**DELHI TECHNOLOGICAL UNIVERSITY**

**DEPARTMENT OF**

**COMPUTER SCIENCE ENGINEERING**

****

**CO327   
Machine Learning**

**SUBMITTED TO: SUBMITTED BY:**

Dr. Kavinder Singh Ayush Tandon

2K22/CO/133

**INDEX**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.No.** | **Experiment** | **Date** | **Signature** |
| 1. | Python Basics, Numpy, Pandas, Matplotlib. |  |
| 2. | Data Visualization, Preprocessing and cleaning. |  |
| 3. | Linear Regression using gradient descent (advertising dataset) |  |
| 4. | Linear Regression using Mean Square Error (advertising dataset) |  |
| 5 | Logistic Regression on Iris Dataset. |  |
| 6. | Decision Tree using Cart Algorithm (breast cancer dataset {binary classification} |  |
| 7. | Decision Tree using ID3 Algorithm (breast cancer dataset {binary classification} |  |
| 8. | Decision Tree using C4.5 Algorithm (breast cancer dataset {binary classification} |  |
| 9. | Implementation of KNN. |  |
| 10. | Implementation of Support Vector Machine. |  |
| 11. | Implementation of Naive Bayes Classifier. |  |
| 12. | Implementation of Random Forest. |  |
| 13. | Implementation of Data classification using  K-Means |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 14. | Implementation of Basic Neural Network. |  |  |
| 15. | Evaluation Metrics in Machine Learning. |  |
| 16. | Implementation of a Convolutional Neural Network (CNN) using the MNIST dataset. |  |
| 17. | Implementing Dimensionality Reduction using Principal Component Analysis (PCA). |  |
| 18. | Build an Artificial Neural Network (ANN) with backpropagation. |  |
| 19. | Image Classification on Fashion MNIST Dataset using Artificial Neural Network (ANN). |  |
| 20. | Build an Artificial Neural Network (ANN) on MNIST Dataset. |  |

**EXPERIMENT - 1**

**Aim:** Python Basics (NumPy, Pandas, Matplotlib)

**Theory:**  
In the fields of data science and machine learning, Python libraries such as NumPy, Pandas, and Matplotlib are indispensable for data manipulation, processing, and visualization.

* **NumPy** is a core library for scientific computing in Python. It supports large, multi-dimensional arrays and matrices, and offers a comprehensive collection of high-level mathematical functions to operate on these arrays efficiently.
* **Pandas** is a powerful library designed for data manipulation and analysis. It provides versatile data structures like DataFrames, which allow for organizing data into rows and columns, similar to a spreadsheet. This makes it easy to filter, sort, and perform complex data operations.
* **Matplotlib** is a comprehensive plotting library used for creating static, animated, and interactive visualizations in Python. It enables the visualization of data distributions, trends, and patterns, which are essential for analyzing data and deriving insights.

**CODE:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

# Numpy Basics

# Creating an array and performing basic operations

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

print("Original Array:", array)

print("Array Squared:", array \*\* 2)

# Pandas Basics

# Creating a DataFrame and performing basic data manipulation

data = {

    'Name': ['Jasprit', 'Rahul', 'Sachin', 'Dravid', 'Ashwin'],

    'Age': [24, 27, 22, 32, 29],

    'City': ['Delhi', 'Mumbai', 'Hyderabad', 'Lucknow', 'Bangalore']

}

df = pd.DataFrame(data)

print("\nDataFrame:")

print(df)

# Filtering data

print("\nPeople aged over 25:")

print(df[df['Age'] > 25])

a = np.array([[1, 2],

[3, 4]])

b = np.array([[4, 3],

[2, 1]])

print ("Adding 1 to every element:", a + 1)

print ("**\n**Subtracting 2 from each element:", b - 2)

# Matplotlib Basics

# Creating a simple line plot

x = np.linspace(0, 10, 100)

y = np.sin(x)

plt.plot(x, y, label="sin(x)")

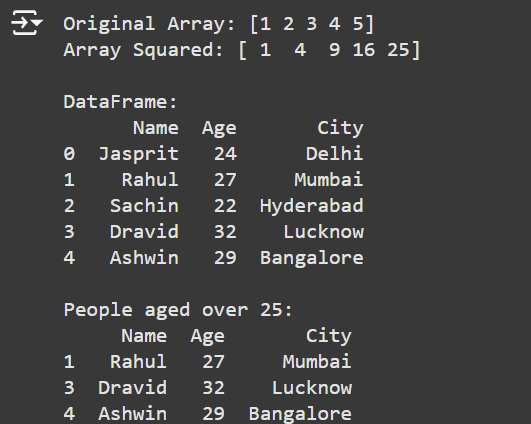
plt.xlabel("X-axis")

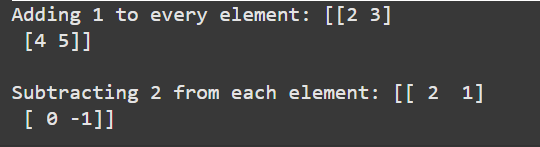
plt.ylabel("Y-axis")

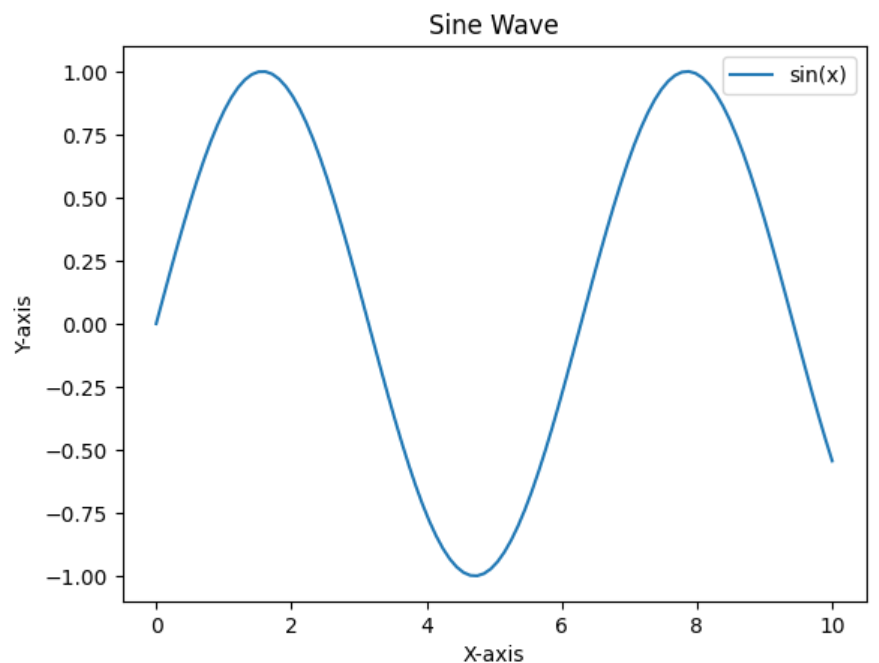
plt.title("Sine Wave")

plt.legend()

plt.show()  
  
**Output:**







**Learning Outcomes**

1. Gained familiarity with NumPy for array creation and manipulation.

2. Understood data handling and filtering with Pandas.

3. Learned how to plot data and visualize patterns using Matplotlib.

**EXPERIMENT – 2**

**AIM:** Data visualization, preprocessing and cleaning.

**THEORY:**

Data preprocessing is a critical step in preparing data for machine learning. It includes data cleaning, handling missing values, encoding categorical variables, and scaling features. Proper data preprocessing improves model performance and helps eliminate noise and inconsistencies.

* Data visualization helps understand data distribution and relationships between features, allowing for informed decisions on preprocessing steps.
* Data cleaning includes handling null values and removing outliers to ensure data consistency.
* Preprocessing often involves encoding categorical variables to numerical form, scaling features to standardize their range, and splitting data into training and testing sets.

**CODE:**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

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

from sklearn.preprocessing import StandardScaler, LabelEncoder

# Loading a sample dataset

data = pd.DataFrame({

'Feature1': [5.1, 4.9, np.nan, 5.0, 5.5, 6.1, 5.8],

'Feature2': [3.5, 3.0, 3.2, np.nan, 3.8, 3.3, 3.0],

'Category': ['A', 'B', 'A', 'A', 'B', 'B', 'A']

})

# Displaying the first few rows

print("Original Data:")

print(data)

# Handling missing values by filling with the mean

data['Feature1'].fillna(data['Feature1'].mean(), inplace=True)

data['Feature2'].fillna(data['Feature2'].mean(), inplace=True)

# Encoding categorical variables

le = LabelEncoder()

data['Category'] = le.fit\_transform(data['Category'])

# Scaling the features

scaler = StandardScaler()

data[['Feature1', 'Feature2']] = scaler.fit\_transform(data[['Feature1', 'Feature2']])

# Visualizing the data distribution

plt.figure(figsize=(10, 5))

sns.histplot(data['Feature1'], kde=True, label='Feature1')

sns.histplot(data['Feature2'], kde=True, label='Feature2')

plt.legend()

plt.title("Feature Distributions")

plt.show()

# Splitting data into training and testing sets

X = data[['Feature1', 'Feature2']]

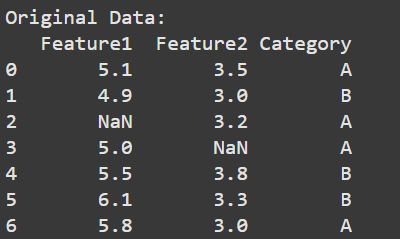
y = data['Category']

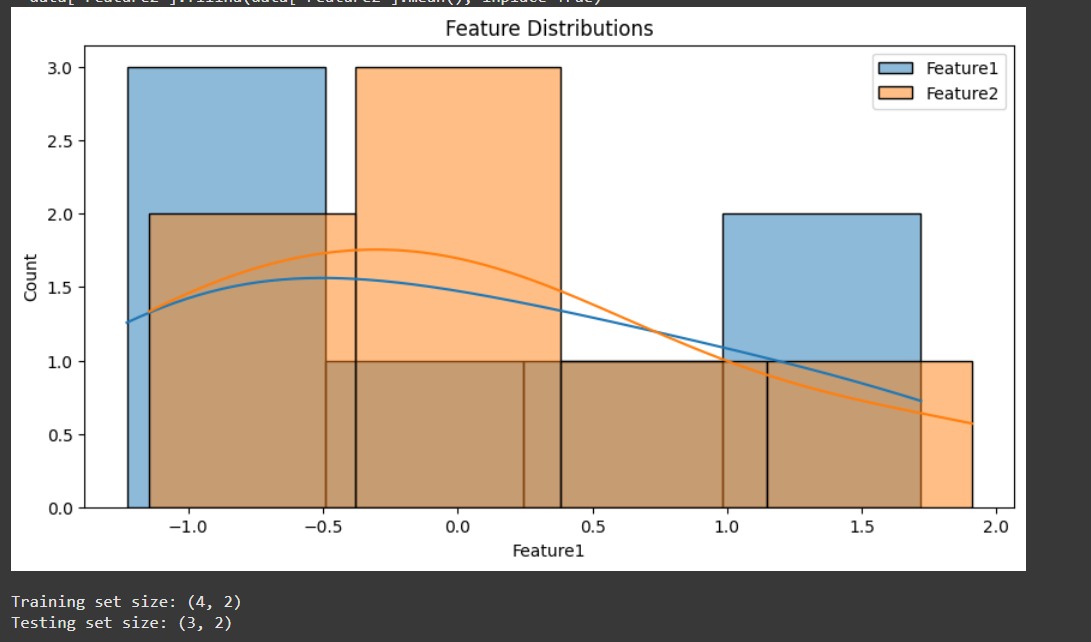
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

print("\nTraining set size:", X\_train.shape)

print("Testing set size:", X\_test.shape)

**Output:**





**Learning Outcome:**

1. Learned how to visualize data distributions using histograms.
2. Understood methods for handling missing values and encoding categorical data.
3. Practiced scaling features and splitting data into training and testing sets.

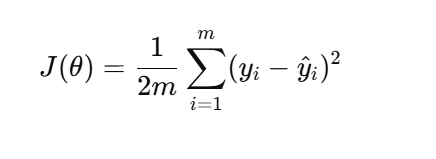
**EXPERIMENT- 3**

**AIM:** Implementation of Linear Regression using Gradient Descent on the advertising dataset.

**THEORY:**

Linear Regression is a statistical method to model the relationship between a dependent variable and one or more independent variables.

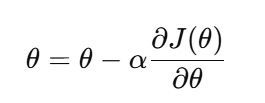
Gradient Descent is an optimization algorithm used to minimize the cost function by iteratively adjusting model parameters to reduce prediction error.

The cost function J(θ) Linear Regression is typically Mean Squared Error (MSE), given by:  


where:

* yi is the actual value,
* yi hat is the predicted value,
* m is the number of training examples.

The Gradient Descent update rule for weights θ is:



**CODE:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

# Load

data = pd.read\_csv("/content/advertising.csv")

# Extract features (X) and target (y)

X = data[['TV', 'Radio', 'Newspaper']].values

y = data['Sales'].values

# Normalize the features for better convergence

X = (X - np.mean(X, axis=0)) / np.std(X, axis=0)

# Add a column of ones to X for the intercept term (bias)

m = len(y)

X = np.hstack((np.ones((m, 1)), X))

# Initialize weights (theta) and hyperparameters

theta = np.zeros(X.shape[1])

learning\_rate = 0.01

num\_iterations = 1000

# Define the cost function (Mean Squared Error)

def compute\_cost(X, y, theta):

    m = len(y)

    predictions = X.dot(theta)

    cost = (1 / (2 \* m)) \* np.sum((predictions - y) \*\* 2)

    return cost

# Implement Gradient Descent

def gradient\_descent(X, y, theta, learning\_rate, num\_iterations):

    m = len(y)

    cost\_history = []

    for i in range(num\_iterations):

        predictions = X.dot(theta)

        errors = predictions - y

        theta -= (learning\_rate / m) \* X.T.dot(errors)

        # Store the cost for each iteration

        cost = compute\_cost(X, y, theta)

        cost\_history.append(cost)

        # Print cost every 100 iterations (optional)

        if i % 100 == 0:

            print(f"Iteration {i}: Cost {cost}")

    return theta, cost\_history

# Run Gradient Descent

theta\_final, cost\_history = gradient\_descent(X, y, theta, learning\_rate, num\_iterations)

# Print final weights

print("\nFinal weights:", theta\_final)

# Plot the cost function history

plt.plot(range(num\_iterations), cost\_history)

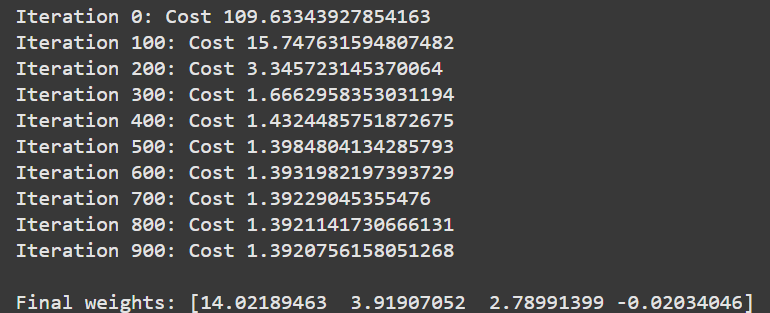
plt.xlabel("Iterations")

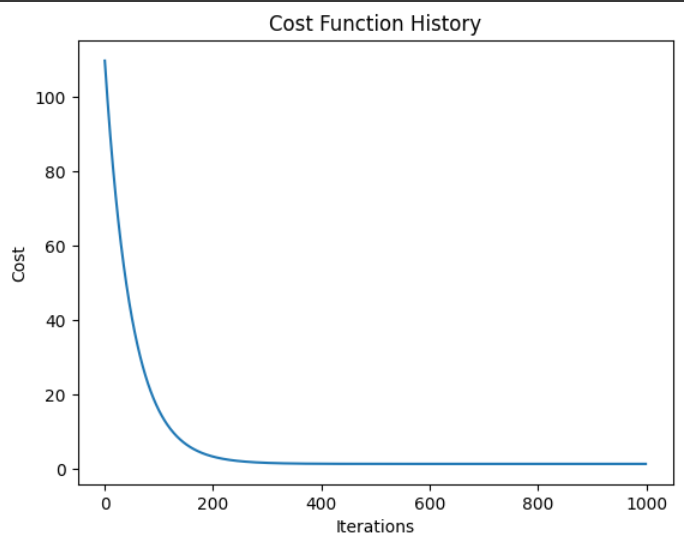
plt.ylabel("Cost")

plt.title("Cost Function History")

plt.show()

**Output:**





**Learning Outcome:**

1. Understood the working of Linear Regression and Gradient Descent.
2. Practiced implementing the gradient descent algorithm to optimize parameters.
3. Visualized the linear relationship between advertising budget and sales.

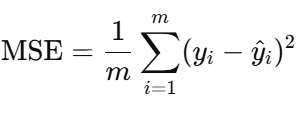
**EXPERIMENT-4**

**Aim:** To implement Linear Regression using Mean Squared Error (MSE)

**Theory:**

Linear Regression is a regression technique that models the relationship between an independent variable(s) and a continuous dependent variable by fitting a line through the data points. This line is the "best fit" line that minimizes the error between predicted and actual values.

The **Mean Squared Error (MSE)** is commonly used as the cost function in linear regression. MSE calculates the average of the squared differences between the actual and predicted values. It is given by:



where:

* yi​ is the actual value,
* yi hat the predicted value,
* m is the number of samples.

Our goal in Linear Regression is to minimize this error by adjusting the parameters (weights) of the model.

**Code:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

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

from sklearn.preprocessing import StandardScaler

# Load the Advertising dataset

data = pd.read\_csv("/content/advertising.csv")  # Ensure the CSV file is in the working directory

# Extract features (X) and target (y)

X = data['TV'].values

y = data['Sales'].values

# Calculating the optimal parameters theta0 and theta1 using the Normal Equation

X\_mean = np.mean(X)

y\_mean = np.mean(y)

theta1 = np.sum((X - X\_mean) \* (y - y\_mean)) / np.sum((X - X\_mean) \*\* 2)

theta0 = y\_mean - theta1 \* X\_mean

# Predicted values

y\_pred = theta0 + theta1 \* X

# Calculating Mean Squared Error

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

# Creating a table to display TV budget, actual sales, and predicted sales

results\_df = pd.DataFrame({

    'TV Advertising Budget': X,

    'Actual Sales': y,

    'Predicted Sales': y\_pred

})

print(results\_df.head())  # Display the first few rows for verification

# Plotting the results

plt.scatter(X, y, color='blue', label="Actual Sales")

plt.plot(X, y\_pred, color='red', label="Predicted Sales")

plt.xlabel("TV Advertising Budget")

plt.ylabel("Sales")

plt.title("Linear Regression using Mean Squared Error")

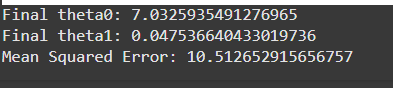
plt.legend()

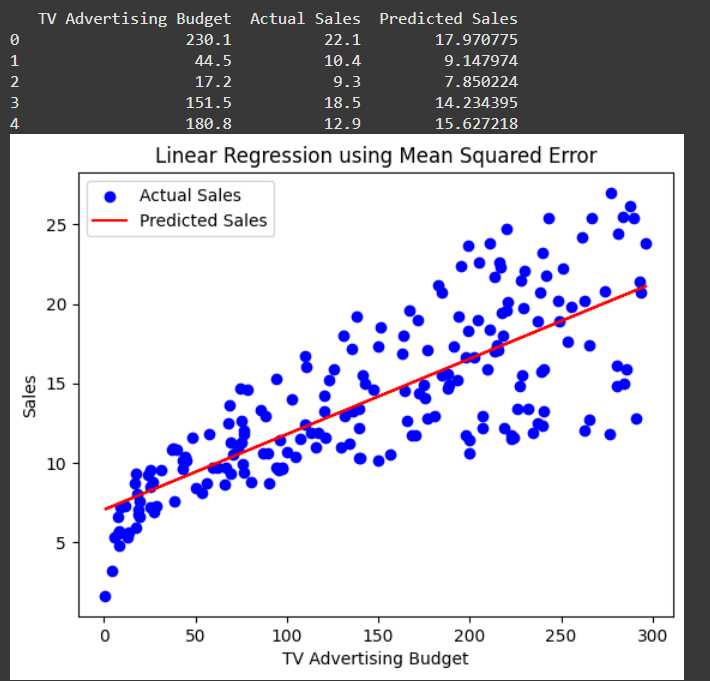
plt.show()

print(f"Final theta0: {theta0}")

print(f"Final theta1: {theta1}")

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

**Output:  
**

****

**Learning Outcomes:**

1. Understood the concept of Mean Squared Error and its role in linear regression.
2. Practiced using MSE as a measure of model performance.
3. Calculated the optimal regression line using the normal equation method for linear regression.

**EXPERIMENT-5**

**AIM:** Implementation of Logistic Regression on IRIS Dataset.

**THEORY:**

Logistic Regression is a classification algorithm used to predict the probability of a categorical dependent variable. It uses the sigmoid function to map predicted values to probabilities.

For binary classification, it predicts the probability that an instance belongs to a particular class. Logistic Regression is widely used in binary classification tasks, but it can also be adapted for multi-class classification using a one-vs-rest approach.

**Code:**

import numpy as np

import pandas as pd

from sklearn.datasets import load\_iris

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

from sklearn.linear\_model import LogisticRegression

from sklearn.metrics import accuracy\_score, confusion\_matrix

import seaborn as sns

import matplotlib.pyplot as plt

# Loading the Iris dataset

iris = load\_iris()

X = iris.data

y = iris.target

# Splitting 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\_state=42)

# Creating and training the Logistic Regression model

model = LogisticRegression(max\_iter=200)

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

# Making predictions

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

# Evaluating model performance

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

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

# Printing accuracy

print(f"Accuracy: {accuracy \* 100:.2f}%")

# Plotting the confusion matrix

sns.heatmap(conf\_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=iris.target\_names, yticklabels=iris.target\_names)

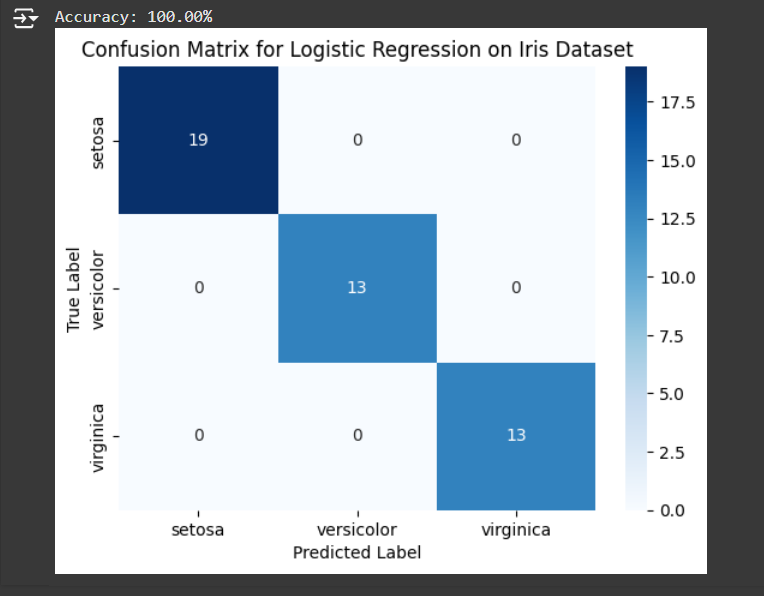
plt.xlabel("Predicted Label")

plt.ylabel("True Label")

plt.title("Confusion Matrix for Logistic Regression on Iris Dataset")

plt.show()

**Output:**



**Learning Outcomes:**

1. Understood the application of Logistic Regression for multi-class classification.
2. Gained experience in training and testing a model on the Iris dataset.
3. Learned to evaluate classification model performance using accuracy and a confusion matrix.

**EXPERIMENT-6**

**AIM:** Implementation of Decision Trees using CART Algorithm on Breast Cancer Dataset.

**THEORY:**

CART (Classification and Regression Trees) is a decision tree algorithm that splits data based on features to create branches, aiming to reduce impurity at each split.

The algorithm uses Gini Impurity as a criterion for split decisions. The goal is to minimize the Gini Impurity, resulting in pure or homogeneous nodes, which improves classification accuracy.

**CODE:**

from sklearn.datasets import load\_breast\_cancer

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

from sklearn.tree import DecisionTreeClassifier

from sklearn.metrics import accuracy\_score, confusion\_matrix

import seaborn as sns

import matplotlib.pyplot as plt

# Loading the Breast Cancer dataset

data = load\_breast\_cancer()

X = data.data

y = data.target

# Splitting 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\_state=42)

# Creating and training the Decision Tree model using CART

model = DecisionTreeClassifier(criterion="gini", random\_state=42)

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

# Making predictions

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

# Evaluating model performance

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

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

# Printing accuracy

print(f"Accuracy: {accuracy \* 100:.2f}%")

# Plotting the confusion matrix

sns.heatmap(conf\_matrix, annot=True, fmt="d", cmap="Blues", xticklabels=data.target\_names, yticklabels=data.target\_names)

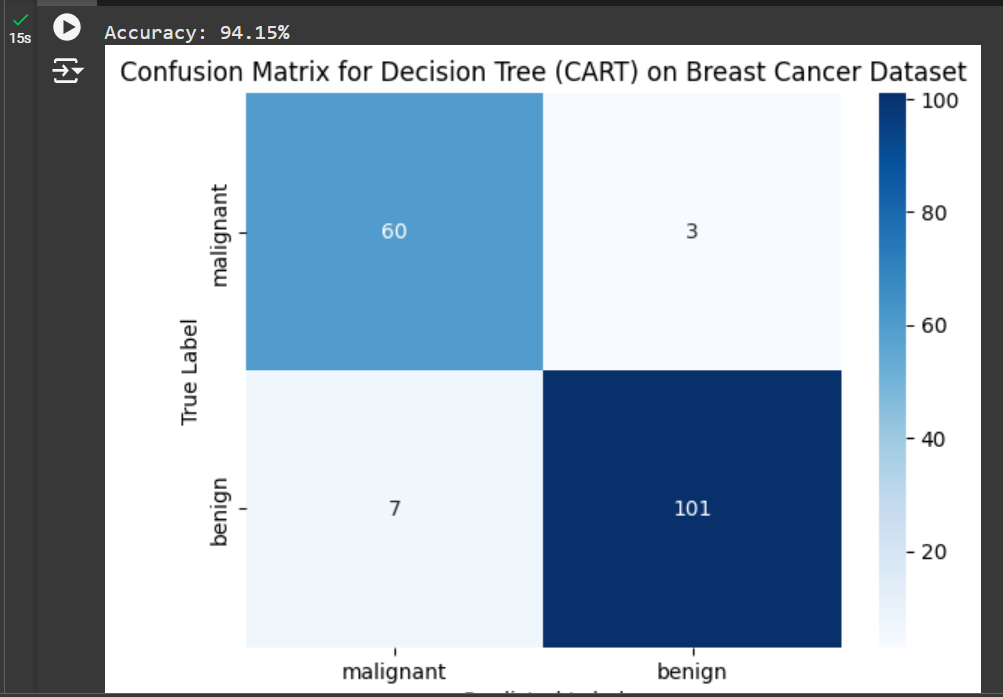
plt.xlabel("Predicted Label")

plt.ylabel("True Label")

plt.title("Confusion Matrix for Decision Tree (CART) on Breast Cancer Dataset")

plt.show()

**Output:**



**Learning Outcomes:**

1. Gained understanding of the CART algorithm and its use of Gini Impurity for decision-making.
2. Applied a Decision Tree model for binary classification on the Breast Cancer dataset.
3. Learned to evaluate model performance using accuracy and a confusion matrix.

**EXPERIMENT - 7**

**Aim:** To build a decision tree for binary classification using the ID3 algorithm on the breast cancer dataset.

**Theory:**  
The ID3 (Iterative Dichotomiser 3) algorithm is a widely used algorithm for constructing decision trees. It uses a top-down, greedy approach, selecting attributes that maximize the information gain.

Information gain is derived from entropy, a measure of uncertainty or impurity in data. ID3 splits the dataset based on the attribute that reduces uncertainty the most, creating branches in the decision tree until reaching pure or mostly pure subsets.

**Steps in the ID3 Algorithm:**

1. **Calculate Entropy**: Measure the uncertainty in the dataset.
2. **Calculate Information Gain**: For each attribute, calculate how much uncertainty is reduced if we split the data by that attribute.
3. **Select Attribute**: Choose the attribute with the highest information gain as the root node.
4. **Split Data**: Divide the dataset based on the chosen attribute.
5. **Recursion**: Repeat the process for each subset until all data points are classified or no further splits are possible.

**Code:**

import numpy as np

import pandas as pd

from sklearn.datasets import load\_breast\_cancer

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

from collections import Counter

data = load\_breast\_cancer()

df = pd.DataFrame(data.data, columns=data.feature\_names)

df['target'] = data.target

def entropy(y):

    counts = np.bincount(y)

    probabilities = counts / len(y)

    entropy\_value = -np.sum([p \* np.log2(p) for p in probabilities if p > 0])

    return entropy\_value

def information\_gain(X, y, feature):

    # Total entropy of the parent node

    parent\_entropy = entropy(y)

    # Threshold to split

    threshold = X[feature].mean()

    left\_indices = X[feature] <= threshold

    right\_indices = X[feature] > threshold

    n\_left, n\_right = sum(left\_indices), sum(right\_indices)

    # Weighted entropy of children

    left\_entropy = entropy(y[left\_indices]) if n\_left > 0 else 0

    right\_entropy = entropy(y[right\_indices]) if n\_right > 0 else 0

    weighted\_entropy = (n\_left / len(y)) \* left\_entropy + (n\_right / len(y)) \* right\_entropy

    info\_gain = parent\_entropy - weighted\_entropy

    return info\_gain

# Best feature to split

def best\_feature\_split(X, y):

    print("\nEvaluating information gain for each feature...")

    gains = {feature: information\_gain(X, y, feature) for feature in X.columns}

    best\_feature = max(gains, key=gains.get)

    print(f"\nBest feature to split on: '{best\_feature}' with gain {gains[best\_feature]}")

    return best\_feature

# tree nodes and make decisions

class DecisionTree:

    def \_\_init\_\_(self, depth=0, max\_depth=5):

        self.depth = depth

        self.max\_depth = max\_depth

        self.feature = None

        self.threshold = None

        self.left = None

        self.right = None

        self.label = None

    def fit(self, X, y):

        # If only one class is present, make it a leaf node

        if len(np.unique(y)) == 1:

            self.label = y.iloc[0]

            return

        # If max depth is reached, set the most common label

        if self.depth >= self.max\_depth:

            self.label = Counter(y).most\_common(1)[0][0]

            return

        # Select best feature to split on

        self.feature = best\_feature\_split(X, y)

        self.threshold = X[self.feature].mean()

        print(f"Splitting on feature '{self.feature}' at threshold {self.threshold}")

        # Split the dataset into two parts

        left\_indices = X[self.feature] <= self.threshold

        right\_indices = X[self.feature] > self.threshold

        # Create left and right child nodes

        self.left = DecisionTree(depth=self.depth + 1, max\_depth=self.max\_depth)

        self.right = DecisionTree(depth=self.depth + 1, max\_depth=self.max\_depth)

        # Recursively fit each child

        self.left.fit(X[left\_indices], y[left\_indices])

        self.right.fit(X[right\_indices], y[right\_indices])

    def predict\_one(self, x):

        if self.label is not None:

            return self.label

        elif x[self.feature] <= self.threshold:

            return self.left.predict\_one(x)

        else:

            return self.right.predict\_one(x)

    def predict(self, X):

        predictions = X.apply(lambda x: self.predict\_one(x), axis=1)

        print("Prediction completed.")

        return predictions

# Train and evaluate the decision tree

def main():

    # Split the dataset into features and target variable

    X = df.drop(columns='target')

    y = df['target']

    # Split data into training and test sets

    X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

    # Initialize and fit the decision tree

    tree = DecisionTree(max\_depth=5)

    print("\nTraining Decision Tree...")

    tree.fit(X\_train, y\_train)

    # Make predictions on the test set

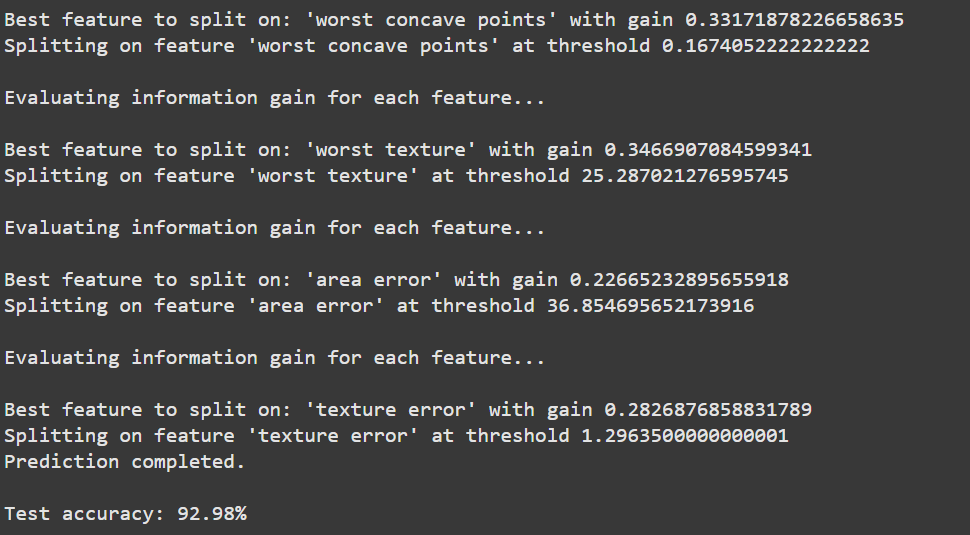
    predictions = tree.predict(X\_test)

    accuracy = np.mean(predictions == y\_test)

    print(f"\nTest accuracy: {accuracy \* 100:.2f}%")

main()

**Output:**



**Code flow:**

1. **Data Loading**: Loads breast cancer dataset and organizes it in a DataFrame for analysis.
2. **Entropy Calculation**: Measures data impurity to guide feature splits.
3. **Information Gain**: Calculates the reduction in entropy when splitting by each feature.
4. **Best Feature Selection**: Chooses the feature with the highest information gain for the next split.
5. **Tree Construction**: Recursively splits data, building branches until max depth or pure nodes are reached.
6. **Leaf Nodes**: Labels branches as malignant or benign when a node is pure or max depth is reached.
7. **Prediction**: Traverses the tree to classify test samples based on their feature values.
8. **Evaluation**: Outputs test accuracy, showing the model’s performance in classifying tumors.

**Learning Outcome:**

1. **Interpretability**: ID3 provides clear decision rules, making the model easy to understand.
2. **Feature Importance**: Key features selected at the top of the tree highlight the most relevant predictors.
3. **Overfitting Risk**: Deep trees may overfit; pruning or depth limits can help mitigate this.

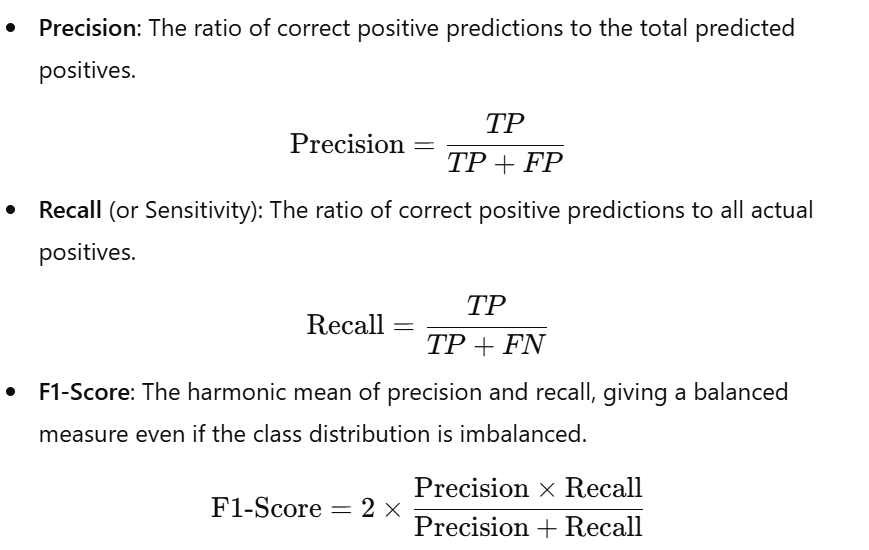
**EXPERIMENT-8**

**Aim:** To develop a Decision Tree classifier using the C4.5 algorithm on a breast cancer dataset to predict binary outcomes (malignant or benign).

**Theory**

The C4.5 algorithm is an extension of the ID3 algorithm, widely used for constructing decision trees. It improves on ID3 by handling both continuous and discrete attributes, handling missing data, and applying a pruning mechanism to reduce overfitting.

The algorithm recursively splits the dataset by selecting the attribute with the highest information gain ratio, which measures the purity of subsets created by each split.   
The resulting tree helps in classifying instances into predefined categories—in this case, predicting whether breast cancer is malignant or benign based on given features.



**Code:**

from sklearn.datasets import load\_breast\_cancer

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

from sklearn.tree import DecisionTreeClassifier

from sklearn import metrics

data = load\_breast\_cancer()

X = data.data

y = data.target

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

# Decision Tree with entropy criterion (similar to C4.5)

clf = DecisionTreeClassifier(criterion='entropy', random\_state=42)

clf.fit(X\_train, y\_train)

# Predict on test set

y\_pred = clf.predict(X\_test)

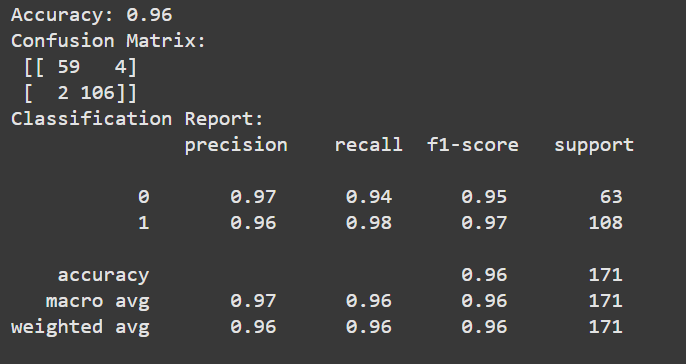
accuracy = metrics.accuracy\_score(y\_test, y\_pred)

print(f"Accuracy: {accuracy:.2f}")

print("Confusion Matrix:\n", metrics.confusion\_matrix(y\_test, y\_pred))

print("Classification Report:\n", metrics.classification\_report(y\_test, y\_pred))

**Output:**



**Learning Outcomes:**

1. Gain practical experience in binary classification using decision trees.
2. Learn to apply the entropy criterion for effective decision-making in C4.5.
3. Evaluate classification model performance using accuracy, precision, and recall.

**EXPERIMENT-9**

**Aim:** To implement the K-Nearest Neighbors (KNN) algorithm

**Theory:**

The K-Nearest Neighbors (KNN) algorithm is a simple, instance-based learning method for classification and regression. It is a non-parametric algorithm, meaning it makes no assumptions about the underlying data distribution. In classification, KNN predicts the class of a sample based on the majority class among its KKK closest neighbors in the feature space.

To classify a new point, the algorithm calculates the distance (commonly Euclidean) from the new point to all points in the training set. Then, it selects the KKK nearest neighbors and assigns the new point to the class that is most common among them.

Breast cancer dataset from sklearn is used, which has features representing characteristics of the cell nuclei and a binary target indicating whether a tumor is benign (class 0) or malignant (class 1).

**Code:**

from sklearn.datasets import load\_breast\_cancer

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

from sklearn.neighbors import KNeighborsClassifier

from sklearn import metrics

data = load\_breast\_cancer()

X = data.data

y = data.target

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

k=5

knn = KNeighborsClassifier(n\_neighbors=k)

print("KNN implemented for k =",k)

knn.fit(X\_train, y\_train)

y\_pred = knn.predict(X\_test)

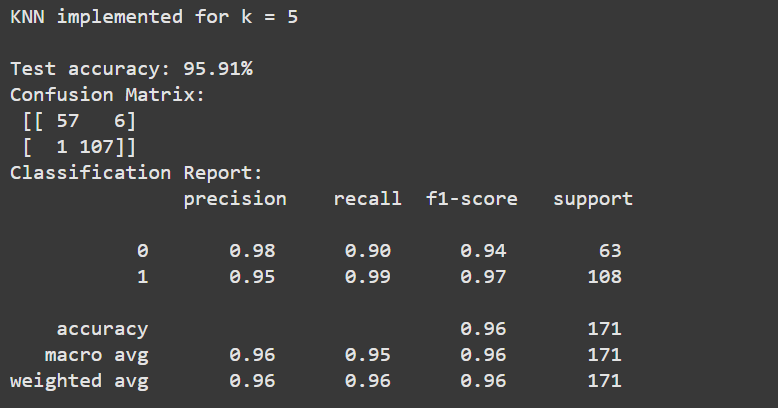
accuracy = metrics.accuracy\_score(y\_test, y\_pred)

print(f"\nTest accuracy: {accuracy \* 100:.2f}%")

print("Confusion Matrix:\n", metrics.confusion\_matrix(y\_test, y\_pred))

print("Classification Report:\n", metrics.classification\_report(y\_test, y\_pred))

**Output:**



**Observations**

* **Accuracy**: Displays the proportion of correctly predicted cases in the test set.
* **Confusion Matrix**: Shows the counts of True Positives, True Negatives, False Positives, and False Negatives, helping to visualize the performance of the model across both classes.
* **Classification Report**: Provides precision, recall, and F1-score for each class, offering deeper insight into how well the model is distinguishing between malignant and benign classes.

**Learning Outcomes:**

1. Understand the fundamentals of the K-Nearest Neighbors algorithm for classification.
2. Learn to use KNN for binary classification tasks using real-world datasets.
3. Evaluate and interpret model performance through accuracy, confusion matrix, and classification report.

**EXPERIMENT-10**

**AIM**: To perform data classification using SVM

**THEORY**:

**Support Vector Machines (SVM):** Support Vector Machines are supervised machine learning algorithms that can be used for classification and regression tasks. The primary goal of SVM is to find the optimal hyperplane that best separates data points of different classes.

The hyperplane is chosen in such a way that it maximizes the margin between the classes, leading to better generalization performance. Key concepts related to SVM:

* 1. **Objective:** The main objective of SVM in classification is to find a hyperplane that maximizes the margin between different classes of data points. This margin represents the distance between the hyperplane and the closest data points from each class.
  2. **Kernel Functions**: SVM can handle non-linear data by transforming it to a higher-dimensional space using kernels like linear, polynomial, and RBF.
  3. **Support Vectors**: These are the closest data points to the hyperplane, crucial for defining its position.
  4. **Advantages**: Effective in high-dimensional spaces, robust to outliers, and supports non-linear data with kernels.
  5. **Disadvantages**: Requires careful selection of kernels and is sensitive to hyperparameters.

**CODE:**

from sklearn import datasets

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

from sklearn.svm import SVC

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix

import matplotlib.pyplot as plt

import seaborn as sns

# Load the Iris dataset

iris = datasets.load\_iris()

X = iris.data

y = iris.target

# Split the data into training and testing sets (70% train, 30% test)

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

svm\_clf = SVC(kernel='linear')  # You can try other kernels like 'rbf', 'poly', etc.

svm\_clf.fit(X\_train, y\_train)

y\_pred = svm\_clf.predict(X\_test)

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

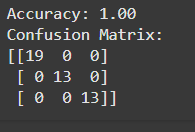
print(f"Accuracy: {accuracy:.2f}")

print("Confusion Matrix:")

cm = confusion\_matrix(y\_test, y\_pred)

print(cm)

**OUTPUT:**



**Learning Outcomes:**

1. Understanding of Support Vector Machines (SVM) as a powerful machine learning algorithm for classification tasks.

2. Knowledge of key concepts related to SVM, including the objective of maximizing the margin, kernel functions, support vectors, and advantages/disadvantages.

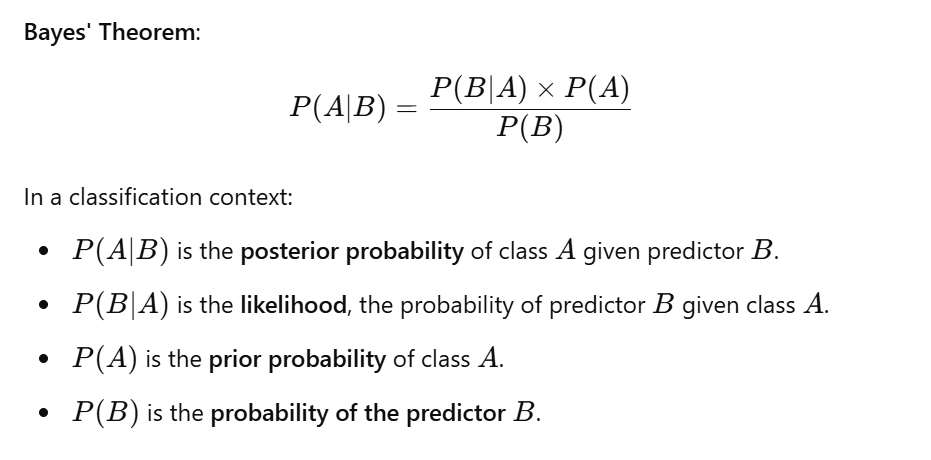
3. Practical experience in implementing SVM for data classification using Python.

**EXPERIMENT-11**

**AIM:** Implementation of Naive Bayes Classifier

**Theory**:

Naive Bayes is a probabilistic classifier based on Bayes' theorem. It assumes that features in a dataset are independent of each other, which is why it's called "naive." Despite this strong and often unrealistic assumption, Naive Bayes performs well in many real-world classification tasks.



The algorithm calculates the posterior probability for each class and selects the class with the highest probability.

**CODE:**

from sklearn import datasets

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

from sklearn.naive\_bayes import GaussianNB

from sklearn.metrics import accuracy\_score, classification\_report

iris = datasets.load\_iris()

X, y = iris.data, iris.target

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

nb\_classifier = GaussianNB()

nb\_classifier.fit(X\_train, y\_train)

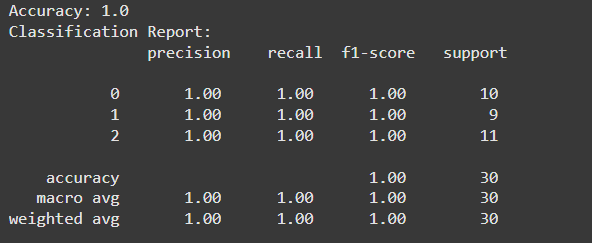
y\_pred = nb\_classifier.predict(X\_test)

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

print("Accuracy:", accuracy)

print("Classification Report:\n", classification\_report(y\_test, y\_pred))

**Output:**



**Learning Outcomes:**

1. **Efficient and Effective**: Naive Bayes is fast, handles large datasets well, and provides a solid baseline for simple classification tasks.
2. **Independence Assumption**: Works best when features are independent; highly correlated features may reduce performance.
3. **Gaussian Naive Bayes**: Assumes a normal distribution for continuous data, making it suitable for datasets like Iris, often achieving high accuracy.

**EXPERIMENT-12**

**AIM:** Implementation of Random Forest

**THEORY:**   
Random Forest is an ensemble learning method used for classification and regression tasks. It works by constructing multiple decision trees during training and outputting the mode of the classes for classification or the mean prediction for regression. This approach increases accuracy and prevents overfitting, as the diversity among the trees helps reduce variance. Random Forest is robust, handles large datasets effectively, and can manage missing values and maintain accuracy even when a portion of the data is missing.

**CODE:**

from sklearn.datasets import load\_iris

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

from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import accuracy\_score

# Load the Iris dataset

iris = load\_iris()

X, y = iris.data, iris.target

# 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)

clf = RandomForestClassifier(n\_estimators=100, random\_state=42)

clf.fit(X\_train, y\_train)

# Make predictions on the test set

y\_pred = clf.predict(X\_test)

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

print("Accuracy:", accuracy)

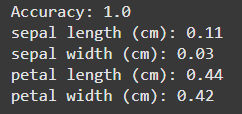
# Display feature importances

importances = clf.feature\_importances\_

for feature, importance in zip(iris.feature\_names, importances):

print(f"{feature}: {importance:.2f}")

**Output:**

****

**Learning Outcomes:**

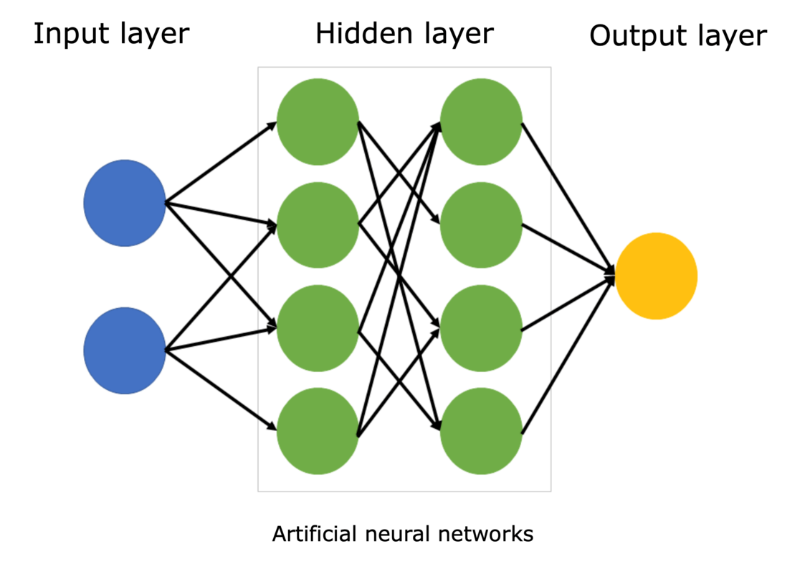
1. **Understanding Ensemble Learning**: Learn how Random Forest combines multiple decision trees to improve accuracy and reduce overfitting.
2. **Feature Importance**: Gain insight into identifying key features for model predictions using feature importance in Random Forest.
3. **Model Evaluation**: Develop skills to evaluate model performance and understand the robustness of Random Forest on different datasets.

**EXPERIMENT-14**

**AIM:** Implementation of Basic Neural Network

**THEORY:**   
A neural network is a fundamental component of deep learning, designed to mimic the human brain's structure for problem-solving tasks.

It consists of layers of nodes, including an input layer, one or more hidden layers, and an output layer. Each node, or artificial neuron, processes inputs using weights, biases, and an activation function to produce an output.



The network is trained using a dataset, adjusting weights through backpropagation and optimizing them using algorithms like gradient descent to minimize errors.

This learning process enables the neural network to recognize complex patterns in data.

**CODE:**

import numpy as np

from sklearn.datasets import load\_iris

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

from sklearn.preprocessing import OneHotEncoder, StandardScaler

iris = load\_iris()

X = iris.data

y = iris.target.reshape(-1, 1)

encoder = OneHotEncoder(sparse\_output=False

y\_encoded = encoder.fit\_transform(y)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y\_encoded, test\_size=0.2, random\_state=42)

scaler = StandardScaler()

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

X\_test = scaler.transform(X\_test)

input\_size = X\_train.shape[1]

hidden\_size = 5

output\_size = y\_train.shape[1]

learning\_rate = 0.01

epochs = 1000

weights\_input\_hidden = np.random.rand(input\_size, hidden\_size)

weights\_hidden\_output = np.random.rand(hidden\_size, output\_size)

bias\_hidden = np.zeros((1, hidden\_size))

bias\_output = np.zeros((1, output\_size))

# Activation function (ReLU and softmax for output)

def sigmoid(x):

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

def sigmoid\_derivative(x):

return x \* (1 - x)

# Training loop

for epoch in range(epochs):

# Forward pass

hidden\_input = np.dot(X\_train, weights\_input\_hidden) + bias\_hidden

hidden\_output = sigmoid(hidden\_input)

final\_input = np.dot(hidden\_output, weights\_hidden\_output) + bias\_output

final\_output = sigmoid(final\_input)

# Calculate error

error = y\_train - final\_output

# Backpropagation

d\_output = error \* sigmoid\_derivative(final\_output)

error\_hidden = d\_output.dot(weights\_hidden\_output.T)

d\_hidden = error\_hidden \* sigmoid\_derivative(hidden\_output)

# Update weights and biases

weights\_hidden\_output += hidden\_output.T.dot(d\_output) \* learning\_rate

weights\_input\_hidden += X\_train.T.dot(d\_hidden) \* learning\_rate

bias\_output += np.sum(d\_output, axis=0, keepdims=True) \* learning\_rate

bias\_hidden += np.sum(d\_hidden, axis=0, keepdims=True) \* learning\_rate

if epoch % 100 == 0:

loss = np.mean(np.square(error))

print(f"Epoch {epoch}, Loss: {loss:.4f}")

# Prediction and accuracy check

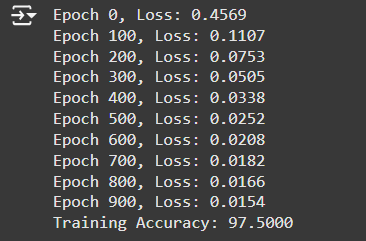
predictions = np.argmax(final\_output, axis=1)

y\_true = np.argmax(y\_train, axis=1)

accuracy = np.mean(predictions == y\_true)

print(f"Training Accuracy: {accuracy\*100:.4f}")

**Output:**



**Learning Outcomes:**

1. Gained foundational insights into neural network architecture, including the forward and backward pass, weight updates, and activation functions.
2. Understood the impact of key hyperparameters, such as learning rate and epoch count, on model learning and performance.
3. Recognized practical challenges in training I.E. convergence speed and accuracy.

**EXPERIMENT-15**

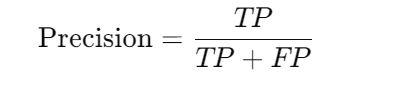
**Aim:** Evaluation Metrics in Machine Learning

**Theory:**

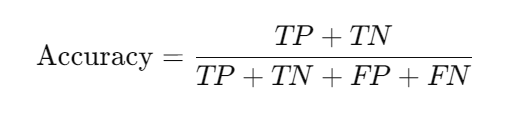
**Confusion Matrix**  
The Confusion Matrix is a table that describes the performance of a classification algorithm by comparing predicted values against actual values. It summarizes results into four categories:

* **True Positives (TP):** Positive instances correctly predicted as positive.
* **True Negatives (TN):** Negative instances correctly predicted as negative.
* **False Positives (FP):** Negative instances incorrectly predicted as positive.
* **False Negatives (FN):** Positive instances incorrectly predicted as negative.

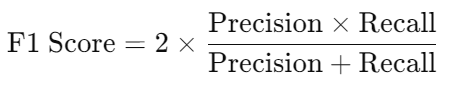
**Precision**  
Precision is the ratio of correctly predicted positive observations to the total predicted positive observations, helping identify how many predicted positives are actually correct.



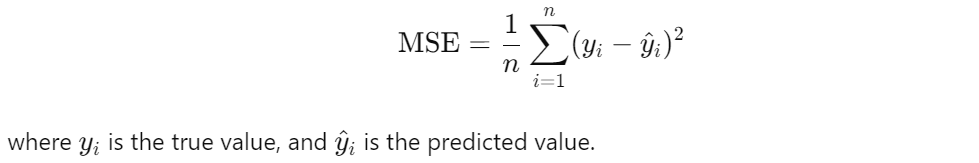
**Accuracy**  
Accuracy is the ratio of correctly predicted instances to the total instances, indicating the model’s overall correctness. However, it can be misleading in cases of class imbalance.



**F1 Score**  
The F1 score is the harmonic mean of precision and recall, balancing the two by considering both false positives and false negatives. It is particularly useful with imbalanced classes.



**Mean Squared Error (MSE)**  
MSE is commonly used for regression models, measuring the average squared difference between actual and predicted values. A lower MSE indicates a better model fit.



**CODE:**

import numpy as np

import pandas as pd

from sklearn.datasets import load\_breast\_cancer

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

from sklearn.metrics import confusion\_matrix, precision\_score, accuracy\_score, f1\_score, mean\_squared\_error

from sklearn.linear\_model import LogisticRegression

data = load\_breast\_cancer()

X = pd.DataFrame(data.data, columns=data.feature\_names) # Features

y = pd.Series(data.target) # Target

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

model = LogisticRegression(max\_iter=5000) # Increase max\_iter for convergence

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

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

conf\_matrix = confusion\_matrix(y\_test, y\_pred)

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

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

f1 = f1\_score(y\_test, y\_pred)

# Mean Squared Error

y\_test\_reg = y\_test[:5].astype(float) # Use first few true labels as regression values

y\_pred\_reg = model.predict\_proba(X\_test[:5])[:, 1] # Probabilities as continuous values

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

print("Confusion Matrix:\n", conf\_matrix)

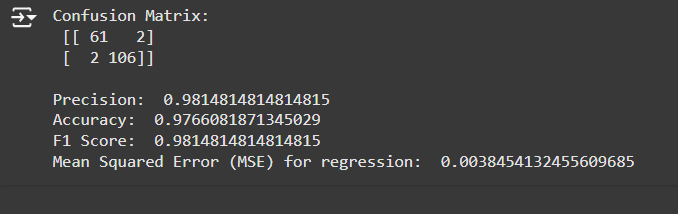
print("\nPrecision: ", precision)

print("Accuracy: ", accuracy)

print("F1 Score: ", f1)

print("Mean Squared Error (MSE) for regression: ", mse)

**Output:**



**Learning Outcomes:**

1. **Confusion Matrix:** Provides insights into model performance by showing correct and incorrect predictions, helping identify error types like false positives and false negatives.
2. **Precision:** Important for scenarios where false positives are critical, ensuring that predicted positives are likely accurate (e.g., in medical diagnostics).
3. **Accuracy:** Gives an overall measure but may be misleading for imbalanced datasets; should be used alongside precision, recall, and F1 score.
4. **F1 Score:** Balances precision and recall, offering a better evaluation for models on imbalanced data.
5. **Mean Squared Error (MSE):** Useful in regression to assess model fit; less relevant for classification tasks.

**EXPERIMENT-16**

**Aim:** Implementation of a Convolutional Neural Network (CNN) using the MNIST dataset.

**Theory:**

Convolutional Neural Networks (CNNs) are a class of deep learning models used for image recognition, object detection, and other tasks involving grid-like data such as images. Unlike traditional neural networks, CNNs take advantage of spatial structure by using convolutional layers, which apply filters to capture spatial hierarchies.

The MNIST dataset is a classic dataset of handwritten digits consisting of 60,000 training images and 10,000 test images. Each image is a 28x28 grayscale image labeled with a digit between 0 and 9. The dataset serves as a standard benchmark for evaluating machine learning algorithms.

**Code:**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow import keras

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

df = pd.read\_csv('/content/tmnist.csv') # Assuming 'tmnist.csv' contains MNIST-like data

print("Shape of the dataset:", df.shape)

X = df.drop(columns=['names', 'labels'], axis=1) # Exclude 'names' and 'labels' from features

y = df['labels']

no\_of\_classes = y.nunique()

print("Number of unique labels:", no\_of\_classes)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Normalize pixel values

X\_train, X\_test = X\_train / 255.0, X\_test / 255.0

model = keras.Sequential([

keras.layers.Dense(128, input\_shape=(784,), activation='relu'), # Flattened input layer

keras.layers.Dense(no\_of\_classes, activation='softmax') # Output layer for the number of classes

])

# Compile the model

model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

# Train the model

history = model.fit(X\_train, y\_train, epochs=20, validation\_split=0.2, verbose=1)

# Evaluate the model on test data

test\_loss, test\_acc = model.evaluate(X\_test, y\_test)

print('Test accuracy:', test\_acc)

# Plot accuracy over epochs

plt.plot(history.history['accuracy'], label='Train Accuracy')

plt.plot(history.history['val\_accuracy'], label='Validation Accuracy')

plt.title('Model Accuracy')

plt.xlabel('Epoch')

plt.ylabel('Accuracy')

plt.legend()

plt.show()

# Plot a random sample of test images with predicted and true labels

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

figure = plt.figure(figsize=(20, 8))

for i, index in enumerate(np.random.choice(X\_test.shape[0], size=10, replace=False)):

ax = figure.add\_subplot(2, 5, i + 1, xticks=[], yticks=[])

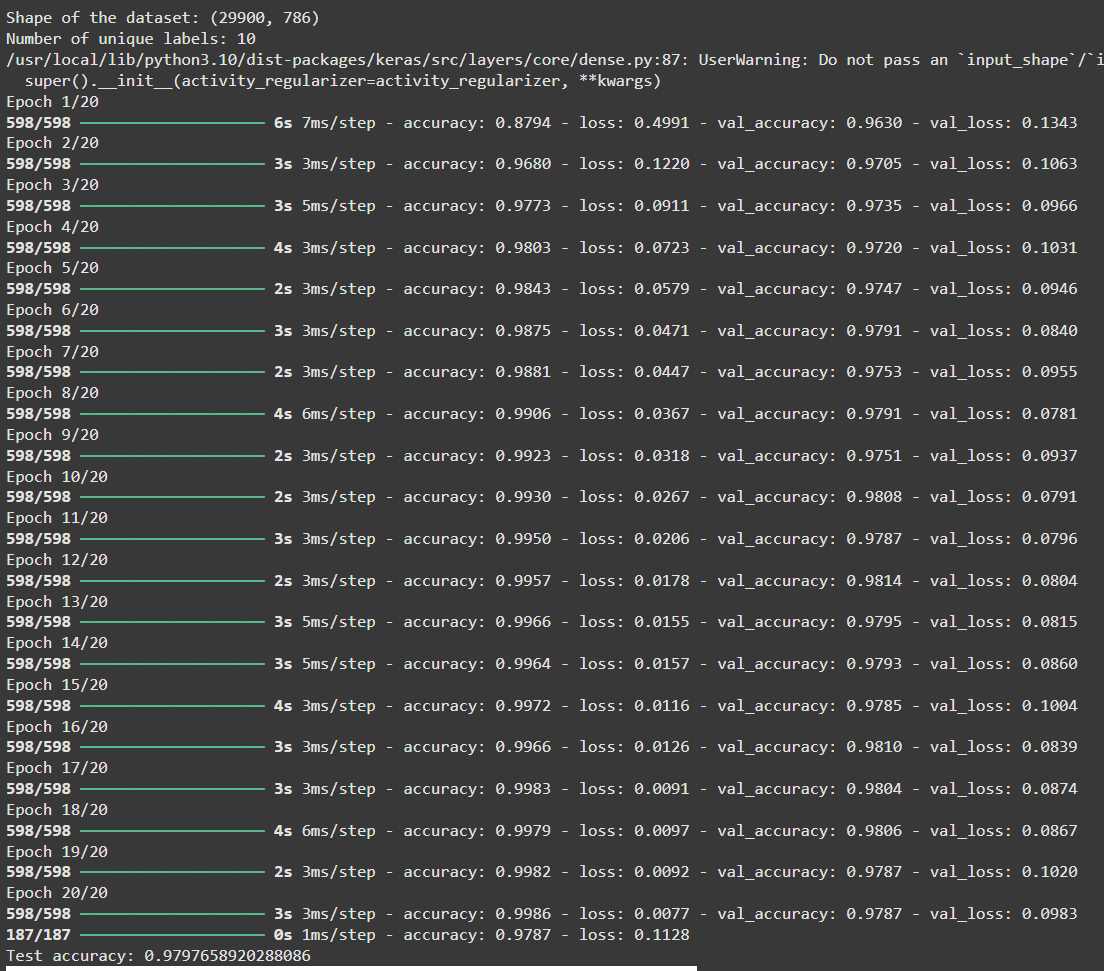
ax.imshow(X\_test.iloc[index].values.reshape(28, 28), cmap='gray')

pred\_label = np.argmax(y\_pred[index])

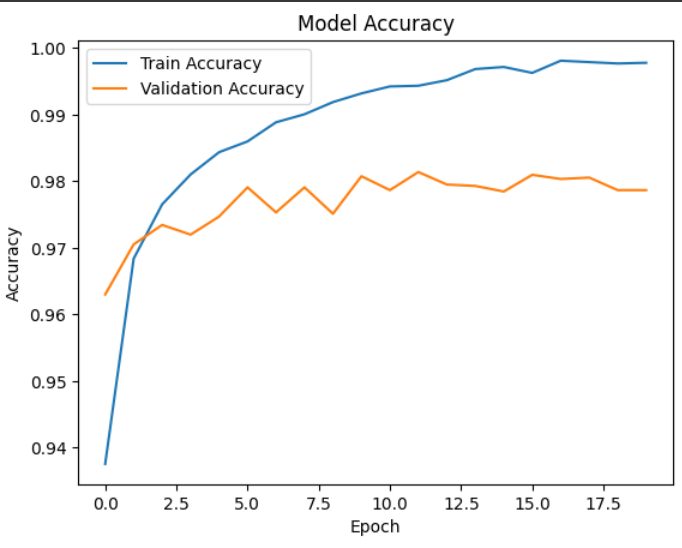
true\_label = y\_test.iloc[index]

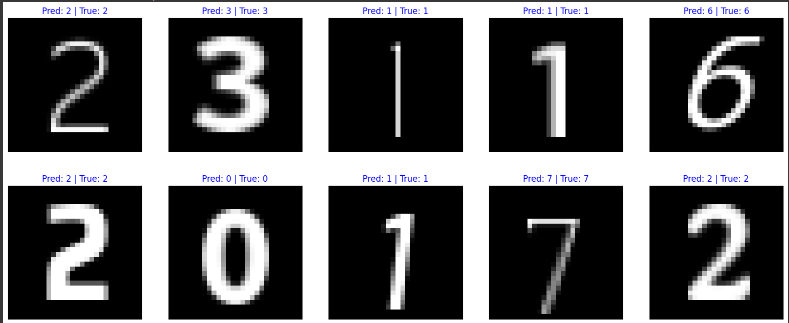
ax.set\_title(f"Pred: {pred\_label} | True: {true\_label}", color=("blue" if pred\_label == true\_label else "red"))

plt.show()

**Output:**

**Accuracy: 97.97%**



****

**Learning Outcomes:**

1. Learned how to implement a basic neural network using Keras and TensorFlow.

2. Explored data preprocessing, including normalization and class distribution visualization.

3. Evaluated model performance using training accuracy, validation accuracy, and test set accuracy.

**EXPERIMENT-17**

**Aim:** Implementing Dimensionality Reduction using Principal Component Analysis (PCA).

**Theory:**

Dimensionality reduction techniques like PCA help reduce the number of input variables in a dataset while retaining most of the important information.

PCA achieves this by finding a set of new uncorrelated variables (principal components) that successively maximize variance.

It is widely used in preprocessing for machine learning, visualization of high-dimensional data, and noise reduction.

**Code:**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.decomposition import PCA

from sklearn.preprocessing import StandardScaler

df = pd.read\_csv('/content/tmnist.csv') # Replace with appropriate data file

print("Labels: ", df['labels'].unique())

no\_of\_classes = df['labels'].nunique()

X = df.drop(columns=['names', 'labels'], axis=1) # Drop unwanted columns

y = df['labels']

X\_std = StandardScaler().fit\_transform(X)

pca = PCA(n\_components=2) # Reduce to 2 dimensions

principal\_components = pca.fit\_transform(X\_std)

principal\_df = pd.DataFrame(data=principal\_components, columns=['Principal Component 1', 'Principal Component 2'])

# Concatenate with target labels for visualization

final\_df = pd.concat([principal\_df, y.reset\_index(drop=True)], axis=1)

# Visualize the data in the 2D PCA space

plt.figure(figsize=(10, 8))

targets = df['labels'].unique()

colors = ['r', 'g', 'b', 'c', 'm', 'y', 'k'] \* (len(targets) // 7 + 1) # Adjust number of colors

for target, color in zip(targets, colors):

indices\_to\_keep = final\_df['labels'] == target

plt.scatter(final\_df.loc[indices\_to\_keep, 'Principal Component 1'],

final\_df.loc[indices\_to\_keep, 'Principal Component 2'], c=color, s=50)

plt.xlabel('Principal Component 1')

plt.ylabel('Principal Component 2')

plt.title('2 Component PCA')

plt.legend(targets, loc='best')

plt.show()

explained\_variance = pca.explained\_variance\_ratio\_

print("Explained Variance Ratio:", explained\_variance)

print("Total Variance Explained by 2 Components:", sum(explained\_variance))

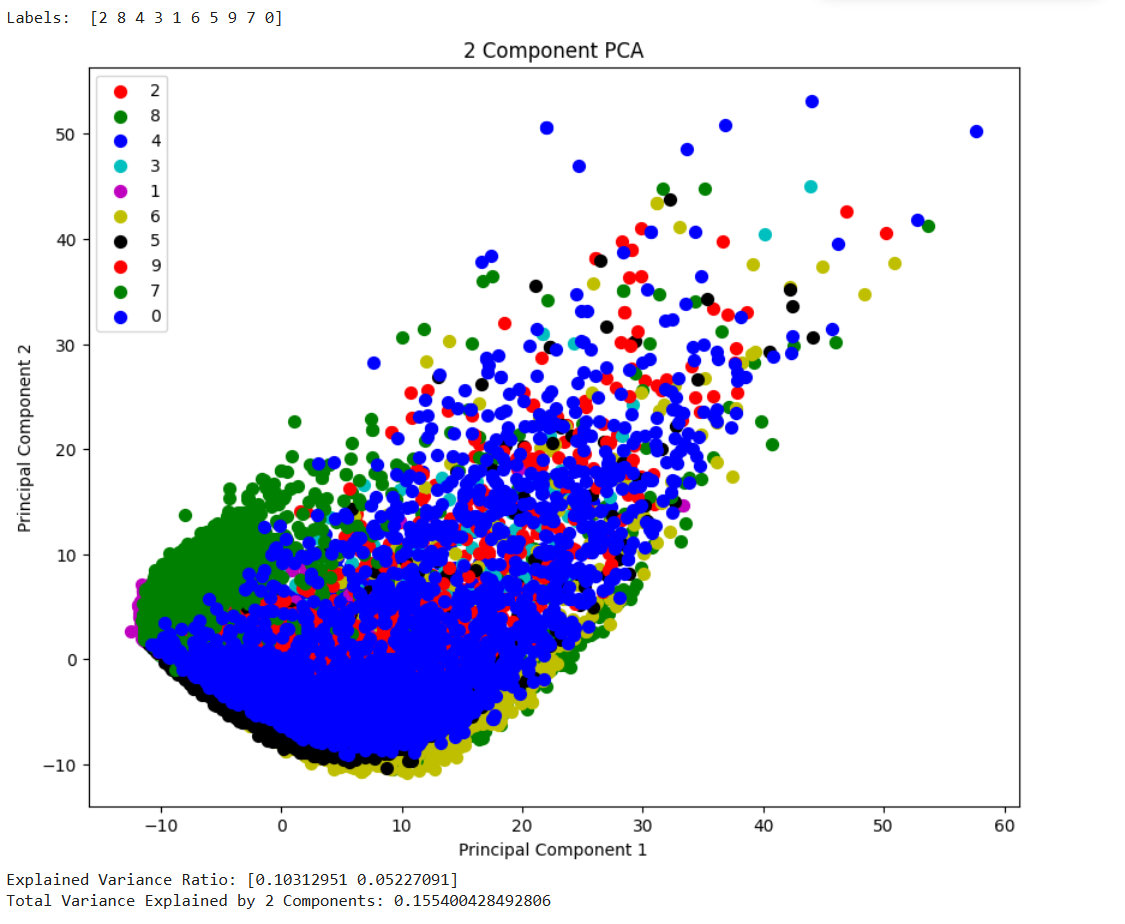
**Learning Outcomes:**

1. PCA effectively reduces the dimensionality of the dataset to 2 components.

2. Visualization in the 2D space helps understand class separability and data clustering.

3. The explained variance shows the amount of information retained by the selected principal components.

**Output:**



**EXPERIMENT-18**

**Aim:** Build an Artificial Neural Network (ANN) with backpropagation.

**Theory:**

An **ANN** is a model inspired by the human brain, consisting of input, hidden, and output layers. During training, data passes through these layers (forward pass), and the error is calculated using a loss function.

**Backpropagation** then adjusts the weights and biases by calculating gradients and updating them via optimization (e.g., gradient descent). This process is repeated over multiple epochs to minimize error.

**Hyperparameters:** Training an ANN involves tuning several hyperparameters:

* **Learning Rate:** Determines the step size for weight updates. A higher learning rate may lead to faster convergence, but it could also overshoot the optimal solution. A lower rate might make training slower but can yield a more accurate solution.
* **Batch Size:** Controls the number of samples passed through the network before updating the weights.
* **Epochs:** The number of complete passes through the entire training dataset.

**Code:**

import keras

import numpy as np

from keras.datasets import mnist

from keras.models import Sequential

from keras.layers import Dense

from keras.utils import to\_categorical

# Load dataset

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

# Scale input values to [0, 1]

x\_train = x\_train.reshape(60000, 784).astype('float32') / 255

x\_test = x\_test.reshape(10000, 784).astype('float32') / 255

# Convert target values to one-hot encoding

y\_train = to\_categorical(y\_train, num\_classes=10)

y\_test = to\_categorical(y\_test, num\_classes=10)

# Build the model

model = Sequential([

Dense(10, activation='sigmoid', input\_shape=(784,)), # Hidden layer

Dense(10, activation='softmax') # Output layer

])

# Compile the model

model.compile(loss="categorical\_crossentropy",

optimizer="sgd",

metrics=['accuracy'])

# Train the model

history = model.fit(x\_train, y\_train, batch\_size=100, epochs=20) # here 20 epochs

# Evaluate the model on test data

test\_loss, test\_acc = model.evaluate(x\_test, y\_test)

print(f'Test accuracy: {round(test\_acc, 4)}')

# Display a sample prediction

example\_index = 11

prediction = model.predict(x\_test[example\_index].reshape(1, 784))

print("Predicted label:", np.argmax(prediction))

print("True label:", np.argmax(y\_test[example\_index]))

**Learning Outcomes:**

1. The ANN trained with backpropagation achieves a certain accuracy on the MNIST test data.

2. The confusion matrix visualizes the performance of the model across different classes.

3. Predictions for a sample input provide insight into model's confidence levels for each class.

**Result:**

**Accuracy: 87.37% for 20 epochs**

More epoch, better the model is trained on dataset and higher is the accuracy.

**Output:**



**EXPERIMENT-19**

**Aim:** Image Classification on Fashion MNIST Dataset using Artificial Neural Network (ANN)

**Theory:**

**Fashion MNIST** is a dataset of grayscale images of 10 different categories of clothing items (like shirts, shoes, and coats). The dataset includes **60,000** training images and 10,000 test images, each of **28x28 pixels.**

Each image is labeled with one of the **10 classes**. Fashion MNIST is often used as a drop-in replacement for MNIST to test image classification models in a slightly more complex setting.

**Artificial Neural Networks (ANN)** are a powerful model type for image classification. They consist of layers of nodes (neurons) where each neuron takes input data, applies a transformation, and passes the output to the next layer.

Using backpropagation, ANN adjusts weights through training to improve accuracy. In image classification, ANN learns patterns across pixel values to classify images accurately.

**Code:**

import numpy as np

import tensorflow as tf

from tensorflow.keras.datasets import fashion\_mnist

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten

from tensorflow.keras.utils import to\_categorical

import matplotlib.pyplot as plt

# Load the Fashion MNIST dataset

(X\_train, y\_train), (X\_test, y\_test) = fashion\_mnist.load\_data()

# Data Preprocessing

X\_train = X\_train / 255.0 # Scale pixel values to [0, 1]

X\_test = X\_test / 255.0

y\_train = to\_categorical(y\_train, 10) # One-hot encode labels

y\_test = to\_categorical(y\_test, 10)

# Building the Model

model = Sequential([

Flatten(input\_shape=(28, 28)), # Flatten 28x28 images to 784-element vectors

Dense(128, activation='relu'), # Hidden layer with 128 neurons

Dense(10, activation='softmax') # Output layer for 10 classes

])

# Compile the Model

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

# Train the Model

history = model.fit(X\_train, y\_train, epochs=10, batch\_size=32, validation\_split=0.2)

# Evaluate the Model on Test Data

test\_loss, test\_accuracy = model.evaluate(X\_test, y\_test)

print(f'Test Accuracy: {test\_accuracy:.4f}')

# Plot Training and Validation Accuracy

plt.plot(history.history['accuracy'], label='Training Accuracy')

plt.plot(history.history['val\_accuracy'], label='Validation Accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.title('Training and Validation Accuracy Over Epochs')

plt.show()

# Predict a sample from test data

sample\_index = 0

sample\_image = X\_test[sample\_index].reshape(1, 28, 28) # Reshape for prediction

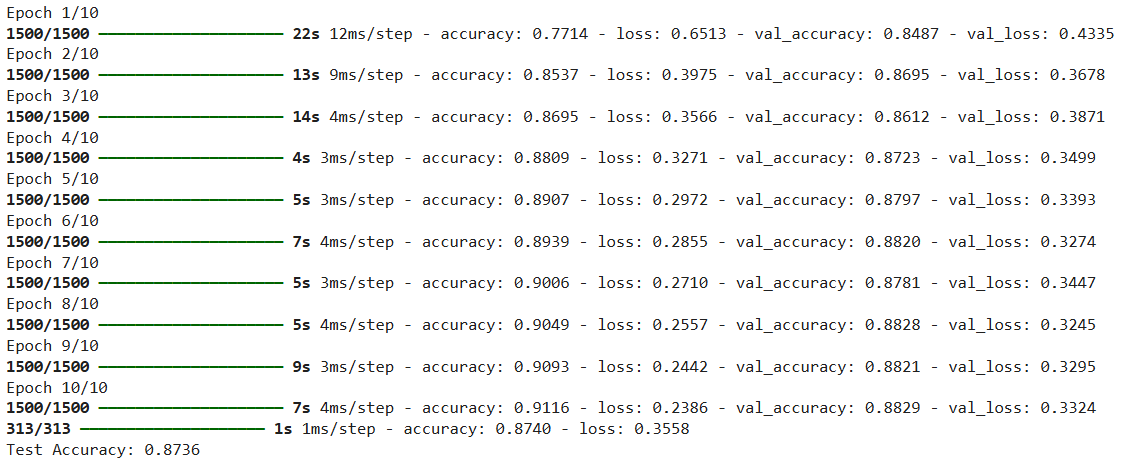
sample\_label = np.argmax(model.predict(sample\_image), axis=1)

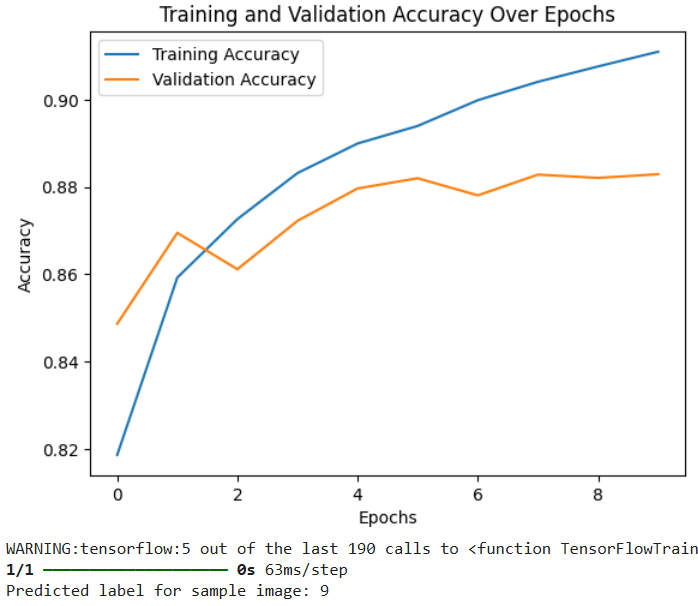
print("Predicted label for sample image:", sample\_label[0])

**Learning Outcomes:**

1. **Data Normalization**: Scaling image pixel values between 0 and 1 improves ANN performance.
2. **Simple Architecture Works**: Even a basic ANN can effectively classify Fashion MNIST images.
3. **Validation Helps**: Monitoring validation accuracy during training detects overfitting early.

**Output:**

**  
Accuracy: 87.36% for 10 epochs**

****

**EXPERIMENT-20**

**Aim:** Build an Artificial Neural Network (ANN) on MNIST Dataset.

**Theory:**

The MNIST dataset contains 60,000 training and 10,000 test images of handwritten digits (0-9), each of size 28x28 pixels.

An Artificial Neural Network (ANN) is a deep learning model inspired by the human brain, consisting of layers of interconnected neurons.

Key components of ANN : input layer, hidden layers, output layer, activation functions, optimizer, loss functions

In the case of image classification, an ANN learns to identify patterns in the pixel values of images to classify them into different categories.

**Code:**

import numpy as np

import tensorflow as tf

from tensorflow.keras.datasets import mnist

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten

from tensorflow.keras.utils import to\_categorical

import matplotlib.pyplot as plt

# Load MNIST dataset

(X\_train, y\_train), (X\_test, y\_test) = mnist.load\_data()

# Data Preprocessing

X\_train = X\_train / 255.0 # Normalize pixel values to [0, 1]

X\_test = X\_test / 255.0

y\_train = to\_categorical(y\_train, 10) # One-hot encode labels

y\_test = to\_categorical(y\_test, 10)

# Build the model

model = Sequential([

Flatten(input\_shape=(28, 28)), # Flatten the 28x28 images into 784 values

Dense(128, activation='relu'), # Hidden layer with 128 neurons

Dense(10, activation='softmax') # Output layer for 10 classes

])

# Compile the model

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

# Train the model

history = model.fit(X\_train, y\_train, epochs=10, batch\_size=32, validation\_split=0.2)

# Evaluate the model on the test dataset

test\_loss, test\_acc = model.evaluate(X\_test, y\_test)

print(f'Test Accuracy: {test\_acc:.4f}')

# Plot the training and validation accuracy

plt.plot(history.history['accuracy'], label='Training Accuracy')

plt.plot(history.history['val\_accuracy'], label='Validation Accuracy')

plt.xlabel('Epochs')

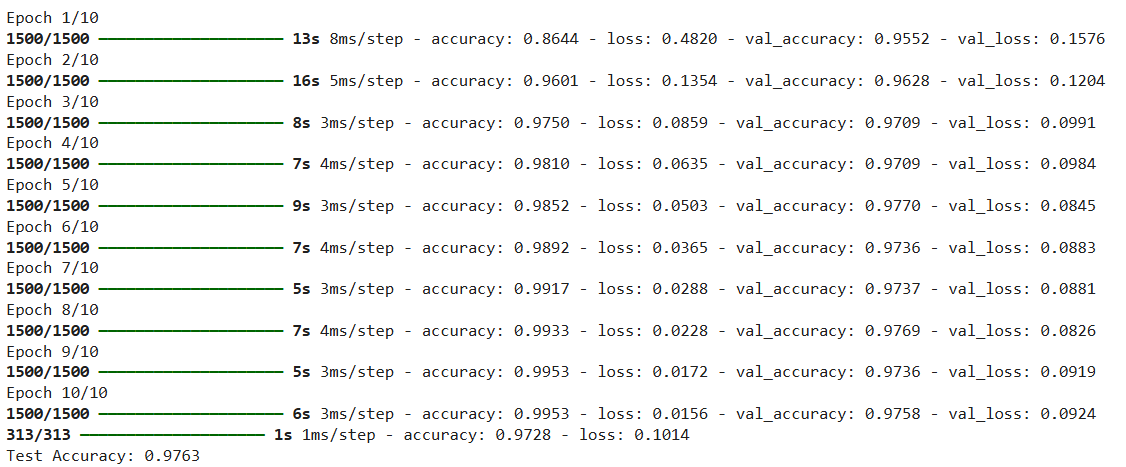
plt.ylabel('Accuracy')

plt.legend()

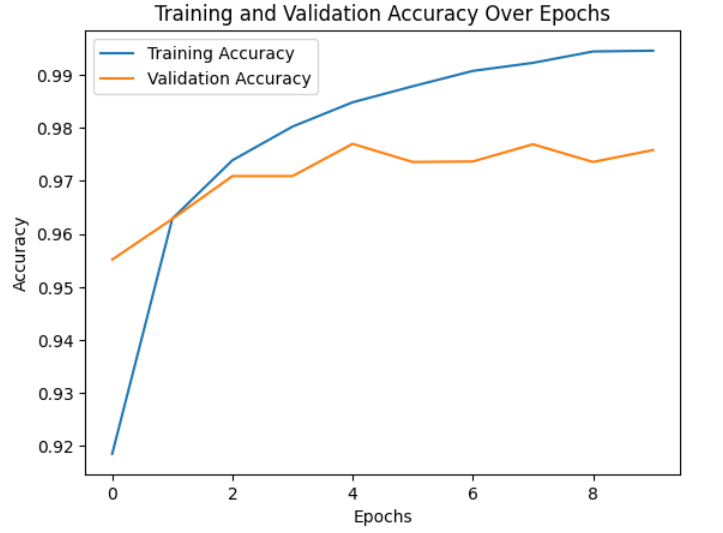
plt.title('Training and Validation Accuracy Over Epochs')

plt.show()

**Output:**

****

**Accuracy: 97.63 % for 10 epochs**

****

**Learning Outcomes:**

1. **Data Preprocessing**: Normalizing image pixel values improves the ANN's ability to learn effectively.
2. **Activation Functions**: Using functions like ReLU and softmax helps the model learn complex patterns and output probabilities for classification.
3. **Model Evaluation**: Tracking accuracy and loss during training helps ensure the model is learning correctly and prevents overfitting.

**EXPERIMENT-13**

**Aim**: To implement Data classification using K-Means.

**Theory:**  
K-Means is an unsupervised machine learning algorithm commonly used for clustering tasks. It groups data points into clusters based on their similarity, aiming to minimize the sum of squared distances between data points and their respective cluster centroids.

**Key Concepts:**

1. **Objective:** The main objective of K-Means is to partition data into clusters so that points within the same cluster are more similar to each other than to points in other clusters.
2. **K Centroids:** The algorithm requires specifying the number of clusters (K) in advance. It initializes K centroids randomly, then assigns each data point to the nearest centroid.
3. **Iterations:** K-Means operates iteratively, calculating the distance of each data point from the centroids, reassigning points to the nearest centroid, and recalculating centroids until convergence.
4. **Advantages:** Simple, fast, and efficient for clustering large datasets. Effectively discovers clusters with spherical shapes.
5. **Disadvantages:** Sensitive to the initial placement of centroids. May not perform well on data with non-spherical clusters.

**Code:**

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

from sklearn.cluster import KMeans

from sklearn.preprocessing import StandardScaler

# Load the Iris dataset

iris = load\_iris()

X = iris.data  # Features (sepal length, sepal width, petal length, petal width)

y = iris.target  # Actual species (used for comparison later)

# Feature scaling for better K-Means convergence

scaler = StandardScaler()

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

# clustering with 3 clusters (since we know there are 3 species)

kmeans = KMeans(n\_clusters=3, random\_state=42)

y\_pred = kmeans.fit\_predict(X\_scaled)

# Visualize the clusters (using the first two features for simplicity)

plt.figure(figsize=(10, 6))

# Plot each cluster with a unique color and label

for cluster\_label in np.unique(y\_pred):

    plt.scatter(

        X\_scaled[y\_pred == cluster\_label, 0],

        X\_scaled[y\_pred == cluster\_label, 1],

        label=f"Cluster {cluster\_label}",

        s=50

    )

plt.scatter(

    kmeans.cluster\_centers\_[:, 0],

    kmeans.cluster\_centers\_[:, 1],

    c='red', marker='X', s=200, label="Centroids"

)

plt.xlabel('Scaled Sepal Length')

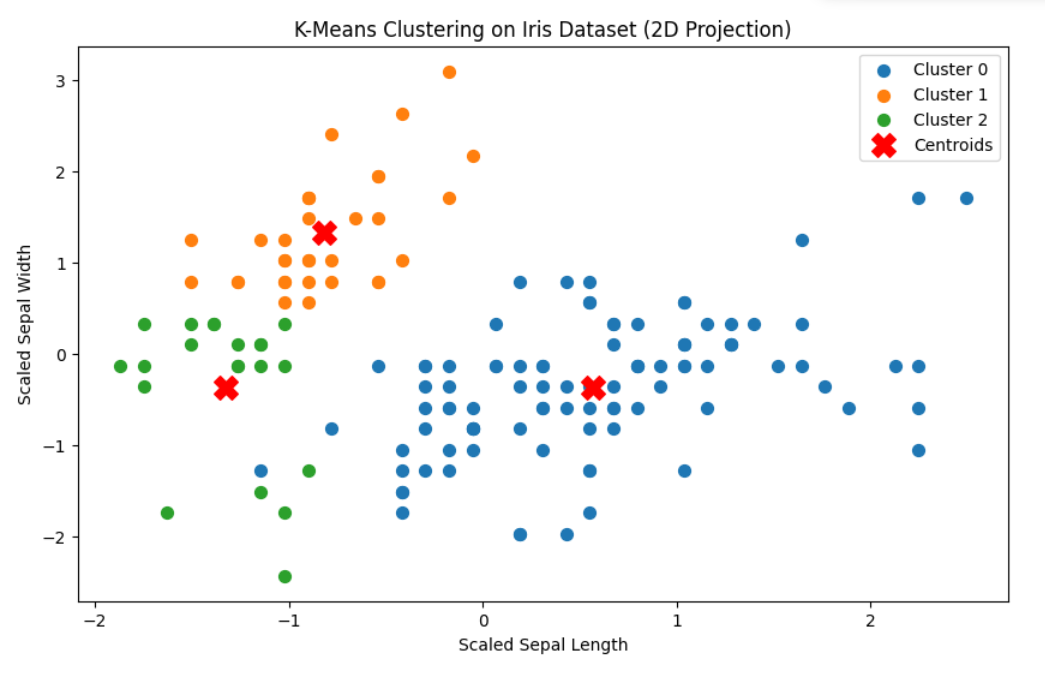
plt.ylabel('Scaled Sepal Width')

plt.title('K-Means Clustering on Iris Dataset (2D Projection)')

plt.legend()

plt.show()

**Output:**



**Learning Outcomes:**

1. Gained an understanding of K-Means clustering, including its iterative process to achieve cluster convergence.
2. Recognized the simplicity and efficiency of K-Means for clustering tasks, especially in large datasets.
3. Identified the limitations of K-Means, such as sensitivity to centroid initialization and challenges with non-spherical clusters.