 5: Creating and Training the Linear Regression Model

```
model = LinearRegression()
model.fit(X_train, y_train)
```

3.6.6.6 Step 6: Making Predictions on the Test Set

```
y_pred = model.predict(X_test)
```

3.6.6.7 Step 7: Evaluating the Model

```
mae = metrics.mean_absolute_error(y_test, y_pred)
mse = metrics.mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)

print(f'Mean Absolute Error: {mae}')
print(f'Mean Squared Error: {mse}')
print(f'Root Mean Squared Error: {rmse}')
```

3.6.6.8 Step 8: Visualizing the Regression Line

```
plt.scatter(X_test, y_test, color='black')
plt.plot(X_test, y_pred, color='blue', linewidth=3)
plt.title('Pizza Price Regression Model')
plt.xlabel('Radius')
plt.ylabel('Price')
plt.show()
```

---

*Task 2:**

3.7 2. Classification Algorithm in Machine Learning

In machine learning, a **classification algorithm** is a type of supervised learning algorithm designed to assign predefined labels or categories to input data based on its features. The primary goal is to learn a mapping between input features and corresponding output labels, enabling the algorithm to make predictions on new, unseen data.

3.7.1 Key Characteristics:

1. **Supervised Learning:**
    - Classification is a form of supervised learning, where the algorithm is trained on a labeled dataset pairing each example with the correct output label. It learns the mapping from input features to output labels during training.
2. **Input Features:**
    - Input to a classification algorithm consists of features or attributes describing the data. These features are used to predict the output labels.
3. **Output Labels or Classes:**
    - The output is a set of predefined labels or classes representing different categories assigned by the algorithm. It could be binary (e.g., "spam" and "non-spam") or multiclass (e.g., digits 0 through 9).
4. **Training Process:**
    - During training, the algorithm is presented with a labeled dataset. It adjusts its internal parameters or model based on input features and corresponding output labels to learn patterns in the data.
5. **Model Representation:**
    - A classification algorithm builds a model representing the learned relationship between input features and output labels. Models can take various forms such as decision trees, support vector machines, logistic regression, or neural networks.
6. **Prediction:**
    - Once trained, the model predicts the output label for new, unseen data. It uses the learned patterns to classify instances based on their input features.
7. **Evaluation Metrics:**
    - Performance is evaluated using metrics like accuracy, precision, recall, F1 score, and confusion matrix. These metrics assess how well the algorithm generalizes to new data and accurately classifies instances.

3.7.2 Common Classification Algorithms:

- Logistic Regression
- Decision Trees
- Random Forests
- Support Vector Machines
- K-Nearest Neighbors (KNN)
- Naive Bayes

The choice of a specific classification algorithm depends on the nature of the data and the problem at hand. Each algorithm has its strengths and weaknesses, making it suitable for different types of classification tasks.

3.8 Logistic Regression and Regularization in Classification

Logistic Regression is a powerful statistical method widely employed for binary and multiclass classification tasks. It models the probability of an instance belonging to a particular class, utilizing the logistic function (sigmoid function). In the context of classification, logistic regression is often enhanced with regularization techniques, such as Ridge Regression and Lasso Regression, to mitigate overfitting and improve model generalization.

3.8.1 a. Logistic Regression

Logistic Regression is a widely-used statistical method for binary and multiclass classification in machine learning. Despite its name, it is primarily used for classification tasks, not regression. The model is well-suited for scenarios where the dependent variable is categorical, and the goal is to predict the probability of an observation belonging to a particular class.

3.8.2 Hypothesis Function

The core of Logistic Regression is the hypothesis function, which uses the logistic or sigmoid function to transform a linear combination of input features into a value between 0 and 1. The hypothesis function $h_\theta(x)$ in logistic regression models the probability that a given input x belongs to the positive class (typically denoted as class 1).

$$h_\theta(x) = \frac{1}{1 + e^{-(\theta^T x)}}$$

This function produces values between 0 and 1, mapping the linear combination of input features (x) and model parameters (θ) to a probability.

3.8.3 Decision Boundary

The decision boundary, where $h_\theta(x) = 0.5$, is defined by $\theta^T x = 0$. This boundary separates instances into different classes based on their predicted probabilities.

3.8.4 Cost Function (Binary Classification)

The cost function in logistic regression is formulated based on the principle of maximum likelihood estimation. The objective is to maximize the likelihood that the observed data (labels) would be generated by the predicted probabilities. In binary logistic regression, the cost function is typically defined using the log-likelihood function.

3.8.4.1 Binary Logistic Regression Cost Function

Denoting the predicted probability that an example belongs to the positive class as $h_\theta(x)$, where x is the input features and θ is the parameter vector. The actual label for the example is y, where $y = 1$ if the example belongs to the positive class and $y = 0$ if it belongs to the negative class.

The logistic regression cost function for a single training example is defined as:

$$J(\theta) = -y \log(h_\theta(x)) - (1 - y) \log(1 - h_\theta(x))$$

This cost function combines two scenarios:

- If $y = 1$: The first term $-y \log(h_\theta(x))$ penalizes the model if the predicted probability $(h_\theta(x))$ is close to 0.
- If $y = 0$: The second term $-(1 - y) \log(1 - h_\theta(x))$ penalizes the model if the predicted probability is close to 1.

For multiple training examples, the overall cost function is the average of these individual costs:

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))]$$

Here, m is the number of training examples, $y^{(i)}$ is the actual label for the i-th example, and $h_\theta(x^{(i)})$ is the predicted probability.

3.8.5 Gradient Descent (Binary Classification)

Gradient descent is employed to minimize the cost function, updating model parameters (θ_j) iteratively.

$$\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)}$$

Here, α is the learning rate, and $x_j^{(i)}$ represents the j-th feature of the i-th example.

3.8.6 Regularization in Logistic Regression

3.8.6.1 Ridge Regression (L2 Regularization)

Ridge Regression introduces a regularization term to the cost function, penalizing the sum of squared coefficients.

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

The additional term $\frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$ restrains the coefficients, where λ is the regularization parameter.

3.8.6.2 Gradient Descent (Ridge Regression)

The gradient descent update rule for Ridge Regression includes the regularization term.

$$\theta_j := \theta_j - \alpha\left(\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} + \frac{\lambda}{m}\theta_j\right)$$

3.8.7 Lasso Regression (L1 Regularization)

Lasso Regression incorporates a regularization term that penalizes the absolute values of the coefficients.

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)}\log(h_\theta(x^{(i)})) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))] + \frac{\lambda}{m}\sum_{j=1}^{n}|\theta_j|$$

The term $\frac{\lambda}{m}\sum_{j=1}^{n}|\theta_j|$ promotes sparsity in the model by encouraging some coefficients to become exactly zero.

3.8.7.1 Gradient Descent (Lasso Regression)

The gradient descent update rule for Lasso Regression includes the regularization term with the sign of the coefficients.

$$\theta_j := \theta_j - \alpha\left(\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} + \frac{\lambda}{m}\text{sign}(\theta_j)\right)$$

These regularization techniques play a crucial role in preventing overfitting, enhancing model robustness, and improving the generalization of logistic regression models in classification tasks.

3.8.7.2 Key Elements of Classification Model Using Logistic Regression and the Iris Dataset

Assuming you have the seaborn library installed, you can use the famous Iris dataset for a classification task. The Iris dataset is commonly used to predict the species of iris flowers based on their features.

```
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix

# Load Iris dataset
iris = sns.load_dataset('iris')

# Display the first few rows of the dataset
print(iris.head())
```

Now, let's proceed with the key elements:

3.8.7.3 Step 1. Data Preparation:

```
# Features (X) and target (y)
X = iris.drop('species', axis=1)
y = iris['species']

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

3.8.7.4 Step 2. Model Training:

For this example, let's use Logistic Regression.

```
# Initialize the Logistic Regression model
logreg = LogisticRegression(max_iter=1000)

# Train the model on the training data
logreg.fit(X_train, y_train)
```

3.8.7.5 Step 3. Model Prediction:

```
# Make predictions on the test set
y_pred = logreg.predict(X_test)
```

3.8.7.6 Step 4. Model Evaluation:

```
# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)

# Display evaluation metrics
print(f'Accuracy: {accuracy:.2f}')
print('Confusion Matrix:')
print(conf_matrix)
print('Classification Report:')
print(classification_rep)
```

3.8.8 b. Decision Trees

Decision Trees are versatile and widely used machine learning algorithms for both classification and regression tasks. These models make decisions based on the features of the input data by recursively partitioning it into subsets. Each partition is determined by asking a series of questions based on the features, leading to a tree-like structure.

3.8.8.1 What is Iterative Dichotomiser3 Algorithm?

ID3 or Iterative Dichotomiser3 Algorithm is used in machine learning for building decision trees from a given dataset. It was developed in 1986 by Ross Quinlan. It is a greedy algorithm that builds a decision tree by recursively partitioning the data set into smaller and smaller subsets until all data points in each subset belong to the same class. It employs a top-down approach, recursively selecting features to split the dataset based on information gain.

Th ID3 (Iterative Dichotomiser 3) algorithm is a classic decision tree algorithm used for both classification and regression tasks.ID3 deals primarily with categorical properties, which means that it can efficiently handle objects with a discrete set of values. This property is consistent with its suitability for problems where the input features are categorical rather than continuous. One of the strengths of ID3 is its ability to generate interpretable decision trees. The resulting tree structure is easily understood and visualized, providing insight into the decision-making process. However, ID3 can be sensitive to noisy data and prone to overfitting, capturing details in the training data that may not adequately account for new unseen data.

3.8.8.2 How ID3 Algorithms work?

The ID3 algorithm works by building a decision tree, which is a hierarchical structure that classifies data points into different categories and splits the dataset into smaller subsets based on the values of the features in the dataset. The ID3 algorithm then selects the feature that provides the most information about the target variable. The decision tree is built top-down, starting with the root node, which represents the entire dataset. At each node, the ID3 algorithm selects the attribute that provides the most information gain about the target variable. The attribute with the highest information gain is the one that best separates the data points into different categories.

ID3 metrices

The ID3 algorithm utilizes metrics related to information theory, particularly entropy and information gain, to make decisions during the tree-building process.

3.8.8.3 Information Gain and Attribute Selection

The ID3 algorithm uses a measure of impurity, such as entropy or Gini impurity, to calculate the information gain of each attribute. Entropy is a measure of disorder in a dataset. The entropy of `X` is calculated using the formula:

$$H(X) = -\sum_{i=0}^{n} P(x_i) log_b P(x_i)$$

Also the information gain also be calculated in ID3 algorithm. Information gain can be defined as the amount of information gained about a random variable or signal from observing another random variable.It can be considered as the difference between the entropy of parent node and weighted average entropy of child nodes. The formula for calculating information gain is:

$$IG(S, A) = H(S) - \sum_{i=0}^{t} P(x) \cdot H(x)$$

A dataset with high entropy is a dataset where the data points are evenly distributed across the different categories. A dataset with low entropy is a dataset where the data points are concentrated in one or a few categories. A detailed discussion of ID3 algorithm is availabile in:

https://www.linkedin.com/pulse/decision-tree-id3-algorithm-maths-behind-fenil-patel/

3.8.8.4 Key Concepts for Decision Tree in Machine Learning

*3.8.8.4.1 1. **Tree Structure:***

A Decision Tree consists of nodes, where each node represents a decision based on a particular feature. The tree structure includes:

- **Root Node:** The topmost node, which makes the initial decision.
- **Internal Nodes:** Nodes that represent decisions based on specific features.
- **Leaf Nodes:** Terminal nodes that provide the final output, either a class label (for classification) or a numerical value (for regression).

*3.8.8.4.2 2. **Decision Criteria:***

At each internal node, a decision criterion is applied to split the data. Common criteria include Gini impurity for classification and mean squared error for regression.

- **Gini Impurity (Classification):**

$$\text{Gini}(t) = 1 - \sum_{i=1}^{c} p(i|t)^2$$

  Here, c is the number of classes, and $p(i|t)$ is the probability of class i at node t.

- **Mean Squared Error (Regression):**

$$\text{MSE}(t) = \frac{1}{n_t} \sum_{i \in D_t} (y_i - \bar{y}_t)^2$$

  Here, D_t is the set of training instances at node t, n_t is the number of instances, y_i is the target value for instance i, and $\bar{y}_t$ is the mean target value at node t.

*3.8.8.4.3 3. **Tree Growing:***

The process of creating a decision tree involves recursively splitting the data based on the chosen criteria until a stopping condition is met. Overfitting can be controlled by setting maximum depth, minimum samples per leaf, or other hyper-parameters.

*3.8.8.4.4 4. **Decision Making:***

To make predictions for a new instance, it traverses the tree from the root to a leaf, following the decisions based on the features.

3.8.9 Use Cases

Decision Trees find applications in various domains:

- **Classification:**
  - Spam email detection.
  - Disease diagnosis.
- **Regression:**
  - Predicting house prices.
  - Forecasting sales.

3.8.10 Advantages

- **Interpretability:** Decision Trees are easy to understand and interpret, making them suitable for explaining model decisions.
- **Handling Non-Linearity:** They can capture complex non-linear relationships in the data.

3.8.11 Limitations

- **Overfitting:** Decision Trees can easily overfit noisy data, necessitating the use of pruning techniques.
- **Instability:** Small changes in the data can lead to significantly different tree structures.

Decision Trees serve as the basis for more complex ensemble methods like Random Forests and Gradient Boosting, contributing to their popularity in machine learning workflows.

3.8.11.1 Minimal Working Example

Problem statement: Consider the following data and create a descision tree based on the information to determine whether one has to play or not play football using ID3 algorithm.

outlook	temp	humidity	windy	play
overcast	hot	high	FALSE	yes
overcast	cool	normal	TRUE	yes
overcast	mild	high	TRUE	yes
overcast	hot	normal	FALSE	yes
rainy	mild	high	FALSE	yes
rainy	cool	normal	FALSE	yes
rainy	cool	normal	TRUE	no
rainy	mild	normal	FALSE	yes
rainy	mild	high	TRUE	no
sunny	hot	high	FALSE	no
sunny	hot	high	TRUE	no

outlook	temp	humidity	windy	play
sunny	mild	high	FALSE	no
sunny	cool	normal	FALSE	yes
sunny	mild	normal	TRUE	yes

Solution

Here there are four independent variables to determine the dependent variable. The independent variables are Outlook, Temperature, Humidity, and Wind. The dependent variable is whether to play football or not. As the first step in the `Python` implementation, create a dataframe that stores the given data. For numerical calculation and data processing we will use the `numpy` and `pandas` libraries. Let's load these libraries first and make some initial arrangements to write functions for entropy and information gain calculations.

```
import numpy as np
import pandas as pd
eps = np.finfo(float).eps
from numpy import log2 as log
```

Creating the dataframe: We will use the following `Python` code to create store the data in a dataframe.

```
#creating data as lists
outlook = 'overcast,overcast,overcast,overcast,rainy,rainy,rainy,rainy,rainy,sunny,sunny,sunny,sun
temp = 'hot,cool,mild,hot,mild,cool,cool,mild,mild,hot,hot,mild,cool,mild'.split(',')
humidity = 'high,normal,high,normal,high,normal,normal,normal,high,high,high,high,normal,normal'.s
windy = 'FALSE,TRUE,TRUE,FALSE,FALSE,FALSE,TRUE,FALSE,TRUE,FALSE,TRUE,FALSE,FALSE,TRUE'.split(',')
play = 'yes,yes,yes,yes,yes,yes,no,yes,no,no,no,no,yes,yes'.split(',')
```

```
#creating a datadrame of input data
dataset ={'outlook':outlook,'temp':temp,'humidity':humidity,'windy':windy,'play':play}
df = pd.DataFrame(dataset,columns=['outlook','temp','humidity','windy','play'])
```

3.8.11.1.1 Computational steps in Learning Decision Tree

1. compute the entropy for data-set
2. for every attribute/feature:

```
1.calculate entropy for all categorical values

2.take average information entropy for the current attribute

3.calculate gain for the current attribute
```

3. pick the highest gain attribute.
4. Repeat until we get the tree we desired

The `python` code to compute the dataset entropy is as follows:

```
# writting a general code for finding entropy
def find_entropy(df):
    Samplespace = df.keys()[-1]   #To make the code generic, changing target variable class name
    entropy = 0
    values = df[Samplespace].unique()
    for value in values:
        fraction = df[Samplespace].value_counts()[value]/len(df[Samplespace])
        entropy += -fraction*np.log2(fraction)
    return entropy
```

Now entropy of various attributes can be found using the following `python` function.

```
def find_entropy_attribute(df,attribute):
  Samplespace = df.keys()[-1]   #To make the code generic, changing target variable class name
  target_variables = df[Samplespace].unique()  #This gives all 'Yes' and 'No'
  variables = df[attribute].unique()    #This gives different features in that attribute (like 'Ho
  entropy2 = 0
  for variable in variables:
      entropy = 0
      for target_variable in target_variables:
          num = len(df[attribute][df[attribute]==variable][df[Samplespace] ==target_variable])
          den = len(df[attribute][df[attribute]==variable])
          fraction = num/(den+eps)
          entropy += -fraction*log(fraction+eps)
      fraction2 = den/len(df)
      entropy2 += -fraction2*entropy
  return abs(entropy2)
```

Now the winner attribute can be identified using the information gain. The following funtion will do that job.

```
def find_winner(df):
    Entropy_att = []
    IG = []
    for key in df.keys()[:-1]:
#         Entropy_att.append(find_entropy_attribute(df,key))
        IG.append(find_entropy(df)-find_entropy_attribute(df,key))
    return df.keys()[:-1][np.argmax(IG)]
```

Once the winner node is selected based on the information gain, it will be set as the decision node at that level and further the dataset will be updated after removing the selected feature. This can be done using the following function.

```
def get_subtable(df, node,value):
  return df[df[node] == value].reset_index(drop=True)
```

Now let's write a function to combine all these functions and create a decision tree.

```
def buildTree(df,tree=None):
    Samplespace = df.keys()[-1]   #To make the code generic, changing target variable class name
    DEC=df.columns[-1]
    #Here we build our decision tree

    #Get attribute with maximum information gain
    node = find_winner(df)

    #Get distinct value of that attribute e.g Salary is node and Low,Med and High are values
    attValue = np.unique(df[node])

    #Create an empty dictionary to create tree
    if tree is None:
        tree={}
        tree[node] = {}

   #We make loop to construct a tree by calling this function recursively.
    #In this we check if the subset is pure and stops if it is pure.

    for value in attValue:

        subtable = get_subtable(df,node,value)
        clValue,counts = np.unique(subtable[DEC],return_counts=True)

        if len(counts)==1:#Checking purity of subset
            tree[node][value] = clValue[0]
        else:
            tree[node][value] = buildTree(subtable) #Calling the function recursively

    return tree
```

Now the time to call the `buildTree()` function with given dataset!

```
#running the code to build the tree with given dataset
tree=buildTree(df)
```

Now internally the problem is solved and the result is stored in the form of a `dictionary`. The output can be displayed using following lines of code.

```
# printing the tree
import pprint
pprint.pprint(tree)
```

A complete tree visualization need additional library. That task is done with following code.

```
#code to visualize the decision tree
import pydot
import uuid

def generate_unique_node():
    """ Generate a unique node label."""
    return str(uuid.uuid1())

def create_node(graph, label, shape='oval'):
    node = pydot.Node(generate_unique_node(), label=label, shape=shape)
    graph.add_node(node)
    return node

def create_edge(graph, node_parent, node_child, label):
    link = pydot.Edge(node_parent, node_child, label=label)
    graph.add_edge(link)
    return link

def walk_tree(graph, dictionary, prev_node=None):
    """ Recursive construction of a decision tree stored as a dictionary """
    for parent, child in dictionary.items():
        # root
        if not prev_node:
            root = create_node(graph, parent)
            walk_tree(graph, child, root)
            continue

        # node
        if isinstance(child, dict):
            for p, c in child.items():
                n = create_node(graph, p)
                create_edge(graph, prev_node, n, str(parent))
                walk_tree(graph, c, n)

        # leaf
        else:
            leaf = create_node(graph, str(child), shape='box')
            create_edge(graph, prev_node, leaf, str(parent))
```

Yes! we did it. Now combine all the sub-functions in the tree traversal and write `plot_tree()` function.

```
def plot_tree(dictionary, filename="DecisionTree.png"):
    graph = pydot.Dot(graph_type='graph')
    walk_tree(graph, tree)
    graph.write_png(filename)
```

```
plot_tree(tree)
```

By this the problem is solved using ID3 algorithm and tree is saved in a file named `DecisionTree.png` in the working directory.

3.8.11.2 Machine Learning Approach in Decision Tree Learning

To implement the ID3 algorithm for a decision tree in Python, you can use the `scikit-learn` library. Since the functions of this machine learning library will accept the data in the numerical format, first we have to use some encoding process (data pre-processing). The `Python` code for this approach is given below:

```
from sklearn.preprocessing import LabelEncoder
Le = LabelEncoder()

df['outlook'] = Le.fit_transform(df['outlook'])
df['temp'] = Le.fit_transform(df['temp'])
df['humidity'] = Le.fit_transform(df['humidity'])
df['windy'] = Le.fit_transform(df['windy'])
df['play'] = Le.fit_transform(df['play'])
```

Instead of using this label encoder function from `sklearn` library, one can directly use data conversion technique from `pandas` library as follows:

```
# Convert categorical variables to numerical representation
df_numeric = pd.get_dummies(df.iloc[:, :-1])
```

This method is just the native `pandas` approach. But the former is `sklearn` approach.

Now lets' use the `sklearn` library for the rest. Here we just split the input data and the out put column and train a `decision tree` model using built-in functions available in this library.

```
#splitting data into input and labels
y = df['play']
X = df.drop(['play'],axis=1)
```

Now the true machine learning steps come!

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

```
# Fitting the model
from sklearn import tree #loading the sub-module for decision tree
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

```
clf = tree.DecisionTreeClassifier(criterion = 'entropy')
clf = fit(X_train, y_train)
```

The two line code did the complete job of creating a decision tree based on the given training data. The first line create a `DecisionTreeClassifier` object and the last line train the model using the given training data. Now the tree can be plot using the following code.

```
# We can visualize the tree using tree.plot_tree
tree.plot_tree(clf)
```

A more visually appealing tree diagram can be created using the `graphviz` library as shown bellow:

```
import graphviz
dot_data = tree.export_graphviz(clf, out_file=None)
graph = graphviz.Source(dot_data)
graph
```

Model Evaluation: Performance of this model can be tested using the various performance measures as follows:

```
# Make predictions on the test set
y_pred = clf.predict(X_test)
# Display accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")
```

We can use other performance measures like `confusion matrix`, `precision`, `recall`, `f1 score`, and the `classification report`. Implementation of these calculations are given below:

```
from sklearn import metrics
cm = confusion_matrix(y_test, y_pred)
print(cm)
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
print("Precision:",metrics.precision_score(y_test, y_pred))
print("Recall:",metrics.recall_score(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

Task2: From the 9 days of market behaviour a table is created. Past trend, Open interest and Trading volume are the factors affecting the market. Create a decision tree to represent the relationship between the features and market behaviour.

Past Trend	Open Interest	Trading Volume	Return
Positive	Low	High	Up
Negative	High	Low	Down
Positive	Low	High	Up
Positive	High	High	Up
Negative	Low	High	Down
Positive	Low	Low	Down
Negative	High	High	Down
Positive	Low	Low	Down
Positive	High	High	Up

3.8.11.3 Direct ID3 algorithm v/s Machine Learning approach in Decision Tree learning

Scikit-Learn Approach: The scikit-learn approach to implementing the ID3 algorithm leverages the simplicity and efficiency of a well-established machine learning library. Using scikit-learn allows for a streamlined development process, where the implementation is abstracted, and the user can easily create, train, and evaluate decision tree models. The library follows a consistent interface for various machine learning algorithms, providing a user-friendly experience. Additionally, scikit-learn integrates seamlessly with other tools for model evaluation and selection, making it suitable for practical, real-world scenarios. However, this convenience comes with a trade-off, as the user has less control over the internal workings of the algorithm, limiting customization options.

Custom Implementation Approach: On the other hand, a custom implementation of the ID3 algorithm offers more flexibility and control over the decision tree's behavior. This approach is particularly valuable for educational purposes, as it allows developers to gain a deeper understanding of the underlying principles and mechanisms of the algorithm. Custom implementations also provide the ability to fine-tune the algorithm to handle specific nuances in the data or meet unique requirements. However, the downside includes increased complexity, especially for those less familiar with the intricacies of the algorithm. Maintenance and performance considerations may also pose challenges, as custom implementations might require more effort to update and may not be as optimized as library solutions.

Considerations: The choice between the two approaches depends on the specific context, goals, and trade-offs. For quick model development and deployment, especially in real-world applications, the scikit-learn approach is often preferable due to its simplicity and efficiency. In contrast, a custom implementation is advantageous when in-depth understanding, educational value, and fine-tuning capabilities are priorities.

3.8.12 3. Support Vector Machines

Support Vector Machines (SVM) is a supervised machine learning algorithm that analyzes and classifies data by finding the optimal hyperplane that best separates different classes in a feature space. The key idea behind SVM is to identify the hyperplane that maximizes the margin, i.e., the distance between the hyperplane and the nearest data points (support vectors) from each class. SVM is effective in high-dimensional spaces, making it suitable for a wide range of applications, including both linear and non-linear data.

In theory, the SVM algorithm, aka the support vector machine algorithm, is linear. What makes the SVM algorithm stand out compared to other algorithms is that it can deal with classification problems using an SVM classifier and regression problems using an SVM regressor. However, one must remember that the SVM classifier is the backbone of the support vector machine concept and, in general, is the aptest algorithm to solve classification problems.

Being a linear algorithm at its core can be imagined almost like a Linear or Logistic Regression. For example, an SVM classifier creates a line (plane or hyper-plane, depending upon the dimensionality of the data) in an N-dimensional space to classify data points that belong to two separate classes. It is also noteworthy that the original SVM classifier had this objective and was originally designed to solve binary classification problems, however unlike, say, linear regression that uses the concept of line of best fit, which is the predictive line that gives the minimum Sum of Squared Error (if using OLS Regression), or Logistic Regression that uses Maximum Likelihood Estimation to find the best fitting sigmoid curve, Support Vector Machines uses the concept of Margins to come up with predictions.

Before understanding how the SVM algorithm works to solve classification and regression-based problems, it's important to appreciate the rich history. SVM was developed by Vladimir Vapnik in the 1970s. As the legend goes, it was developed as part of a bet where Vapnik envisaged that coming up with a decision boundary that tries to maximize the margin between the two classes will give great results and overcome the problem of overfitting. Everything changed, particularly in the '90s when the kernel method was introduced that made it possible to solve non-linear problems using SVM. This greatly affected the importance and development of neural networks for a while, as they were extremely complicated. At the same time, SVM was much simpler than them and still could solve non-linear classification problems with ease and better accuracy. In the present time, even with the advancement of Deep Learning and Neural Networks in general, the importance and reliance on SVM have not diminished, and it continues to enjoy praises and frequent use in numerous industries that involve machine learning in their functioning.

The best way to understand the SVM algorithm is by focusing on its primary type, the SVM classifier. The idea behind the SVM classifier is to come up with a hyper-lane in an N-dimensional space that divides the data points belonging to different classes. However, this hyper-pane is chosen based on margin as the hyperplane providing the maximum margin between the two classes is considered. These margins are calculated using data points known as Support Vectors. Support Vectors are those data points that are near to the hyper-plane and help in orienting it.

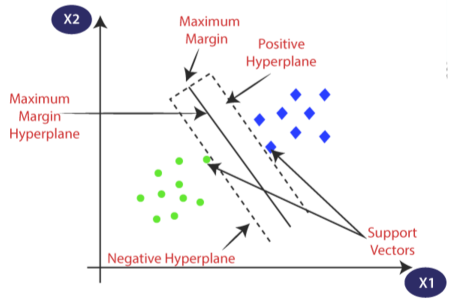

FIGURE 3.1 Schematic Diagram of SVM

All the key elements of the SVM binary classifier is explained in the Figure 3.1.

3.8.12.1 Linear Decision Function

SVM starts with a linear decision function. Given a dataset with feature vectors x and corresponding labels y in $\{-1, 1\}$, the decision function is $f(x) = w * x + b$, where w is the weight vector and b is the bias term. The classification rule is straightforward: if $f(x) >= 0$, assign the data point to class 1; otherwise, assign it to class -1.

3.8.12.2 Geometric Interpretation

The hyperplane in SVM is chosen to maximize the margin, which is the distance between the hyperplane and the nearest data points from each class.

These crucial data points are called support vectors, and the optimization goal is to maximize the margin for a more robust and generalized model.

3.8.13 Optimization Process

3.8.13.1 Objective Function and Lagrangian

To formalize the optimization problem, SVM introduces an objective function that reflects the margin. The goal is to maximize $M = 2/||w||$ subject to the constraint $y_i(w * x_i + b) >= 1$ for all data points. This is a constrained optimization problem, and we formulate the Lagrangian:

$$\mathcal{L}(w, b, \alpha) = \frac{1}{2}||w||^2 - \sum_{i=1}^{n} \alpha_i [y_i(w * x_i + b) - 1]$$

3.8.13.2 Dual Problem and Optimization

By setting the partial derivatives of the Lagrangian with respect to w and b to zero, we derive the dual problem. The solution involves finding the optimal values of Lagrange multipliers ($alpha$), which correspond to the support vectors. The decision function becomes $f(x) = \sum(\alpha_i * y_i * x_i * x) + b$, emphasizing the importance of support vectors in defining the hyperplane.

3.8.14 Non-Linear Extension with Kernels

3.8.14.1 Kernel Trick

SVM's applicability extends beyond linearly separable data. The kernel trick allows SVM to handle non-linear data by mapping the input space into a higher-dimensional space. The decision function becomes $f(x) = \sum(\alpha_i * y_i * K(x_i, x)) + b$, where $K(x_i, x)$ is the kernel function. Common kernels include the linear kernel $K(x_i, x) = x_i * x)$, polynomial kernel, and radial basis function (RBF) kernel.

3.8.14.2 Soft Margin SVM

In realistic scenarios, data might not be perfectly separable. Soft Margin SVM introduces slack variables to allow for some mis-classification, balancing the trade-off between maximizing the margin and minimizing mis-classification.

3.8.14.3 Multiclass SVM

While SVM was originally designed for binary classification, it can be extended to handle multiple classes using strategies like one-vs-one or one-vs-all.

Example: Design a SVM classifier to classify the samples in the iris dataset into defined classes and use suitable measures for report the performance of the model.

Solution:

The `Python` code for solving this problem is shown bellow:

```
# Import required libraries
from sklearn import datasets
from sklearn.model_selection import train_test_split,GridSearchCV
from sklearn.svm import SVC # SVC stands for support vector classifier
from sklearn.metrics import accuracy_score
```

```
# Load the iris dataset
iris = datasets.load_iris()
```

```
# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.3, random_
```

```
# Create an SVM classifier object
svm = SVC(kernel='linear', C=1, random_state=42)
```

```
# Train the SVM classifier on the training data
svm.fit(X_train, y_train)
```

```
# predicting on traing dataset
y_pred = svm.predict(X_train)
```

```
# Evaluate the accuracy of the classifier on training
accuracy = accuracy_score(y_train, y_pred)
print('Accuracy:', accuracy)
```

```
# Use the trained SVM classifier to make predictions on the testing data
y_pred = svm.predict(X_test)
```

```
# Evaluate the accuracy of the classifier
accuracy = accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy)
```

```
# Identifying the best parameters
grid = {
    'C':[0.01,0.1,1,10],
    'kernel' : ["linear","poly","rbf","sigmoid"],
    'degree' : [1,3,5,7],
    'gamma' : [0.01,1]
}
```

```
svm  = SVC ()
svm_cv = GridSearchCV(svm, grid, cv = 5)
svm_cv.fit(X_train,y_train)
print("Best Parameters:",svm_cv.best_params_)
print("Train Score:",svm_cv.best_score_)
print("Test Score:",svm_cv.score(X_test,y_test))
```

```
svm_cv.predict(X_test)
```

```
# visualize seperation boundaries

svm.fit(X_train, y_train)
import numpy as np
import matplotlib.pyplot as plt
# Create a meshgrid of points to plot the decision boundary
x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))
Z = svm.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Plot the decision boundary and the training points
plt.contourf(xx, yy, Z, alpha=0.4)
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, alpha=0.8)
plt.xlabel('Sepal length')
plt.ylabel('Sepal width')
plt.title('SVM classification on Iris dataset')
plt.show()
```

3.8.15 Ensemble methods in Machine Learning

Ensemble methods in machine learning represent a sophisticated approach to enhancing predictive performance by combining the strengths of multiple models. The core idea is rooted in the concept that the collective knowledge and insights of diverse models can compensate for individual model limitations, leading to more accurate and robust predictions. These methods have gained immense popularity due to their ability to significantly improve both classification and regression tasks.

One prevalent category of ensemble methods is Bagging, which involves training multiple instances of the same base model on different subsets of the training data. This process, known as Bootstrap Aggregating, helps mitigate overfitting and improves the overall stability of the model. Random Forest, a popular algorithm, employs bagging with decision trees as base models, resulting in a robust and accurate predictive model.

Boosting, another key category, focuses on sequentially training weak learners, each emphasizing the misclassified instances of the previous ones. AdaBoost and Gradient Boosting Machines are notable algorithms within this category, emphasizing the importance of learning from mistakes to iteratively improve the model's predictive power.

Stacking, a more advanced ensemble technique, incorporates diverse base models, and a meta-model is trained to combine their predictions effectively. Stacking leverages the strengths of different algorithms and can yield superior performance compared to individual models. Additionally, voting classifiers/regressors provide a simple yet effective ensemble strategy, making predictions based on the majority vote or average of individual models' predictions.

Ensemble methods offer several advantages, including increased accuracy, improved generalization, robustness to noisy data, and versatility across various machine learning domains. However, they come with challenges, such as increased computational complexity, potential loss of interpretability, and the necessity of selecting diverse and high-quality base models. Overall, understanding and harnessing the potential of ensemble methods empower machine learning practitioners to build more resilient and powerful models for diverse applications.

3.8.16 Comparison between Regression and Classification Models

A brief comparison of regression and classification models.

Aspect	Regression Model	Classification Model
Nature of Target Variable	Continuous numeric value	Categorical (discrete) classes
Output Representation	Single numeric value	Class label or category
Model Function	Captures relationship with continuous output	Defines decision boundaries for classes
Evaluation Metrics	Mean Squared Error (MSE), Mean Absolute Error (MAE), R-squared	Accuracy, Precision, Recall, F1 Score, Confusion Matrix
Application	Predicting quantities (e.g., sales, temperature)	Categorizing instances (e.g., spam detection, image recognition)
Decision Boundary	Not applicable (continuous output)	Defines regions separating different classes
Example	Predicting house prices, stock prices	Classifying emails as spam or non-spam

4

Un-supervised Learning

Unsupervised learning is a fundamental paradigm in machine learning that deals with the exploration and extraction of patterns from unlabeled data. Unlike supervised learning, where the model is trained on labeled examples with explicit input-output pairs, unsupervised learning operates on data lacking predefined labels. The primary objective is to uncover inherent structures, relationships, or representations within the data without explicit guidance.

4.1 Clustering

Clustering is a fundamental unsupervised learning technique that involves grouping similar data points together based on certain criteria, aiming to discover intrinsic patterns within the dataset. The goal is to create homogeneous groups or clusters, where data points within the same cluster share more similarities with each other than with those in other clusters. This process helps in uncovering the inherent structure of the data, providing valuable insights for various applications such as customer segmentation, anomaly detection, and data organization.

4.1.1 Different clustering approaches

There are various types of clustering methods, each with its own approach and characteristics. Here are some commonly used types of clustering:

1. Centroid-Based Clustering
2. Hierarchical Clustering
3. Density-Based Clustering
4. Distribution-Based Clustering
5. Graph-Based Clustering
6. Fuzzy Clustering

7. Model-Based Clustering

In this foundation course, we will discuss only the first two types of clustering algorithms.

4.1.2 1. Centroid-Based Clustering

Centroid-based clustering is a type of clustering algorithm that partitions the data into clusters, where each cluster is represented by a central point known as the centroid. The goal is to minimize the distance between data points in the same cluster and maximize the separation between different clusters. One of the most widely used centroid-based clustering algorithms is K-Means.

4.1.2.1 K-Means Algorithm:

4.1.2.2 Mathematical Formulation:

Given a dataset $X = \{x_1, x_2, \dots, x_n\}$ with n data points in a p-dimensional space, and a predefined number of clusters k, the K-Means algorithm aims to find k centroids $c_1, c_2, \dots, c_k$ and assign each data point to the cluster with the nearest centroid.

1. **Initialization:**
   - Randomly select k initial centroids $c_1^{(0)}, c_2^{(0)}, ..., c_k^{(0)}$.
2. **Assignment Step:**
   - For each data point x_i, compute its distance to each centroid c_j:

     $$d(x_i, c_j) = \sqrt{\sum_{l=1}^{p} (x_{il} - c_{jl})^2}$$

   - Assign x_i to the cluster with the nearest centroid:

     $$\text{Cluster}(x_i) = \arg\min_j d(x_i, c_j)$$
3. **Update Step:**
   - Recalculate the centroids based on the data points in each cluster:

     $$c_j^{(t+1)} = \frac{1}{|C_j|} \sum_{x \in C_j} x$$

   - Here, t represents the iteration, C_j is the set of data points assigned to cluster j, and $|C_j|$ is the number of data points in cluster j.

4. **Convergence:**
    - Repeat the Assignment and Update steps until convergence criteria are met (e.g., minimal change in centroids or a predefined number of iterations).

4.1.2.3 Objective Function (Cost Function):

The K-Means algorithm minimizes the following objective function, often referred to as the "within-cluster sum of squares" or "inertia":

$$J = \sum_{j=1}^{k} \sum_{x \in C_j} \|x - c_j\|^2$$

Here, $\|x - c_j\|^2$ represents the squared Euclidean distance between data point x and centroid c_j.

4.1.2.3.1 Limitations:

- K-Means is sensitive to the initial choice of centroids, and different initializations may lead to different final clusters.
- The algorithm assumes spherical and equally-sized clusters, making it less suitable for non-spherical or unevenly sized clusters.

Despite its limitations, K-Means remains a widely used and computationally efficient method for centroid-based clustering in various applications.

4.1.2.4 Pseudocode:

```
# Input
X = $\{x_1, x_2, ..., x_n\}$
k = Number of clusters

# Initialization
Randomly select k initial centroids $c_1^(0), c_2^(0), ..., c_k^(0)$

# Repeat until convergence
while not converged:
  # Assignment Step
  for each data point x_i:
    Compute distances to all centroids: d(x_i, c_j) for j = 1 to k
    Assign x_i to the cluster with the nearest centroid: Cluster(x_i) = arg min_j d(x_i, c_j)
  # Update Step
  for each cluster j = 1 to k:
    Recalculate the centroid: c_j^(t+1) = (1/|C_j|) * \sum_{x in C_j} x

  # Convergence criteria
  Check if centroids do not change significantly or reach a predefined number of iterations.
```

4.1.2.5 Simple example of K-means clustering using `iris` dataset.

In this example we will demonstrate how to use clustering to find an approximate classification of iris flowers using the input features only.

Step 1: Loading libraries

```
import numpy as np
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.datasets import load_iris
import matplotlib.pyplot as plt
```

Loading the dataset

```
# Load the Iris dataset
iris = load_iris()
X = iris.data # create only input data not target
```

Visulazing data point distribution:

```
# Visualize the clusters
plt.scatter(X[:, 0], X[:, 1])
#plt.scatter(centroids[:, 0], centroids[:, 1], marker='X', s=200, c='red')
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1])
plt.title('Distribution of Iris data')
plt.show()
```

Finding the best value of k using elbow method:

```
# Initialize an empty list to store the within-cluster sum of squares
wcss = []

# Try different values of k
for k in range(1, 11):
    kmeans = KMeans(n_clusters=k,n_init='auto', random_state=42)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)# here inertia calculate sum of square distance in each cluster
```

Creating elbow plot:

```
# Plot the within-cluster sum of squares for different values of k
plt.plot(range(1, 11), wcss, marker='o')
plt.xlabel('Number of Clusters (k)')
plt.ylabel('Within-Cluster Sum of Squares (WCSS)')
plt.title('Elbow Method')
plt.show()
```

Using Gridsearch method:

```
# Define the parameter grid
from sklearn.model_selection import GridSearchCV

param_grid = {'n_clusters': [2, 3, 4, 5, 6]}

# Create a KMeans object
kmeans = KMeans(n_init='auto',random_state=42)

# Create a GridSearchCV object
grid_search = GridSearchCV(kmeans, param_grid, cv=5)

# Perform grid search
grid_search.fit(X)

# Get the best parameters and the best score
best_params = grid_search.best_params_
best_score = grid_search.best_score_
```

Showing best k-value:

```
print("Best Parameters:", best_params)
print("Best Score:", best_score)
```

Implementing K-means clustering:

```
# Perform k-means clustering
k = 6 # Number of clusters
kmeans = KMeans(n_clusters=k,n_init='auto', random_state=42)
kmeans.fit(X)
```

Extracting labels and cluster centers:

```
# Get the cluster labels and centroids
labels = kmeans.labels_
centroids = kmeans.cluster_centers_

# Add the cluster labels to the DataFrame
df['Cluster'] = labels
```

Glimpse of cluster labels:

```
df.head()
```

Visualizing the clustering using first two features:

```
# Visualize the clusters
plt.scatter(X[:, 0], X[:, 1], c=labels, cmap='viridis')
plt.scatter(centroids[:, 0], centroids[:, 1], marker='X', s=200, c='red')
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1])
plt.title('K-means Clustering')
plt.show()
```

Conclusion: Originally there are only three categories of iris flowers. But with all possible four features, the model predict there are six types in iris flowers. So clustering is not a good option to label the iris dataset using input features.

4.1.3 2.Hierarchical Clustering

Hierarchical clustering is a type of clustering algorithm that organizes data points into a tree-like structure, known as a dendrogram. This method creates a hierarchy of clusters, revealing relationships and structures within the dataset. Hierarchical clustering can be either agglomerative (bottom-up) or divisive (top-down), and it does not require a predefined number of clusters.

4.1.3.1 Agglomerative Hierarchical Clustering:

Concept of Similarity and `proximity-matrix`:

Take the distance between the centroids of these clusters. The points having the least distance are referred to as similar points and we can merge them. We can refer to this as a distance-based algorithm as well (since we are calculating the distances between the clusters).

In hierarchical clustering, we have a concept called a `proximity matrix`. This stores the distances between each point. Let's take an example to understand this matrix as well as the steps to perform hierarchical clustering.

Example: Suppose a teacher wants to divide her students into different groups. She has the marks scored by each student in an assignment and based on these marks, she wants to segment them into groups. There's no fixed target here as to how many groups to have. Since the teacher does not know what type of students should be assigned to which group, it cannot be solved as a supervised learning problem. So, we will try to apply hierarchical clustering here and segment the students into different groups.

Let's take a sample of 5 students:

1	10
2	7
3	28
4	20
5	35

Creating a Proximity Matrix First, we will create a proximity matrix which will tell us the distance between each of these points. Since we are calculating the distance of each point from each of the other points, we will get a square matrix of shape n X n (where n is the number of observations).

Let's make the 5 x 5 proximity matrix for our example:

	1	2	3	4	5
1	0	3	18	10	25
2	3	0	21	13	28
3	18	21	0	8	7
4	10	13	8	0	15
5	25	28	7	15	0

The diagonal elements of this matrix will always be 0 as the distance of a point with itself is always 0. We will use the Euclidean distance formula to calculate the rest of the distances.

4.2 Steps to Perform Hierarchical Clustering

1. Step 1: First, we assign all the points to an individual cluster:
2. Step 2: Next, we will look at the smallest distance in the proximity matrix and merge the points with the smallest distance. We then update the proximity matrix:

Note: Here, the smallest distance is 3 and hence we will merge point 1 and 2: Here, we have taken the maximum of the two marks (7, 10) to replace the marks for this cluster. Instead of the maximum, we can also take the minimum value or the average values as well. Now, we will again calculate the proximity matrix for these clusters:

ID	$Marks$
$(1,2)$	10
3	28
4	20
5	35

Updating proximity-matrix:

Now update the proximity matrix as:

ID	$(1,2)$	3	4	5
$(1,2)$	0	18	10	25
3	18	0	8	7
4	10	8	0	15
5	25	7	15	0

Next level clusters:

3 and 5 together form new cluster. Updated data is:

$(1,2)$	10
$(3,5)$	35
4	20

New proximity-matrix:

ID	$(1,2)$	$(3,5)$	4
$(1,2)$	0	25	10
$(3,5)$	25	0	15
4	10	15	0

New cluster (1,2,4). Updated data:

$(1,2,4)$	20
$(3,5)$	35

4.2.1 Mathematical Formulation:

Given a dataset $X = \{x_1, x_2, ..., x_n\}$ with n data points in a p-dimensional space, agglomerative hierarchical clustering proceeds as follows:

1. **Initialization:**
    - Treat each data point as a singleton cluster, resulting in n initial clusters.
2. **Merge Step:**
    - Identify the two clusters with the smallest dissimilarity or linkage and merge them into a new cluster.
    - The dissimilarity or linkage between two clusters can be measured using methods such as:
        - **Single Linkage:** Minimum distance between any two points in the clusters.
        - **Complete Linkage:** Maximum distance between any two points in the clusters.
        - **Average Linkage:** Average distance between all pairs of points in the clusters.
3. **Update Dissimilarity Matrix:**
    - Recalculate the dissimilarity or linkage between the new cluster and the remaining clusters.
4. **Repeat:**
    - Repeat the Merge and Update steps until all data points belong to a single cluster, forming a dendrogram.

4.2.2 Dendrogram:

A dendrogram visually represents the hierarchy of clusters and the order in which they are merged. The vertical lines in the dendrogram represent clusters, and the height at which two clusters merge indicates their dissimilarity.

4.2.3 `Python` code to implement Heirarchical Clustering.

Task: Suppose a teacher wants to divide her students into different groups. She has the marks scored by each student in an assignment and based on these marks, she wants to segment them into groups. There's no fixed target here as to how many groups to have. Since the teacher does not know what type of students should be assigned to which group, it cannot be solved as a supervised learning problem. So, we will try to apply hierarchical clustering here and segment the students into different groups.

Let's take a sample of 5 students:

1	10
2	7
3	28
4	20
5	35

```
#loading libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

```
df=pd.DataFrame([10,7,28,20,35],columns=["Marks"])
```

```
import scipy.cluster.hierarchy as shc
plt.figure(figsize=(10, 7))
plt.title("Dendrograms")
dend = shc.dendrogram(shc.linkage(df, method='ward'))
plt.axhline(y=3, color='r', linestyle='--')
```

```
# running clustering
from sklearn.cluster import AgglomerativeClustering
cluster = AgglomerativeClustering(n_clusters=2, affinity='euclidean', linkage='ward')
cluster.fit_predict(df)
```

4.2.4 Limitations:

- The time complexity of agglomerative hierarchical clustering can be high, especially for large datasets.
- It is sensitive to noise and outliers.

Despite its limitations, hierarchical clustering is valuable for revealing structures within the data, providing insights into the relationships between data points at different scales.

4.2.5 Pseudocode:

```
# Input
X = {x_1, x_2, ..., x_n}

# Initialization
Create n singleton clusters, one for each data point

# Repeat until a single cluster is formed
while number of clusters > 1:
  Find the two clusters with the minimum dissimilarity or linkage
  Merge the two clusters into a new cluster
  Update the dissimilarity matrix

# Dendrogram Visualization
Plot the dendrogram to visualize the hierarchy of clusters
```

4.3 Dimensionality Reduction

Dimensionality reduction is a critical technique in machine learning and data analysis, playing a pivotal role in feature engineering. As datasets grow in complexity with an increasing number of features, the curse of dimensionality becomes a significant challenge. Dimensionality reduction addresses this challenge by transforming high-dimensional data into a lower-dimensional representation, preserving essential information while improving computational efficiency and model performance.

4.3.1 The Challenge of High Dimensionality

In many real-world applications, datasets contain a large number of features, each representing a different aspect of the data. High-dimensional datasets can suffer from several issues, such as increased computational complexity, elevated risk of overfitting, and challenges in visualizing and interpreting data. Dimensionality reduction aims to mitigate these problems by extracting the most relevant information from the original features while reducing redundancy and noise.

4.3.2 Key Objectives of Dimensionality Reduction

1. **Computational Efficiency:** High-dimensional datasets often require extensive computational resources. By reducing the number of features, dimensionality reduction accelerates the training and evaluation of machine learning models, making them more scalable and efficient.
2. **Overfitting Prevention:** With an abundance of features, models can become overly complex and may fit noise in the data rather than capturing the underlying patterns. Dimensionality reduction helps prevent overfitting by focusing on the most significant features, enhancing a model's generalization to new, unseen data.
3. **Improved Model Performance:** Reducing dimensionality often results in a more concise and informative representation of the data. This can lead to improved model performance, especially in scenarios where the original feature space is noisy or redundant.

4.3.3 Role in Feature Engineering

Feature engineering involves the process of selecting, transforming, or creating features to enhance the performance of machine learning models. Dimensionality reduction is a crucial aspect of feature engineering as it allows practitioners to distill the most critical aspects of the data into a more manageable form. This distilled representation serves as a foundation for building more effective models, simplifying the feature space without sacrificing predictive power.

4.3.4 Common Dimensionality Reduction Techniques

1. **Principal Component Analysis (PCA):** PCA is a widely-used linear technique that identifies orthogonal axes (principal components) along which the variance of the data is maximized. It projects

the data onto a lower-dimensional subspace while retaining as much variance as possible.

2. **t-Distributed Stochastic Neighbor Embedding (t-SNE):** t-SNE is a non-linear dimensionality reduction technique that excels in visualizing high-dimensional data in lower-dimensional spaces. It is particularly useful for exploring the local relationships between data points.
3. **Autoencoders:** Autoencoders are neural network architectures that learn a compressed, meaningful representation of the input data. They consist of an encoder and a decoder, working together to reduce dimensionality while preserving important features.

Dimensionality reduction is a crucial tool in the realm of feature engineering. By transforming high-dimensional data into a more compact and informative representation, dimensionality reduction techniques contribute to more efficient, interpretable, and accurate machine learning models. This process is especially valuable when dealing with complex datasets where the sheer number of features poses computational and modeling challenges.

4.3.4.1 1. Principal Component Analysis

PCA transforms high-dimensional data into a lower-dimensional representation, capturing the most significant variance in the data. The resulting components, called principal components, provide a new basis for the data that emphasizes its essential features.

Mathematical Backgrounds: Given a dataset X with n data points, each of dimension p, PCA aims to find a set of orthogonal vectors (principal components) that represent the directions of maximum variance in the data. The principal components are linear combinations of the original features. The steps involved in PCA are as follows:

Step 1: Data Mean-Centering

Mean-Centering: Subtract the mean of each feature from the dataset.

$$\bar{x}_j = x_j - \frac{1}{n}\sum_{i=1}^{n} x_{ij}$$

This ensures that the data is centered around the origin.

Step 2: Covariance Matrix Calculation

Covariance Matrix (C): Calculate the covariance matrix for the mean-centered data.

$$C_{jk} = \frac{1}{n-1}\sum_{i=1}^{n} \bar{x}_{ij}\bar{x}_{ik}$$

Step 3: Eigen decomposition of Covariance Matrix

Eigendecomposition: Compute the eigenvectors ($\mathbf{v}_i$) and eigenvalues (λ_i) of the covariance matrix (C).

$$C\mathbf{v}_i = \lambda_i \mathbf{v}_i$$

Each $\mathbf{v}_i$ represents a principal component direction.

Step 4: Principal Components Selection

Principal Components: Select the top k eigenvectors corresponding to the k largest eigenvalues to form the projection matrix (W).

- Arrange the selected eigenvectors as columns in W.
- The data can be projected onto the lower-dimensional subspace using $Z = XW$.

- The eigenvalues λ_i represent the amount of variance captured by each principal component.
- Larger eigenvalues correspond to more significant variability in the data.
- Choosing k depends on the desired level of dimensionality reduction and the cumulative explained variance.

4.3.4.2 Demonstration of `PCA` on MNIST digit dataset

This section is devoted to the illustration of the Principal Components Analysis in a built-in dataset. As a first step, load the necessary `Python` libraries and dependancies for this example.

```
from __future__ import print_function
import time
import numpy as np
import scipy
import pandas as pd
from sklearn.datasets import fetch_openml
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
%matplotlib inline
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sns
```

Next is loading the dataset from the repository and check the dimension.

```
mnist = fetch_openml('mnist_784', cache=False)
X = mnist.data / 255.0 # scaling of data
y = mnist.target
print(X.shape, y.shape)
```

We are going to convert the matrix and vector to a `Pandas` DataFrame. This is very similar to the DataFrames used in `R` and will make it easier for us to plot it later on.

```
df = pd.DataFrame(X)
#feat_cols = [ 'pixel'+str(i) for i in range(1,X.shape[1]) ]
feat_cols=df.columns
df['y'] =y
df['label'] = df['y'].apply(lambda i: str(i))
#X, y = None, None
print('Size of the dataframe: {}'.format(df.shape))
```

Also convert the target variable into `int` type.

```
df.y=df.y.astype(int)
```

Now just have a look at the cleaned dataset.

```
df.head()
```

Next stage is the randomization and sampling (the systematic approach used in Statistical Analysis to avoid bias). Because we dont want to be using 70,000 digits in some calculations we'll take a random subset of the digits. The randomization is important as the dataset is sorted by its label (i.e., the first seven thousand or so are zeros, etc.). To ensure randomization we'll create a random permutation of the number 0 to 69,999 which allows us later to select the first five or ten thousand for our calculations and visualizations.

```
# For reproducability of the results
np.random.seed(42)
rndperm = np.random.permutation(df.shape[0])
rndperm
```

We now have our dataframe and our randomization vector. Lets first check what these numbers actually look like (checking sampling power). To do this we'll generate 30 plots of randomly selected images.

```
plt.gray()
fig = plt.figure( figsize=(16,7) )
for i in range(1,16):
    ax = fig.add_subplot(3,5,i, title="Digit: {}".format(str(df.loc[rndperm[i],'label'])) )
    ax.matshow(df.loc[rndperm[i],feat_cols].values.reshape((28,28)).astype(float))
plt.show()
```

The images are all essentially 28-by-28 pixel images and therefore have a total of 784 `dimensions`, each holding the value of one specific pixel. What we can do is reduce the number of dimensions drastically whilst trying to retain as much of the 'variation' in the information as possible. This is where we get to dimensionality reduction. Lets first take a look at the Principal Component Analysis.

```
pca = PCA(n_components=3)
pca_result = pca.fit_transform(df[feat_cols].values)
df['pca-one'] = pca_result[:,0]
df['pca-two'] = pca_result[:,1]
df['pca-three'] = pca_result[:,2]
print('Explained variation per principal component: {}'.format(pca.explained_variance_ratio_))
```

Now, given that the first two components account for about 25% of the variation in the entire dataset lets see if that is enough to visually set the different digits apart. What we can do is create a `scatterplot` of the first and second principal component and color each of the different types of digits with a different color. If we are lucky the same type of digits will be positioned (i.e., clustered) together in groups, which would mean that the first two principal components actually tell us a great deal about the specific types of digits.

```
plt.figure(figsize=(16,10))
sns.scatterplot(
    x="pca-one", y="pca-two",
    hue="y",
    palette=sns.color_palette("hls", 10),
    data=df.loc[rndperm,:],
    legend="full",
    alpha=0.3
)
```

From the graph we can see the two components definitely hold some information, especially for specific digits, but clearly not enough to set all of them apart. Luckily there is another technique that we can use to reduce the number of dimensions that may prove more helpful. For a 3d-version of the same plot

```
from mpl_toolkits import mplot3d
ax = plt.figure(figsize=(16,10))
ax = plt.axes(projection ='3d')
ax.scatter(
    xs=df.loc[rndperm,:]["pca-one"],
    ys=df.loc[rndperm,:]["pca-two"],
    zs=df.loc[rndperm,:]["pca-three"],
    c=df.loc[rndperm,:]["y"],
    cmap='tab10'
)
```

```
ax.set_xlabel('pca-one')
ax.set_ylabel('pca-two')
ax.set_zlabel('pca-three')
plt.show()
```

In the next few paragraphs we are going to take a look at that technique and explore if it gives us a better way of reducing the dimensions for visualization. The method we will be exploring is known as t-SNE (t-Distributed Stochastic Neighbouring Entities).

4.3.4.3 2. t-Distributed Stochastic Neighbouring Entities

What is the power of t-SNE

> "t-Distributed stochastic neighbor embedding (t-SNE) minimizes the divergence between two distributions: a distribution that measures pairwise similarities of the input objects and a distribution that measures pairwise similarities of the corresponding low-dimensional points in the embedding".

Essentially what this means is that it looks at the original data that is entered into the algorithm and looks at how to best represent this data using less dimensions by matching both distributions. The way it does this is computationally quite heavy and therefore there are some (serious) limitations to the use of this technique. For example one of the recommendations is that, in case of very high dimensional data, you may need to apply another dimensionality reduction technique before using t-SNE:

Drawback of t-SNE

> "Since t-SNE scales quadratically in the number of objects N, its applicability is limited to data sets with only a few thousand input objects; beyond that, learning becomes too slow to be practical (and the memory requirements become too large)".

4.3.4.4 Develop a `t-SNE` model in `Python`

```
time_start = time.time()
tsne = TSNE(n_components=2, verbose=1, perplexity=40, n_iter=300)
tsne_results = tsne.fit_transform(data_subset)
print('t-SNE done! Time elapsed: {} seconds'.format(time.time()-time_start))
```

Now let's compare the t-SNE with a compatiable PCA model. For that purpose, create a PCA instance too.

```
N = 10000
df_subset = df.loc[rndperm[:N],:].copy()
data_subset = df_subset[feat_cols].values
pca = PCA(n_components=3)
pca_result = pca.fit_transform(data_subset)
df_subset['pca-one'] = pca_result[:,0]
df_subset['pca-two'] = pca_result[:,1]
df_subset['pca-three'] = pca_result[:,2]
print('Explained variation per principal component: {}'.format(pca.explained_variance_ratio_))
```

Now that we have the two resulting dimensions we can again visualise them by creating a scatter plot of the two dimensions and coloring each sample by its respective label.

```
df_subset['tsne-2d-one'] = tsne_results[:,0]
df_subset['tsne-2d-two'] = tsne_results[:,1]
plt.figure(figsize=(16,10))
sns.scatterplot(
    x="tsne-2d-one", y="tsne-2d-two",
    hue="y",
    palette=sns.color_palette("hls", 10),
    data=df_subset,
    legend="full",
    alpha=0.3
)
```

This is already a significant improvement over the PCA visualisation we used earlier. We can see that the digits are very clearly clustered in their own sub groups. If we would now use a clustering algorithm to pick out the seperate clusters we could probably quite accurately assign new points to a label. Just to compare PCA & T-SNE:

```
plt.figure(figsize=(16,7))
ax1 = plt.subplot(1, 2, 1)
sns.scatterplot(
    x="pca-one", y="pca-two",
    hue="y",
    palette=sns.color_palette("hls", 10),
    data=df_subset,
    legend="full",
    alpha=0.3,
    ax=ax1
)
ax2 = plt.subplot(1, 2, 2)
sns.scatterplot(
    x="tsne-2d-one", y="tsne-2d-two",
    hue="y",
    palette=sns.color_palette("hls", 10),
    data=df_subset,
    legend="full",
    alpha=0.3,
    ax=ax2
)
```

Model improvement stage: We'll now take the recommendations to heart and actually reduce the number of dimensions before feeding the data into the t-SNE algorithm. For this we'll use PCA again. We will first create a new dataset containing the fifty dimensions generated by the PCA reduction algorithm. We can then use this dataset to perform the t-SNE to improve performance.

```
pca_50 = PCA(n_components=50)
pca_result_50 = pca_50.fit_transform(data_subset)
print('Cumulative explained variation for 50 principal components: {}'.format(np.sum(pca_50.explai
```

The first 50 components roughly hold around 85% of the total variation in the data. So 50 dimensions is enough to capture data information. Now lets try and feed this data into the t-SNE algorithm. This time we'll use 10,000 samples out of the 70,000 to make sure the algorithm does not take up too much memory and CPU.

```
time_start = time.time()
tsne = TSNE(n_components=2, verbose=0, perplexity=40, n_iter=300)
tsne_pca_results = tsne.fit_transform(pca_result_50)
print('t-SNE done! Time elapsed: {} seconds'.format(time.time()-time_start))
```

A final model visualization of all the above mentioned models are created using the following `Python` code.

```
df_subset['tsne-pca50-one'] = tsne_pca_results[:,0]
df_subset['tsne-pca50-two'] = tsne_pca_results[:,1]
plt.figure(figsize=(16,4))
ax1 = plt.subplot(1, 3, 1)
sns.scatterplot(
    x="pca-one", y="pca-two",
    hue="y",
    palette=sns.color_palette("hls", 10),
    data=df_subset,
    legend="full",
    alpha=0.3,
    ax=ax1
)
ax2 = plt.subplot(1, 3, 2)
sns.scatterplot(
    x="tsne-2d-one", y="tsne-2d-two",
    hue="y",
    palette=sns.color_palette("hls", 10),
    data=df_subset,
    legend="full",
    alpha=0.3,
    ax=ax2
)
ax3 = plt.subplot(1, 3, 3)
sns.scatterplot(
    x="tsne-pca50-one", y="tsne-pca50-two",
    hue="y",
    palette=sns.color_palette("hls", 10),
    data=df_subset,
    legend="full",
    alpha=0.3,
    ax=ax3
)
```

In conclusion, a PCA+t-SNE combination performs well in many complex data structure.

4.4 Anomaly Detection

Anomaly detection detects data points in data that does not fit well with the rest of the data. It has a wide range of applications such as fraud detection, surveillance, diagnosis, data cleanup, and predictive maintenance. An interesting and comprehensive explanation of anomaly detection in a general setup is given in the blog post-

https://iwringer.wordpress.com/2015/11/17/anomaly-detection-concepts-and-techniques/

A

Declaration

This document is in draft format.