MAR THOMA INSTITUTE OF INFORMATION TECHNOLOGY

CHADAYAMANGALAM P.O, AYUR

(Affiliated to the University of Kerala & Approved by the AICTE)

DEPARTMENT OF COMPUTER APPLICATIONS

Bonafide record of work done by …….....
(Reg.No…….) in the*.....*Lab of Mar Thoma Institute of Information Technology, Ayur during the year…….....

Staff in charge **Head of the Department**

Submitted for the MCA Degree Practical Examination held at Mar Thoma Institute of Information Technology, Ayur on……

Internal Examiner **External Examiner**

INDEX

ITERATIVE DEEPENING

Iterative deepening depth-first search (IDDFS) is an algorithm that is an important part of an Uninformed search strategy just like BFS and DFS. We can define IDDFS as an algorithm of an amalgam of BFS and DFS searching techniques. In IDDFS, We have found certain limitations in BFS and DFS so we have done hybridization of both the procedures for eliminating the demerits lying in them individually. We do a limited depth-first search up to a fixed "limited depth". Then we keep on incrementing the depth limit by iterating the procedure unless we have found the goal node or have traversed the whole tree whichever is earlier.

WORKING OF ITERATIVE DEEPENING

In the uninformed searching strategy, the BFS and DFS have not been so ideal in searching the element in optimum time and space. The algorithms only guarantee that the path will be found in exponential time and space. So we found a method where we can use the amalgamation of space competence of DFS and optimum solution approach of BFS methods, and there we develop a new method called iterative deepening using the two of them. The main idea here lies in utilizing the re-computation of entities of the boundary instead of stocking them up. Every re-computation is made up of DFS and thus it uses less space. Now let us also consider using BFS in iterative deepening search.

- Consider making a breadth-first search into an iterative deepening search.
- We can do this by having aside a DFS which will search up to a limit. It first does searching to a pre-defined limit depth to depth and then generates a route length1.
- This is done by creating routes of length 1 in the DFS way. Next, it makes way for routes of depth limit 2, 3 and onwards.
- It even can delete all the preceding calculation all-time at the beginning of the loop and iterate. Hence at some depth eventually the solution will be found if there is any in the tree because the enumeration takes place in order.

PROGRAM NO :1

AIM:

Implementation Of Iterative Deepening Algorithm

ALGORITHM:

Step 1: For each child of the current node

Step 2: If it is the target node, return

Step 3: If the current maximum depth is reached, return

Step 4: Set the current node to this node and go back to 1.

Step 5: After having gone through all children, go to the next child of the parent (the next sibling)

Step 6: After having gone through all children of the start node, increase the maximum depth and go back to 1.

Step 7: If we have reached all leaf (bottom) nodes, the goal node doesn't exist.

CODE:

```
graph = {
   'A' : ['B','C'],
   'B' : ['D','E'],
   'C' : ['G'],
   'D' : [],
   'E' : ['F'],
   'G' : [],
   'F' : []
   }

def DFS(currentNode,destination,graph,maxDepth):


   print("Checking for destination",currentNode)
   if currentNode == destination:
      return True
```

```
    if maxDepth <=0 :
        return False
    for node in graph[currentNode]:
        #print (node)
        if DFS(node,destination,graph,maxDepth-1):
            return True

    return False

def iterativeDDFS(currentNode,destination,graph,maxDepth):
    for i in range(maxDepth):
        if DFS(currentNode,destination,graph,i):
            return True
    return False

if not iterativeDDFS('A','E',graph, 4):
    print("Path is not available")
else:
    print("A Path Exists")
```

OUT PUT:

```
Checking for destination A
Checking for destination A
Checking for destination B
Checking for destination C
Checking for destination A
Checking for destination B
Checking for destination D
Checking for destination E
A Path Exists
```

A* ALGORITHM

It is a searching algorithm that is used to find the shortest path between an initial and a final point.

It is a handy algorithm that is often used for map traversal to find the shortest path to be taken. A* was initially designed as a graph traversal problem, to help build a robot that can find its own course. It still remains a widely popular algorithm for graph traversal.

It searches for shorter paths first, thus making it an optimal and complete algorithm. An optimal algorithm will find the least cost outcome for a problem, while a complete algorithm finds all the possible outcomes of a problem.

Another aspect that makes A* so powerful is the use of weighted graphs in its implementation. A weighted graph uses numbers to represent the cost of taking each path or course of action. This means that the algorithms can take the path with the least cost, and find the best route in terms of distance and time.

BASIC CONCEPT OF A* ALGORITHM

A heuristic algorithm sacrifices optimality, with precision and accuracy for speed, to solve problems faster and more efficiently.

All graphs have different nodes or points which the algorithm has to take, to reach the final node. The paths between these nodes all have a numerical value, which is considered as the weight of the path. The total of all paths transverse gives you the cost of that route.

Initially, the Algorithm calculates the cost to all its immediate neighbouring nodes n, and chooses the one incurring the least cost. This process repeats until no new nodes can be chosen and all paths have been traversed. Then, you should consider the best path among them. If f(n) represents the final cost, then it can be denoted as :

f(n) = g(n) + h(n), where :

g(n) = cost of traversing from one node to another. This will vary from node to node

h(n) = heuristic approximation of the node's value. This is not a real value but an approximation cost

PROGRAM NO :2

AIM:

Implementation Of A* Algorithm

ALGORITHM:

Step 1: Define A List OPEN

Initally,OPEN Consists Solely Of A Single,Node,The Start Node S

Step 2: if the list empty,return failur and exit

Step 3: remove node n with the smallest value of f(n) from OPEN and move it to list CLOSE

If node n in goal state,return success and exit.

Step 4: Expand node n.

Step 5: if any successor to n is goal node,return success and the solution by tracing the path from goal node to s

Otherwise,go,to step 6

Step 6: For each sucessor node,

Applay the evaluation function f to the node

If the node has not been in either list,add it to OPEN

Step 7: Go back to step 2

CODE:

```
class Graph:
    def __init__(self, adjac_lis):
        self.adjac_lis = adjac_lis

    def get_neighbors(self, v):
        return self.adjac_lis[v]

    # This is heuristic function which is having equal values for all nodes
    def h(self, n):
```

```
        H = {
            'S': 5,
            'A': 3,
            'B': 4,
            'C': 2,
            'D': 6,
            'G': 0
        }
        return H[n]
    def a_star_algorithm(self, start, stop):
        # In this open_lst is a lisy of nodes which have been visited, but who's
        # neighbours haven't all been always inspected, It starts off with the start
#node
        # And closed_lst is a list of nodes which have been visited
        # and who's neighbors have been always inspected
        open_lst = [start]
        closed_lst = []
        # path_len has present distances from start to all other nodes
        # the default value is +infinity
        path_len = {}
        path_len[start] = 0
        # par contains an adjac mapping of all nodes
        par = {}
        par[start] = start
        while len(open_lst) > 0:
            n = None
            # it will find a node with the lowest value of f() -
            for v in open_lst:
                if n == None or path_len[v] + self.h(v) < path_len[n] + self.h(n):
                    n = v;
            if n == None:
                print('Path does not exist!')
```

```
    return None
# if the current node is the stop
# then we start again from start
if n == stop:
    reconst_path = []
    while par[n] != n:
        reconst_path.append(n)
        n = par[n]
    reconst_path.append(start)
    reconst_path.reverse()
    print('Path found: {}'.format(reconst_path))
    return reconst_path

# for all the neighbors of the current node do
for (m, weight) in self.get_neighbors(n):
  # print(m)
 # if the current node is not presentin both open_lst and closed_lst
   # add it to open_lst and note n as it's par
   if m not in open_lst and m not in closed_lst:
       open_lst.append(m)
       par[m] = n
       path_len[m] = path_len[n] + weight

   # otherwise, check if it's quicker to first visit n, then m
   # and if it is, update par data and path_len data
   # and if the node was in the closed_lst, move it to open_lst
   else:
       if path_len[m] > path_len[n] + weight:
           path_len[m] = path_len[n] + weight
           par[m] = n

           if m in closed_lst:
```

```
                closed_lst.remove(m)
                open_lst.append(m)

        # remove n from the open_lst, and add it to closed_lst
        # because all of his neighbors were inspected
        open_lst.remove(n)
        closed_lst.append(n)

    print('Path does not exist!')
    return None
adjac_lis = {
  'S': [('A',1),('G',10)],
  'A': [('B', 2), ('C', 1)],
  'B': [('D', 5)],
  'C': [('D', 3),('G',4)],
  'D': [],
  'G': []
}
graph1 = Graph(adjac_lis)
graph1.a_star_algorithm('A', 'G')
```

OUT PUT:

```
Path found: ['A', 'C', 'G']
```

K-NEAREST NEIGHBOR (KNN)

- K-Nearest Neighbour is one of the simplest Machine Learning algorithms based on Supervised Learning technique.
- K-NN algorithm assumes the similarity between the new case/data and available cases and put the new case into the category that is most similar to the available categories.
- K-NN algorithm stores all the available data and classifies a new data point based on the similarity. This means when new data appears then it can be easily classified into a well suite category by using K- NN algorithm.
- K-NN algorithm can be used for Regression as well as for Classification but mostly it is used for the Classification problems.
- K-NN is a **non-parametric algorithm**, which means it does not make any assumption on underlying data.
- It is also called a **lazy learner algorithm** because it does not learn from the training set immediately instead it stores the dataset and at the time of classification, it performs an action on the dataset.
- KNN algorithm at the training phase just stores the dataset and when it gets new data, then it classifies that data into a category that is much similar to the new data.

Advantages of KNN Algorithm:

- It is simple to implement.
- It is robust to the noisy training data
- It can be more effective if the training data is large.

Disadvantages of KNN Algorithm:

- Always needs to determine the value of K which may be complex some time.
- The computation cost is high because of calculating the distance between the data points for all the training samples.

Steps to implement the K-NN algorithm:

- Data Pre-processing step
- Fitting the K-NN algorithm to the Training set
- Predicting the test result
- Test accuracy of the result(Creation of Confusion matrix)
- Visualizing the test set result.

PROGRAM:3

AIM:

Implementation of KNN Algorithm

ALGORITHM:

Step 1: Import necessary libraries

Step 2: Read dataset from Data/diabetes.csv

Step3: create a data frame with all training data except the target column and separate target values

Step 4: **Split dataset**

Split the dataset into train and test data. Split up our dataset into inputs (X) and our target (y).

Step 4: **Training Model**

Here we are using KNeighboursClassifier imported from the module sklearn.neighbours library for implementing the k-nearest neighbors vote. First, we will create a new k-NN classifier and set ‘n_neighbors’ to 10. The number 'k' is an integer value specified by the user. We calculate the distances between the points of the sample and the object to be classified. To determine the similarity between two instances, we will use the Euclidean distance.

Next, we need to train the model. In order to train our new model, we will use the ‘fit’ function and pass in our training data as parameters to fit our model to the training data.

Step 5: Running Predictions

Once the model is trained, we can use the ‘predict’ function on our model to make predictions on our test data.

Step 6: Checking accuracy

Now let’s see how accurate our model is on the full test set. To do this, we will use the ‘score’ function and pass in our test input and target data to see how well our model predictions match up to the actual results.

CODE:

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

data = pd.read_csv("Data/apples_and_oranges.csv")
'''
Splitting the data set into two. If test_size is float, should be between 0.0 and 1.0 and
represent the proportion of the dataset to include in the test split.'''
training_set, test_set = train_test_split(data, test_size=0.4,,random_state=1)
X_train = training_set.iloc[:, 0:2].values
Y_train = training_set.iloc[:, 2].values
X_test = test_set.iloc[:, 0:2].values
Y_test = test_set.iloc[:, 2].values
print("Initial Test Set\n", test_set)
classifier = KNeighborsClassifier(n_neighbors=15)
classifier.fit(X_train, Y_train)
Y_pred = classifier.predict(X_test)
test_set["Predictions"] = Y_pred
print("\nTest Set after Prediction\n\n", test_set)
accuracy=classifier.score(X_test, Y_test)
print("\nAccuracy of KNN for the Given Dataset : ", accuracy)
```

OUT PUT:

Test Set after prediction

	Weight	Size	Class	predictions
0	69	4.39	orange	orange
3	72	5.85	apple	apple
38	70	5.59	apple	orange
8	74	5.36	apple	apple
20	66	4.13	orange	orange
24	67	4.25	orange	orange
10	73	5.79	apple	apple
13	68	4.47	orange	orange
9	65	4.27	orange	orange
28	74	5.25	apple	apple
18	67	4.18	orange	orange
15	65	4.48	orange	orange
23	68	4.08	orange	orange
11	70	5.47	apple	orange
36	69	4.76	orange	orange
39	73	5.03	apple	apple

Accuracy of KNN for the given dataset: 0.875

SUPPORT VECTOR MACHINE

Support Vector Machine or SVM is one of the most popular Supervised Learning algorithms, which is used for Classification as well as Regression problems. However, primarily, it is used for Classification problems in Machine Learning.

The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyper plane.

SVM chooses the extreme points/vectors that help in creating the hyper plane. These extreme cases are called as support vectors, and hence algorithm is termed as Support Vector Machine. Consider the below diagram in which there are two different categories that are classified using a decision boundary or hyper plane.

SVM WORKING

Just for the sake of understanding, we will leave the machines out of the picture for a minute. Now how would a human being like you and me classify a set of objects scattered on the surface of a table? Of course we will consider all their physical and visual characteristics and then identify based on our prior knowledge. We can easily identify and distinguish apples and oranges based on their colour, texture, shape etc.

Now bringing back the machines, how would a machine identify an apple or an orange. Not surprisingly, it is based on the characteristics that we provide the machine with. It can be size, shape, weight etc. The more features we consider the easier it is to identify and distinguish both.

For the time being, we will just focus on the weight and size(diameter) of apples and oranges. Now how would a machine using SVM, classify a new fruit as either apple or orange just based on the data on the size and weights of some 20 apples and oranges that were observed and labelled? The below image depicts how.

SVM CAN BE OF TWO TYPES:

Linear SVM: Linear SVM is used for linearly separable data, which means if a dataset can be classified into two classes by using a single straight line, then such data is termed as

linearly separable data, and classifier is used called as Linear SVM classifier.

Non-linear SVM: Non-Linear SVM is used for non-linearly separated data, which means if a dataset cannot be classified by using a straight line, then such data is termed as non-linear data and classifier used is called as Non-linear SVM classifier.

PROGRAM:4

AIM:

Implementation Of SVM Algorithm

ALGORITHM:

Step 1: Import necessary libraries such as pandas, svm model etc

Step 2: Read dataset from Data/apples_and_oranges.csv

Step 3: **Split dataset**

Split the dataset into train and test data. iloc can be used to fetch records based on the index values from the datasets and split up our dataset into inputs (X) and our target (y).

Step 4: Create a svm Classifier

classifier = SVC(kernel='linear')
We're going to be using the SVC (support vector classifier) SVM (support vector machine). Our kernel is going to be linear. The objective of a Linear SVC (Support Vector Classifier) is to fit to the data provided, returning a "best fit" hyperplane that divides, or categorizes, our data. From there, after getting the hyperplane, we can then feed some features to our classifier to see what the "predicted" class is.

Step 5: **Training Model**

Train the model using the training sets
Classifier..fit(X_train, y_train)

Step 5: Running Predictions

Once the model is trained, we can use the 'predict' function on our model to make predictions on our test data.

Step 6: Checking accuracy

Now let's see how accurate our model is on the full test set. To do this, we will use the 'score' function and pass in our test input and target data to see how well our model predictions match up to the actual results.

CODE:

```
import pandas as pd
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC

data = pd.read_csv("Data/apples_and_oranges.csv")
'''
Splitting the data set into two. If test_size is float, should be between 0.0 and 1.0 and
represent the proportion of the dataset to include in the test split.'''
training_set, test_set = train_test_split(data, test_size=0.2,random_state=1)
X_train = training_set.iloc[:, 0:2].values
Y_train = training_set.iloc[:, 2].values
X_test = test_set.iloc[:, 0:2].values
Y_test = test_set.iloc[:, 2].values
print("Initial Test Set\n", test_set)
classifier = SVC(kernel='linear')
classifier.fit(X_train, Y_train)
Y_pred = classifier.predict(X_test)
test_set["Predictions"] = Y_pred
print("\nTest Set after Prediction\n\n", test_set)
cm = confusion_matrix(Y_test, Y_pred)
accuracy = float(cm.diagonal().sum()) / len(Y_test)
print("\nAccuracy of SVM For The Given Dataset : ", accuracy)
```

OUT PUT:

initial Test set

	Weight	Size	Class
2	65	4.09	orange
31	66	4.68	orange
3	72	5.85	apple
21	70	4.83	orange
27	70	4.22	orange
29	71	5.26	apple
22	69	4.61	orange
39	73	5.03	apple

Test Set after prediction

	Weight	Size	Class	predictions
2	65	4.09	orange	orange
31	66	4.68	orange	orange
3	72	5.85	apple	apple
21	70	4.83	orange	apple
27	70	4.22	orange	orange
29	71	5.26	apple	apple
22	69	4.61	orange	orange
39	73	5.03	apple	apple

Accuracy of SVM for the given dataset: 0.875

RANDOM FOREST ALGORITHM

Random Forest is a popular machine learning algorithm that belongs to the supervised learning technique. It can be used for both Classification and Regression problems in ML. It is based on the concept of ensemble learning, which is a process of combining multiple classifiers to solve a complex problem and to improve the performance of the model.

As the name suggests, "Random Forest is a classifier that contains a number of decision trees on various subsets of the given dataset and takes the average to improve the predictive accuracy of that dataset." Instead of relying on one decision tree, the random forest takes the prediction from each tree and based on the majority votes of predictions, and it predicts the final output.

ADVANTAGES OF RANDOM FOREST

- Random Forest is capable of performing both Classification and Regression tasks.
- It is capable of handling large datasets with high dimensionality.
- It enhances the accuracy of the model and prevents the over fitting issue.

DISADVANTAGES OF RANDOM FOREST

- Although random forest can be used for both classification and regression tasks, it is not more suitable for Regression tasks.

PROGRAM:5

AIM:

Implementation Of Random Forest Algorithm

ALGORITHM:

Step 1: Import necessary libraries

Step 2: Read dataset from Data/apples_and_oranges.csv

Step 3: **Split dataset**

Split the dataset into train and test data. iloc can be used to fetch records based on the index values from the datasets and split up our dataset into inputs (X) and our target (y).

Step 4: **Training Model**

We can train our random forests to solve this classification problem

For classification,RandomForestClassifier class of the sklearn.ensemble library is used. A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting.RandomForestClassifier class takes as a parameter.

The parameter defines the number of trees in our random forest. We will start with 100 trees again. In order to train our new model, we will use the 'fit' function and pass in our training data as parameters to fit our model to the training data.

Step 5: Running Predictions

Once the model is trained, we can use the 'predict' function on our model to make predictions on our test data.

Step 6: Checking accuracy

Now let's see how accurate our model is on the full test set. To do this, we will use the 'score' function and pass in our test input and target data to see how well our model predictions match up to the actual results.

CODE:

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

data = pd.read_csv("Data/apples_and_oranges.csv")
'''
Splitting the data set into two. If test_size is float, should be between 0.0 and 1.0 and
represent the proportion of the dataset to include in the test split.'''
training_set, test_set = train_test_split(data, test_size=0.2,,random_state=1)
X_train = training_set.iloc[:, 0:2].values
Y_train = training_set.iloc[:, 2].values
X_test = test_set.iloc[:, 0:2].values
Y_test = test_set.iloc[:, 2].values
print("Initial Test Set\n", test_set)
classifier = RandomForestClassifier(n_estimators=100)
classifier.fit(X_train, Y_train)
Y_pred = classifier.predict(X_test)
test_set["Predictions"] = Y_pred
print("\nTest Set after Prediction\n\n", test_set)
accuracy=classifier.score(X_test, Y_test)
print("\nAccuracy of Random Forest for the Given Dataset : ", accuracy)
```

OUT PUT:

initial Test set

	Weight	Size	Class
31	66	4.68	orange
35	69	4.11	orange
25	71	5.35	apple
34	68	4.83	orange
11	70	5.47	apple
5	73	5.68	apple
1	69	4.21	orange
8	74	5.36	apple

Test Set after prediction

	Weight	Size	Class	predictions
31	66	4.68	orange	orange
35	69	4.11	orange	orange
25	71	5.35	apple	apple
34	68	4.83	orange	orange
11	70	5.47	apple	apple
5	73	5.68	apple	apple
1	69	4.21	orange	orange
8	74	5.36	apple	apple

Accuracy of Random Forest for the given dataset: 1.0

LINEAR REGRESSION

Linear Regression is a machine learning algorithm based on supervised learning. It performs a regression task. Regression models a target prediction value based on independent variables. Linear regression algorithm shows a linear relationship between a dependent (y) and one or more independent (y) variables, hence called as linear regression. Since linear regression shows the linear relationship, which means it finds how the value of the dependent variable is changing according to the value of the independent variable.

Mathematically, we can represent a linear regression as:

$$y= a_0+a_1x+ \varepsilon$$

Y= Dependent Variable (Target Variable)

X= Independent Variable (predictor Variable)

a0= intercept of the line (Gives an additional degree of freedom)

a1 = Linear regression coefficient (scale factor to each input value).

ε = random error

The values for x and y variables are training datasets for Linear Regression model representation.

TYPES OF LINEAR REGRESSION

Linear regression can be further divided into two types of the algorithm:

- **SIMPLE LINEAR REGRESSION:**
  If a single independent variable is used to predict the value of a numerical dependent variable, then such a Linear Regression algorithm is called Simple Linear Regression.
- **MULTIPLE LINEAR REGRESSION:**
  If more than one independent variable is used to predict the value of a numerical dependent variable, then such a Linear Regression algorithm is called Multiple Linear Regression.

PROGRAM:6

AIM:

Implementation of linear regression

ALGORITHM:

Step 1: Loading data

Firstly, we will use the Python Pandas library to read our CSV data.

Step 2: Splitting data into a training set and a test set

We can use the Python scikit-learn train_test_split function to randomly split our data into a training and test set.

Step 3:Data Transformation

Python scikit-learn only accepts the training and test data in a 2-dimensional array format. We have to perform data transformation on our training set and test set.

Step 4: Training Model

Use the Scikit-Learn LinearRegression function to create a model object.

Fit the training set to the model.

Step 5: Predicting Salary using Linear Model

At this stage, we have trained a linear model and we first use it to predict the salary on our training set to see how well it fit on the data. Use the linear model to predict the salary based on the training set. Use the Matplotlib to create a plot to visualize the predicted results.

Step 6: Model Evaluation

Next we will use some quantitative methods to obtain a more precise performance evaluation of our linear model.

Mean Square Error — The average of the squares of the difference between the true values and the predicted values.

Explained Variance Score — A measurement to examine how well a model can handle the variation of values in the dataset.

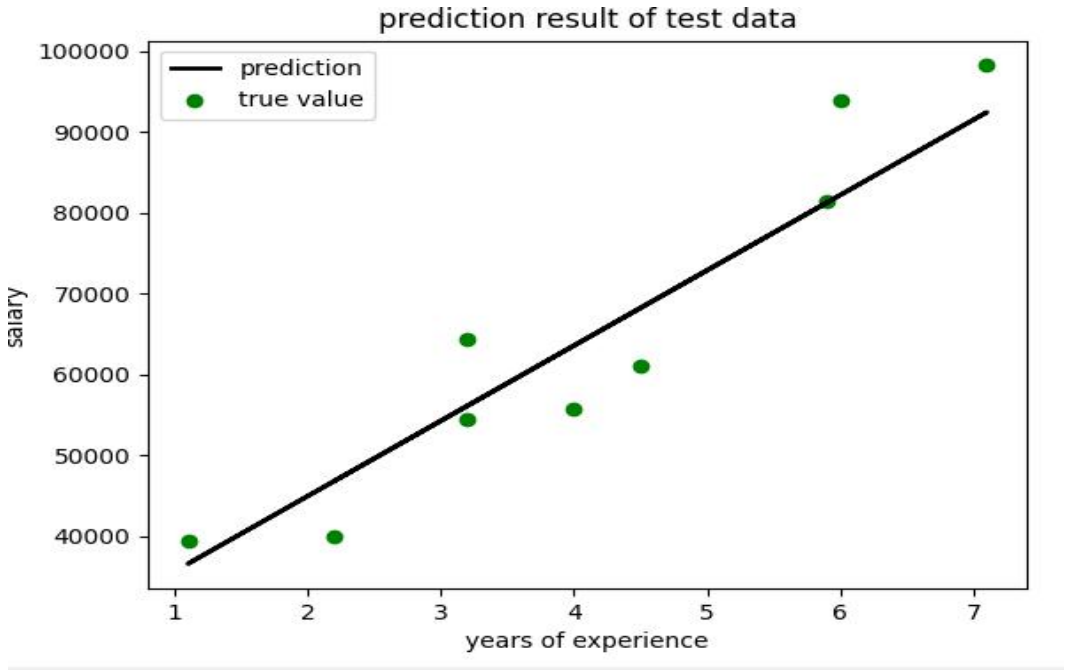
R2 Score — A measurement to examine how well our model can predict values based on the test set (unknown samples). The perfect score is 1.0.

CODE:

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

data = pd.read_csv("Data/Salary_Data.csv")
X = data['YearsExperience']
y = data['Salary']
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.3, random_state=42)
X_train = np.array(X_train).reshape((len(X_train),1))
y_train = np.array(y_train).reshape((len(y_train),1))
X_test = np.array(X_test).reshape(len(X_test), 1)
y_test = np.array(y_test).reshape(len(y_test), 1)
model = linear_model.LinearRegression()
model.fit(X_train, y_train)
model = linear_model.LinearRegression()
model.fit(X_train, y_train)
y_train_pred = model.predict(X_train)
plt.figure()
plt.scatter(X_train, y_train, color='blue', label="True Value")
plt.plot(X_train, y_train_pred, color='black', linewidth=2, label="Prediction")
plt.xlabel("Years of Experiences")
plt.ylabel("Salary")
plt.title('Prediction Result of Training Data')
```

```
plt.legend() plt.show()
y_test_pred = model.predict(X_test)
plt.figure()
plt.scatter(X_test, y_test, color='green', label='True Value') plt.plot(X_test,
y_test_pred, color='black', linewidth=2, label='Prediction')
plt.xlabel("Years of Experiences")
plt.ylabel("Salary")
plt.title('Prediction Result of Test data')
plt.legend()
plt.show() print("lope",model.coef_)
print("Intercept",model.intercept_)
print("Mean squared error =", round(sm.mean_squared_error(y_test, y_test_pred), 2))
print("Explain variance score =", round(sm.explained_variance_score(y_test, y_test_pred),2))
print("R2 score =", round(sm.r2_score(y_test, y_test_pred), 2))
```

OUT PUT:



Slope [[9320.02614507]]

Intercept [26313.22881299]

mean squared error= 45550929.47

explain variance score=0.89

R2 score=0.89

PROGRAM:7

AIM:

Implementation of Linear Discriminant Analysis(LDA)

ALGORITHM:

Step1: Import the required libraries:

Step 2: Define a function get_color(feature) that returns the color based on the feature value:

- If feature is 'Iris-setosa', return 'red'
- If feature is 'Iris-versicolor', return 'black'
- If feature is any other value, return 'blue'

Step 3: Read the data from the "Data/Iris.csv" file into a pandas DataFrame called data.

Step 4: Split the data into a training set and a test set using train_test_split with a test size of 0.4.

Step 5: Extract the features and labels from the training set and test set

Step 6 : Apply Linear Discriminant Analysis (LDA) to reduce the dimensionality of the training and test sets to 2:

Step 7: Plot a scatter plot of the transformed training data using plt.scatter:

Step 8 : Create an instance of RandomForestClassifier called classifier.

Step 9: Train the classifier using the transformed training data and corresponding labels**:**

Step 10: Predict the labels for the transformed test data using the trained classifier:

Step 11: Calculate the accuracy of the classifier on the test data:

Step 12: Print the accuracy of the Random Forest classifier for the given dataset:

CODE:

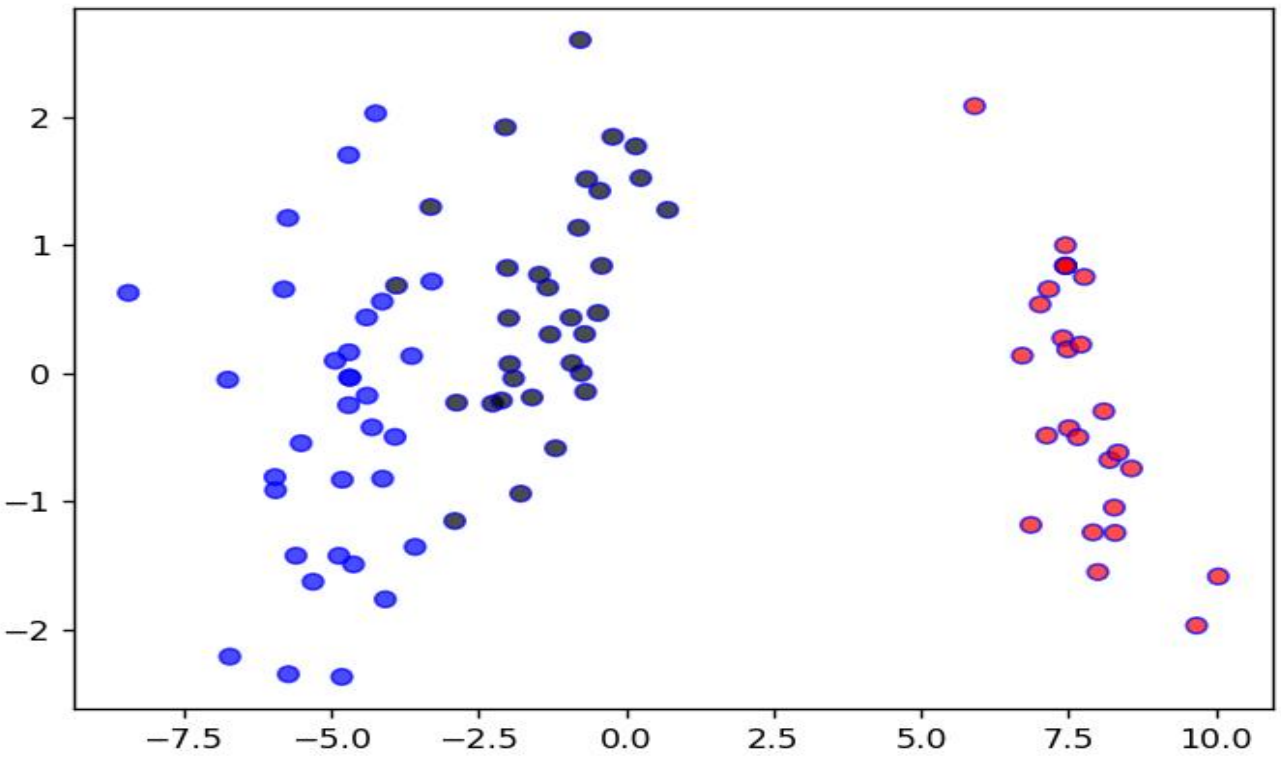
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn import datasets
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.model_selection import train_test_split

def get_color(feature):
        return "red" if feature == 'Iris-setosa' else 'black' if feature == 'Iris-versicolor' else 'blue'
data = pd.read_csv("Data/Iris.csv")
print(data)
training_set, test_set = train_test_split(data, test_size=0.4)
X_train = training_set.iloc[:, 0:4].values
Y_train = training_set.iloc[:, 4].values
X_test = test_set.iloc[:, 0:4].values
Y_test = test_set.iloc[:, 4].values

# apply Linear Discriminant Analysis
lda = LinearDiscriminantAnalysis(n_components=2)
X_train = lda.fit_transform(X_train, Y_train)
X_test = lda.transform(X_test)
colors = np.array([get_color(feature) for feature in Y_train])

# plot the scatter plot
plt.scatter( X_train[:, 0], X_train[:, 1], c=colors, cmap='rainbow',  alpha=0.7, edgecolors='b')
plt.show()

# classify using random forest classifier
classifier = RandomForestClassifier()
classifier.fit(X_train, Y_train)
```

```
Y_pred = classifier.predict(X_test)
accuracy=classifier.score(X_test, Y_test)
print("\nAccuracy of Random Forest for the Given Dataset : ", accuracy)
```

OUTPUT:

	SepalLengthCm	SepalWidthCm	PetalLengthCm	PetalWidthCm	Species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
..	...	...	...	...	...
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

[150 rows x 5 columns]



PROGRAM:8

AIM:

Implementation of Lasso Regression

ALGORITHM:

Step 1: Import the required libraries:

Step 2: Load the Iris dataset using datasets.load_iris() and assign it to the variable **bh**.

Step 3: Assign **bh.data** to **X** to store the features and **bh.target** to **y** to store the labels.

Step 4: Split the data into training and test sets using train_test_split:

Call **train_test_split(X, y, test_size=0.5, random_state=42)**.

Assign the returned values to **X_train**, **X_test**, **y_train**, and **y_test**.

Step 5: Create an instance of the Lasso regression model:

Initialize **lasso** as **Lasso(alpha=1.0)**.

Step 6: Fit the Lasso model to the training data:

Call **lasso.fit(X_train, y_train)**.

Step 7: Calculate the model score on the test data:

Print the score using **print(lasso.score(X_test, y_test))**

CODE:

```
from sklearn import datasets
from sklearn.linear_model import Lasso
from sklearn.model_selection import train_test_split
bh =datasets.load_iris()
X = bh.data
y = bh.target
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=42)
lasso = Lasso(alpha=1.0)
lasso.fit(X_train, y_train)
print(lasso.score(X_test, y_test))
```

OUTPUT:

```
0.32634596271631844
```

PROGRAM:9

AIM:

Implementation of K Means Algorithm

ALGORITHM:

Step 1: Import the required libraries:

Step 2: Create a dictionary 'data' containing the X and Y coordinates of data points.

Step 3: Create a pandas DataFrame 'df' using the 'data' dictionary.

Step 4: Create an instance of KMeans clustering with n_clusters=4 and fit the data:
Initialize 'kmeans' as 'KMeans(n_clusters=4).fit(df)'.

Step 5: Retrieve the cluster centroids from the fitted KMeans model:
Assign 'centroids' as 'kmeans.cluster_centers_'.

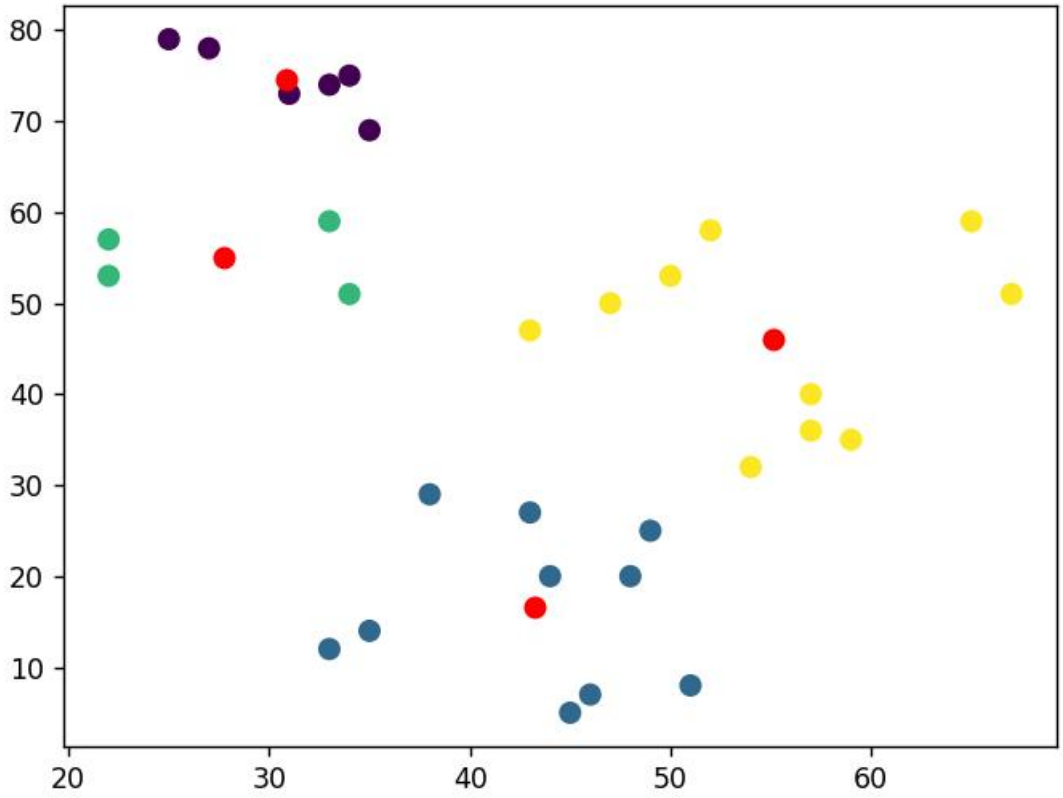
Step 6: Plot the data points and cluster centroids:

CODE:

```
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
data={
'X':[25,34,22,27,33,33,31,22,35,34,67,54,57,43,50,57,59,52,65,47,49,48,35,33,44,45,38,43,51,46],
'Y':[79,51,53,78,59,74,73,57,69,75,51,32,40,47,53,36,35,58,59,50,25,20,14,12,20,5,29,27,8,7] }
df=pd.DataFrame(data)
kmeans=KMeans(n_clusters=4).fit(df)
centroids=kmeans.cluster_centers_
print(centroids)
plt.scatter(df['X'],df['Y'],c=kmeans.labels_.astype(float),s=50,alpha=1.0)
plt.scatter(centroids[:,0],centroids[:,1],c='red',s=50)
plt.show()
```

OUTPUT:

```
[[30.83333333 74.66666667]
 [43.2        16.7       ]
 [27.75       55.        ]
 [55.1        46.1       ]]
```



PROGRAM:10

AIM:

Implementation of hill climbing using travelling salesman problem

ALGORITHM:

Step 1: Import the required library:

Step 2: Define the function randomSolution(tsp):

- Initialize an empty list cities containing indices from 0 to len(tsp) - 1.
- Initialize an empty list solution.
- Iterate len(tsp) times:
  - Generate a random index randomCity using random.randint(0, len(cities) - 1).
  - Append the city at index randomCity to solution.
  - Remove the city at index randomCity from cities.
- Return solution.

Step 3: Define the function routeLength(tsp, solution):

- Initialize routeLength to 0.
- Iterate over the indices of solution:
  - Add the distance between the current city and the previous city to routeLength using tsp[solution[i - 1]][solution[i]].
- Return routeLength.

Step 4: Define the function getNeighbours(solution):

- Initialize an empty list neighbours.
- Iterate over the indices of solution:
- Iterate over the indices from i + 1 to len(solution):
  - Create a copy of solution called neighbour.
  - Swap the cities at indices i and j in neighbour.
  - Append neighbour to neighbours.
- Return neighbours.

Step 5: Define the function getBestNeighbour(tsp, neighbours):

- Initialize bestRouteLength with the route length of the first neighbour.
- Initialize bestNeighbour as the first neighbour.
- Iterate over each neighbour in neighbours:

- Calculate the currentRouteLength using routeLength(tsp, neighbour).
- If currentRouteLength is smaller than bestRouteLength, update bestRouteLength and bestNeighbour.

- Return bestNeighbour and bestRouteLength.

Step 6: Define the function hillClimbing(tsp):

- Initialize currentSolution with a random solution using randomSolution(tsp).
- Initialize currentRouteLength as the route length of currentSolution.
- Initialize neighbours with the neighbours of currentSolution using getNeighbours(currentSolution).
- Find the best neighbour and its route length using getBestNeighbour(tsp, neighbours), and assign them to bestNeighbour ,bestNeighbourRouteLength.
- Enter a loop that continues as long as bestNeighbourRouteLength is less than currentRouteLength:
  - Update currentSolution and currentRouteLength with the values of bestNeighbour and bestNeighbourRouteLength.
  - Recalculate neighbours with the neighbours of currentSolution.
  - Find the best neighbour and its route length using
  - getBestNeighbour(tsp, neighbours), and assign them to bestNeighbour and bestNeighbourRouteLength.
- Return currentSolution and currentRouteLength.

Step 7: Define the function main():

- Create the tsp matrix with the distances between cities.
- Print the result of hillClimbing(tsp).

CODE:

```
import random
def randomSolution(tsp):
        cities = list(range(len(tsp)))
        solution = []
        for i in range(len(tsp)):
                randomCity = cities[random.randint(0, len(cities) - 1)]
                solution.append(randomCity)
                cities.remove(randomCity)
        return solution

def routeLength(tsp, solution):
        routeLength = 0
        for i in range(len(solution)):
                routeLength += tsp[solution[i - 1]][solution[i]]
        return routeLength

def getNeighbours(solution):
         neighbours = []
        for i in range(len(solution)):
                for j in range(i + 1, len(solution)):
                        neighbour = solution.copy()
                         neighbour[i] = solution[j]
                        neighbour[j] = solution[i]
                        neighbours.append(neighbour)
        return neighbours

def getBestNeighbour(tsp, neighbours):
        bestRouteLength = routeLength(tsp, neighbours[0])
        bestNeighbour = neighbours[0]
        for neighbour in neighbours:
                currentRouteLength = routeLength(tsp, neighbour)
                if currentRouteLength < bestRouteLength:
```

```
                    bestRouteLength = currentRouteLength
                    bestNeighbour = neighbour
        return bestNeighbour, bestRouteLength

def hillClimbing(tsp):
        currentSolution = randomSolution(tsp)
        currentRouteLength = routeLength(tsp, currentSolution)
        neighbours = getNeighbours(currentSolution)
        bestNeighbour, bestNeighbourRouteLength = getBestNeighbour(tsp, neighbours)
        while bestNeighbourRouteLength < currentRouteLength:
                currentSolution = bestNeighbour
                currentRouteLength = bestNeighbourRouteLength
                neighbours = getNeighbours(currentSolution)
                bestNeighbour, bestNeighbourRouteLength = getBestNeighbour(tsp, neighbours)
        return currentSolution, currentRouteLength

def main():
        tsp = [
                [0, 400, 500, 300],
                [400, 0, 300, 500]      ,
                [500, 300, 0, 400],
                [300, 500, 400, 0]
                ]
        print(hillClimbing(tsp))
if __name__ == "__main__":
        main()
```

OUTPUT:

```
([3, 2, 1, 0], 1400)
```