DY Patil International University

TC- 1

Fundamentals of Artificial Intelligence and Machine learning

Year - 3rd

Semester - 5th

Lab Manual

Index

9	**Clustering** a. Apply EM algorithm to cluster a set of data stored in a .CSV file. Use the same data set for clustering using the k-Means algorithm. b. Compare the results of these two algorithms and comment on the quality of clustering. You can add Java/Python ML library classes/API in the program	37.
10	**Deep Learning** To build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets.	43.

Lab – 1

Aim: - Optimization algorithm To find the parameters or coefficients of a function where the function has a minimum value using gradient Descent optimization algorithms

Theory :

Gradient Descent is a popular optimization algorithm used to find parameters or coefficients of a function, typically in the context of machine learning or optimization problems where the goal is to minimize a cost or loss function. Here's a theoretical overview of how Gradient Descent works:

1. Objective function: You start with an objective function, often referred to as $J(\theta)$, where θ represents the parameters or coefficients you want to optimize. The goal is to find values of θ that minimize $J(\theta)$.
2. Initial guess: Initialize θ with arbitrary values. This can be done randomly or with some initial values.
3. Compute gradient: Compute the gradient of the objective function with respect to θ. The gradient represents the direction and magnitude of the steepest increase in the objective function. It is calculated as:

$$\nabla J(\theta) = [\partial J(\theta)/\partial \theta_0, \partial J(\theta)/\partial \theta_1, ..., \partial J(\theta)/\partial \theta\_n]$$

Here n is the number of parameters and $\partial J(\theta)/\partial \theta\_i$ represents the partial derivative of $J(\theta)$ with respect to θ_i.

4. Update parameters: Update the θ parameters using transition information. This is done iteratively and can be expressed as:

$$\theta\_new = \theta\_old - \alpha * \nabla J(\theta\_old)$$

where α (alpha) is the learning rate,

5. Convergence criterion: Repeat steps 3 and 4 until the convergence criterion is met. This criterion can be a maximum number of iterations or a threshold for changing the objective function.

6. Result: Once the algorithm converges, the values of θ will represent the parameters or coefficients that minimize the objective function J(θ).

There are different variants of gradient descent, such as Stochastic Gradient Descent (SGD), Mini-Batch Gradient Descent, and others, which modify the way gradients are calculated or update the learning rate to improve convergence and computational efficiency.

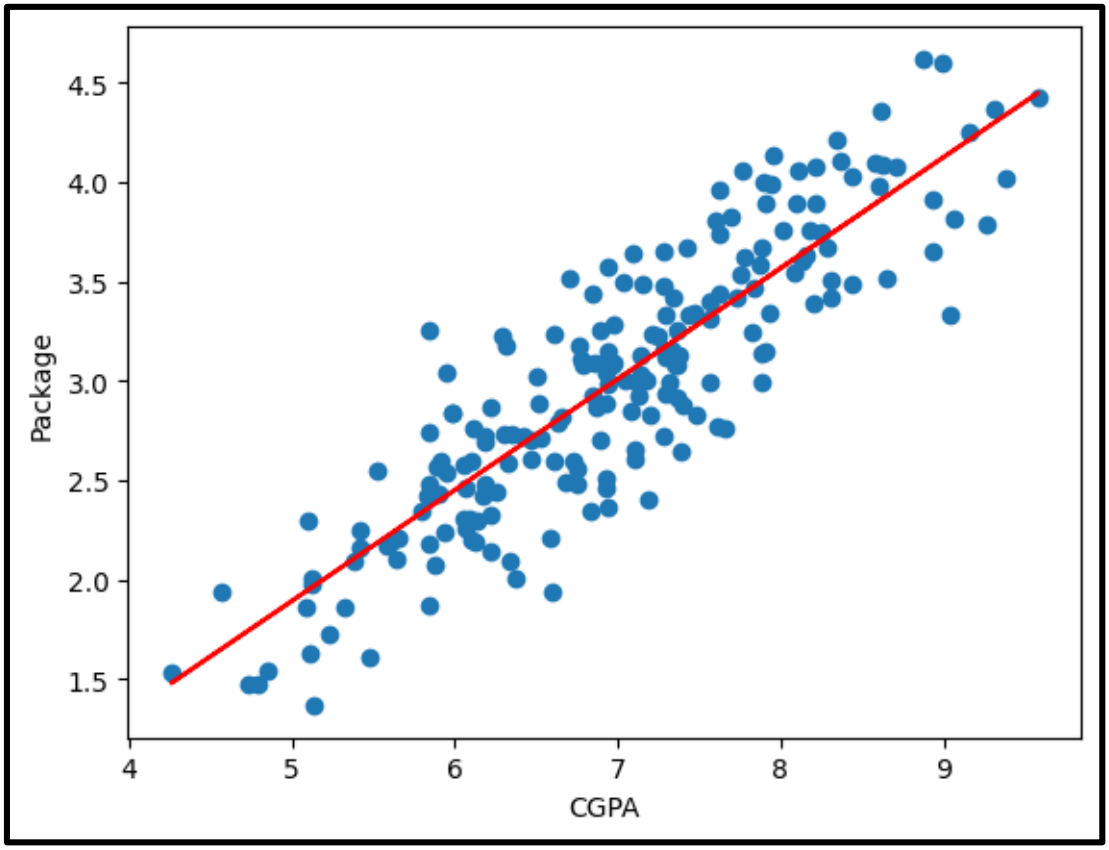
Code:-

```
import pandas as pd
import random
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import numpy as np
df = pd.read_csv("/content/placement.csv")
X = df.iloc[:,0].values
Y = df.iloc[:,1].values
class Linear:
  def __init__(self):
    self.m = None
    self.b = None
  def fit(self, X_train, Y_train):
    num = 0
    den = 0

    for i in range(X_train.shape[0]):
      num += ((X_train[i] - X_train.mean()) * (Y_train[i]-
Y_train.mean()))
      den += ((X_train[i] - X_train.mean()) * (X_train[i]-X_train.mean()))

      self.m = num/den
      self.b = Y_train.mean() - (self.m * X_train.mean())

    print("Slope is:", self.m)
    print("Intercept is:", self.b)

  def predict(self, X_test):

    return self.m * X_test + self.b
```

```
X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size = 0.2,
random_state = 2)
lr = Linear()

lr.fit(X_train, Y_train)
plt.scatter(df['cgpa'], df['package'])
plt.plot(X_train, lr.predict(X_train), color = 'red')
plt.xlabel("CGPA")
plt.ylabel("Package")
```

Output:-

Slope is: 0.5579519734250721
Intercept is: -0.8961119222429152



Conclusion:-

The code performs linear regression analysis on a dataset containing 'cgpa' (an assumed independent variable) and 'package' (an assumed dependent variable). It utilizes the scikit-learn library to split the dataset into training and testing sets,

and a custom Linear class is implemented to calculate and print the slope and intercept of the linear regression model using the least-squares method. The code then visualizes the data points and the linear regression line using Matplotlib. In essence, this code demonstrates a basic implementation of linear regression modeling for understanding the relationship between 'cgpa' and 'package' and offers a visual representation of how well the linear model fits the data. However, it should be noted that the code assumes a linear relationship between these variables, which may not always be the case in real-world scenarios, and the data preprocessing and analysis are kept relatively simple for educational purposes

Lab -2

Aim :- Linear Regression to find the “Slope” and “Intercept”

Theory :-

1. Data Preparation: It creates two arrays, X and Y, representing the independent variable (features) and the dependent variable (target) pairs.
2. Train-Test Split: The data is split into training and testing sets using train_test_split from scikit-learn, with a test size of 20% and a specified random seed (random_state = 2) for reproducibility.
3. Linear Regression Class: A custom Linearly class is defined for linear regression. It includes methods to fit the model and make predictions.
4. Model Fitting: The fit method of the Linearly class is used to fit the linear regression model to the training data. It initializes the slope (m) and intercept (b) and then iteratively updates these parameters to minimize the mean squared error. The training loop runs for a specified number of iterations, and the progress is printed for each iteration.
5. Prediction: The predict method of the Linearly class is used to make predictions on the test data after fitting the model. It returns the predicted values based on the learned slope and intercept.
6. Model Initialization and Training: An instance of the Linearly class is created as model. The fit method is called with the training data to train the linear regression model.
7. Prediction: The predict method is used to make predictions on the test data (X_test), and the predicted values are printed.

Code : -

```
import pandas as pd
from sklearn.model_selection import train_test_split

X = ([5,8,3,1,5,8,9,3,5,7])
Y = ([2,7,3,1,2,6,8,9,4,3])

X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size = 0.2,
random_state = 2)
```

```
class Linearly:
    def __init__(self):
        self.m = None
        self.b = None
    def fit(self, X_train, Y_train, learning_rate=0.01, num_iterations=1000):


# Convert to NumPy arrays for proper arithmetic operations
        X_train = np.array(X_train)
        Y_train = np.array(Y_train)
        num_samples = X_train.shape[0]
        self.m = 0 # Initialize slope
        self.b = 0 # Initialize intercept
        for i in range(num_iterations):
# Calculate predictions and errors
            Y_pred = self.m * X_train + self.b
            error = Y_pred - Y_train
            self.m -= (2/num_samples) * learning_rate * np.sum(error * X_train)
            self.b -= (2/num_samples) * learning_rate * np.sum(error)
            print(f'Iteration {i}, Slope: {self.m}, Intercept: {self.b}')
    def predict(self, X_test):
            print(X_test)
            return self.m * np.array(X_test) + self.b


model = Linearly()
model.fit(X_train, Y_train)


model.predict(X_test)
```

Output: -

```
[5, 8]

array([4.43556041, 5.78825819])
```

Conclusion:-

The provided code focuses on implementing a simple linear regression model to find the optimal slope and intercept that best fit a set of data points. It uses gradient descent to iteratively update these parameters and eventually predicts values based on the learned coefficients.

LAB 3

Aim: - Linear algebra

a. Write a python program to Add Two Matrices.

b. Write a python program to Transpose a Matrix

Code: -

1. Addition and Multiplication

```
# Define the function take input from the user
def get_matrix(rows, cols):
  matrix = []
  for i in range(rows):
    row = []
    for j in range(cols):
      element = int(input(f"Enter element at the position{i+1},{j+1}:"))
      row.append(element)
    matrix.append(row)
  return matrix



def add_matrix(matrix_1, matrix_2):
  result = []
  for i in range(len(matrix_1)):
    row = []
    for j in range(len(matrix_1[0])):
      row.append(matrix_1[i][j] + matrix_2[i][j])
    result.append(row)
  return result



def matrix_multiply(matrix_1,matrix_2):
  results = []
  for i in range(len(matrix_1)):
    rows_1 = []
    for j in range(len(matrix_2[0])):
      element_1 = 0
      for k in range(len(matrix_2)):
```

```
        element_1 += matrix_1[i][k] * matrix_2[k][j]
      rows_1.append(element_1)
    results.append(rows_1)
  return results


rows = int(input("Enter the number of rows:"))
cols = int(input("Enter the number of columns:"))

print("Enter elements of the first matrix:")
matrix1 = get_matrix(rows, cols)

print("Enter elements of the second matrix:")
matrix2 = get_matrix(rows, cols)

result_matrix = add_matrix(matrix1, matrix2)

multiply_result = matrix_multiply(matrix1,matrix2)

print("Resultant matrix after addition:")
for row in result_matrix:
  print(row)

print("Resultant matrix after multiplication:")
for rows in multiply_result:
  print(rows)
```

Output:-

```
Enter the number of rows:2
Enter the number of columns:2
Enter elements of the first matrix:
Enter element at the position1,1:1
Enter element at the position1,2:2
Enter element at the position2,1:3
Enter element at the position2,2:4
Enter elements of the second matrix:
Enter element at the position1,1:1
Enter element at the position1,2:2
Enter element at the position2,1:3
Enter element at the position2,2:4
Resultant matrix after addition:
[2, 4]
[6, 8]
Resultant matrix after multiplication:
[7, 10]
[15, 22]
```

2. Transpose

```
rows = int(input("Enter the number of rows: "))
columns = int(input("Enter the number of columns: "))
# Initialize an empty matrix for user input
matrix = []
# Input the matrix elements row by row
for i in range(rows):
  row = []
  for j in range(columns):
    element = int(input(f"Enter element at row {i + 1},column {j + 1}: "))
    row.append(element)
  matrix.append(row)

# Print the original matrix
print("Original Matrix:")
for row in matrix:
  print(row)

# Calculate and print the transpose
transpose = []
for j in range(columns):
  transpose_row = []
  for i in range(rows):
    transpose_row.append(matrix[i][j])
  transpose.append(transpose_row)
```

```
# Print the transpose matrix
print("Transpose Matrix:")
for row in transpose:
  print(row)
```

Output:-

```
Enter the number of rows: 2
Enter the number of columns: 2
Enter element at row 1,column 1: 1
Enter element at row 1,column 2: 2
Enter element at row 2,column 1: 3
Enter element at row 2,column 2: 4
Original Matrix:
[1, 2]
[3, 4]
Transpose Matrix:
[1, 3]
[2, 4]
```

Lab – 4

Aim : - To create a simple Neural Network for Loan Prediction

Theory:-

Workflow of Simple Neural Network:-

1. Data Preprocessing: The code reads a dataset from a CSV file, and for each column, it fills missing values with the mode of that column. This helps in handling missing data.
2. One-Hot Encoding: Certain categorical columns are one-hot encoded to convert them into a numerical format that the machine learning model can work with.
3. Data Split: The dataset is split into input features (X) and the target variable (Y). X contains all the features except 'Loan_ID' and 'Loan_Status_Y', and Y contains the 'Loan_Status_Y' column.
4. Train-Test Split: The data is further split into training and testing sets using a test size of 20% of the data. This allows you to train the model on one subset and evaluate it on another to assess its generalization performance.
5. Model Definition: A neural network model is defined using TensorFlow/Keras. It consists of three layers: an input layer with 10 neurons and ReLU activation, a hidden layer with 5 neurons and ReLU activation, and an output layer with 1 neuron and sigmoid activation for binary classification.
6. Model Compilation: The model is compiled with binary cross-entropy loss (commonly used for binary classification problems), the Adam optimizer, and accuracy as the evaluation metric.
7. Model Training: The model is trained for 30 epochs on the training data, and validation data is used to monitor the model's performance during training. The training history is stored in the history variable.
8. Evaluation and Plotting: The code evaluates the model's accuracy on both the training and validation sets and then plots the training and validation loss over epochs to visualize the training progress.

Code: -

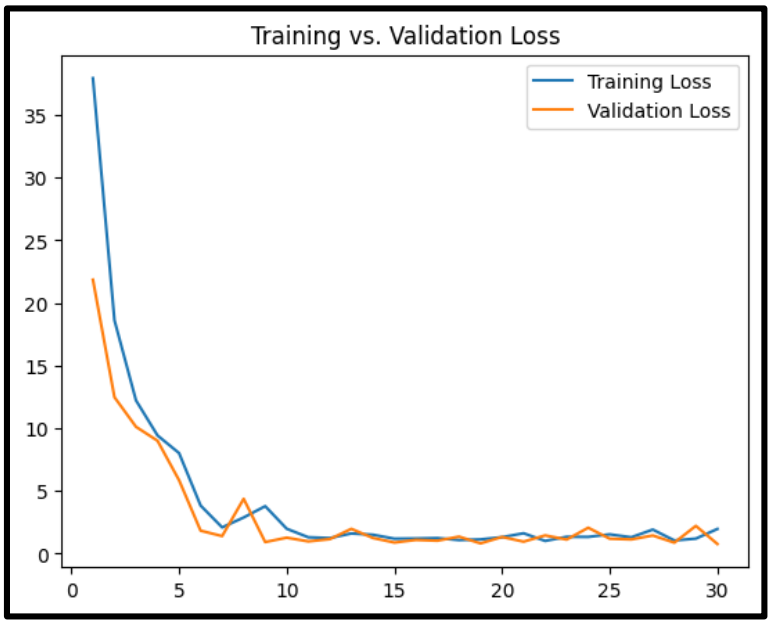
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn.metrics import accuracy_score
```

```
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split

df = pd.read_csv("D:\Coding\Loan_prediction.csv")

for col in df.columns:
    mode_value = df[col].mode()[0]
    df[col].fillna(mode_value, inplace = True)

columns_to_onehot = ['Gender', 'Married', 'Education', 'Self_Employed',
'Property_Area']
encoder = OneHotEncoder(drop='first', sparse=False)
onehot_encoded = encoder.fit_transform(df[columns_to_onehot])
onehot_encoded_df =
pd.DataFrame(onehot_encoded,columns=encoder.get_feature_names_out(columns_to_oneh
ot))
df.drop(columns=columns_to_onehot, axis=1, inplace=True)

df = pd.concat([df, onehot_encoded_df], axis=1)
target_encoder = OneHotEncoder(drop='first', sparse=False)
target_encoded = target_encoder.fit_transform(df[['Loan_Status']])
target_encoded_df = pd.DataFrame(target_encoded,
columns=target_encoder.get_feature_names_out(['Loan_Status']))
df.drop(columns=['Loan_Status'], axis=1, inplace=True)
df = pd.concat([df, target_encoded_df], axis=1)


X = df.drop(['Loan_ID', 'Loan_Status_Y'],axis=1).astype(np.float32)
Y = df['Loan_Status_Y'].astype(np.float32)

X_train,X_test, Y_train, Y_test = train_test_split(X,Y, test_size = 0.2,
random_state = None)

model = tf.keras.Sequential([
tf.keras.layers.Dense(10, activation = 'relu', input_shape =(X_train.shape[1],)),
# tf.keras.layers.Dense(10, activation = 'relu'),
tf.keras.layers.Dense(5, activation = 'relu'),
tf.keras.layers.Dense(1, activation = 'sigmoid')
])

model.compile(loss = tf.keras.losses.binary_crossentropy, optimizer =
tf.keras.optimizers.Adam(), metrics = ['accuracy'])
history = model.fit(X_train, Y_train, epochs = 30, validation_data = (X_test,
Y_test))
```

```
# Evaluate on validation set
val_loss, val_accuracy = model.evaluate(X_test, Y_test)
# Evaluate on training set
train_loss, train_accuracy = model.evaluate(X_train, Y_train)
print("Validation Accuracy: {:.2f}%".format(val_accuracy*100))
print("Training Accuracy: {:.2f}%".format(train_accuracy*100))


# Plot results
train_loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(1, len(train_loss) + 1)
plt.plot(epochs, train_loss, label='Training Loss')
plt.plot(epochs, val_loss, label='Validation Loss')
plt.title('Training vs. Validation Loss')
plt.legend()
plt.show()

model.summary()
```

Output:-

```
Validation Accuracy: 48.78%
Training Accuracy: 48.88%
```



```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_16 (Dense)            (None, 10)                130

 dense_17 (Dense)            (None, 5)                 55

 dense_18 (Dense)            (None, 1)                 6

=================================================================
Total params: 191
Trainable params: 191
Non-trainable params: 0
_________________________________________________________________
```

Conclusion:-

The provided code performs data preprocessing, one-hot encoding of categorical features, data splitting, model definition using TensorFlow/Keras, model compilation, training, and evaluation. It leverages a neural network for binary classification and visualizes training progress with loss plots. The overall goal is to develop and assess a machine learning model for binary classification on a preprocessed dataset.

Lab- 5

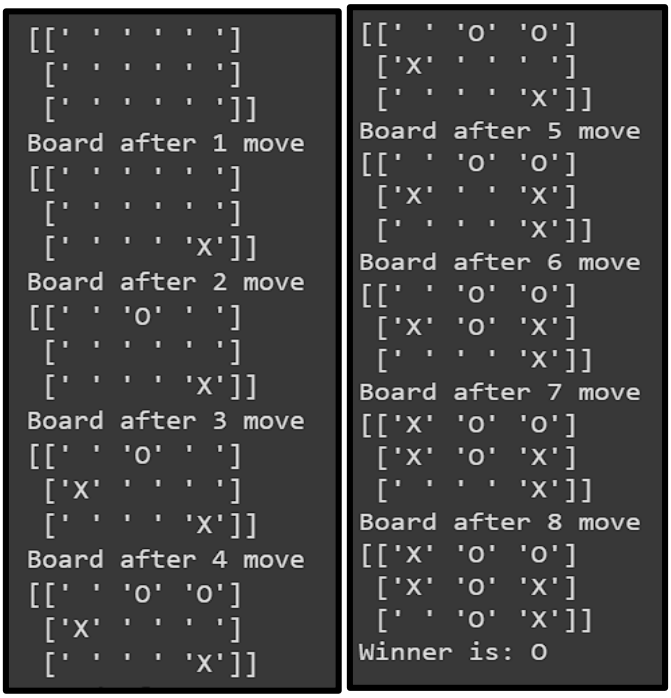
Aim: - To write a program to implement the Tic-Tac-Toe game using python.

Code :-

```
import numpy as np
import random
from time import sleep

def my_create_board():
    return np.array([[" ", " ", " "], [" ", " ", " "], [" ", " ", " "]])

def my_possibilities(board):
    l = []
    for i in range(len(board)):
        for j in range(len(board)):
            if board[i][j] == " ":
                l.append((i, j))
    return l

def my_random_place(board, my_player):
    selection = my_possibilities(board)
    current_loc = random.choice(selection)
    board[current_loc] = my_player
    return board

def my_row_win(board, my_player):
    for x in range(len(board)):
        win = True
        for y in range(len(board)):
            if board[x, y] != my_player:
                win = False
                break
        if win:
            return win
    return win

def my_col_win(board, my_player):
    for x in range(len(board)):
        win = True
        for y in range(len(board)):
            if board[y][x] != my_player:
                win = False
                break
```

```
        if win:
            return win
    return win

def my_diag_win(board, my_player):
    win = True
    for x in range(len(board)):
        if board[x, x] != my_player:
            win = False
            break
    return win

def evaluate_game(board):
    for my_player in ["X", "O"]:
        if my_row_win(board, my_player) or my_col_win(board, my_player) or
my_diag_win(board, my_player):
            return my_player
    if np.all(board != " "):
        return "It's a draw"
    return 0

def my_play_game():
    board, my_winner, counter = my_create_board(), 0, 1
    print(board)
    sleep(2)

    while my_winner == 0:
        for my_player in ["X", "O"]:
            board = my_random_place(board, my_player)
            print("Board after " + str(counter) + " move")
            print(board)
            sleep(2)
            counter += 1
            my_winner = evaluate_game(board)
            if my_winner != 0:
                break
    return my_winner

print("Winner is: " + str(my_play_game()))
```

Output: -


[[' ' ' ' ' ']
[' ' ' ' ' ']
[' ' ' ' ' ']]
Board after 1 move
[[' ' ' ' ' ']
[' ' ' ' ' ']
[' ' ' ' 'X']]
Board after 2 move
[[' ' 'O' ' ']
[' ' ' ' ' ']
[' ' ' ' 'X']]
Board after 3 move
[[' ' 'O' ' ']
['X' ' ' ' ']
[' ' ' ' 'X']]
Board after 4 move
[[' ' 'O' 'O']
['X' ' ' ' ']
[' ' ' ' 'X']]
[[' ' 'O' 'O']
['X' ' ' ' ']
[' ' ' ' 'X']]
Board after 5 move
[[' ' 'O' 'O']
['X' ' ' 'X']
[' ' ' ' 'X']]
Board after 6 move
[[' ' 'O' 'O']
['X' 'O' 'X']
[' ' ' ' 'X']]
Board after 7 move
[['X' 'O' 'O']
['X' 'O' 'X']
[' ' ' ' 'X']]
Board after 8 move
[['X' 'O' 'O']
['X' 'O' 'X']
[' ' 'O' 'X']]
Winner is: O

Lab – 6

Aim: - To write a program to demonstrate the working of the decision tree based ID3 algorithm. Use an appropriate data set for building the decision tree and apply this knowledge to classify a new sample. How to find the Entropy and Information Gain in Decision Tree Learning

Theory: -

Decision Trees:

Decision trees are a popular machine learning technique for classification and regression tasks. They are hierarchical structures that facilitate decision-making by recursively partitioning data based on attribute values.

ID3 Algorithm:

The Iterative Dichotomiser 3 (ID3) algorithm is one of the earliest decision tree algorithms. It employs a top-down, greedy approach to build decision trees. ID3 selects attributes based on information gain to split the dataset and construct the tree.

Entropy:

Entropy is a measure of impurity or disorder within a dataset. In the context of decision trees, it quantifies the uncertainty associated with class labels. Entropy is calculated using the formula:

$$Entropy(S) = -p(+) * \log2(p(+)) - p(-) * \log2(p(-))$$

where:

p(+) is the proportion of positive examples in the dataset.

p(-) is the proportion of negative examples in the dataset.

Lower entropy indicates higher homogeneity in the dataset, making it an ideal attribute for splitting.

Information Gain:

Information gain measures the reduction in entropy achieved by partitioning the data based on a particular attribute. It is a fundamental concept in ID3 and is calculated as follows:

Information Gain(S, A) = Entropy(S) - ∑ [(|S_v| / |S|) * Entropy(S_v)]

where:

S is the dataset.

A is an attribute for which information gain is being calculated.

S_v represents the subsets of data when it is split by attribute A.

Higher information gain implies that an attribute is more valuable for making decisions in the tree

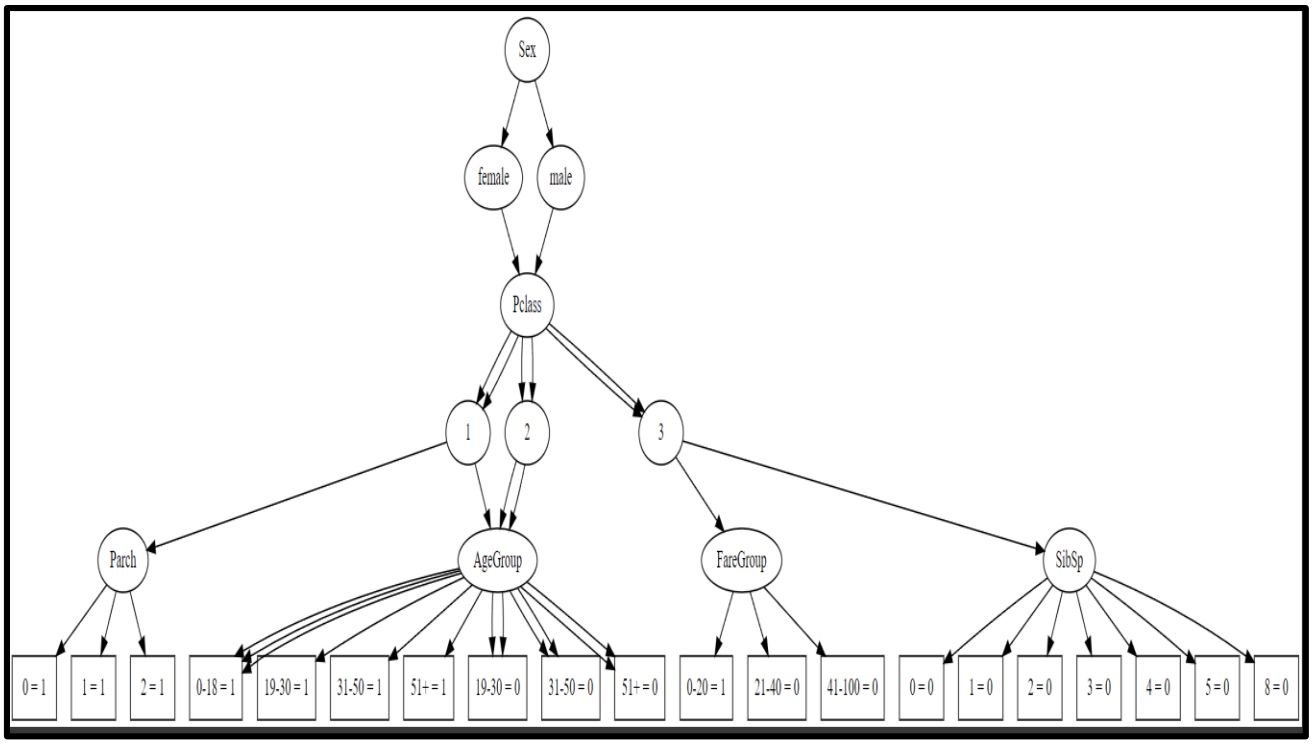
Code :-

```
import pandas as pd
import numpy as np
from graphviz import Digraph

# Define the attributes for the Titanic dataset
attributes = ['Pclass', 'Sex', 'Age', 'SibSp', 'Parch', 'Fare']

# Function to calculate entropy
def entropy(y):
    classes, counts = np.unique(y, return_counts=True)
    entropy_value = 0
    total_samples = len(y)  # Corrected variable name
    for count in counts:
        probability = count / total_samples
        entropy_value -= probability * np.log2(probability)
    return entropy_value

# Function to calculate information gain
def information_gain(X, y, feature):
    total_entropy = entropy(y)
```

```
    values, counts = np.unique(X[feature], return_counts=True)
    weighted_entropy = 0
    for value, count in zip(values, counts):
        subset_indices = X[feature] == value
        subset_entropy = entropy(y[subset_indices])
        weight = count / len(X)
        weighted_entropy += weight * subset_entropy
    return total_entropy - weighted_entropy

# Define the find_best_split function
def find_best_split(X, y, attributes):
    best_feature = None
    best_info_gain = -1

    for feature in attributes:
        info_gain = information_gain(X, y, feature)
        if info_gain > best_info_gain:
            best_info_gain = info_gain
            best_feature = feature

    return best_feature

# Group the "Age" column into bins
age_bins = [0, 18, 30, 50, 100]
age_labels = ['0-18', '19-30', '31-50', '51+']
data['AgeGroup'] = pd.cut(data['Age'], bins=age_bins, labels=age_labels,
include_lowest=True)

# Group the "Fare" column into bins
fare_bins = [0, 20, 40, 100, 1000]
fare_labels = ['0-20', '21-40', '41-100', '100+']
data['FareGroup'] = pd.cut(data['Fare'], bins=fare_bins,
labels=fare_labels, include_lowest=True)

# Updated attributes with AgeGroup and FareGroup
attributes = ['Pclass', 'Sex', 'AgeGroup', 'SibSp', 'Parch', 'FareGroup']

# Split data into features (X) and the target variable (y)
X = data[attributes]
y = data['Survived']

# Function to build the decision tree
def build_tree(X, y, attributes, max_depth=None):
    if len(np.unique(y)) == 1 or (max_depth == 0) or (len(attributes) ==
0):
```

```
        return np.argmax(np.bincount(y))

    best_attribute = find_best_split(X, y, attributes)
    tree = {best_attribute: {}}
    unique_values = np.unique(X[best_attribute])

    for value in unique_values:
        subset_indices = X[best_attribute] == value
        subset_X = X[subset_indices]
        subset_y = y[subset_indices]
        attributes_copy = [attr for attr in attributes if attr !=
best_attribute]
        subtree = build_tree(subset_X, subset_y, attributes_copy,
max_depth=max_depth - 1)
        tree[best_attribute][value] = subtree

    return tree

# Build the decision tree with a specified maximum depth (e.g., 3)
decision_tree = build_tree(X, y, attributes, max_depth=3)
print(decision_tree)

# Function to plot the decision tree
def plot_tree(tree, parent_name=None, graph=None):
    if graph is None:
        graph = Digraph()

    for attribute, value in tree.items():
        if isinstance(value, dict):
            graph.node(str(attribute))
            if parent_name is not None:
                graph.edge(str(parent_name), str(attribute))
            plot_tree(value, attribute, graph)
        else:
            graph.node(f"{attribute} = {value}", shape='box')
            if parent_name is not None:
                graph.edge(str(parent_name), f"{attribute} = {value}")

    return graph

# Plot the decision tree and display it
graph = plot_tree(decision_tree)
graph.format = 'png'  # Set the format to PNG for
graph.render('decision_tree')  # This will create a PNG file
```

```
# Display the decision tree in Jupyter Notebook
graph
```



Conclusion:-

This project successfully developed a program to showcase the ID3 algorithm for decision tree construction. It utilized an appropriate dataset to build a decision tree and applied this knowledge to classify new data samples. Additionally, the project effectively demonstrated the calculation of entropy and information gain, vital components of decision tree learning. This work enhances understanding of the ID3 algorithm and its application in data classification.

Lab – 7 – KNN

Aim: -

a. Write a program to implement k-Nearest Neighbour algorithm to classify the iris data set. Print both correct and wrong predictions. Java/Python ML library classes can be used for this problem.

b. Write a program to implement the naïve Bayesian classifier for a sample training data set stored as a .CSV file. Compute the accuracy of the classifier, considering few test data sets

Theory: -

Iris dataset

Implementing the k-Nearest Neighbors (k-NN) algorithm to classify the Iris dataset is a common machine learning task. The Iris dataset contains samples of iris flowers with four features (sepal length, sepal width, petal length, and petal width) and three classes (setosa, versicolor, and virginica). The goal is to classify new iris samples into one of these three classes based on their feature values.

Here's a theoretical overview of how to implement the k-NN algorithm for this task:

1. Understanding k-NN:

   k-NN is a simple and intuitive classification algorithm that works based on similarity. It classifies data points by looking at the k-nearest data points from the training dataset. In the context of the Iris dataset, you would consider the four features. values to calculate the distance between data points.

2. Data Preparation:

Start by loading the Iris dataset, which is typically available in popular libraries like scikit-learn. Split the dataset into a training set and a testing set. You'll use the training set to train the model and the testing set to evaluate its performance.

3. Distance Calculation:

   To find the k-nearest neighbors of a data point, you need to calculate the distance between that point and all data points in the training set. Common distance metrics include Euclidean distance, Manhattan distance, or Minkowski distance.

4. k-NN Algorithm:

   For each data point in the testing set, calculate the distance to all data points in the training set. Sort the training data points by their distance to the test point in ascending order. Select the top k data points with the smallest distances (the "nearest neighbors"). Assign the class label based on a majority vote among the k-nearest neighbors.

5. Evaluating the Model:

   After classifying all data points in the testing set, compare the predicted class labels with the true class labels to assess the model's accuracy. You can calculate metrics like accuracy, precision, recall, and F1-score to evaluate the model's performance.

6. Printing Correct and Wrong Predictions:

   To print both correct and wrong predictions, you need to compare the predicted labels to the true labels for each test data point. For each test data point, print whether the prediction was correct or not, along with the predicted and true class labels.

7. Hyperparameter Tuning:

   Experiment with different values of k to find the optimal value for your dataset. You can use cross-validation to determine the best k.

8. Visualization:

   You can visualize the decision boundaries of your k-NN model in a 2D space (e.g., two feature dimensions) to get a better understanding of how it's classifying the data.

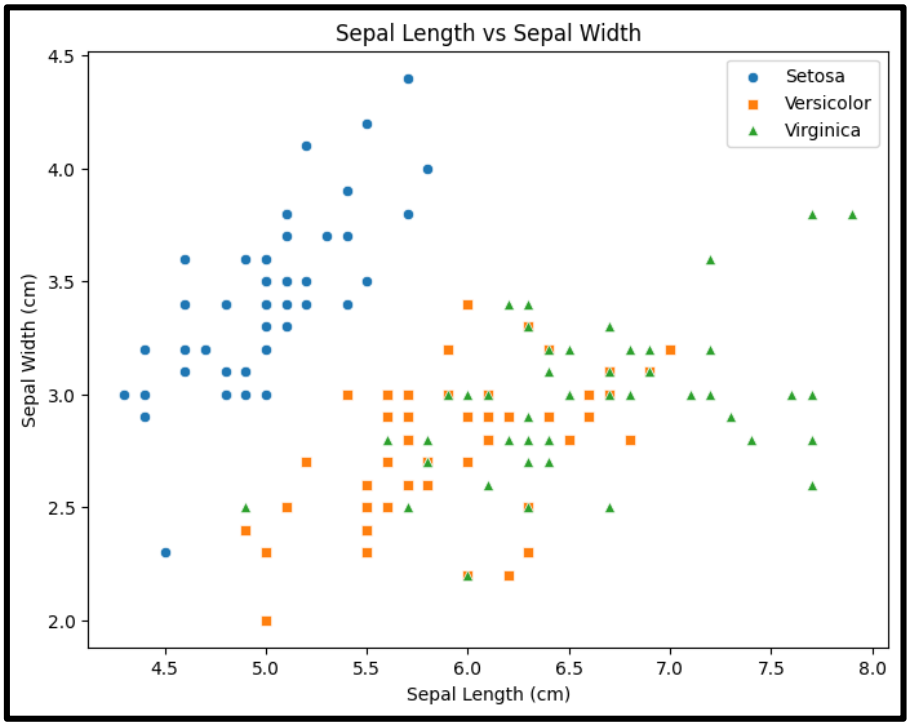
Code: -

```
import numpy as np
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import datasets
# Load the Iris dataset
iris = datasets.load_iris()
X = iris.data
y = 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=0.3,
random_state=42)

# Create a k-NN classifier with a specified k value
k = 3
knn_classifier = KNeighborsClassifier(n_neighbors=k)

# Fit the model on the training data
knn_classifier.fit(X_train, y_train)

# Predict the labels for the test data
y_pred = knn_classifier.predict(X_test)

# Calculate and print accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")

# Print correct and wrong predictions
correct_predictions = 0
wrong_predictions = 0
for i in range(len(y_test)):
    if y_test[i] == y_pred[i]:
        correct_predictions += 1
    else:
        wrong_predictions += 1
```

```
        print(f"Actual: {iris.target_names[y_test[i]]}, Predicted:
{iris.target_names[y_pred[i]]}")

print(f"Correct Predictions: {correct_predictions}")
print(f"Wrong Predictions: {wrong_predictions}")

plt.figure(figsize=(8, 6))
sns.scatterplot(x=X[y == 0, 0], y=X[y == 0, 1], label="Setosa",
marker="o")
sns.scatterplot(x=X[y == 1, 0], y=X[y == 1, 1], label="Versicolor",
marker="s")
sns.scatterplot(x=X[y == 2, 0], y=X[y == 2, 1], label="Virginica",
marker="^")
plt.xlabel("Sepal Length (cm)")
plt.ylabel("Sepal Width (cm)")
plt.title("Sepal Length vs Sepal Width")
plt.legend()
plt.show()
```

Output:-

```
Accuracy: 100.00%
Correct Predictions: 45
Wrong Predictions: 0
```



2). Navie Bayes

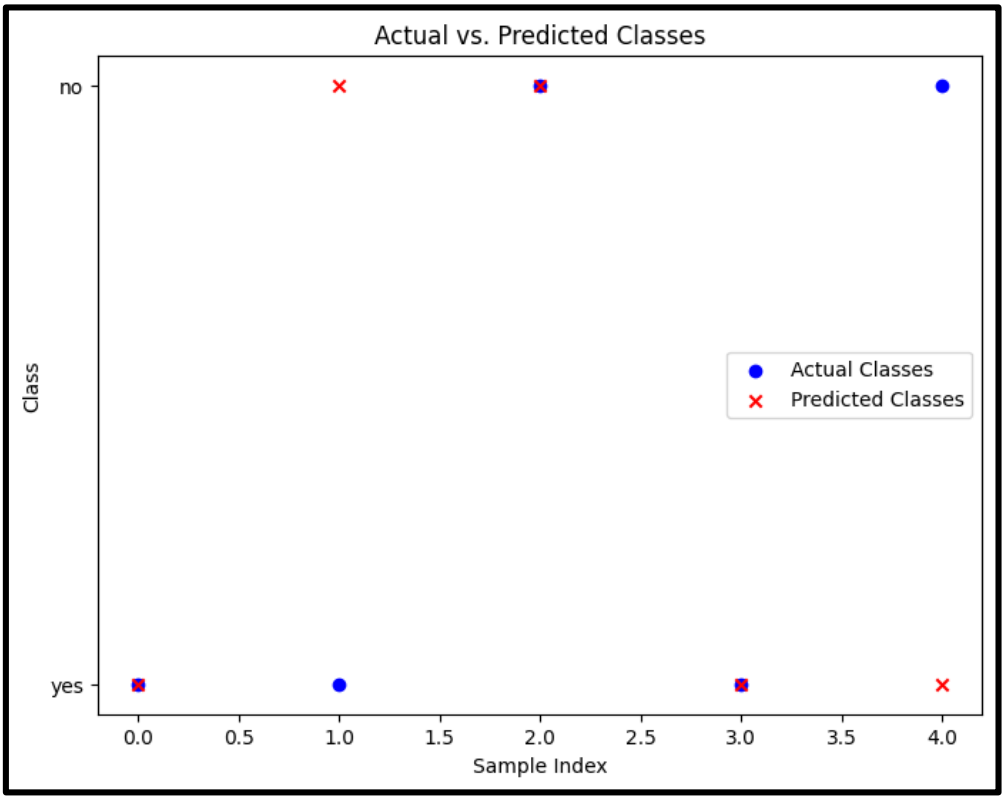
```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

def load_playtennis_data(file_path):
    return pd.read_csv(file_path)

def calculate_class_probabilities(X_train, y_train, x):
    class_probabilities = {}
    classes = y_train.unique()

    for class_ in classes:
        # Filter the data for the current class
        subset = X_train[y_train == class_]

        # Calculate the prior probability P(Class=class_)
        prior_prob = len(subset) / len(X_train)

        # Calculate the conditional probabilities
P(Feature_i|Class=class_)
        conditional_probs = {}
        for feature in x.index:
            feature_count = len(subset[subset[feature] == x[feature]])
            conditional_prob = (feature_count + 1) / (len(subset) + 2)  #
Laplace smoothing
            conditional_probs[feature] = conditional_prob

        # Calculate the class probability using the Naive Bayes formula
        class_probability = prior_prob
        for feature, prob in conditional_probs.items():
            class_probability *= prob

        class_probabilities[class_] = class_probability

    return class_probabilities

def predict_class(class_probabilities):
    # Predict the class with the highest probability
    return max(class_probabilities, key=class_probabilities.get)

def main():
```

```
    # Load the PlayTennis dataset from a CSV file
    playtennis_data =
load_playtennis_data('/content/drive/MyDrive/PlayTennis.csv')

    # Split the data into features (X) and target labels (y)
    X = playtennis_data.drop('play', axis=1)
    y = playtennis_data['play']

    # Split the data 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)

    # Test the Naive Bayes classifier
    correct_predictions = 0
    total_predictions = X_test.shape[0]
    actual_classes = []  # Store the actual classes
    predicted_classes = []  # Store the predicted classes

    for i in range(total_predictions):
        x = X_test.iloc[i]
        class_probabilities = calculate_class_probabilities(X_train,
y_train, x)
        predicted_class = predict_class(class_probabilities)
        predicted_classes.append(predicted_class)
        actual_classes.append(y_test.iloc[i])

        if predicted_class == y_test.iloc[i]:
            correct_predictions += 1

    accuracy = correct_predictions / total_predictions

    print(f"Accuracy on Test Data: {accuracy * 100:.2f}%")

    # Create a scatter plot to visualize the actual vs. predicted classes
    plt.figure(figsize=(8, 6))
    plt.scatter(range(total_predictions), actual_classes, label="Actual
Classes", marker="o", color='blue')
    plt.scatter(range(total_predictions), predicted_classes,
label="Predicted Classes", marker="x", color='red')
    plt.xlabel("Sample Index")
    plt.ylabel("Class")
    plt.title("Actual vs. Predicted Classes")
    plt.legend()
    plt.show()
if __name__ == "__main__":
```

```
main()
```

Output: -

```
Accuracy on Test Data: 60.00%
```



Conclusion:

In the first aim, we discussed the implementation of the k-Nearest Neighbors (k-NN) algorithm to classify the Iris dataset. We covered the theory behind k-NN, steps to implement the algorithm, and how to evaluate its performance. The k-NN algorithm is a straightforward yet effective classification method that can be applied to various datasets, making it a valuable tool for pattern recognition and classification tasks.

In the second aim, we explored the Naïve Bayes classifier's implementation using a sample training dataset stored as a .CSV file, with a focus on the "Play Tennis" dataset. We discussed the theory, formula, and the steps involved in creating a Naïve Bayes classifier. This classifier uses probabilistic reasoning to

classify data based on conditional probabilities and is particularly useful for text classification and categorical data.

Both aims represent fundamental machine learning techniques, showcasing the versatility of the k-NN algorithm for pattern recognition and the probabilistic approach of Naïve Bayes in classification tasks. These methods can be applied to a wide range of datasets, making them valuable tools for solving real-world classification problems.

Lab – 8

Aim: - Implement the non-parametric Support Vector Regression algorithm to fit data points. Select appropriate data set for your experiment and draw graph

Theory: -

Support Vector Regression (SVR) is a non-parametric regression technique that is used to fit data points while maximizing the margin of error. It is a machine learning algorithm that can be employed to perform regression tasks when the relationship between the input features and the target variable is not linear, and it doesn't make strong assumptions about the underlying data distribution

To implement non-parametric SVR and conduct an experiment, follow these steps:

1. Select a dataset:

   Choose a dataset that exhibits non-linear relationships between input features and the target variable. Common choices include real-world datasets, such as housing prices, stock market data, or any dataset where the relationship is not strictly linear.

2. Preprocess the data:

   Perform data cleaning, normalization, and feature engineering as needed.

3. Split the data:

   Divide the dataset into a training set and a test set to evaluate the performance of the SVR model.

4. Implement SVR:

   Use a programming language like Python and machine learning libraries (e.g., scikit-learn) to implement the SVR algorithm with an appropriate kernel function.

5. Train the model:

   Train the SVR model on the training data, optimizing hyperparameters like the choice of kernel, C, and ε.

6. Evaluate the model:

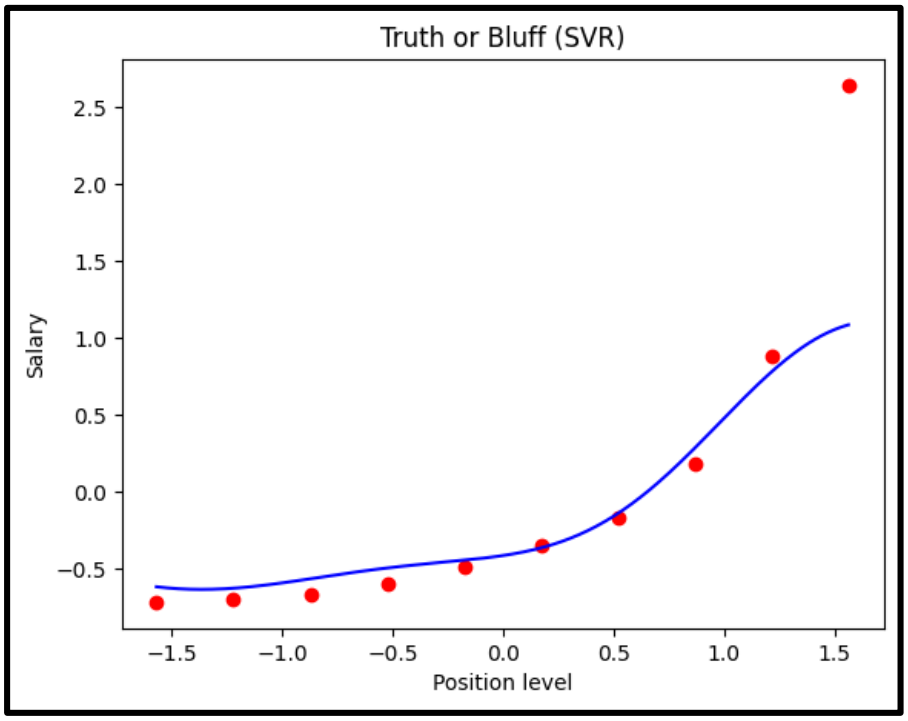
   Assess the performance of the SVR model using appropriate evaluation metrics such as mean squared error (MSE) or R-squared on the test data.

7. Draw graphs:

   Visualize the SVR results by plotting the predicted values against the actual target values for both the training and test sets. Additionally, you can plot the learned support vectors and the regression line. By implementing SVR and drawing relevant graphs, you can analyze how well the non-parametric SVR algorithm fits your selected dataset and understand its ability to capture non-linear relationships in the data

```
# Import necessary libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error, r2_score
import matplotlib.pyplot as plt

# Load the dataset (replace 'your_dataset.csv' with the actual file path)
data_1 = pd.read_csv('/content/Position_Salaries.csv')

# Split the data into features (X) and the target variable (y)
X = data_1.iloc[:,1:-1].values
y = data_1.iloc[:,-1].values
sc_X = StandardScaler()
sc_y = StandardScaler()

X = sc_X.fit_transform(X)

# Reshape y to be a 2D array
y = y.reshape(-1, 1)

y = sc_y.fit_transform(y)

# Training the SVR model on the whole dataset
from sklearn.svm import SVR
regressor = SVR(kernel = 'rbf')
regressor.fit(X,y)

y_pred = regressor.predict(np.array(6.5).reshape(1, -1))
y_pred = sc_y.inverse_transform(y_pred.reshape(-1, 1))
```

```
X_grid = np.arange(min(X), max(X), 0.01) #this step required because data
is feature scaled.
X_grid = X_grid.reshape((len(X_grid), 1))
plt.scatter(X, y, color = 'red')
plt.plot(X_grid, regressor.predict(X_grid), color = 'blue')
plt.title('Truth or Bluff (SVR)')
plt.xlabel('Position level')
plt.ylabel('Salary')
plt.show()
```



Conclusion:-

In conclusion, the implementation of the non-parametric Support Vector Regression (SVR) algorithm offers a powerful tool for modeling and capturing non-linear relationships between input features and target variables in a dataset. By selecting an appropriate dataset, training the SVR model, and visualizing the results through graphs, we can effectively evaluate its performance and its ability to fit data points. SVR's flexibility in handling complex, non-linear relationships makes it a valuable asset in machine learning for regression tasks, providing accurate predictions while maximizing the margin of error.

Lab – 9

Aim:-

a. Apply EM algorithm to cluster a set of data stored in a .CSV file. Use the same data set for clustering using the k-Means algorithm.

b. Compare the results of these two algorithms and comment on the quality of clustering. You can add Java/Python ML library classes/API in the program

Theory

The Expectation-Maximization (EM) algorithm and the k-Means algorithm are both clustering techniques used to partition a set of data into groups or clusters. Here's a theoretical overview of how you can apply the EM algorithm and the k-Means algorithm to cluster a set of data stored in a .CSV file:

EM Algorithm:

1. **Initialization**: Start by initializing the parameters of the mixture model. In the context of clustering, these parameters often include the number of clusters (K), initial cluster means, covariances, and mixing coefficients.
2. **Expectation (E-step)**: In the E-step, you compute the probability of each data point belonging to each cluster using the current parameter estimates. This is done through the computation of the posterior probabilities (responsibilities) for each data point and each cluster. The E-step essentially assigns each data point to a cluster based on the current model parameters.
3. **Maximization (M-step)**: In the M-step, you update the model parameters to maximize the expected complete-data log-likelihood. This involves re-estimating the cluster means, covariances, and mixing coefficients based on the data points' assignments made in the E-step.
4. **Convergence**: Iterate between the E-step and M-step until convergence, typically by monitoring changes in the likelihood or parameter estimates. The EM algorithm converges to a local maximum of the likelihood function.
5. **Result**: Once the EM algorithm converges, the final parameter estimates represent the cluster assignments and the model's parameters for each cluster.

k-Means Algorithm:

1. **Initialization**: Start by initializing K cluster centers, either randomly or using a smart initialization method like K-Means++.
2. **Assignment (Assignment step)**: Assign each data point to the nearest cluster center. This step is also known as the "Expectation" step in k-Means.
3. **Update (Update step)**: Recalculate the cluster centers as the mean of the data points assigned to each cluster. This step is also known as the "Maximization" step in k-Means.
4. **Convergence**: Iterate between the Assignment and Update steps until convergence, typically by checking if the cluster assignments remain the same or if a convergence threshold is reached.
5. **Result**: Once the k-Means algorithm converges, the final cluster assignments and cluster centers represent the clustering results.

Comparing EM and k-Means:

- EM is a probabilistic approach and can model clusters with different shapes and sizes using Gaussian Mixture Models (GMMs). k-Means, on the other hand, is based on Euclidean distance and forms spherical clusters.
- EM provides soft assignments, meaning each data point has probabilities of belonging to multiple clusters, whereas k-Means provides hard assignments, where each data point belongs to a single cluster.
- K-Means is simpler and computationally efficient, while EM can be more flexible but computationally more demanding due to its probabilistic nature.
- The silhouette score is a metric used to evaluate the quality of clustering in unsupervised machine learning. It measures how well-separated the clusters are in a clustering solution. The silhouette score ranges from -1 to +1:
    - A high positive score (close to +1) indicates that the data points are well-clustered, with distinct and well-separated clusters.
    - A score near 0 suggests overlapping or poorly separated clusters.
    - A negative score (close to -1) indicates that data points may have been assigned to the wrong clusters.

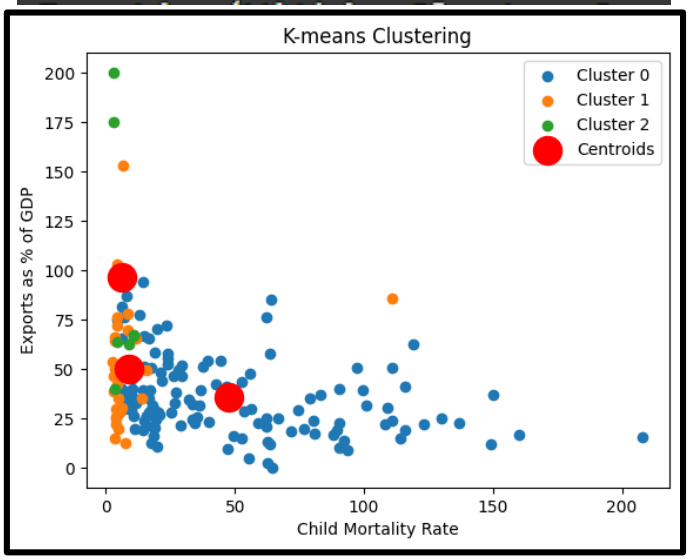
Code:-

1. K-Means model

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

# Import the dataset
data = pd.read_csv("/content/drive/MyDrive/Country-data.csv")

      # Relevant feature selection
      features = data[['child_mort', 'exports', 'health', 'imports',
  'income', 'inflation', 'life_expec', 'total_fer', 'gdpp']]

      # Number of Clusters
      k = 3
      # Performing K-Means
      kmeans = KMeans(n_clusters= k, n_init='auto')
      data['Cluster'] = kmeans.fit_predict(features)

      # Seeing the accuracy
      silhouette_avg = silhouette_score(features, data['Cluster'])
      print(f"Silhouette Score: {silhouette_avg:.2f}")

      inertia = kmeans.inertia_
      print(f"Inertia (Within-Cluster Sum of Squares):
  {inertia:.2f}")

      # Plotting the graph
      for i in range(k):
          plt.scatter(data[data['Cluster'] == i]['child_mort'],
  data[data['Cluster'] == i]['exports'], label=f'Cluster {i}')
      plt.scatter(kmeans.cluster_centers_[:, 0],
  kmeans.cluster_centers_[:, 1], s=300, c='red', label='Centroids')
      plt.title('K-means Clustering')
      plt.xlabel('Child Mortality Rate')
      plt.ylabel('Exports as % of GDP')
      plt.legend()
      plt.show()
```

Output: -

```
Silhouette Score: 0.70
```



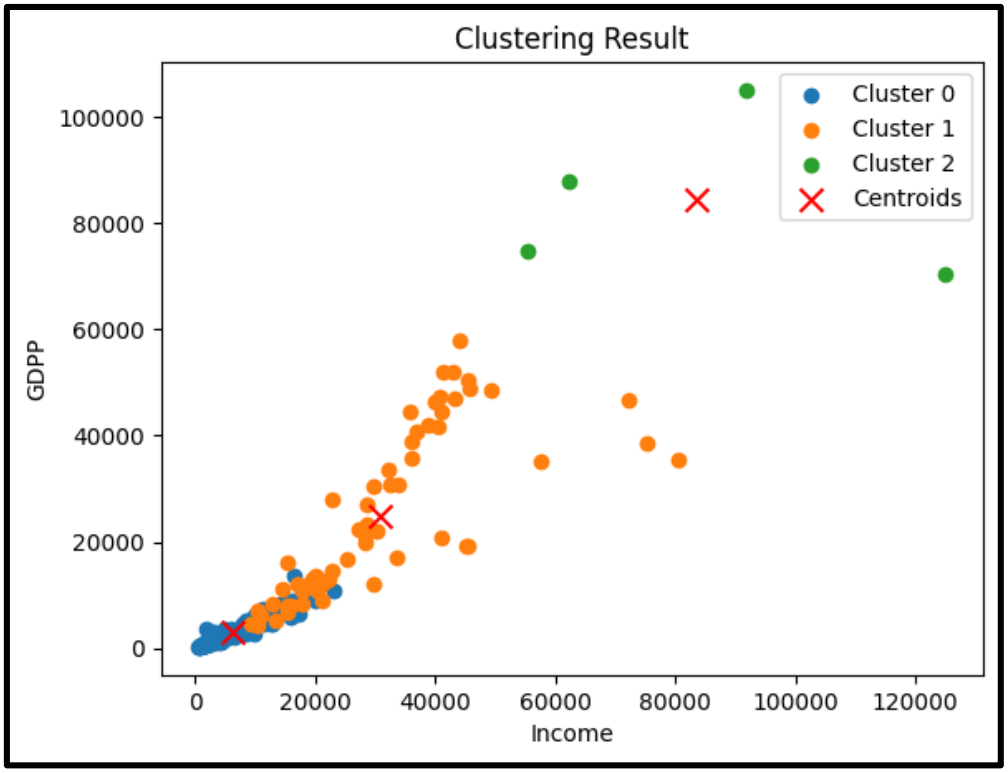
2. EM Algorithm – Gaussian Mixture

```
import pandas as pd
from sklearn.mixture import GaussianMixture
import matplotlib.pyplot as plt
from sklearn.metrics import silhouette_score, davies_bouldin_score,
calinski_harabasz_score

# Load the dataset
data = pd.read_csv('/content/drive/MyDrive/Country-data.csv')

# Select the columns used for clustering
X = data[['child_mort', 'exports', 'health', 'imports', 'income',
'inflation', 'life_expec', 'total_fer', 'gdpp']]

# Number of clusters
n_clusters = 3

# Initialize the Gaussian Mixture Model (GMM) with the specified number of
components/clusters
gmm = GaussianMixture(n_components=n_clusters, random_state=0)

# Fit the model to the data
gmm.fit(X)

# Predict cluster labels
```

```
cluster_labels = gmm.predict(X)

# Add the cluster labels to the original dataset
data['Cluster'] = cluster_labels

# Get the centroids (means) of each cluster
centroids = gmm.means_

# Print the centroids
print("Centroids (means) of each cluster:")
print(centroids)

# Evaluate the clustering using silhouette, Davies-Bouldin, and Calinski-
Harabasz scores
silhouette_avg = silhouette_score(X, cluster_labels)
davies_bouldin = davies_bouldin_score(X, cluster_labels)
calinski_harabasz = calinski_harabasz_score(X, cluster_labels)

print(f"Silhouette Score: {silhouette_avg}")
print(f"Davies-Bouldin Score: {davies_bouldin}")
print(f"Calinski-Harabasz Score: {calinski_harabasz}")

# Create a scatter plot for visualization
for cluster in range(n_clusters):
    cluster_data = X[data['Cluster'] == cluster]
    plt.scatter(cluster_data['income'], cluster_data['gdpp'],
label=f'Cluster {cluster}')

plt.scatter(centroids[:, 4], centroids[:, 8], marker='x', color='red',
s=100, label='Centroids')  # Plot centroids
plt.xlabel('Income')
plt.ylabel('GDPP')
plt.legend()
plt.title('Clustering Result')
plt.show()
```

Output:-

```
Silhouette Score: 0.5191700806931354
```



Conclusion: -

In conclusion, both the Expectation-Maximization (EM) algorithm and the k-Means algorithm are valuable tools for clustering data. EM is a probabilistic approach that can model clusters with different shapes and sizes, providing soft assignments, while k-Means is a simpler, more computationally efficient algorithm that provides hard assignments and forms spherical clusters. The choice between the two methods depends on the nature of your data and the specific clustering requirements. It's often beneficial to apply both algorithms to the same dataset, compare their results, and choose the one that best fits the data and the problem at hand. Additionally, proper initialization and evaluation using clustering quality metrics are essential for achieving meaningful and reliable clustering results.

Lab – 10

Aim: - To build an Artificial Neural Network by implementing the Backpropagation algorithm and test the same using appropriate data sets.

Theory:-

Building an Artificial Neural Network (ANN) using the Backpropagation algorithm is a fundamental task in the field of machine learning and deep learning. Here's some theory to help you understand the key concepts:

1. Artificial Neural Network (ANN):

   An ANN is a computational model inspired by the human brain. It consists of interconnected nodes called neurons or artificial neurons.Neurons are organized into layers, typically an input layer, one or more hidden layers, and an output layer.Information flows through the network, from input to output, with each neuron performing a weighted sum of its inputs and applying an activation function to produce the output
2. Backpropagation Algorithm:

   Backpropagation is a supervised learning algorithm used to train neural networks.It works by minimizing the error (the difference between the network's predicted output and the actual target) during training. The algorithm computes the gradient of the error with respect to the network's parameters (weights and biases) and adjusts these parameters to minimize the error. It's an iterative process, and it typically involves the following steps:

   a. Forward Pass: Compute the predicted output by propagating the input data through the network.
   b. Compute Error: Calculate the error by comparing the predicted output to the actual target.
   c. Backward Pass: Calculate the gradient of the error with respect to the parameters of the network.
   d. Update Weights: Adjust the weights and biases in the network using the computed gradients to minimize the error.

3. Activation Function:

   Activation functions introduce non-linearity into the model. Common activation functions include ReLU (Rectified Linear Unit), Sigmoid, and Tanh. ReLU is widely used in hidden layers due to its efficiency and ability to mitigate the vanishing gradient problem. Sigmoid and Tanh are used in the output layer for binary and multi-class classification, respectively.

4. Training and Testing:

   During training, the ANN learns to make predictions and adapt its weights to minimize the error on the training data.After training, the model is evaluated on the testing data to assess its performance on unseen examples.Common evaluation metrics for classification problems include accuracy, precision, recall, F1-score, and confusion matrices.

5. Hyperparameters and Architecture:

   Designing a neural network involves choosing the number of layers, the number of neurons in each layer, the activation functions, and other hyperparameters. Hyperparameter tuning is crucial for finding the best-performing model.

6. Overfitting:

   Neural networks are prone to overfitting, where the model becomes too complex and fits the training data noise. Techniques like regularization and early stopping are used to prevent overfitting.

Code :-

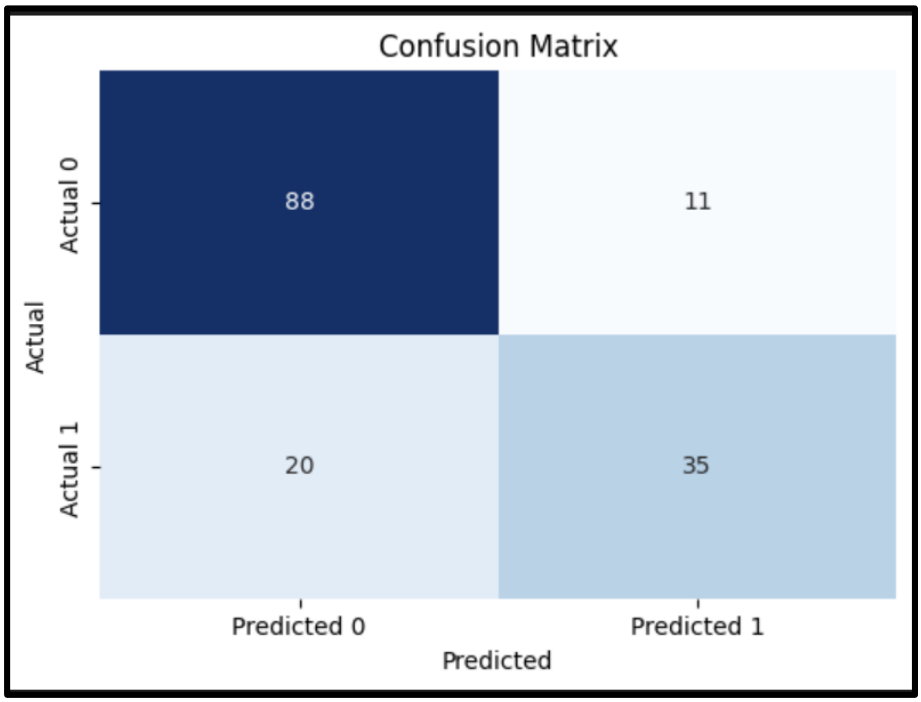
```
import numpy as np
import tensorflow as tf
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, confusion_matrix

# Load the dataset from a CSV file
data = pd.read_csv("D:\Coding\diabetes.csv")

# Separate the features (X) and target (y)
X = data.drop(columns=["Outcome"])
```

```
y = data["Outcome"]

# 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)

# Normalize the data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Build a simple feedforward neural network using TensorFlow
model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(X_train.shape[1],)),  # Input layer
    tf.keras.layers.Dense(8, activation='relu'),
    tf.keras.layers.Dense(10, activation='relu'),
    tf.keras.layers.Dense(10, activation='relu'),  # Hidden layer with 64 units
and ReLU activation
    tf.keras.layers.Dense(1, activation = 'relu')  # Output layer (1 unit for
binary classification)
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=100, batch_size=32, verbose=1)

# Evaluate the model on the test set
y_pred = (model.predict(X_test) > 0.5).astype(int)  # Convert probabilities to
binary predictions

accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)
confusion = confusion_matrix(y_test, y_pred)

# Print the results
print(f'Test Accuracy: {accuracy}')
print(f'Precision: {precision}')
print(f'Recall: {recall}')
print(f'F1 Score: {f1}')

# Create a heatmap for the confusion matrix
```

```
plt.figure(figsize=(6, 4))
sns.heatmap(confusion, annot=True, fmt="d", cmap="Blues", cbar=False,
            xticklabels=['Predicted 0', 'Predicted 1'],
            yticklabels=['Actual 0', 'Actual 1'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

Output: -

```
Test Accuracy: 0.7727272727272727
Precision: 0.6785714285714286
Recall: 0.6909090909090909
F1 Score: 0.6846846846846847
```



Conclusion: -

In conclusion, building an Artificial Neural Network (ANN) with the Backpropagation algorithm is a foundational technique in machine learning and deep learning. ANNs are composed of interconnected neurons organized in layers, and Backpropagation is the process by which the network is trained to minimize errors. The key steps include forward and backward passes, weight updates, and the use of activation functions.The design of the ANN, hyperparameter choices, and the prevention of overfitting are essential aspects of building effective neural networks. Overall, ANNs with Backpropagation have wide-ranging applications in classification, regression, and other machine learning tasks, making them a fundamental tool for data analysis and prediction.