

DELHI TECHNOLOGICAL UNIVERSITY

DEPARTMENT OF **COMPUTER SCIENCE ENGINEERING**



CO327

Machine Learning

SUBMITTED TO:

Dr. Kavinder Singh

SUBMITTED BY:

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 ("\nSubtracting 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:

```

➦ Original Array: [1 2 3 4 5]
  Array Squared: [ 1  4  9 16 25]

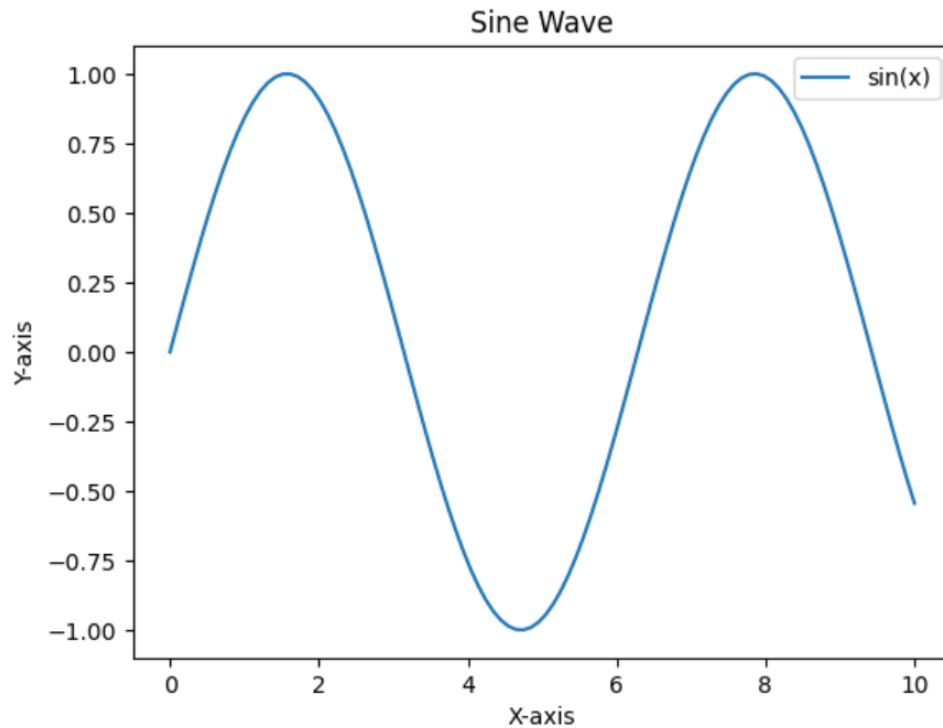
DataFrame:
   Name  Age  City
0  Jasprit  24  Delhi
1   Rahul  27  Mumbai
2   Sachin  22  Hyderabad
3   Dravid  32  Lucknow
4   Ashwin  29  Bangalore

People aged over 25:
   Name  Age  City
1   Rahul  27  Mumbai
3  Dravid  32  Lucknow
4  Ashwin  29  Bangalore

```

```
Adding 1 to every element:  $\begin{bmatrix} 2 & 3 \\ 4 & 5 \end{bmatrix}$ 
```

```
Subtracting 2 from each element:  $\begin{bmatrix} 2 & 1 \\ 0 & -1 \end{bmatrix}$ 
```



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()

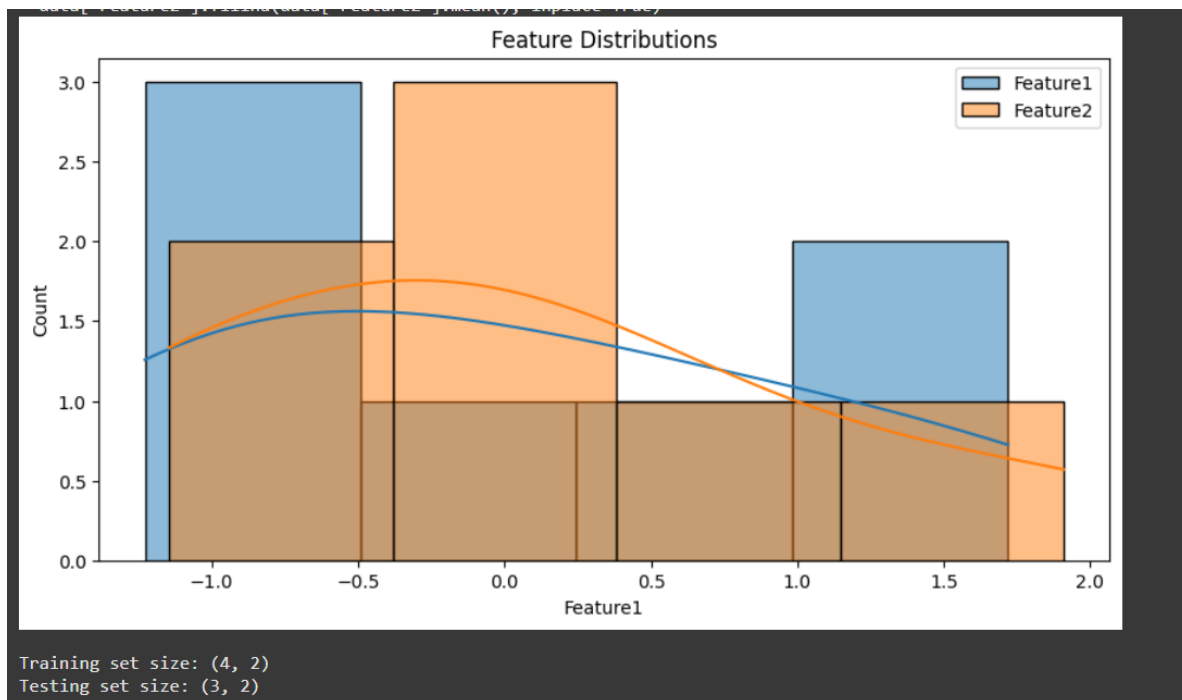
# Splitting data into training and testing sets
X = data[['Feature1', 'Feature2']]
y = data['Category']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

print("\nTraining set size:", X_train.shape)
print("Testing set size:", X_test.shape)

```

Output:

Original Data:			
	Feature1	Feature2	Category
0	5.1	3.5	A
1	4.9	3.0	B
2	NaN	3.2	A
3	5.0	NaN	A
4	5.5	3.8	B
5	6.1	3.3	B
6	5.8	3.0	A



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 $J(\theta)$ Linear Regression is typically Mean Squared Error (MSE), given by:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^m (y_i - \hat{y}_i)^2$$

where:

- y_i is the actual value,
- \hat{y}_i is the predicted value,
- m is the number of training examples.

The Gradient Descent update rule for weights θ is:

$$\theta = \theta - \alpha \frac{\partial J(\theta)}{\partial \theta}$$

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)

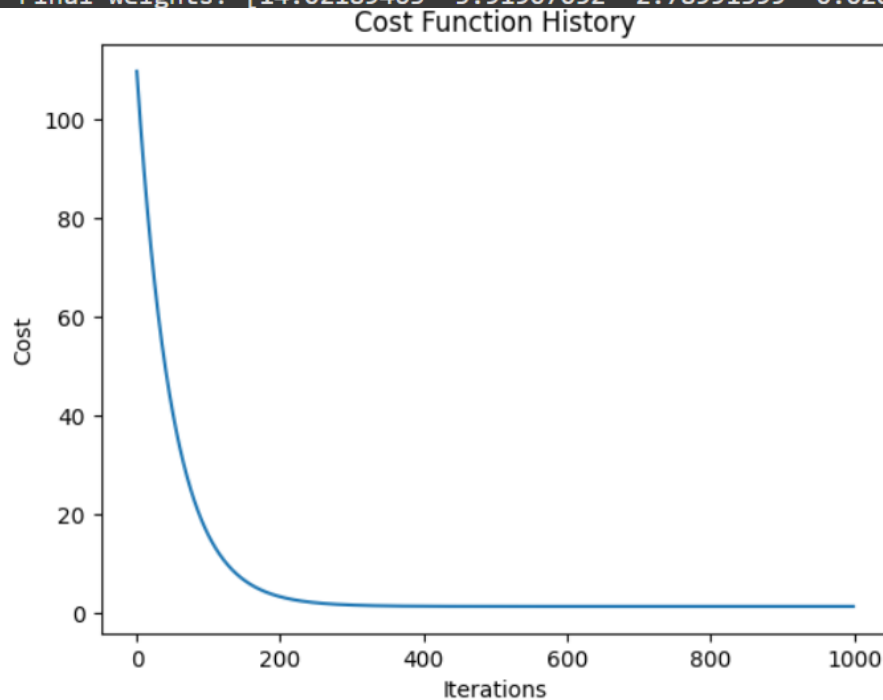
```

```
# 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:

```
Iteration 0: Cost 109.63343927854163
Iteration 100: Cost 15.747631594807482
Iteration 200: Cost 3.345723145370064
Iteration 300: Cost 1.6662958353031194
Iteration 400: Cost 1.4324485751872675
Iteration 500: Cost 1.3984804134285793
Iteration 600: Cost 1.3931982197393729
Iteration 700: Cost 1.39229045355476
Iteration 800: Cost 1.3921141730666131
Iteration 900: Cost 1.3920756158051268

Final weights: [14.02189463  3.91907052  2.78991399 -0.02034046]
```



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:

$$\text{MSE} = \frac{1}{m} \sum_{i=1}^m (y_i - \hat{y}_i)^2$$

where:

- y_i is the actual value,
- \hat{y}_i is 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}")

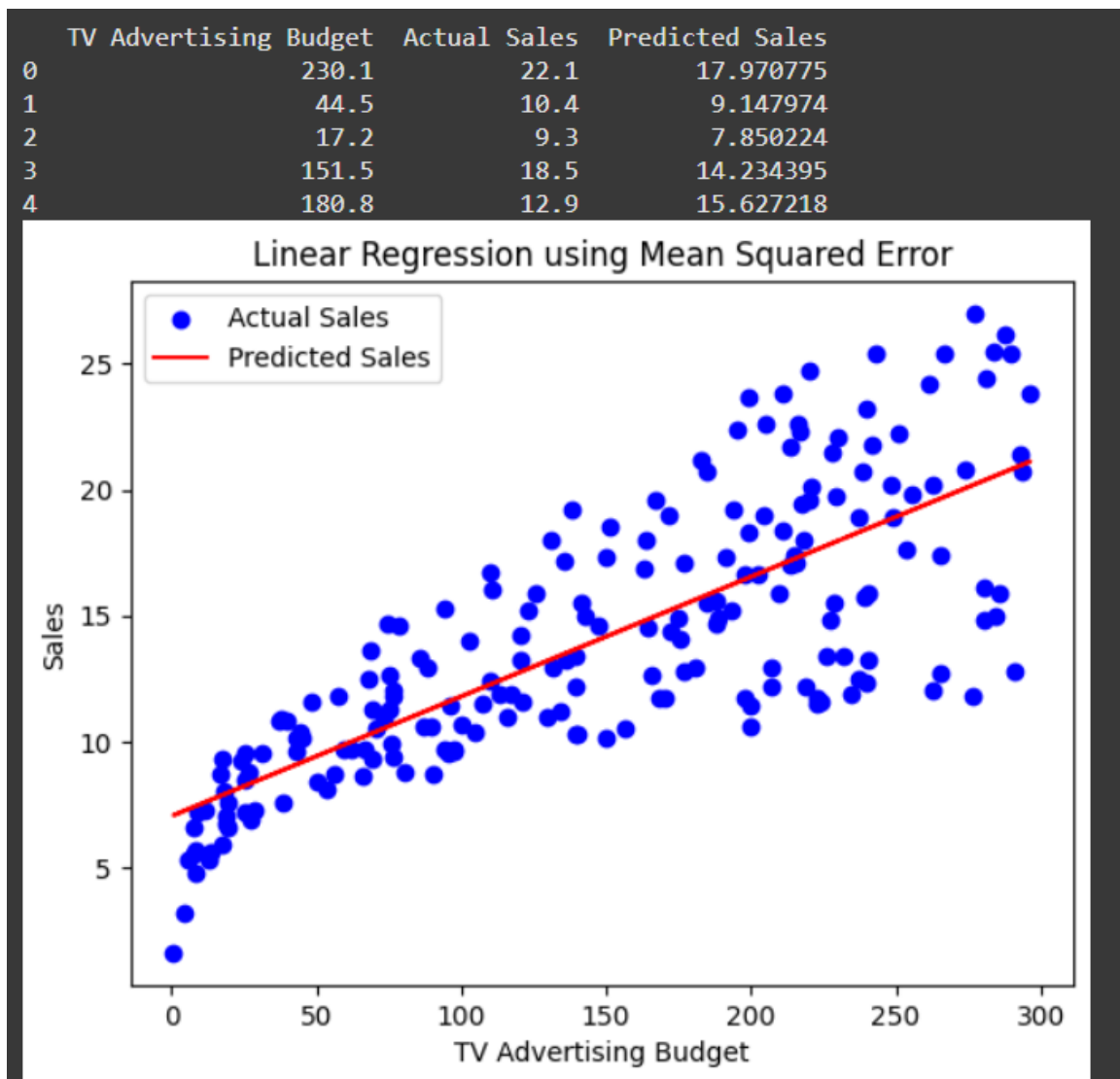
```

Output:

```

Final theta0: 7.0325935491276965
Final theta1: 0.047536640433019736
Mean Squared Error: 10.512652915656757

```



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)

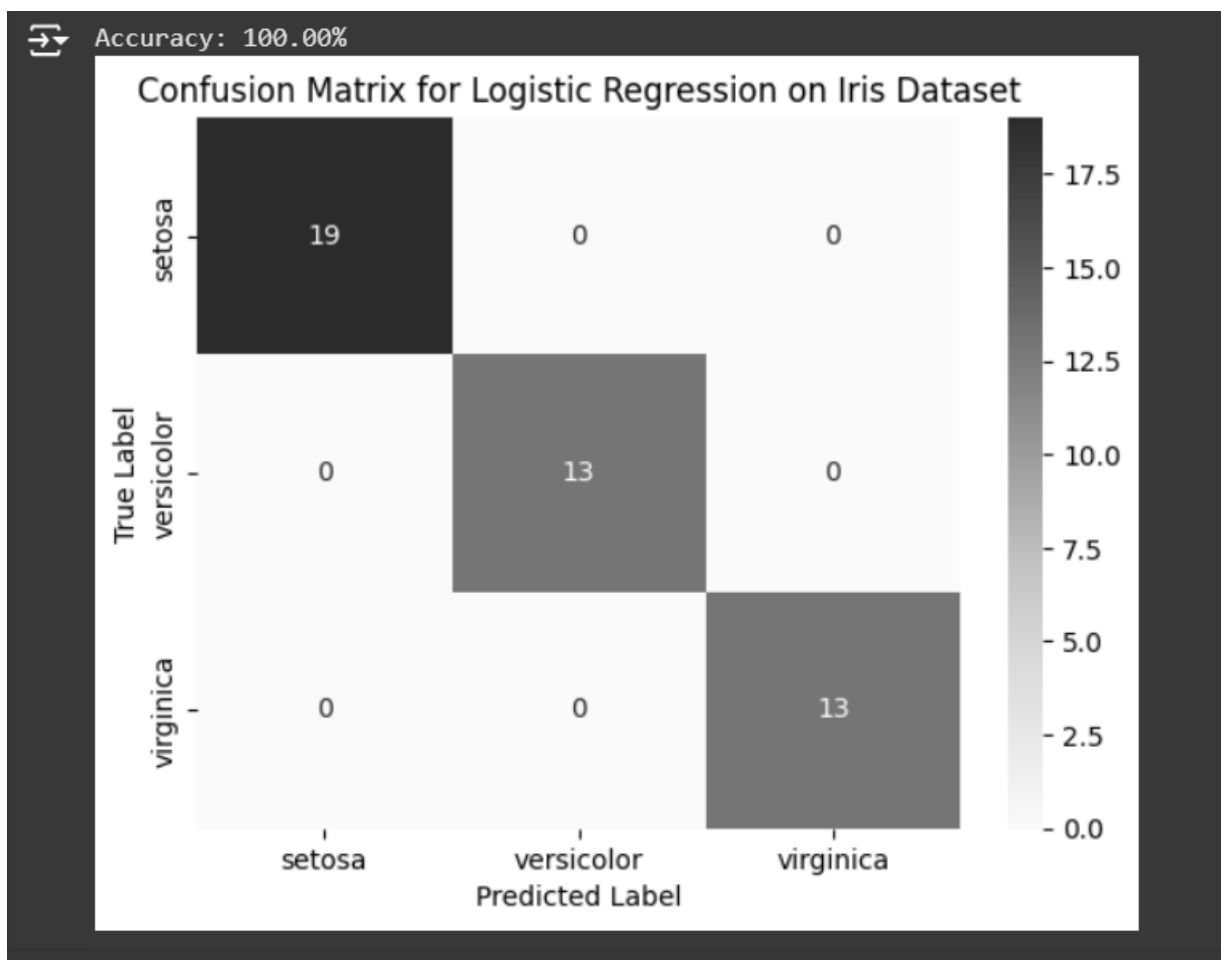
# Printing accuracy
```



```
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)

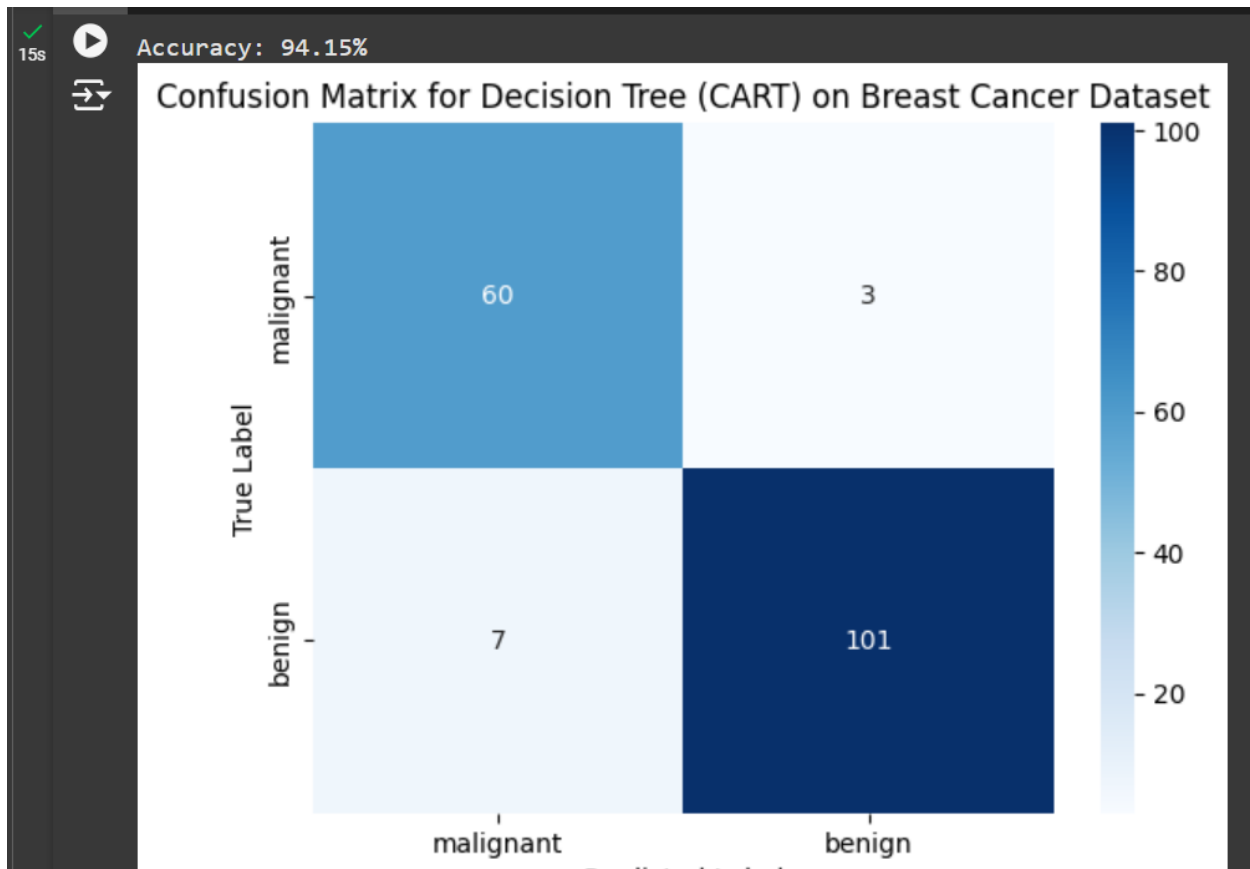
# Evaluating model performance
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)

# 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:

```
Best feature to split on: 'worst concave points' with gain 0.33171878226658635
Splitting on feature 'worst concave points' at threshold 0.1674052222222222

Evaluating information gain for each feature...

Best feature to split on: 'worst texture' with gain 0.3466907084599341
Splitting on feature 'worst texture' at threshold 25.287021276595745

Evaluating information gain for each feature...

Best feature to split on: 'area error' with gain 0.22665232895655918
Splitting on feature 'area error' at threshold 36.854695652173916

Evaluating information gain for each feature...

Best feature to split on: 'texture error' with gain 0.2826876858831789
Splitting on feature 'texture error' at threshold 1.2963500000000001
Prediction completed.

Test accuracy: 92.98%
```

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.

- **Precision:** The ratio of correct positive predictions to the total predicted positives.

$$\text{Precision} = \frac{TP}{TP + FP}$$

- **Recall (or Sensitivity):** The ratio of correct positive predictions to all actual positives.

$$\text{Recall} = \frac{TP}{TP + FN}$$

- **F1-Score:** The harmonic mean of precision and recall, giving a balanced measure even if the class distribution is imbalanced.

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

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:

```
Accuracy: 0.96
Confusion Matrix:
[[ 59   4]
 [  2 106]]
Classification Report:
              precision    recall  f1-score   support

     0       0.97       0.94       0.95         63
     1       0.96       0.98       0.97        108
```

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:

```
KNN implemented for k = 5

Test accuracy: 95.91%
Confusion Matrix:
[[ 57   6]
 [  1 107]]
Classification Report:
              precision    recall  f1-score   support

     0           0.98       0.90       0.94         63
     1           0.95       0.99       0.97        108

 accuracy          0.96
 macro avg         0.96
 weighted avg      0.96
```

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

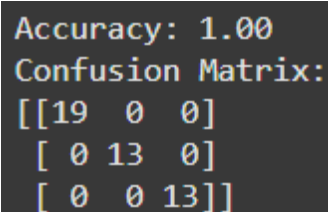
# Split the data into training and testing sets (70% train, 30% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

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:



```
Accuracy: 1.00
Confusion Matrix:
[[19  0  0]
 [ 0 13  0]
 [ 0  0 13]]
```

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.

Bayes' Theorem:

$$P(A|B) = \frac{P(B|A) \times P(A)}{P(B)}$$

In a classification context:

- $P(A|B)$ is the **posterior probability** of class A given predictor B .
- $P(B|A)$ is the **likelihood**, the probability of predictor B given class A .
- $P(A)$ is the **prior probability** of class A .
- $P(B)$ is the **probability of the predictor B** .

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:

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

     0           1.00       1.00       1.00        10
     1           1.00       1.00       1.00         9
     2           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
```

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:

```
Accuracy: 1.0
sepal length (cm): 0.11
sepal width (cm): 0.03
petal length (cm): 0.44
petal width (cm): 0.42
```

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.

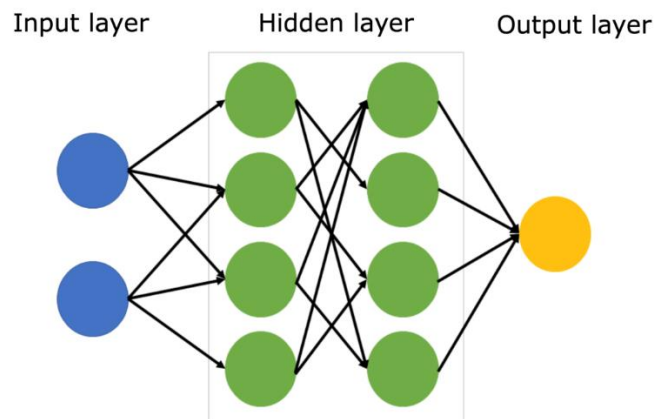
EXPERIMENT-14

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
```

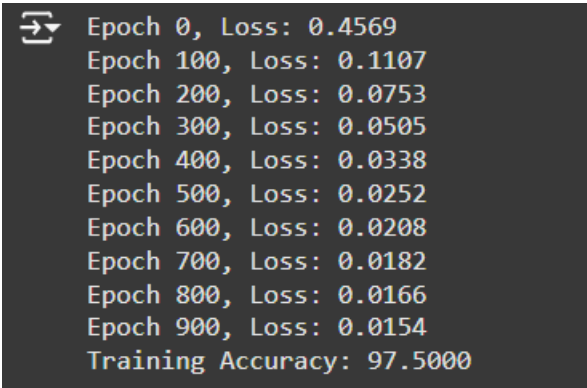
```
if epoch % 100 == 0:
```

```
    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:



```
Epoch 0, Loss: 0.4569
Epoch 100, Loss: 0.1107
Epoch 200, Loss: 0.0753
Epoch 300, Loss: 0.0505
Epoch 400, Loss: 0.0338
Epoch 500, Loss: 0.0252
Epoch 600, Loss: 0.0208
Epoch 700, Loss: 0.0182
Epoch 800, Loss: 0.0166
Epoch 900, Loss: 0.0154
Training Accuracy: 97.5000
```

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.

$$\text{Precision} = \frac{TP}{TP + FP}$$

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.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

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.

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

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.

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

where y_i is the true value, and \hat{y}_i is the predicted value.

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)

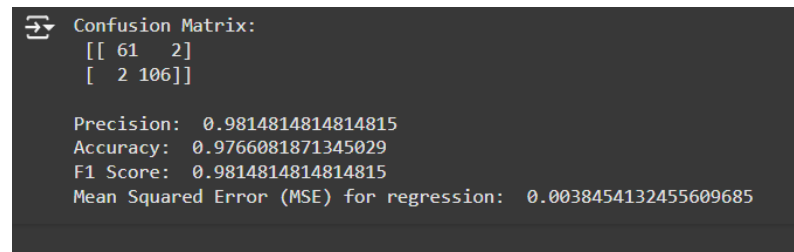
print("\nPrecision: ", precision)

print("Accuracy: ", accuracy)

print("F1 Score: ", f1)

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

Output:

A terminal window with a dark background and light-colored text. It displays the output of the Python code, including the Confusion Matrix, Precision, Accuracy, F1 Score, and Mean Squared Error (MSE) for regression.

```
↵ Confusion Matrix:
[[ 61   2]
 [  2 106]]

Precision:  0.9814814814814815
Accuracy:   0.9766081871345029
F1 Score:   0.9814814814814815
Mean Squared Error (MSE) for regression:  0.0038454132455609685
```

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,)), # 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()

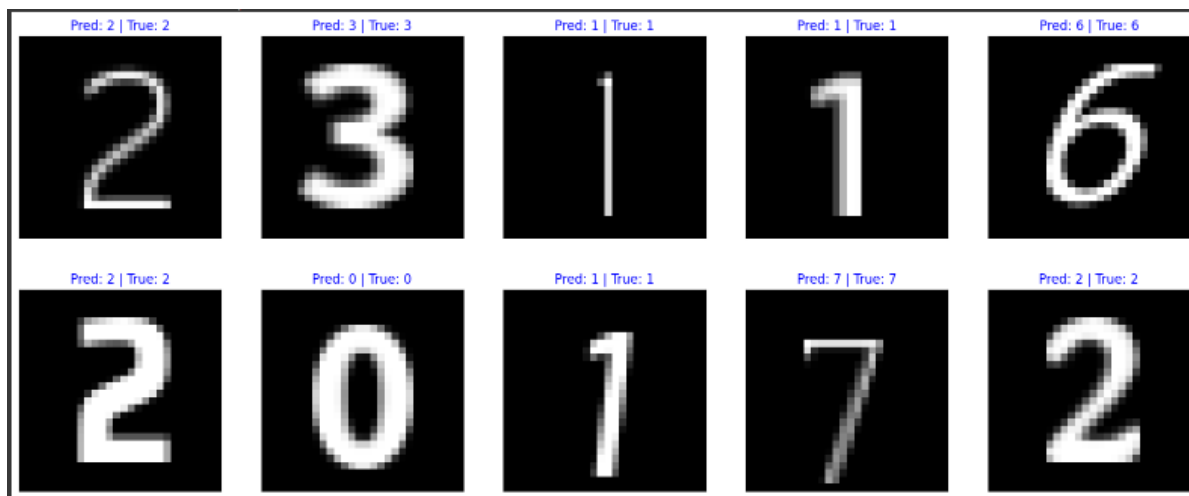
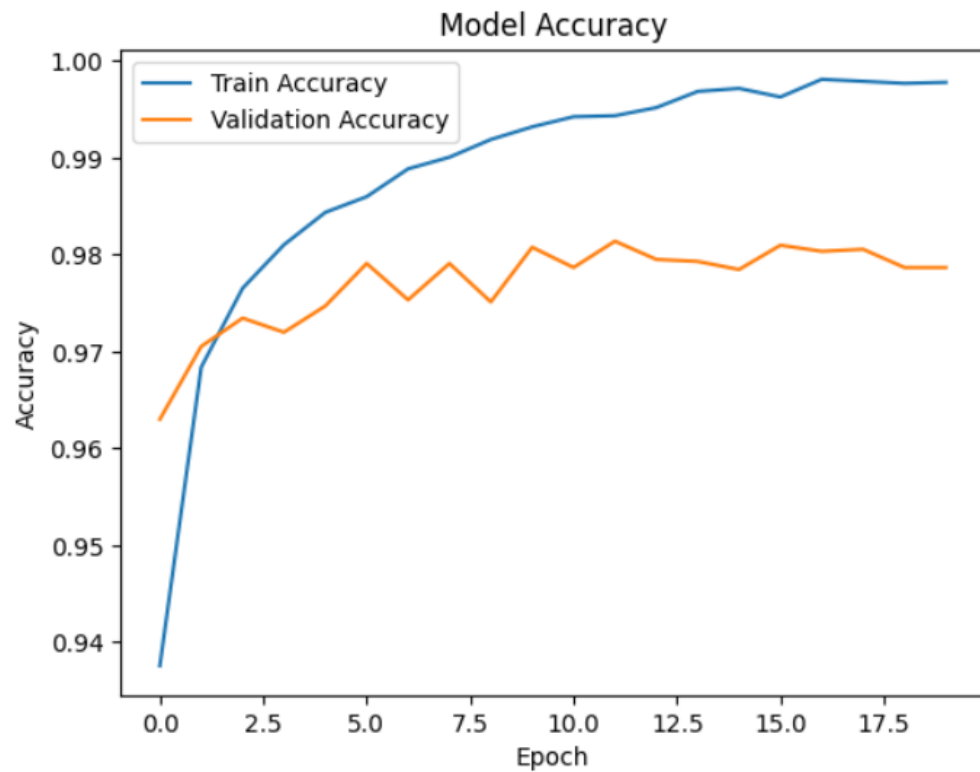
# Plot a random sample of test images with predicted and true labels
y_pred = model.predict(X_test)
figure = plt.figure(figsize=(20, 8))
for i, index in enumerate(np.random.choice(X_test.shape[0], size=10, replace=False)):
    ax = figure.add_subplot(2, 5, i + 1, xticks=[], yticks=[])
    ax.imshow(X_test.iloc[index].values.reshape(28, 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:

```
Shape of the dataset: (29900, 786)
Number of unique labels: 10
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/'i
    super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/20
598/598 ————— 6s 7ms/step - accuracy: 0.8794 - loss: 0.4991 - val_accuracy: 0.9630 - val_loss: 0.1343
Epoch 2/20
598/598 ————— 3s 3ms/step - accuracy: 0.9680 - loss: 0.1220 - val_accuracy: 0.9705 - val_loss: 0.1063
Epoch 3/20
598/598 ————— 3s 5ms/step - accuracy: 0.9773 - loss: 0.0911 - val_accuracy: 0.9735 - val_loss: 0.0966
Epoch 4/20
598/598 ————— 4s 3ms/step - accuracy: 0.9803 - loss: 0.0723 - val_accuracy: 0.9720 - val_loss: 0.1031
Epoch 5/20
598/598 ————— 2s 3ms/step - accuracy: 0.9843 - loss: 0.0579 - val_accuracy: 0.9747 - val_loss: 0.0946
Epoch 6/20
598/598 ————— 3s 3ms/step - accuracy: 0.9875 - loss: 0.0471 - val_accuracy: 0.9791 - val_loss: 0.0840
Epoch 7/20
598/598 ————— 2s 3ms/step - accuracy: 0.9881 - loss: 0.0447 - val_accuracy: 0.9753 - val_loss: 0.0955
Epoch 8/20
598/598 ————— 4s 6ms/step - accuracy: 0.9906 - loss: 0.0367 - val_accuracy: 0.9791 - val_loss: 0.0781
Epoch 9/20
598/598 ————— 2s 3ms/step - accuracy: 0.9923 - loss: 0.0318 - val_accuracy: 0.9751 - val_loss: 0.0937
Epoch 10/20
598/598 ————— 2s 3ms/step - accuracy: 0.9930 - loss: 0.0267 - val_accuracy: 0.9808 - val_loss: 0.0791
Epoch 11/20
598/598 ————— 3s 3ms/step - accuracy: 0.9950 - loss: 0.0206 - val_accuracy: 0.9787 - val_loss: 0.0796
Epoch 12/20
598/598 ————— 2s 3ms/step - accuracy: 0.9957 - loss: 0.0178 - val_accuracy: 0.9814 - val_loss: 0.0804
Epoch 13/20
598/598 ————— 3s 5ms/step - accuracy: 0.9966 - loss: 0.0155 - val_accuracy: 0.9795 - val_loss: 0.0815
Epoch 14/20
598/598 ————— 3s 5ms/step - accuracy: 0.9964 - loss: 0.0157 - val_accuracy: 0.9793 - val_loss: 0.0860
Epoch 15/20
598/598 ————— 4s 3ms/step - accuracy: 0.9972 - loss: 0.0116 - val_accuracy: 0.9785 - val_loss: 0.1004
Epoch 16/20
598/598 ————— 3s 3ms/step - accuracy: 0.9966 - loss: 0.0126 - val_accuracy: 0.9810 - val_loss: 0.0839
Epoch 17/20
598/598 ————— 3s 3ms/step - accuracy: 0.9983 - loss: 0.0091 - val_accuracy: 0.9804 - val_loss: 0.0874
Epoch 18/20
598/598 ————— 4s 6ms/step - accuracy: 0.9979 - loss: 0.0097 - val_accuracy: 0.9806 - val_loss: 0.0867
Epoch 19/20
598/598 ————— 2s 3ms/step - accuracy: 0.9982 - loss: 0.0092 - val_accuracy: 0.9787 - val_loss: 0.1020
Epoch 20/20
598/598 ————— 3s 3ms/step - accuracy: 0.9986 - loss: 0.0077 - val_accuracy: 0.9787 - val_loss: 0.0983
187/187 ————— 0s 1ms/step - accuracy: 0.9787 - loss: 0.1128
Test accuracy: 0.9797658920288086
```

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))

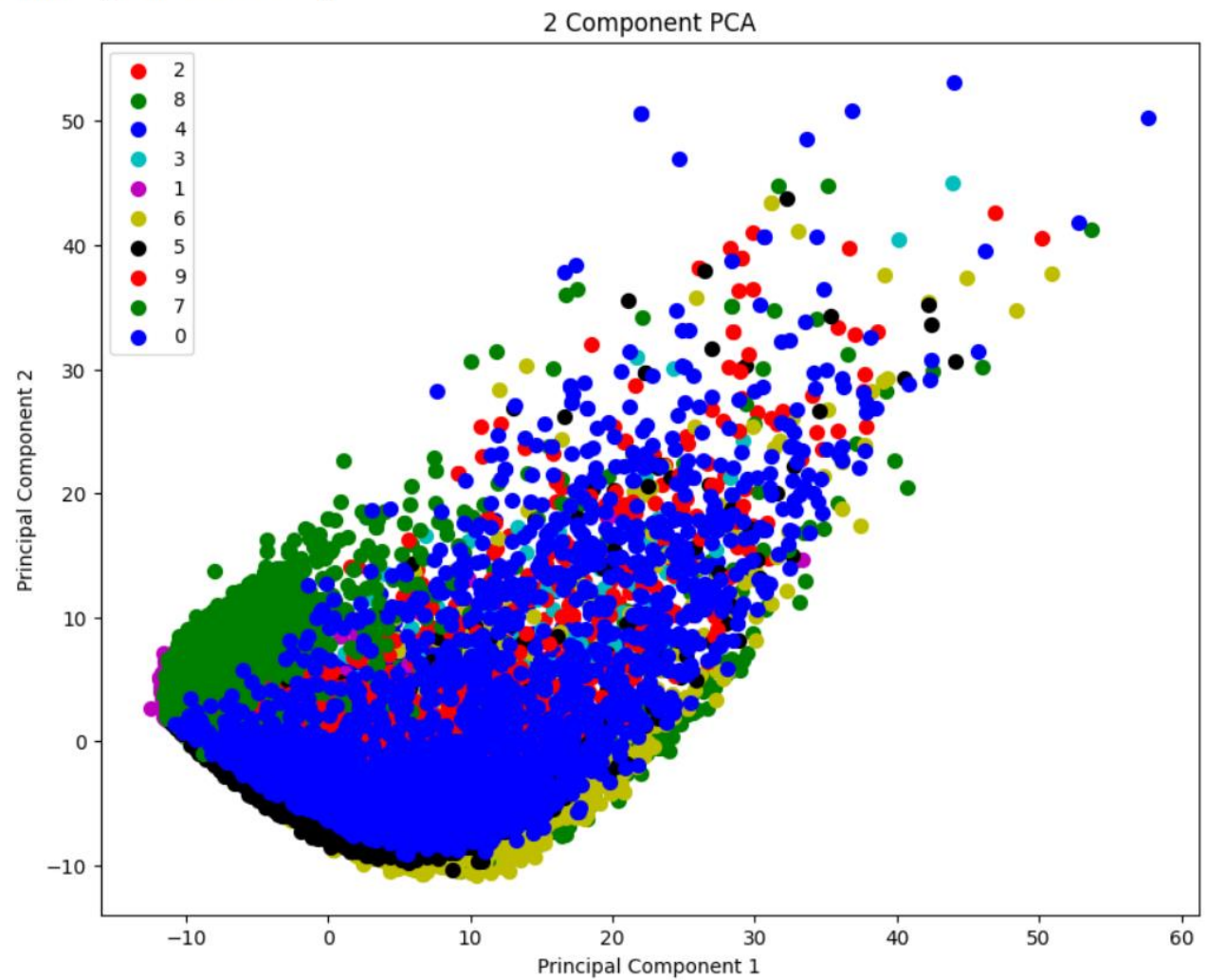
```

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:

Labels: [2 8 4 3 1 6 5 9 7 0]



Explained Variance Ratio: [0.10312951 0.05227091]

Total Variance Explained by 2 Components: 0.155400428492806

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:

```
Epoch 1/20
600/600 ————— 4s 3ms/step - accuracy: 0.2034 - loss: 2.2523
Epoch 2/20
600/600 ————— 2s 2ms/step - accuracy: 0.5131 - loss: 1.9400
Epoch 3/20
600/600 ————— 1s 2ms/step - accuracy: 0.6092 - loss: 1.6985
Epoch 4/20
600/600 ————— 1s 2ms/step - accuracy: 0.6771 - loss: 1.4936
Epoch 5/20
600/600 ————— 1s 2ms/step - accuracy: 0.7190 - loss: 1.3370
Epoch 6/20
600/600 ————— 1s 2ms/step - accuracy: 0.7593 - loss: 1.1986
Epoch 7/20
600/600 ————— 1s 2ms/step - accuracy: 0.7840 - loss: 1.0891
Epoch 8/20
600/600 ————— 2s 3ms/step - accuracy: 0.7999 - loss: 1.0014
Epoch 9/20
600/600 ————— 2s 2ms/step - accuracy: 0.8143 - loss: 0.9251
Epoch 10/20
600/600 ————— 2s 2ms/step - accuracy: 0.8254 - loss: 0.8607
Epoch 11/20
600/600 ————— 1s 2ms/step - accuracy: 0.8333 - loss: 0.8067
Epoch 12/20
600/600 ————— 1s 2ms/step - accuracy: 0.8395 - loss: 0.7644
Epoch 13/20
600/600 ————— 1s 2ms/step - accuracy: 0.8462 - loss: 0.7245
Epoch 14/20
600/600 ————— 1s 2ms/step - accuracy: 0.8529 - loss: 0.6836
Epoch 15/20
600/600 ————— 1s 2ms/step - accuracy: 0.8555 - loss: 0.6553
Epoch 16/20
600/600 ————— 1s 2ms/step - accuracy: 0.8595 - loss: 0.6269
Epoch 17/20
600/600 ————— 2s 3ms/step - accuracy: 0.8609 - loss: 0.6108
Epoch 18/20
600/600 ————— 2s 2ms/step - accuracy: 0.8642 - loss: 0.5883
Epoch 19/20
600/600 ————— 1s 2ms/step - accuracy: 0.8639 - loss: 0.5759
Epoch 20/20
600/600 ————— 1s 2ms/step - accuracy: 0.8700 - loss: 0.5536
313/313 ————— 1s 1ms/step - accuracy: 0.8557 - loss: 0.5895
Test accuracy: 0.8737
1/1 ————— 0s 42ms/step
Predicted label: 6
True label: 6
```

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])

```

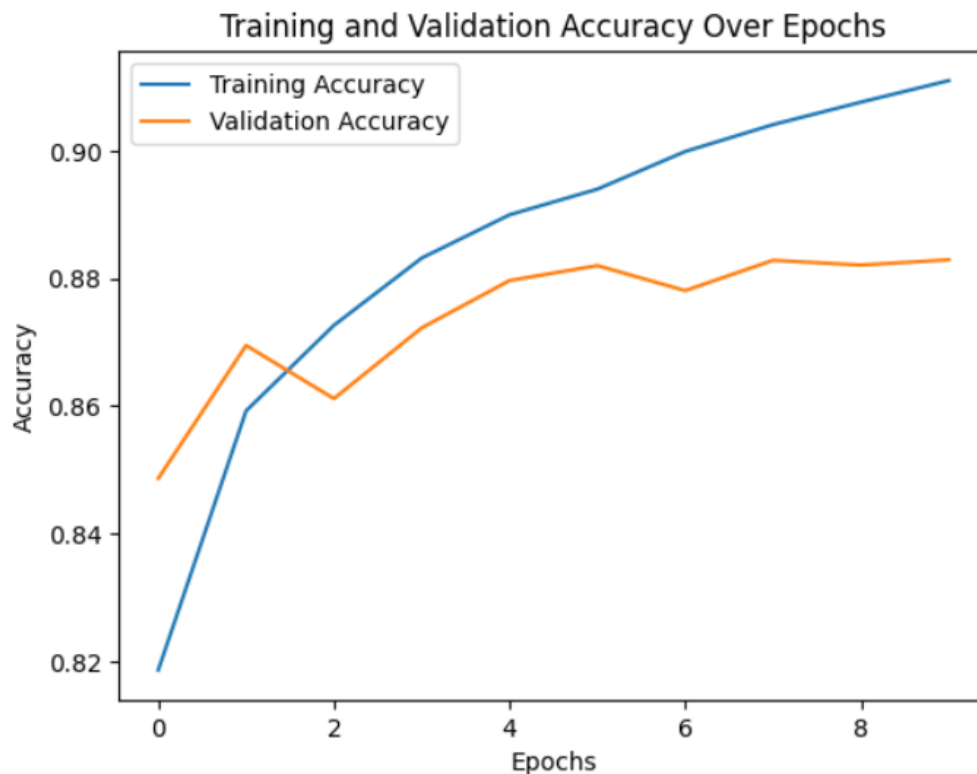
Learning Outcomes:

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:

```
Epoch 1/10
1500/1500 ————— 22s 12ms/step - accuracy: 0.7714 - loss: 0.6513 - val_accuracy: 0.8487 - val_loss: 0.4335
Epoch 2/10
1500/1500 ————— 13s 9ms/step - accuracy: 0.8537 - loss: 0.3975 - val_accuracy: 0.8695 - val_loss: 0.3678
Epoch 3/10
1500/1500 ————— 14s 4ms/step - accuracy: 0.8695 - loss: 0.3566 - val_accuracy: 0.8612 - val_loss: 0.3871
Epoch 4/10
1500/1500 ————— 4s 3ms/step - accuracy: 0.8809 - loss: 0.3271 - val_accuracy: 0.8723 - val_loss: 0.3499
Epoch 5/10
1500/1500 ————— 5s 3ms/step - accuracy: 0.8907 - loss: 0.2972 - val_accuracy: 0.8797 - val_loss: 0.3393
Epoch 6/10
1500/1500 ————— 7s 4ms/step - accuracy: 0.8939 - loss: 0.2855 - val_accuracy: 0.8820 - val_loss: 0.3274
Epoch 7/10
1500/1500 ————— 5s 3ms/step - accuracy: 0.9006 - loss: 0.2710 - val_accuracy: 0.8781 - val_loss: 0.3447
Epoch 8/10
1500/1500 ————— 5s 4ms/step - accuracy: 0.9049 - loss: 0.2557 - val_accuracy: 0.8828 - val_loss: 0.3245
Epoch 9/10
1500/1500 ————— 9s 3ms/step - accuracy: 0.9093 - loss: 0.2442 - val_accuracy: 0.8821 - val_loss: 0.3295
Epoch 10/10
1500/1500 ————— 7s 4ms/step - accuracy: 0.9116 - loss: 0.2386 - val_accuracy: 0.8829 - val_loss: 0.3324
313/313 ————— 1s 1ms/step - accuracy: 0.8740 - loss: 0.3558
Test Accuracy: 0.8736
```

Accuracy: 87.36% for 10 epochs



```
WARNING:tensorflow:5 out of the last 190 calls to <function TensorFlowTrain
1/1 ————— 0s 63ms/step
Predicted label for sample image: 9
```

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}')
```

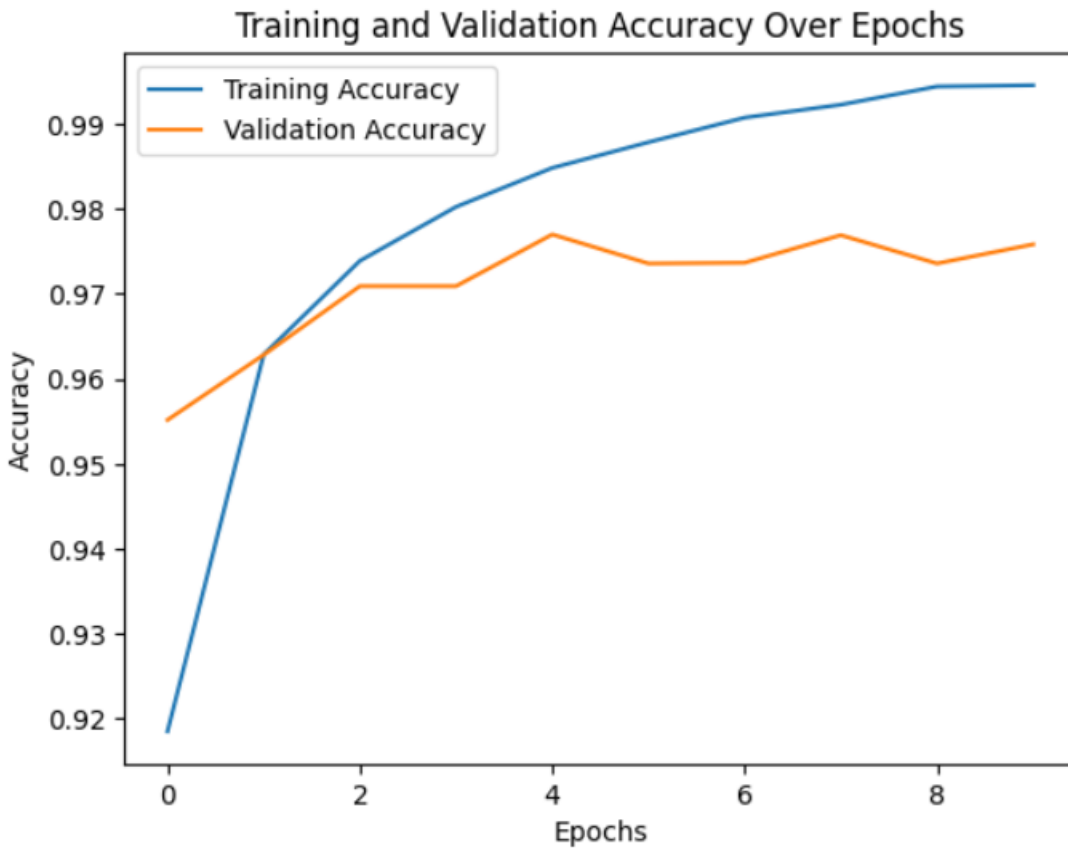
```
# Plot the training and validation accuracy
```

```
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:

```
Epoch 1/10  
1500/1500 ————— 13s 8ms/step - accuracy: 0.8644 - loss: 0.4820 - val_accuracy: 0.9552 - val_loss: 0.1576  
Epoch 2/10  
1500/1500 ————— 16s 5ms/step - accuracy: 0.9601 - loss: 0.1354 - val_accuracy: 0.9628 - val_loss: 0.1204  
Epoch 3/10  
1500/1500 ————— 8s 3ms/step - accuracy: 0.9750 - loss: 0.0859 - val_accuracy: 0.9709 - val_loss: 0.0991  
Epoch 4/10  
1500/1500 ————— 7s 4ms/step - accuracy: 0.9810 - loss: 0.0635 - val_accuracy: 0.9709 - val_loss: 0.0984  
Epoch 5/10  
1500/1500 ————— 9s 3ms/step - accuracy: 0.9852 - loss: 0.0503 - val_accuracy: 0.9770 - val_loss: 0.0845  
Epoch 6/10  
1500/1500 ————— 7s 4ms/step - accuracy: 0.9892 - loss: 0.0365 - val_accuracy: 0.9736 - val_loss: 0.0883  
Epoch 7/10  
1500/1500 ————— 5s 3ms/step - accuracy: 0.9917 - loss: 0.0288 - val_accuracy: 0.9737 - val_loss: 0.0881  
Epoch 8/10  
1500/1500 ————— 7s 4ms/step - accuracy: 0.9933 - loss: 0.0228 - val_accuracy: 0.9769 - val_loss: 0.0826  
Epoch 9/10  
1500/1500 ————— 5s 3ms/step - accuracy: 0.9953 - loss: 0.0172 - val_accuracy: 0.9736 - val_loss: 0.0919  
Epoch 10/10  
1500/1500 ————— 6s 3ms/step - accuracy: 0.9953 - loss: 0.0156 - val_accuracy: 0.9758 - val_loss: 0.0924  
313/313 ————— 1s 1ms/step - accuracy: 0.9728 - loss: 0.1014  
Test Accuracy: 0.9763
```

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)

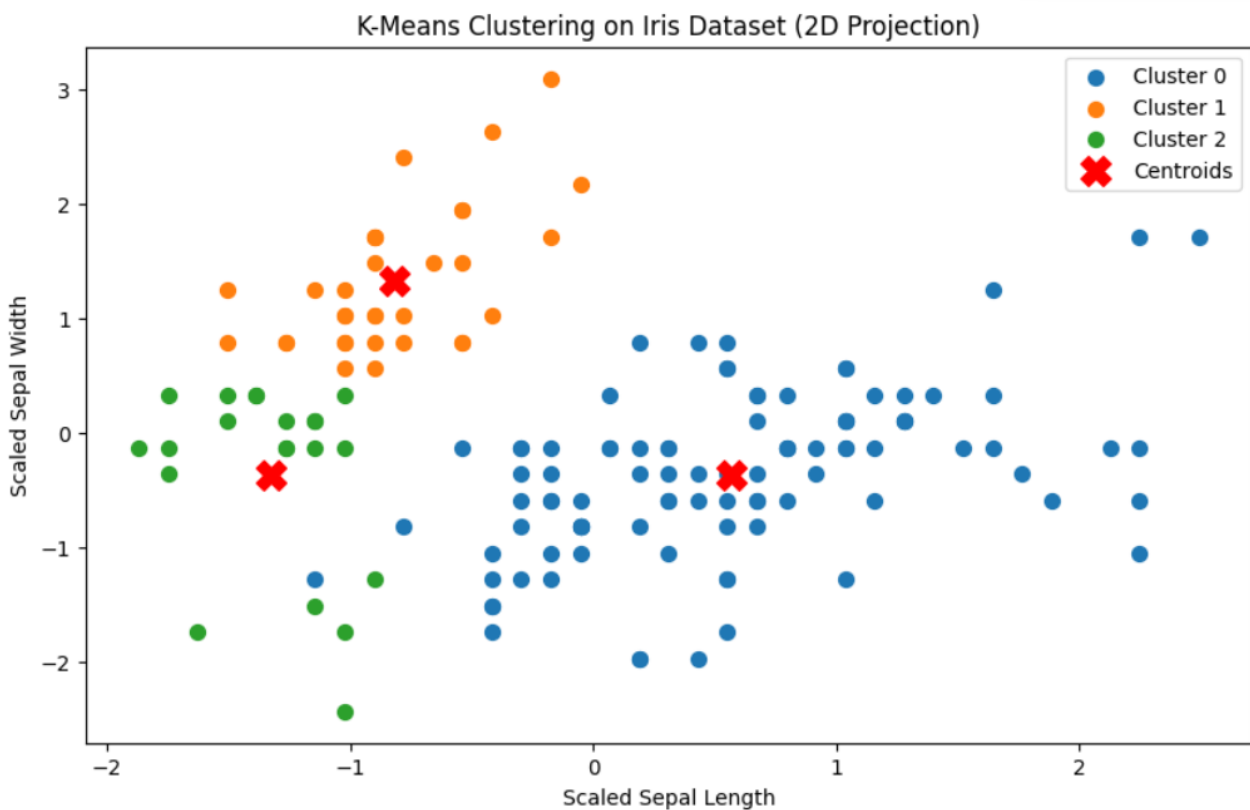
# Visualize the clusters (using the first two features for simplicity)
plt.figure(figsize=(10, 6))

# 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.