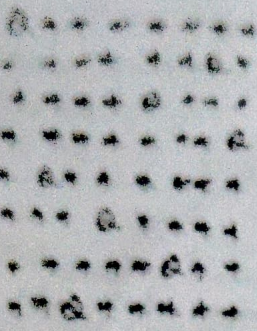
N - Queens Problem

Code:

```
N = int(input("Enter the number of queens: "))
board = [[0] * N for _ in range(N)]

def is_safe(board, row, col):
    return all(board[row][i] == 0 for i in range(col)) and \
           all(board[i][j] == 0 for i, j in zip(range(row-1, -1, -1),
                                                range(col, -1, -1))) \
                                                and \
           all(board[i][j] == 0 for i, j in zip(range(row, N),
                                                range(col, -1, -1)))

def solve_nqueens(board, col=0):
    if col == N: return True
    for i in range(N):
        if is_safe(board, i, col):
            board[i][col] = 1
            if solve_nqueens(board, col+1): return True
            board[i][col] = 0
    return False

if solve_nqueens(board):
    for row in board:
        print(' '.join('Q' if x else '*' for x in row))

else:
    print("No solution exists")
```

Output:

Enter the number of Queens: 8



RESULT

Thus the above program to solve the N-Queens problem has been executed successfully.

DFS SEARCH

Code:

```
def dfs_recursive(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start)
    for neighbour in graph[start]:
        if neighbour not in visited:
            dfs_recursive(graph, neighbour, visited)

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

print("DFS RECURSIVE")
dfs_recursive(graph, 'A')

def dfs_iterative(graph, start):
    visited = set()
    stack = [start]
    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            print(vertex)
            visited.add(vertex)
            stack.extend(neighbour for neighbour in
                         graph(vertex) if neighbour
                         not in visited)

print("DFS ITERATIVE")
dfs_iterative(graph, 'A')
```

OUTPUT:

A B D E F C

RESULT:

Thus the above python code for dfs executed successfully

A* SEARCH

Code:

```
import heapq

def heuristic (a,b):
    return abs ( a[0] - b[0]) + abs ( a[1] - b[1]

def astar ( grid , start , end):

    open list = [ ]
    heapq. heap push (open_list , (0 + heuristic ( start, end ) ,0, start )

    g_cost ={ start:0}

    came_from = { }

    directions =[(0,1), (1,0), (0,-1), (-1,0)]

    while open_list:
        -1 current_g . current = heapq. heappop ( open_list)

        if current == end:
            path =[ ]
            while current in came_from:
                path . append (current)
                current =came_from [current]
            path . append (start)
            return path [::-1]
        for dx, dy in directions:
            neighbour = ( current [0] + dx, current [1]+dy)
            if (0<= neighbour [0] < len (grid) and
                0<= neighbour [1] < len (grid[0]) and
                grid [neighbour [0] ] [neighbour [1]] == 0):
            tentative_g = current_g +1
            if neighbour not in g_cost or tentative_g <
                                    g cost [neighbour ]

                g_cost [ neighbour ]= tentative_g
                g_cost [ ] = tentative_g + heuristic (neighbour , end)
```

heapq.heappush(open_list, (g_cost, tentative_g, neighbour))

came_from[neighbour] = current.

return None

```
grid = [ [0 0 0 0 0],
         [0 1 1 1 0],
         [0 0 0 1 0],
         [0 1 0 0 0],
         [0 0 0 0 0]
       ]

start = (0,0)

end = (4,4)

path = astar(grid, start, end)

print("Path found", path)
```

OUTPUT:

Path found: [(0,0), (0,1), (0,2), (0,3), (0,4), (1,4), (2,4), (3,4), (4,4)]

RESULT:

AO* SEARCH

Code:

```
class graph:
    def __init__(self, graph, heuristic):
        self.graph = graph
        self.heuristic = heuristic
        self.solution = {}

    def ao_star(self, node):
        print(f"Expanding : {node}")
        if node not in self.graph or not self.graph[node]:
            return
        children = self.graph[node]
        best_path = None
        min_cost = float('inf')
        for group in children:
            cost = sum(self.heuristic[child] for child in group)
            if cost < min_cost:
                min_cost = cost
                best_path = group
        self.solution[node] = best_path
        print(f"Best Path for {node}: {best_path}
                with cost {min_cost}")
        for child in best_path:
            self.ao_star(child)

    def get_solution(self):
        return self.solution

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

```
heuristic = {'A': 0, 'B': 1, 'C': 2, 'D': 4, 'E': 1, 'G': 3,
             'H': 5}

graph_obj = Graph(graph, heuristic)
graph_obj = AOStar('A')
solution = graph_obj.get_solution()
print("solution", solution)
```

OUTPUT:

```
Expanding: A
Best Path for A: ['B', 'C']
Expanding: B
Best path for B: ['E']
Expanding: E
expanding: C
Best path C: ['G']
expanding: G
Solution: {A: [B,C], 'B': ['E'], 'C': ['G']}
```

RESULT:

Thus the python program for A* Algorithm was executed successfully.

Implementation of Decision Tree Classification Techniques

Code:

```
from sklearn.tree import DecisionTreeClassifier
import numpy as np
x = np.array([[170, 65, 42], [180, 75, 44], [160, 50, 39],
              [175, 70, 43], [165, 55, 39], [180, 80, 45]])
y = np.array([0, 1, 0, 1, 0, 1])
clf = DecisionTreeClassifier()
clf.fit(x, y)
new_data = np.array([[172, 68, 43]])
prediction = clf.predict(new_data)
print("Predicted gender: ", "Male" if prediction[0] == 1 else "female")
```

OUTPUT:

Predicted gender : Male

RESULT:

Thus implementation of Decision Tree classification technique is done & executed successfully.

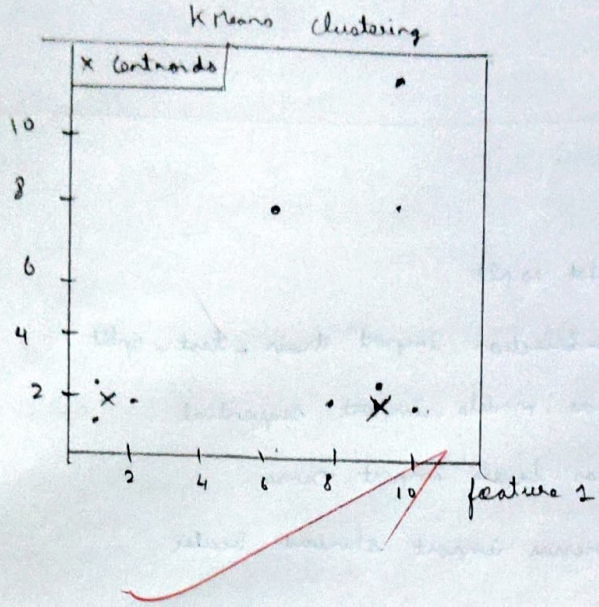
Implementation of Clustering Techniques

K - Means

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
X = np.array([[1,2],[1.5,1.8],[5,8],[8,8],[1,0.6],
              [9,11],[8,2],[10,2],[9,3]])
Kmeans = KMeans(n_clusters=3)
Kmeans.fit(X)
Centroids = Kmeans.cluster_centers_
labels = Kmeans.labels_
plt.scatter(X[:,0], X[:,1], c=labels, cmap='viridis',
            marker='x', label='centroids')
plt.title('Kmeans Clustering')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.legend()
plt.show()
```

OUTPUT:



RESULT:

Thus the above program to implement K-means has been executed successfully.

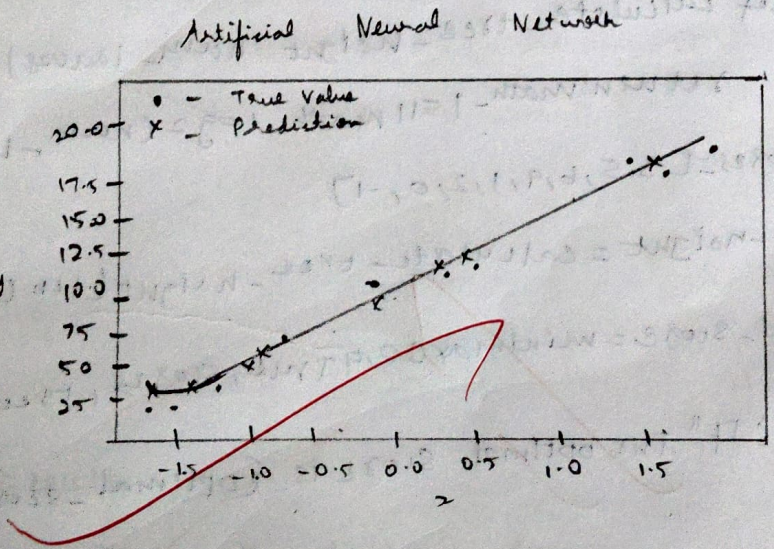
Implementation of Artificial Neural Network in Regression

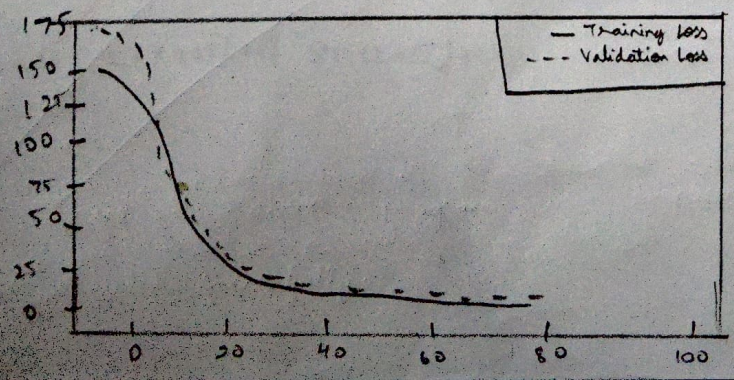
Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.preprocessing import StandardScaler

np.random.seed(42)
x = np.linspace(0,10,100)
y = 2*x + 1 + np.random.normal(0,1,100)
X = x.reshape(-1,1)
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.2,
                                   random_state=42)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
model = Sequential()
model.add(Dense(units=64, activation='relu', input_dim=1))
model.add(Dense(units=32, activation='relu'))
model.add(Dense(units=1))
model.compile(optimizer='adam', loss='mean_squared_error')
history = model.fit(X_train, y_train, epochs=100, batch_size=10,
                    validation_split=0.2)
y_pred = model.predict(X_test)
```

```
plt.scatter(X_test, y_test, colour='blue', label='True_Values')
plt.scatter(X_test, y_pred, colour='red', label='Predictions')
plt.plot(X_test, y_pred, colour='red', linewidth=2)
plt.title('Artificial Neural Network Regression')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.show()

plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

OUTPUT:





MINIMAX

Aim:

To implement minimax algorithm.

Program

```
Import math
def minimax(depth, node_index, is_maximizer,
                                    scores, height):
    if depth == height:
        return scores[node_index]
    if is_maximizer:
        return max(minimax(depth+1, node_index*2,
            False, scores, height), minimax(depth,
            node_index*2+1, false score, height))
    else
        return max(minimax(depth+1, node_index*2,
            true, s[mes, height], minmax, minimax
            (depth+1, node_index*2+1)
                true, scores, height))
    def calculate_tree_height(num_leaves)
        return math.ceil(math.log2(num_leaves))
    scores = [3.5, 6, 9, 1, 2, 0, -1]
tree_height = calculate_tree_height(len(scores))
optimal_score = minimax(0, 0, True, scores, tree_height
Print("The optimal score is {optimal_score}")
```

output:

The optimal score is =5.

Result

Thus the decision tree program has been executed successfully.

Introduction to prolog

Aim

To learn prolog terminology and write basic programs.

TERMINOLOGIES

1) Atomic terms.

They are usually strings made up of leaves and uppercase letters. digits and the underscore, starting with a lowercase letter.

Eg. dog. ab_C_321

2) Variables.

They are string of letters, digits and the underscore, starting with a capital letter or an underscore,

Eg: Dog. Apple_420

3) Compond terms.

Compond terms are made up of a prolog atom and a number of arguments enclosed in Paranthesis and seperates by comas.

Eg: is_bigger(elephant, X). f(g(X, _), Y)

4) Fact

A fact is a prediction followed by a dot

Eg: bigger_animal(whale). life_a_beautiful

5) Rules

A rules consists of a head and body

Eg: u_smaller(Y, Y. Yes = bigger Y, X) aunt(Aunt, child
== sister(Aunt, parent) parent
(parent, child)

Sourse code:

KB1

woman(mia)

woman(jody).

woman(yolanda)

Playsairguitar(jody)

party.

query1 = ? - woman(mia)

query2 = ? - playsAirguitar(mia)

query3 = ? - party

query4 = ? - concert;

~~query~~

output:

? - woman(mia)

true

? - playsairguitar(mia)

false

? - wohmat

Error = unknow procedure. concert/0.

KB2

happy(yolanda)

Listens2music(mia).

Listen2music(yolanda) :- happy(yolanda)

playsair guitar(mia) :- listen2music(mia

playsairguitar(yolanda) :- listen2music(yolanda)

output

? - likes(dan,x)

X = sally

? - married(dan, sally)

true

? - married(john, brittney)

false

KB4

food (burger)

food (sandwich)

food (pizza)

lunch (sandwich)

dinner (pizza)

meal (X) = food(X)

output.

? - food (pizza)

true

? - meal (X), lunch (X)

X = sandwich

KB5

owns (jack, car (bmw))

owns (john, car (chevy))

owns (olivia, car (civic))

own (jane, car (chevy))

sedan (car (bmw))

truck (car (chevy))

output

? owns (john, X)

X = car (chevy)

? - own (john, _)

true

Result:

Thus the basic Prolog Programs has been executed successfully.

Prolog – Family tree.

Aim

To develop a family tree program using Prolog with all possible facts, rules and queries.

Source - code

Knowledge-base:

/* Facts: */

male(peter)

male(john)

male(chris)

male(kelvin)

female(betty).

female(jeny)

female(lisa)

female(kellin)

parent of(chris, peter)

parent of(chris, betty)

parent of(kellin, chris)

parent of(jeny, john)

parent of(jeny, kelvin)

/* RULES ==*/

/* son, parent

* son, grandparent */

father(x,y) :- male(y) parent of(x,y)

mother(x,y) :- female(y), parent of(x,y)

grandfather(x,y) :- male(y), parent of(x,z), parent of(z,y)

grand mother(x,y) : female(y), parent of(x,z), parent of(z,y).

brother (X, Y) :- male (Y), father (X, Z) father (Y, co), Z
==w

sister (X, Y) - female (X). father (X, Z), father (Y, co)
Z == w

output.

male (peter)
true

father (chris - peter)
true

father (chris, betty)
false

mother (chris, X)
X = bully

brother (chris, helen)
false.

Result.

Thus prolog for family trees program has been executed successfully.