# INDEX

| S.No. | Objective | Date | Signature |
|---|---|---|---|
| 1 | To study about numpy, pandas and matplotlib libraries in python. | | |
| 2 | To perform data preprocessing and data summarization on iris dataset. | | |
| 3 | To perform data preprocessing and data visualization on iris dataset. | | |
| 4 | To implement k means clustering. | | |
| 5 | To implement data classification using KNN. | | |
| 6 | To implement decision tree using ID3 algorithm. | | |
| 7 | To implement decision tree using CART algorithm. | | |
| 8 | To implement decision tree using C4.5 algorithm. | | |
| 9 | To implement multi layer neural network. | | |

# Experiment 1

## Objective:

To study about numpy, pandas and matplotlib libraries in python.

## Theory

In Python programming, libraries such as NumPy, Pandas, and Matplotlib are fundamental for data manipulation, analysis, and visualization. These libraries streamline complex operations, enabling data scientists, analysts, and developers to handle large datasets, perform numerical computations, and visualize data effectively.

**NumPy (Numerical Python)** is a powerful library primarily used for numerical and matrix computations. It introduces the `ndarray` object, an N-dimensional array for efficiently storing and manipulating large arrays of homogeneous data. NumPy arrays are faster and more memory-efficient than traditional Python lists due to their fixed size and storage of elements in contiguous memory. This efficiency allows for rapid mathematical computations and operations across entire arrays without the need for explicit loops. NumPy also provides a range of mathematical functions, including linear algebra, Fourier transforms, and random number generation. The broadcasting feature in NumPy enables arithmetic operations on arrays of different shapes, making it flexible for various data manipulation tasks. NumPy is often used in fields like machine learning, scientific computing, and engineering, where heavy numerical computation is required.

**Pandas** is another essential library in Python, designed specifically for data manipulation and analysis. It provides two primary data structures: `Series` (1D) and `DataFrame` (2D), which are built on top of NumPy arrays. A DataFrame can hold heterogeneous data types across columns, making it ideal for handling and analyzing structured data. Pandas offers a variety of functions for data cleaning, filtering, grouping, merging, and aggregation. It supports handling missing data, which is a common issue in real-world datasets. Pandas also has tools for time series analysis, making it valuable for financial and temporal data. The ability to handle large datasets in-memory, apply various operations on data frames, and reshape data makes Pandas a powerful tool for any data-driven task. Through Pandas, data can be imported from numerous file formats, such as CSV, Excel, and SQL databases, facilitating easy data integration and analysis.

**Matplotlib** is a popular Python library for data visualization. Its primary goal is to provide an easy way to generate visual representations of data, making it easier to understand complex patterns and relationships. Matplotlib offers an extensive range of plotting options, including line plots, scatter plots, bar charts, histograms, and more. The library's core, `pyplot`, provides a MATLAB-like interface for creating interactive and customizable plots. It allows users to control various aspects of a plot, such as labels, colors, styles, and legends, enabling high levels of customization. Matplotlib is often combined with Pandas, as Pandas' built-in plotting functions use Matplotlib as the backend, simplifying the visualization of DataFrames directly. Visualizations are essential in data analysis, as they provide insights and highlight trends that may not be evident through raw numbers alone.

Together, NumPy, Pandas, and Matplotlib form the foundation of Python's data science stack. NumPy handles numerical computations, Pandas manages data manipulation, and Matplotlib provides visualization tools, making it easier for data scientists to analyze and interpret data.

## Result

As a result of this Experiment, we successfully wrote and executed the program to study about numpy, pandas and matplotlib libraries in python.

## Learning Outcomes

Understand and utilize Python libraries NumPy, Pandas, and Matplotlib for numerical operations, data manipulation, and data visualization in data science tasks.

# Experiment 2

## Objective:

To perform data preprocessing and data summarization on iris dataset.

## Theory

Data preprocessing and summarization are critical steps in the data analysis pipeline, particularly when working with machine learning models. These processes ensure that data is clean, consistent, and ready for analysis. The Iris dataset, one of the most well-known datasets in data science, serves as an excellent example for demonstrating these techniques. This dataset includes 150 samples of iris flowers, each described by four features: sepal length, sepal width, petal length, and petal width. Additionally, each sample is labelled as belonging to one of three species of iris: Iris-setosa, Iris-versicolor, and Iris-virginica.

**Data Preprocessing** is a series of steps used to prepare raw data for analysis or modelling. It involves cleaning and transforming the data, handling missing values, and ensuring consistency in data formats. In the case of the Iris dataset, data preprocessing might involve verifying that each feature is numeric and consistent in scale. Since the dataset does not contain missing values, a typical first step is to check for any outliers or data inconsistencies, though the Iris dataset is known for its clean structure. However, in more complex datasets, preprocessing could include filling missing values using techniques like mean imputation, median imputation, or even more sophisticated methods like k-nearest neighbours.

Another preprocessing step is **data normalization or standardization**, especially when working with distance-based machine learning models such as k-nearest neighbours. Normalization scales features to a range, typically [0,1], while standardization scales them to have a mean of 0 and a standard deviation of 1. For the Iris dataset, where all four features are continuous and on different scales, these transformations can ensure that each feature contributes equally to model performance.

**Data Summarization** follows preprocessing and is used to understand the characteristics of the dataset. Summary statistics provide insights into the distribution and spread of the data, aiding in pattern identification. Descriptive statistics like mean, median, standard deviation, minimum, and maximum values are calculated for each feature. For instance, the mean sepal length and standard deviation help provide a quick understanding of the central tendency and spread of this feature. Summarization also includes visual techniques such as histograms, box plots, and pair plots. For example, a pair plot can illustrate relationships between sepal length, sepal width, petal length, and petal width across the different species in the Iris dataset. Box plots can reveal the distribution of each feature, highlighting any outliers or variability between species.

Furthermore, **data visualization** is a part of data summarization that provides a graphical representation of statistical summaries. In the case of the Iris dataset, scatter plots of petal length versus petal width, colored by species, can reveal clusters of species and help in visualizing decision boundaries. These visualizations are crucial when interpreting data patterns before any modeling phase.

## Code & OUTPUT

In [1]:
```
print("Experiment No 02 : To perform data preprocessing and data summarization on iris dataset.")
```

```
Experiment No 02 : To perform data preprocessing and data summarization on iris dataset.
```

In [2]:
```
# Load libraries
import pandas as pd
from sklearn.datasets import load_iris
print("OUTPUT:\n\n")
# Load iris dataset
iris = load_iris()
df = pd.DataFrame(iris.data, columns=iris.feature_names)
df['species'] = iris.target
# Summarization
print(df.describe())
print(df.info())
# Checking for missing values
print(df.isnull().sum())
# Data Preprocessing (Normalizing the data)
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
df_scaled = pd.DataFrame(scaler.fit_transform(df.iloc[:, :-1]), columns=iris.feature_names)
```

```
OUTPUT:


       sepal length (cm)  sepal width (cm)  petal length (cm)  \
count         150.000000        150.000000         150.000000
mean            5.843333          3.057333           3.758000
std             0.828066          0.435866           1.765298
min             4.300000          2.000000           1.000000
25%             5.100000          2.800000           1.600000
50%             5.800000          3.000000           4.350000
75%             6.400000          3.300000           5.100000
max             7.900000          4.400000           6.900000

       petal width (cm)     species
count        150.000000  150.000000
mean           1.199333    1.000000
std            0.762238    0.819232
min            0.100000    0.000000
25%            0.300000    0.000000
50%            1.300000    1.000000
75%            1.800000    2.000000
max            2.500000    2.000000
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   sepal length (cm)  150 non-null    float64
 1   sepal width (cm)   150 non-null    float64
 2   petal length (cm)  150 non-null    float64
 3   petal width (cm)   150 non-null    float64
 4   species            150 non-null    int32
dtypes: float64(4), int32(1)
memory usage: 5.4 KB
None
sepal length (cm)    0
sepal width (cm)     0
petal length (cm)    0
petal width (cm)     0
species              0
dtype: int64
Scaled Data:
    sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0          -0.900681          1.019004          -1.340227         -1.315444
1          -1.143017         -0.131979          -1.340227         -1.315444
2          -1.385353          0.328414          -1.397064         -1.315444
3          -1.506521          0.098217          -1.283389         -1.315444
4          -1.021849          1.249201          -1.340227         -1.315444
```

## Result

As a result of this Experiment, we successfully wrote and executed the program to perform data preprocessing and data summarization on iris dataset.

.

## Learning Outcomes

Understand and apply data preprocessing and summarization techniques to clean, normalize, and analyse datasets, gaining insights into feature distributions and relationships.

# Experiment 3

## Objective:

To perform data preprocessing and data visualization on iris dataset.

## Theory

Data preprocessing and visualization are foundational steps in data analysis, particularly in machine learning and data science. They prepare data for deeper analysis and help reveal underlying patterns. Using the Iris dataset—a widely studied dataset in data science that includes 150 observations of iris flowers—allows us to practice these techniques effectively. This dataset contains four numerical features: sepal length, sepal width, petal length, and petal width, as well as a categorical target label representing the species of the iris flower: Iris-setosa, Iris-versicolor, and Iris-virginica.

**Data Preprocessing** involves cleaning and transforming data to improve its quality and suitability for analysis. Preprocessing can include multiple steps, such as handling missing values, scaling features, encoding categorical data, and removing duplicates. In the Iris dataset, although no missing values are present, it is still useful to check the dataset for any irregularities. Scaling or normalizing the features is also essential, especially if the data will be used in models sensitive to feature scales, such as k-nearest neighbors. Two popular scaling methods are normalization, which transforms features to a [0, 1] range, and standardization, which scales features to have a mean of 0 and a standard deviation of 1.

**Data Visualization** provides a visual interpretation of data and allows us to identify patterns, trends, and relationships within the dataset. Visualizations make it easier to compare features, observe correlations, and understand data distribution. Several plotting techniques can be applied to the Iris dataset:

1. **Scatter Plots**: By plotting feature pairs such as petal length vs. petal width and color-coding points by species, scatter plots reveal natural clusters. For example, petal length and petal width are particularly effective in distinguishing species clusters in the Iris dataset, as Iris-setosa tends to form a distinct group from Iris-versicolor and Iris-virginica.
2. **Box Plots**: Box plots show the distribution of each feature across the three iris species. They reveal the range, quartiles, and potential outliers within each feature, highlighting differences in feature distributions across species.
3. **Histograms**: Histograms help visualize the frequency distribution of each feature. For instance, a histogram of sepal length can show whether the values are normally distributed and if any values stand out as outliers.
4. **Pair Plots (or Scatterplot Matrix)**: A pair plot displays scatter plots for each pair of features, with color coding by species. This approach provides a comprehensive view of feature relationships and how they may relate to the species classification.
5. **Violin Plots**: These combine the features of box plots and histograms to show the distribution of each feature across species, providing insights into both the range and density of values within each species.

## Code & OUTPUT

In [1]:
```
print("Experiment No 03 : To perform data preprocessing and data visualization on iris dataset.")
```

```
Experiment No 03 : To perform data preprocessing and data visualization on iris dataset.
```

In [4]:
```
# Import necessary libraries
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
print("OUTPUT:\n\n")

# Load the iris dataset
iris = load_iris()
df = pd.DataFrame(iris.data, columns=iris.feature_names)
df['species'] = iris.target
df['species'] = df['species'].apply(lambda x: iris.target_names[x])

# Display basic information about the dataset
print("Dataset preview:")
print(df.head())
print("\nDataset summary:")
print(df.describe())
print("\nClass distribution:")
print(df['species'].value_counts())

# Data Preprocessing
# Separate features and target
X = df.iloc[:, :-1]  # Features (sepal and petal measurements)
y = df['species']    # Target (species)

# 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)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

print("\nData preprocessing complete.")

# Data Visualization

# Pair Plot
sns.pairplot(df, hue='species', markers=["o", "s", "D"])
plt.suptitle("Pairplot of Iris Dataset", y=1.02)
plt.show()

# Box Plot for Sepal Length and Petal Length by Species
plt.figure(figsize=(10, 6))
sns.boxplot(x='species', y='sepal length (cm)', data=df)
plt.title("Sepal Length by Species")
plt.show()

plt.figure(figsize=(10, 6))
sns.boxplot(x='species', y='petal length (cm)', data=df)
plt.title("Petal Length by Species")
plt.show()

# Violin Plot for Sepal Width and Petal Width by Species
plt.figure(figsize=(10, 6))
sns.violinplot(x='species', y='sepal width (cm)', data=df)
plt.title("Sepal Width by Species")
plt.show()

plt.figure(figsize=(10, 6))
sns.violinplot(x='species', y='petal width (cm)', data=df)
plt.title("Petal Width by Species")
plt.show()

# Heatmap of Correlation Matrix (excluding species column)
plt.figure(figsize=(8, 6))
sns.heatmap(df.iloc[:, :-1].corr(), annot=True, cmap='coolwarm', square=True)
plt.title("Correlation Heatmap of Iris Dataset")
plt.show()

# Histograms of Each Feature
df.iloc[:, :-1].hist(edgecolor='black', linewidth=1.2, figsize=(10, 8))
plt.suptitle("Feature Distributions")
plt.show()
```

OUTPUT:

```
Dataset preview:
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)  \
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

  species
0  setosa
1  setosa
2  setosa
3  setosa
4  setosa

Dataset summary:
       sepal length (cm)  sepal width (cm)  petal length (cm)  \
count         150.000000        150.000000         150.000000
mean            5.843333          3.057333           3.758000
std             0.828066          0.435866           1.765298
min             4.300000          2.000000           1.000000
25%             5.100000          2.800000           1.600000
50%             5.800000          3.000000           4.350000
75%             6.400000          3.300000           5.100000
max             7.900000          4.400000           6.900000

       petal width (cm)
count        150.000000
mean           1.199333
std            0.762238
min            0.100000
25%            0.300000
50%            1.300000
75%            1.800000
max            2.500000

Class distribution:
species
setosa        50
versicolor    50
virginica     50
Name: count, dtype: int64

Data preprocessing complete.
c:\Users\hp\anaconda3\Lib\site-packages\seaborn\axisgrid.py:118: UserWarning: The figure layout has changed to tight
  self._figure.tight_layout(*args, **kwargs)
```

## Result

As a result of this Experiment, we successfully wrote and executed the program to perform data preprocessing and data visualization on iris dataset.

## Learning Outcomes

Understand and apply data preprocessing techniques and various visualization methods to clean, explore, and interpret patterns in the Iris dataset effectively.

# Experiment 4

## Objective:

To implement k means clustering.

## Theory

K-means clustering is a popular unsupervised machine learning algorithm used to identify and group similar data points into clusters. The algorithm is commonly applied in exploratory data analysis, customer segmentation, pattern recognition, and image compression. Unlike supervised learning algorithms that rely on labeled data, K-means clustering operates without labels, identifying inherent patterns in the data based solely on the similarities among data points.

The K-means Algorithm aims to partition data points into k distinct clusters, where k is a user-defined parameter. Each cluster is represented by its centroid (the average of all points in the cluster). The algorithm works as follows:

Initialization: Choose k initial centroids, either by selecting random data points or by using methods like the K-means++ algorithm, which ensures initial centroids are spread out.
Assignment Step: Each data point is assigned to the nearest centroid based on a distance metric (typically Euclidean distance). This step forms k clusters, where each cluster contains the points closest to a specific centroid.

Update Step: After assigning data points to clusters, the centroids are recalculated by taking the mean of all points in each cluster. This updated centroid represents the new center of the cluster.
Iterate: The assignment and update steps are repeated until the centroids stabilize (i.e., they no longer change significantly) or until a maximum number of iterations is reached.
The objective of K-means is to minimize the within-cluster sum of squares (WCSS), which measures the variance within each cluster. By minimizing this, the algorithm ensures that data points within each cluster are as close as possible to their respective centroid, creating compact clusters.

Choosing the Number of Clusters (k) is a critical step in K-means. One commonly used technique is the elbow method, which involves running the algorithm for a range of k values and plotting the WCSS for each. The "elbow point," or the point where the WCSS reduction slows significantly, often suggests an optimal k. Another method is the silhouette score, which measures how similar points are within clusters compared to points in other clusters.

Advantages and Limitations: K-means is computationally efficient, especially for large datasets, and is relatively easy to implement. It performs well with spherical or well-separated clusters. However, K-means has limitations: it requires the number of clusters to be specified in advance, which is not always straightforward in unsupervised tasks. Additionally, it is sensitive to outliers and may perform poorly with non-spherical or overlapping clusters.
Image Compression: By clustering pixel colors, K-means can reduce the number of colors in an
In practical implementation, libraries like scikit-learn in Python provide an efficient K-means function that automates many aspects of the process, including initialization and convergence monitoring. By adjusting the parameter k and applying techniques like the elbow method, K-means can help reveal valuable insights from complex datasets.

## Code & OUTPUT

In [1]:
```
print("Experiment No 04 : To implement k means clustering.")
```

```
Experiment No 04 : To implement k means clustering.
```

In [5]:
```
# Import necessary libraries
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.decomposition import PCA

print("OUTPUT:\n\n")

# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features
y = iris.target  # True labels for comparison

# Initialize the KMeans model
# Assuming 3 clusters as there are 3 species in the Iris dataset
kmeans = KMeans(n_clusters=3, random_state=42)

# Fit the model and predict clusters
y_kmeans = kmeans.fit_predict(X)

# Show cluster centers
print("Cluster Centers:")
print(kmeans.cluster_centers_)

# Visualize the clusters in a 2D plot using PCA (Principal Component Analysis) for dimensionality reduction
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)

# Plot the results
plt.figure(figsize=(10, 6))
sns.scatterplot(x=X_pca[:, 0], y=X_pca[:, 1], hue=y_kmeans, palette="viridis", s=60)
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], c='red', marker='X', s=200, label='Centroid
plt.title("K-Means Clustering on Iris Dataset")
plt.xlabel("PCA Component 1")
plt.ylabel("PCA Component 2")
plt.legend(title="Cluster")
plt.show()

# Evaluate by comparing with true labels
# Note: K-Means does not use true labels, this is just for evaluation purposes.
from sklearn.metrics import accuracy_score, confusion_matrix

# Map clusters to actual labels (for evaluation purposes only)
# In this case, we need to manually assign the clusters to labels for evaluation
# This can vary each time you run it
label_map = {0: 'setosa', 1: 'versicolor', 2: 'virginica'}
predicted_labels = [label_map[label] for label in y_kmeans]

# Display confusion matrix
print("\nConfusion Matrix:")
print(confusion_matrix(y, y_kmeans))
```

OUTPUT:

```
c:\Users\hp\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1412: FutureWarning: The default value of `n_init
` will change from 10 to 'auto' in 1.4. Set the value of `n_init` explicitly to suppress the warning
  super()._check_params_vs_input(X, default_n_init=10)
c:\Users\hp\anaconda3\Lib\site-packages\sklearn\cluster\_kmeans.py:1436: UserWarning: KMeans is known to have a memo
ry leak on Windows with MKL, when there are less chunks than available threads. You can avoid it by setting the envi
ronment variable OMP_NUM_THREADS=1.
  warnings.warn(
Cluster Centers:
[[5.9016129  2.7483871  4.39354839 1.43387097]
 [5.006      3.428      1.462      0.246     ]
 [6.85       3.07368421 5.74210526 2.07105263]]
```

## Result

As a result of this Experiment, we successfully wrote and executed the program to implement k means clustering.

## Learning Outcomes

Understand and apply the K-means clustering algorithm to group data points into clusters, selecting optimal k and interpreting results effectively.

# Experiment 5

## Objective:

To implement data classification using KNN.

## Theory

The K-nearest neighbors (KNN) algorithm is a simple, yet powerful, supervised machine learning method used for classification and regression tasks. It is based on the principle of similarity between data points, where the classification of a given data point is determined by its closest neighbors in the feature space. KNN is non-parametric, meaning it does not assume a specific form for the underlying data distribution, making it versatile across various types of datasets.

The KNN Algorithm for classification operates as follows:

Choose the Value of K: K represents the number of nearest neighbors to consider when classifying a new data point. A smaller K can lead to a more sensitive model, but may also be more prone to noise, while a larger K may generalize better but could blur class boundaries. Calculate Distance: For each new data point, KNN calculates the distance between this point and all points in the training data. Common distance metrics include Euclidean distance, Manhattan distance, and Minkowski distance. Euclidean distance is often used as it provides a straightforward measure of similarity.

Identify Nearest Neighbors: The algorithm then selects the K data points closest to the new point.Determine the Class: The new data point is assigned the class label that is most frequent among its K nearest neighbors. For instance, if K=5 and three of the neighbors belong to Class A while two belong to Class B, the new point will be classified as Class A.

Choosing the Value of K is crucial to KNN's performance. Common methods include cross-validation, where various K values are tested to determine which yields the best model accuracy. Low values of K may lead to overfitting, where the model is too sensitive to noise. Higher values of K provide smoother decision boundaries but may miss finer patterns.

Advantages and Disadvantages: KNN is easy to understand and implement, making it popular for beginners in machine learning. It performs well on smaller datasets where the data points are easily separable in the feature space. However, it has limitations: it is computationally intensive for large datasets, as it requires calculating distances for all training points for each new prediction. KNN is also sensitive to feature scaling; therefore, normalization or standardization of features is essential to prevent features with large ranges from dominating the distance calculations.

Implementing KNN: In Python, the scikit-learn library provides a simple implementation of KNN through KNeighborsClassifier. Users can set the K value, distance metric, and weighting options. We can evaluate KNN models using performance
Overall, KNN is an effective algorithm for classification tasks where data points exhibit clear clusters or similarities. It is widely used for its simplicity and intuitive approach, providing meaningful insights, especially in tasks where interpretability is key.

## Code & OUTPUT

In [2]:
```
print("Experiment No 05 : To implement data classification using KNN.")
```

```
Experiment No 05 : To implement data classification using KNN.
```

In [4]:
```
# Import necessary libraries
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

print("OUTPUT:\n\n")

# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features (sepal length, sepal width, petal length, petal width)
y = iris.target  # Target (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)

# Initialize the K-Nearest Neighbors classifier with k=3
knn = KNeighborsClassifier(n_neighbors=3)

# Train the model
knn.fit(X_train, y_train)

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

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")

# Display a detailed classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred, target_names=iris.target_names))

# Plotting the Confusion Matrix
ConfusionMatrixDisplay.from_predictions(y_test, y_pred, display_labels=iris.target_names)
plt.title("Confusion Matrix of KNN Classifier")
plt.show()
```

```
OUTPUT:


Accuracy: 1.00

Classification Report:
              precision    recall  f1-score   support

      setosa       1.00      1.00      1.00        10
  versicolor       1.00      1.00      1.00         9
   virginica       1.00      1.00      1.00        11

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

## Result

As a result of this Experiment, we successfully wrote and executed the program to implement data classification using KNN.

## Learning Outcomes

Understand and apply the K-means clustering algorithm to group data points into clusters, selecting optimal k and interpreting results effectively.

# Experiment 6

## Objective:

To implement decision tree using ID3 algorithm.

## Theory

The ID3 (Iterative Dichotomiser 3) algorithm is a popular algorithm for building decision trees, particularly for classification tasks. It is a supervised learning algorithm developed by Ross Quinlan and is widely used for its intuitive, rule-based approach to classification. Decision trees classify data by making a sequence of decisions based on feature values, dividing data into subsets to arrive at class predictions. The ID3 algorithm uses entropy and information gain to determine the best features for splitting data at each node of the tree.

The Decision Tree Structure: A decision tree consists of nodes and branches. Each internal node represents a decision based on a feature, branches represent the outcomes of that decision, and leaf nodes represent the final class labels. By traversing the tree from the root to a leaf node, we can make predictions for any data point based on its feature values.

How the ID3 Algorithm Works:

- Entropy Calculation: Entropy is a measure of uncertainty or impurity within a set of data. For a binary classification problem, entropy

$$E(S) = -p_+ \log_2(p_+) - p_- \log_2(p_-)$$

- Information Gain**:** The core of the ID3 algorithm is the selection of the feature that maximizes information gain. Information gain is the reduction in entropy achieved by partitioning data based on a feature. For a feature AA, information gain IG(S,A)IG(S,A) is calculated as:

$$IG(S, A) = E(S) - \sum_{v \in \text{values}(A)} \frac{|S_v|}{|S|} E(S_v)$$

The feature with the highest information gain is selected to split the data at each node, as it results in the most significant reduction in impurity.

- Recursive Partitioning: Once a feature is selected for a node, the dataset is split into subsets based on this feature's values. The ID3 algorithm is then applied recursively to each subset to build child nodes. This recursive process continues until all nodes are pure (contain only one class) or no further information gain is achievable.
- Stopping Criteria: ID3 stops growing the tree if all instances belong to the same class, if there are no more features to split on, or if the information gain for all features is zero. Additionally, a maximum depth can be set to prevent overfitting.

Advantages and Disadvantages of ID3:

ID3 is easy to interpret, as the resulting decision tree provides a clear path from features to class labels. It is particularly useful when understanding decision logic is crucial. However, ID3 has limitations: it can produce overly complex trees (overfitting) if not pruned, and it does not handle numeric or continuous features well without discretization. Furthermore, it is biased toward features with more values, as they may artificially inflate information gain. This issue can be addressed by using modified versions like C4.5 or CART.

## Code & OUTPUT

In [1]:
```
print("Experiment No 06 : To implement decision tree using ID3 algorithm.")
```

```
Experiment No 06 : To implement decision tree using ID3 algorithm.
```

In [4]:
```
import pandas as pd
import numpy as np
from collections import Counter

print("\tOUTPUT:\n\n")

# Function to calculate entropy of a dataset
def entropy(data):
    total_count = len(data)
    counts = Counter(data)
    ent = 0
    for count in counts.values():
        prob = count / total_count
        ent -= prob * np.log2(prob) if prob > 0 else 0
    return ent

# Function to calculate information gain for a feature
def information_gain(data, feature):
    total_entropy = entropy(data.iloc[:, -1])  # The target column is the last column
    feature_values = data[feature].unique()

    weighted_entropy = 0
    for value in feature_values:
        subset = data[data[feature] == value]
        weighted_entropy += (len(subset) / len(data)) * entropy(subset.iloc[:, -1])

    return total_entropy - weighted_entropy

# Function to build the decision tree using ID3
def id3(data, features):
    # Base case: If all rows have the same class, return the class label
    if len(set(data.iloc[:, -1])) == 1:
        return data.iloc[0, -1]

    # Base case: If no features left to split, return the majority class label
    if len(features) == 0:
        return Counter(data.iloc[:, -1]).most_common(1)[0][0]

    # Select the feature with the highest information gain
    gains = [information_gain(data, feature) for feature in features]
    best_feature = features[np.argmax(gains)]

    # Create a decision tree node
    tree = {best_feature: {}}
    remaining_features = [feature for feature in features if feature != best_feature]

    # Split the data based on the selected feature and recursively build subtrees
    for value in data[best_feature].unique():
        subset = data[data[best_feature] == value]
        subtree = id3(subset, remaining_features)
        tree[best_feature][value] = subtree

    return tree

# Function to make predictions using the decision tree
def predict(tree, instance):
    if isinstance(tree, dict):
        feature = list(tree.keys())[0]
        value = instance[feature]
        subtree = tree[feature].get(value)
        return predict(subtree, instance)
    else:
        return tree

# Example of how to use ID3 to classify the Iris dataset

from sklearn.datasets import load_iris
import pandas as pd

# Load Iris dataset and create DataFrame
iris = load_iris()
data = pd.DataFrame(iris.data, columns=iris.feature_names)
data['species'] = iris.target

# Convert the numeric target values to actual species names for better readability
data['species'] = data['species'].map({0: 'setosa', 1: 'versicolor', 2: 'virginica'})

# Features to consider for splitting (excluding the target variable)
features = data.columns[:-1]  # Extract feature column names

# Build the decision tree using ID3
tree = id3(data, features)

# Print the decision tree
import pprint
pprint.pprint(tree)

# Making predictions on a new instance (just an example)
new_instance = data.iloc[0]  # You can choose any instance to test
prediction = predict(tree, new_instance)
print(f"\n\t Prediction for the first instance: {prediction}")
```

OUTPUT:

```
{'petal length (cm)': {1.0: 'setosa',
                       1.1: 'setosa',
                       1.2: 'setosa',
                       1.3: 'setosa',
                       1.4: 'setosa',
                       1.5: 'setosa',
                       1.6: 'setosa',
                       1.7: 'setosa',
                       1.9: 'setosa',
                       3.0: 'versicolor',
                       3.3: 'versicolor',
                       3.5: 'versicolor',
                       3.6: 'versicolor',
                       3.7: 'versicolor',
                       3.8: 'versicolor',
                       3.9: 'versicolor',
                       4.0: 'versicolor',
                       4.1: 'versicolor',
                       4.2: 'versicolor',
                       4.3: 'versicolor',
                       4.4: 'versicolor',
                       4.5: {'sepal length (cm)': {4.9: 'virginica',
                                                   5.4: 'versicolor',
                                                   5.6: 'versicolor',
                                                   5.7: 'versicolor',
                                                   6.0: 'versicolor',
                                                   6.2: 'versicolor',
                                                   6.4: 'versicolor'}},
                       4.6: 'versicolor',
                       4.7: 'versicolor',
                       4.8: {'sepal length (cm)': {5.9: 'versicolor',
                                                   6.0: 'virginica',
                                                   6.2: 'virginica',
                                                   6.8: 'versicolor'}},
                       4.9: {'sepal width (cm)': {2.5: 'versicolor',
                                                  2.7: 'virginica',
                                                  2.8: 'virginica',
                                                  3.0: 'virginica',
                                                  3.1: 'versicolor'}},
                       5.0: {'sepal length (cm)': {5.7: 'virginica',
                                                   6.0: 'virginica',
                                                   6.3: 'virginica',
                                                   6.7: 'versicolor'}},
                       5.1: {'sepal length (cm)': {5.8: 'virginica',
                                                   5.9: 'virginica',
                                                   6.0: 'versicolor',
                                                   6.3: 'virginica',
                                                   6.5: 'virginica',
                                                   6.9: 'virginica'}},
                       5.2: 'virginica',
                       5.3: 'virginica',
                       5.4: 'virginica',
                       5.5: 'virginica',
                       5.6: 'virginica',
                       5.7: 'virginica',
                       5.8: 'virginica',
                       5.9: 'virginica',
                       6.0: 'virginica',
                       6.1: 'virginica',
                       6.3: 'virginica',
                       6.4: 'virginica',
                       6.6: 'virginica',
                       6.7: 'virginica',
                       6.9: 'virginica'}}
```

Prediction for the first instance: setosa

In [ ]:

## Result

As a result of this Experiment, we successfully wrote and executed the to implement k means clustering.

## Learning Outcomes

Understand and implement the ID3 algorithm for decision trees, utilizing entropy and information gain to build an interpretable classification model.

# Experiment 7

## Objective:

To implement decision tree using CART algorithm.

## Theory

The Classification and Regression Trees (CART) algorithm is a popular approach to building decision trees for both classification and regression tasks. Developed by Breiman et al., CART is highly versatile and forms the basis for more advanced tree-based algorithms like random forests and boosting methods. Unlike the ID3 algorithm, which is primarily used for classification, CART supports both classification and regression, making it widely applicable across a range of machine learning tasks.

Decision Tree Structure in CART:
A decision tree consists of internal nodes representing decisions based on feature values, branches representing the outcomes of those decisions, and leaf nodes where final predictions (either classes for classification or values for regression) are made. The CART algorithm builds binary trees, meaning each internal node splits the data into two subsets based on a chosen feature and threshold value.

The CART Algorithm Process:

Gini Impurity (Classification): For classification, CART uses the Gini impurity criterion to evaluate splits. Gini impurity measures the likelihood of incorrectly classifying a randomly chosen element if it were randomly labeled according to the distribution of labels in a subset. For a node m with classes c, the Gini impurity G is calculated as:

$$G(m) = 1 - \sum_{c=1}^{C} p_c^2$$

where p is the probability of selecting a data point from class c in node m. The lower the Gini impurity, the purer the node. CART chooses the feature and threshold that minimize the Gini impurity for each split.
Mean Squared Error (Regression): For regression tasks, CART uses mean squared error (MSE) to measure the quality of splits. It aims to minimize the variance of target values within each leaf node, ensuring that each region represents a homogeneous output. MSE at node
m is calculated as:

$$\text{MSE} = \frac{1}{N_m} \sum_{i=1}^{N_m} (y_i - \bar{y}_m)^2$$

Recursive Partitioning: The algorithm applies recursive binary splits to partition the dataset. At each step, CART evaluates all possible splits and selects the feature and threshold that minimize impurity (Gini for classification, MSE for regression).
Tree Pruning: Unlike ID3, CART includes a pruning step to prevent overfitting. Pruning removes branches that provide little predictive power, improving the model's generalization on new data.
Advantages and Limitations of CART: CART is highly interpretable, offering an intuitive way to model complex decision boundaries through simple rules. It performs well with both numeric and categorical data and handles multi-class classification naturally.

## Code & OUTPUT

In [1]:
```
print("Experiment No 07 : To implement decision tree using CART algorithm.")
```

```
Experiment No 07 : To implement decision tree using CART algorithm.
```

In [4]:
```
# Import necessary libraries
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report

# Load a sample dataset (Iris dataset for this example)
data = load_iris()
X = data.data
y = data.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_sta

# Initialize the Decision Tree Classifier with CART algorithm
# criterion="gini" for CART (classification)
model = DecisionTreeClassifier(criterion="gini", random_state=42)

# Fit the model
model.fit(X_train, y_train)

# Predict on the test set
y_pred = model.predict(X_test)

print("OUTPUT:\n\n")

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred)

print("Accuracy:", accuracy)
print("Classification Report:\n", report)
```

```
OUTPUT:


Accuracy: 1.0
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```

In [ ]:
```

```

## Result

As a result of this Experiment, we successfully wrote and executed the program to implement decision tree using CART algorithm.

## Learning Outcomes

Understand and implement the CART algorithm for decision trees, using Gini impurity or MSE for splits and applying pruning to improve model interpretability and prevent overfitting.

# Experiment 8

## Objective:

To implement decision tree using C4.5 algorithm.

## Theory

The C4.5 algorithm, developed by Ross Quinlan, is an extension of the ID3 algorithm and is widely used for building decision trees, especially for classification tasks. C4.5 improves on ID3 by addressing several of its limitations, making it a more powerful and flexible algorithm. C4.5 can handle both categorical and continuous data, perform automatic pruning to reduce overfitting, and manage missing values effectively.

**Decision Tree Structure in C4.5**: A decision tree consists of nodes (where decisions are made based on features), branches (representing the outcomes of these decisions), and leaf nodes (which provide the final class label). Like ID3, C4.5 uses entropy and information gain to decide on splits but introduces gain ratio to avoid biases toward features with many distinct values.

**The C4.5 Algorithm Process**:

1. **Entropy and Information Gain**: Like ID3, C4.5 uses **entropy** to measure the impurity of a node and **information gain** to evaluate the effectiveness of each feature for splitting the data. However, C4.5 improves upon ID3 by implementing a new metric, called the **gain ratio**, to prevent the algorithm from favoring features with numerous unique values.

   Information gain is calculated as:

$$IG(S, A) = E(S) - \sum_{v \in \text{values}(A)} \frac{|S_v|}{|S|} E(S_v)$$

   where E(S)E(S) is the entropy of the dataset SS, and SvSv represents subsets of SS based on the values of feature AA.

2. **Gain Ratio**: While ID3 selects the feature with the highest information gain for splitting, C4.5 uses **gain ratio** to mitigate the bias toward features with multiple distinct values. Gain ratio is calculated as:

$$\text{Gain Ratio} = \frac{\text{Information Gain}}{\text{Split Information}}$$

   where **split information** measures the potential of each feature to split data into subsets. C4.5 selects the feature with the highest gain ratio, ensuring more balanced splits.

3. **Handling Continuous Data**: C4.5 can handle continuous or numeric data by determining a threshold value for splitting. For each feature, it identifies possible

thresholds and chooses the one that yields the highest gain ratio. This is a significant improvement over ID3, which only supports categorical features.
4. **Handling Missing Values**: C4.5 accommodates missing values by considering the probability of each possible outcome. When splitting on a feature with missing values, C4.5 assigns weights to branches based on the distribution of known values, allowing the algorithm to include data points with missing values rather than discarding them.
5. **Tree Pruning**: C4.5 includes a post-pruning step to reduce overfitting. **Subtree replacement** and **subtree raising** are used to prune branches that do not improve classification accuracy on test data. This step simplifies the model, enhancing its generalizability and interpretability.

**Advantages and Limitations of C4.5**: C4.5 produces interpretable decision trees, handling continuous and categorical data, missing values, and avoiding biases toward features with numerous values. Its automatic pruning reduces the risk of overfitting, making it suitable for a wide range of classification tasks. However, C4.5 is computationally more intensive than ID3, as calculating gain ratio and handling continuous features require additional processing. Additionally, it may not perform optimally on large datasets without optimization.

**Applications of C4.5**:

1. **Medical Diagnosis**: C4.5 helps build decision trees for diagnostic systems, classifying diseases based on symptoms and test results with rules that are easily interpretable.
2. **Customer Segmentation**: In marketing, C4.5 is used to classify customers based on purchasing behavior and demographics, allowing companies to tailor their strategies.
3. **Credit Scoring**: Financial institutions apply C4.5 to assess applicants' creditworthiness based on various financial and demographic features, generating interpretable decision rules.

**Implementing C4.5**: While C4.5 is not directly available in common libraries like `scikit-learn`, similar functionality can be achieved using `DecisionTreeClassifier` with advanced settings or by using the Quinlan-developed C4.5 in older software. Alternatively, tools like WEKA provide a direct implementation of C4.5.

In summary, C4.5 is a powerful, flexible algorithm that addresses several limitations of ID3, producing interpretable trees with improved handling of continuous data and missing values, as well as automatic pruning.

## Code & OUTPUT

In [1]:
```
print("Experiment No 08 : To implement decision tree using C4.5 algorithm.")
```

```
Experiment No 08 : To implement decision tree using C4.5 algorithm.
```

In [3]:
```
# Import necessary libraries
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report

# Load a sample dataset (Iris dataset for this example)
data = load_iris()
X = data.data
y = data.target

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, rando

# Initialize the Decision Tree Classifier with C4.5-like settings
# criterion="entropy" for information gain (similar to C4.5)
model = DecisionTreeClassifier(criterion="entropy", random_state=42)

# Fit the model
model.fit(X_train, y_train)

# Predict on the test set
y_pred = model.predict(X_test)

print("OUTPUT:\n\n")

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred)

print("Accuracy:", accuracy)
print("Classification Report:\n", report)
```

```
OUTPUT:


Accuracy: 0.9777777777777777
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       0.93      1.00      0.96        13
           2       1.00      0.92      0.96        13

    accuracy                           0.98        45
   macro avg       0.98      0.97      0.97        45
weighted avg       0.98      0.98      0.98        45
```

## Result

As a result of this Experiment, we successfully wrote and executed the program to implement decision tree using C4.5 algorithm.

## Learning Outcomes

Understand and implement the C4.5 algorithm to build robust decision trees for classification tasks, with an emphasis on gain ratio and pruning to improve accuracy and interpretability.

# Experiment 9

## Objective:

To implement multi layer neural network.

## Theory

A **Multi-Layer Neural Network (MLNN)**, often referred to as a **feedforward neural network**, is a type of artificial neural network composed of multiple layers of nodes (neurons). It is one of the fundamental models in deep learning and is used for various tasks such as classification, regression, and pattern recognition. The architecture of a multi-layer neural network typically consists of an input layer, one or more hidden layers, and an output layer.

**1. Structure of a Multi-Layer Neural Network**:

- **Input Layer**: The input layer receives the raw data. Each neuron in this layer represents one feature of the input data.
- **Hidden Layers**: These layers exist between the input and output layers and allow the network to learn complex patterns. Each neuron in a hidden layer performs a weighted sum of inputs and passes the result through an activation function.
- **Output Layer**: The output layer produces the final prediction or classification. The number of neurons in the output layer depends on the type of task—one neuron for binary classification, multiple neurons for multi-class classification, or a single neuron for regression tasks.

**2. The Neuron**: Each neuron in the network performs a simple mathematical operation. Given an input vector x=[x1,x2,...,xn], the output of a neuron is calculated as:

$$y = f(w_1x_1 + w_2x_2 + ... + w_nx_n + b)$$

where:

- w1,w2,..., wn are the weights,
- bb is the bias term,
- ff is the activation function, which introduces non-linearity into the model.

**3. Activation Functions**: Activation functions are crucial in neural networks as they allow the network to learn complex, non-linear relationships between the inputs and outputs. Some commonly used activation functions include:

- **Sigmoid**: Maps outputs to a range between 0 and 1. Often used in binary classification problems. $\sigma(x) = \frac{1}{1 + e^{-x}}$
- **ReLU (Rectified Linear Unit)**: The most widely used activation function in deep learning, which outputs the input directly if it is positive; otherwise, it outputs zero. $\text{ReLU}(x) = \max(0, x)$
- **Tanh**: Similar to sigmoid but outputs values between -1 and 1, often used when the data is centered around zero. tanh$f_0$(x)=ex−e−xex+e−xtanh(x)=ex+e−xex−e−x

**4. Training a Multi-Layer Neural Network**: Training a neural network involves finding the optimal weights and biases that minimize the error between the predicted output and the actual target. This is achieved through the following steps:

- **Forward Propagation**: During forward propagation, the input data passes through the network, and an output is generated at the output layer.
- **Loss Function**: The loss function calculates the difference between the predicted output and the true output. Common loss functions include:
    - **Mean Squared Error (MSE)** for regression tasks.
    - **Cross-Entropy** for classification tasks.
- **Backpropagation**: The backpropagation algorithm is used to update the weights and biases by calculating the gradient of the loss function with respect to each weight in the network. This is achieved through the **chain rule** of differentiation. The gradients are then used to update the weights using an optimization algorithm such as **Stochastic Gradient Descent (SGD)** or **Adam**.

**5. Optimization**: Optimization algorithms are used to adjust the weights to minimize the loss function. Some common optimization techniques are:

- **Gradient Descent**: Gradually adjusts the weights in the opposite direction of the gradient to minimize the loss function.
- **Adam**: A more advanced optimization method that adapts the learning rate based on estimates of first and second moments of the gradients, providing faster convergence.

**6. Overfitting and Regularization**: Overfitting occurs when the model learns the training data too well, including noise and outliers, which hurts its generalization to new data. To combat overfitting:

- **Dropout**: Randomly drops neurons during training to prevent over-reliance on specific neurons.
- **L2 Regularization (Ridge)**: Adds a penalty term to the loss function based on the magnitude of the weights, encouraging smaller weight values and reducing overfitting.

**7. Multi-Layer Neural Networks in Practice**: Multi-layer neural networks are powerful models and are the building blocks of more complex architectures like convolutional neural networks (CNNs) and recurrent neural networks (RNNs). They are widely used in image classification, speech recognition, and natural language processing. Popular deep learning libraries such as **TensorFlow**, **Keras**, and **PyTorch** provide tools to build and train multi-layer neural networks.

**Applications**:

1. **Image Classification**: MLNNs are used in image classification tasks, such as identifying objects in images or recognizing handwritten digits.
2. **Natural Language Processing**: In NLP, multi-layer neural networks are used for tasks like sentiment analysis, machine translation, and named entity recognition.
3. **Speech Recognition**: Neural networks are utilized in speech-to-text systems to convert spoken language into written text.

## Code & OUTPUT

In [1]:
```
print("Experiment No 09 : To implement multi layer neural network.")
```

```
Experiment No 09 : To implement multi layer neural network.
```

In [7]:
```
!pip install torchvision --user
```

```
Collecting torchvision
  Obtaining dependency information for torchvision from https://files.pythonhosted.org/packages/69/55/ce836703ff77bb
21582c3098d5311f8ddde7eadc7eab04be9561961f4725/torchvision-0.20.1-cp311-cp311-win_amd64.whl.metadata
  Using cached torchvision-0.20.1-cp311-cp311-win_amd64.whl.metadata (6.2 kB)
Requirement already satisfied: numpy in c:\users\hp\anaconda3\lib\site-packages (from torchvision) (1.24.3)
Requirement already satisfied: torch==2.5.1 in c:\users\hp\anaconda3\lib\site-packages (from torchvision) (2.5.1)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in c:\users\hp\anaconda3\lib\site-packages (from torchvision)
(9.4.0)
Requirement already satisfied: filelock in c:\users\hp\anaconda3\lib\site-packages (from torch==2.5.1->torchvision)
(3.9.0)
Requirement already satisfied: typing-extensions>=4.8.0 in c:\users\hp\anaconda3\lib\site-packages (from torch==
2.5.1->torchvision) (4.12.2)
Requirement already satisfied: networkx in c:\users\hp\anaconda3\lib\site-packages (from torch==2.5.1->torchvision)
(3.1)
Requirement already satisfied: jinja2 in c:\users\hp\anaconda3\lib\site-packages (from torch==2.5.1->torchvision)
(3.1.2)
Requirement already satisfied: fsspec in c:\users\hp\anaconda3\lib\site-packages (from torch==2.5.1->torchvision) (2
023.4.0)
Requirement already satisfied: sympy==1.13.1 in c:\users\hp\anaconda3\lib\site-packages (from torch==2.5.1->torchvis
ion) (1.13.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in c:\users\hp\anaconda3\lib\site-packages (from sympy==1.13.1->to
rch==2.5.1->torchvision) (1.3.0)
Requirement already satisfied: MarkupSafe>=2.0 in c:\users\hp\anaconda3\lib\site-packages (from jinja2->torch==
2.5.1->torchvision) (2.1.1)
Using cached torchvision-0.20.1-cp311-cp311-win_amd64.whl (1.6 MB)
Installing collected packages: torchvision
Successfully installed torchvision-0.20.1
```

In [8]:
```python
# Import necessary libraries
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

# Define transformations for the dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))  # Normalize to mean 0.5 and std 0.5 for simplicity
])

# Load the MNIST dataset
train_dataset = datasets.MNIST(root='./data', train=True, transform=transform, download=True)
test_dataset = datasets.MNIST(root='./data', train=False, transform=transform, download=True)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

# Define the neural network model
class NeuralNet(nn.Module):
    def __init__(self):
        super(NeuralNet, self).__init__()
        self.fc1 = nn.Linear(28*28, 128)  # First hidden layer with 128 neurons
        self.fc2 = nn.Linear(128, 64)     # Second hidden layer with 64 neurons
        self.fc3 = nn.Linear(64, 10)      # Output layer with 10 classes

    def forward(self, x):
        x = x.view(-1, 28*28)            # Flatten the input
        x = torch.relu(self.fc1(x))       # Apply ReLU activation
        x = torch.relu(self.fc2(x))       # Apply ReLU activation
        x = self.fc3(x)                   # Output layer without activation (for logits)
        return x

# Instantiate the model, define loss function and optimizer
model = NeuralNet()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training the model
epochs = 5
for epoch in range(epochs):
    for images, labels in train_loader:
        optimizer.zero_grad()               # Zero the gradients
        outputs = model(images)             # Forward pass
        loss = criterion(outputs, labels)   # Compute loss
        loss.backward()                     # Backward pass
        optimizer.step()                    # Update weights
    print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")

# Evaluating the model
correct = 0
total = 0
with torch.no_grad():  # No need to calculate gradients for evaluation
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print("OUTPUT:\n\n")
print(f"Accuracy of the model on the 10,000 test images: {100 * correct / total:.2f}%")
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Failed to download (trying next):
<urlopen error [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed: certificate has expired (_ssl.c:1006)>

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-images-idx3-ubyte.gz to ./data\MNIST\raw\train-image
s-idx3-ubyte.gz
100%|████████| 9.91M/9.91M [04:45<00:00, 34.7kB/s]
```

```
Extracting ./data\MNIST\raw\train-images-idx3-ubyte.gz to ./data\MNIST\raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Failed to download (trying next):
<urlopen error [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed: certificate has expired (_ssl.c:1006)>

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/train-labels-idx1-ubyte.gz to ./data\MNIST\raw\train-label
s-idx1-ubyte.gz
100%|██████████| 28.9k/28.9k [00:00<00:00, 110kB/s]
Extracting ./data\MNIST\raw\train-labels-idx1-ubyte.gz to ./data\MNIST\raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Failed to download (trying next):
<urlopen error [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed: certificate has expired (_ssl.c:1006)>

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-images-idx3-ubyte.gz to ./data\MNIST\raw\t10k-images-
idx3-ubyte.gz
100%|██████████| 1.65M/1.65M [01:13<00:00, 22.5kB/s]
Extracting ./data\MNIST\raw\t10k-images-idx3-ubyte.gz to ./data\MNIST\raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Failed to download (trying next):
<urlopen error [SSL: CERTIFICATE_VERIFY_FAILED] certificate verify failed: certificate has expired (_ssl.c:1006)>

Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz
Downloading https://ossci-datasets.s3.amazonaws.com/mnist/t10k-labels-idx1-ubyte.gz to ./data\MNIST\raw\t10k-labels-
idx1-ubyte.gz
100%|██████████| 4.54k/4.54k [00:00<00:00, 911kB/s]
Extracting ./data\MNIST\raw\t10k-labels-idx1-ubyte.gz to ./data\MNIST\raw

Epoch [1/5], Loss: 0.2970
Epoch [2/5], Loss: 0.1825
Epoch [3/5], Loss: 0.0771
Epoch [4/5], Loss: 0.0327
Epoch [5/5], Loss: 0.0104
OUTPUT:


Accuracy of the model on the 10,000 test images: 96.79%
```

In [ ]:

## Result

As a result of this Experiment, we successfully wrote and executed the to implement multi layer neural network in python.

## Learning Outcomes

Understand and implement a multi-layer neural network, including forward propagation, backpropagation, and optimization, for classification and regression tasks.