https://github.com/Moksh081/Python/blob/main/AI%20lab%20ass1.ipynb

8 puzzle blind approach :

```
import sys

import copy


q = []


def compare(s, g):

    if s == g:

        return 1

    else:

        return 0


def find_pos(s):

    for i in range(len(s)):

        for j in range(len(s[0])):

            if s[i][j] == 0:

                return [i, j]


def up(s, pos):

    i = pos[0]

    j = pos[1]


    if i > 0:

        temp = copy.deepcopy(s)

        temp[i][j] = temp[i - 1][j]

        temp[i - 1][j] = 0

        return temp

    else:

        return s
```

```
def down(s, pos):
    i = pos[0]
    j = pos[1]

    if i < 2:
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i + 1][j]
        temp[i + 1][j] = 0
        return temp
    else:
        return s

def right(s, pos):
    i = pos[0]
    j = pos[1]

    if j < 2:
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i][j + 1]
        temp[i][j + 1] = 0
        return temp
    else:
        return s

def left(s, pos):
    i = pos[0]
    j = pos[1]

    if j > 0:
        temp = copy.deepcopy(s)
```

```
        temp[i][j] = temp[i][j - 1]
        temp[i][j - 1] = 0
        return temp
    else:
        return s

def enqueue(s):
    global q
    q = q + [s]

def dequeue():
    global q
    elem = q[0]
    del q[0]
    return elem

def search(s, g):
    curr_state = copy.deepcopy(s)
    if s == g:
        return
    c = 0
    while True:
        pos = find_pos(curr_state)
        new = up(curr_state, pos)
        if new != curr_state:
            if compare(new, g):
                print("Found")
                return
            else:
                enqueue(new)
```

```
        new = down(curr_state, pos)
        if new != curr_state:
            if compare(new, g):
                print("Found")
                return
            else:
                enqueue(new)

        new = right(curr_state, pos)
        if new != curr_state:
            if compare(new, g):
                print("Found")
                return
            else:
                enqueue(new)

        new = left(curr_state, pos)
        if new != curr_state:
            if compare(new, g):
                print("Found")
                return
            else:
                enqueue(new)

        if len(q) > 0:
            curr_state = dequeue()
        else:
            print("Not found")
            return

def main():
```

```
    s = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]
    g = [[2, 8, 1], [0, 4, 3], [7, 6, 5]]
    pos = find_pos(s)
    search(s, g)

if __name__ == "__main__":
    main()
```

WATER JUG :

```
def water_jug_solver(jug1, jug2, aim):
    visited = set()
    stack = [(0, 0)]

    while stack:
        amt1, amt2 = stack.pop()

        if (amt1, amt2) in visited:
            continue

        print(amt1, amt2)
        visited.add((amt1, amt2))

        if amt1 == aim or amt2 == aim:
            print(f"Reached the aim: ({amt1}, {amt2})")
            return True

        # Fill jug1
        stack.append((jug1, amt2))
```

```
        # Fill jug2
        stack.append((amt1, jug2))

        # Empty jug1
        stack.append((0, amt2))

        # Empty jug2
        stack.append((amt1, 0))

        # Pour from jug1 to jug2
        pour_amt = min(amt1, jug2 - amt2)
        stack.append((amt1 - pour_amt, amt2 + pour_amt))

        # Pour from jug2 to jug1
        pour_amt = min(jug1 - amt1, amt2)
        stack.append((amt1 + pour_amt, amt2 - pour_amt))

    return False

# Jug capacities and aim
jug1, jug2, aim = 4, 3, 2

print("Steps:")
water_jug_solver(jug1, jug2, aim)
```

TRAVELLING SALESMAN :

```
def travel(g, v, pos, n, count, cost, dst):
    if count == n and g[pos][s]:
```

```
        cost += g[pos][s]
        dst.append(cost)
        return
    for i in range(n):
        if not v[i] and g[pos][i]:
            v[i] = True
            travel(g, v, i, n, count + 1, cost + g[pos][i], dst)
            v[i] = False

n = 4
g = [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]]

s = int(input("Enter a number between 1 and 4: ")) - 1
v = [False] * n
v[s] = True
dst = []

travel(g, v, s, n, 1, 0, dst)

print(dst)
print(min(dst))
```

BFS :

```
#bfs
import collections

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

```
def bfs(graph, root):
    visited = set()
    queue = collections.deque([root])

    while queue:
        vertex = queue.popleft()
        visited.add(vertex)

        for i in graph[vertex]:
            if i not in visited:
                queue.append(i)

        print(vertex, end=" ")

bfs(graph, 'A')
```

DFS

```
#dfs in python

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

visited = []

def dfs(visited, graph, root):
    if root not in visited:
        print(root)
        visited.append(root)
        for neighbor in graph[root]:
            dfs(visited, graph, neighbor)
```

```
dfs(visited, graph, 'A')
```

Block problem with dfs

```
def dfs(start, goal, visited=None):
    if visited is None:
        visited = set()

    if start == goal:
        return [start]

    visited.add(start)

    for neighbor in get_neighbors(start):
        if neighbor not in visited:
            path = dfs(neighbor, goal, visited)
            if path:
                return [start] + path

    return None

def get_neighbors(state):
    neighbors = []
    for i, block in enumerate(state):
        if i == 0:
            neighbors.append(tuple(sorted([block] + list(state[1:]))))
        elif i == len(state) - 1:
            neighbors.append(tuple(sorted(list(state[:-1]) + [block])))
```

```
        else:
            neighbors.append(tuple(sorted(list(state[:i]) + [block] + list(state[i+1:]))))
    return neighbors


start_state = (3, 2, 1)
goal_state = (1, 2, 3)

path = dfs(start_state, goal_state)
if path:
    print("DFS Path:", path)
else:
    print("No path found")
```

Block problem with BFS

```
from collections import deque

def bfs(start, goal):
    visited = set()
    queue = deque([(start, [])])

    while queue:
        state, path = queue.popleft()

        if state == goal:
            return path + [state]

        if state in visited:
```

```
            continue

        visited.add(state)

        for neighbor in get_neighbors(state):
            if neighbor not in visited:
                queue.append((neighbor, path + [state]))

    return None

start_state = (3, 2, 1)
goal_state = (1, 2, 3)

path = bfs(start_state, goal_state)
if path:
    print("BFS Path:", path)
else:
    print("No path found")
```

BLOCK PROBLEM WITH DLS

```
def dls(start, goal, depth_limit, visited=None):
    if visited is None:
        visited = set()

    if start == goal:
        return [start]

    if depth_limit == 0:
```

```
        return None

    visited.add(start)

    for neighbor in get_neighbors(start):
        if neighbor not in visited:
            path = dls(neighbor, goal, depth_limit - 1, visited)
            if path:
                return [start] + path

    return None

start_state = (3, 2, 1)
goal_state = (1, 2, 3)
depth_limit = 10

path = dls(start_state, goal_state, depth_limit)
if path:
    print("DLS Path:", path)
else:
    print("No path found within depth limit")
```

BLOCK PROBLEM WITH IDS

```
def ids(start, goal, max_depth):
    for depth in range(max_depth):
        path = dls(start, goal, depth)
        if path:
            return path
```

```
    return None

start_state = (3, 2, 1)
goal_state = (1, 2, 3)
max_depth = 10

path = ids(start_state, goal_state, max_depth)
if path:
    print("IDS Path:", path)
else:
    print("No path found within max depth")
```

UCS :

```
import heapq

def ucs(graph, start, goal):
    # Priority queue to store nodes to be visited
    priority_queue = [(0, start)]

    # Dictionary to store the shortest path costs
    cost_so_far = {start: 0}

    while priority_queue:

        current_cost, current_node = heapq.heappop(priority_queue)

        if current_node == goal:
            return cost_so_far[current_node]
```

```
        for neighbor, weight in graph[current_node].items():
            new_cost = current_cost + weight

            if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                cost_so_far[neighbor] = new_cost
                heapq.heappush(priority_queue, (new_cost, neighbor))

    return float('inf')

graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}

# Start and goal nodes
start_node = 'A'
goal_node = 'D'

shortest_path_cost = ucs(graph, start_node, goal_node)

if shortest_path_cost != float('inf'):
    print(f"The shortest path cost from {start_node} to {goal_node} is {shortest_path_cost}.")
else:
    print(f"There is no path from {start_node} to {goal_node}.")
```

8 puzzle , misplaced heuristic , bfs :

```python
import sys
import copy

q = []
visited = []

def compare(s,g):
    if s==g:
        return(1)
    else:
        return(0)

def find_pos(s):

    for i in range(3):
        for j in range(3):
            if s[i][j] == 0:
                return([i,j])



def up(s,pos):

    i = pos[0]
    j = pos[1]

    if i > 0:
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i-1][j]
        temp[i-1][j] = 0
```

```
        return (temp)
    else:
        return (s)


def down(s,pos):

    i = pos[0]
    j = pos[1]

    if i < 2:
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i+1][j]
        temp[i+1][j] = 0
        return (temp)
    else:
        return (s)


def right(s,pos):

    i = pos[0]
    j = pos[1]

    if j < 2:
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i][j+1]
        temp[i][j+1] = 0
        return (temp)
```

```
    else:
        return (s)



def left(s,pos):

    i = pos[0]
    j = pos[1]

    if j > 0:
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i][j-1]
        temp[i][j-1] = 0
        return (temp)
    else:
        return (s)

def enqueue(s,val):
    global q
    q = q + [(val,s)]

def heuristic(s,g):
    d = 0
    for i in range(3):
        for j in range(3):
            if s[i][j] != g[i][j]:
                d += 1
    return d
```

```
def dequeue():

    global q
    global visited

    q.sort()
    visited = visited + [q[0][1]]

    elem = q[0][1]
    del q[0]
    return (elem)

def search(s,g):

    curr_state = copy.deepcopy(s)
    if s == g:
        return

    global visited
    while(1):

        pos = find_pos(curr_state)
        new = up(curr_state,pos)

        if new != curr_state:
            if new == g:
                print ("found!! The intermediate states are:")
                print (visited + [g])
```

```
        return
    else:
        if new not in visited:
            enqueue(new,heuristic(new,g))


new = down(curr_state,pos)

if new != curr_state:
    if new == g:
        print ("found!! The intermediate states are:")
        print (visited + [g])
        return
    else:
        if new not in visited:
            enqueue(new,heuristic(new,g))

new = right(curr_state,pos)

if new != curr_state:
    if new == g:
        print ("found!! The intermediate states are:")
        print (visited + [g])
        return
    else:
        if new not in visited:
            enqueue(new,heuristic(new,g))

new = left(curr_state,pos)
```

```
        if new != curr_state:
            if new == g:
                print ("found!! The intermediate states are:")
                print (visited + [g])
                return
            else:
                if new not in visited:
                    enqueue(new,heuristic(new,g))

        if len(q) > 0:
            curr_state = dequeue()
        else:
            print ("not found")
            return


def main():
    s = [[2,0,3],[1,8,4],[7,6,5]]
    g = [[1,2,3],[8,0,4],[7,6,5]]
    global q
    global visited
    q = q
    visited = visited + [s]

    search(s,g)

if __name__ == "__main__":
    main()
```

8 puzzle , heuristic , hill climb

```
import sys

import copy


curr_min = sys.maxsize

q = []

visited = []


def compare(s,g):

    if s==g:

        return(1)

    else:

        return(0)


def find_pos(s):


    for i in range(3):

        for j in range(3):

            if s[i][j] == 0:

                return([i,j])




def up(s,pos):


    i = pos[0]

    j = pos[1]
```

```
    if i > 0:
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i-1][j]
        temp[i-1][j] = 0
        return (temp)
    else:
        return (s)


def down(s,pos):

    i = pos[0]
    j = pos[1]

    if i < 2:
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i+1][j]
        temp[i+1][j] = 0
        return (temp)
    else:
        return (s)


def right(s,pos):

    i = pos[0]
    j = pos[1]

    if j < 2:
```

```
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i][j+1]
        temp[i][j+1] = 0
        return (temp)
    else:
        return (s)


def left(s,pos):

    i = pos[0]
    j = pos[1]

    if j > 0:
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i][j-1]
        temp[i][j-1] = 0
        return (temp)
    else:
        return (s)

def enqueue(s):
    global q
    q = q + [s]

def heuristic(s,g):
    d = 0
    for i in range(len(s)):
        for j in range(len(s[0])):
```

```
        if s[i][j] != g[i][j]:
            d += 1
    return d



def dequeue(g):

    h = []
    global q
    global visited
    global curr_min

    for i in range(len(q)):
        h = h + [heuristic(q[i],g)]

    if min(h) < curr_min:
        curr_min = min(h)
        index = h.index(min(h))
        visited = visited + [q[index]]
        elem = q[index]
        q = []
        return (elem)
    else:
        print ("optimal solution found !! The intermediate states are: ")
        print (visited)
        exit()
```

```
def search(s,g):

    curr_state = copy.deepcopy(s)
    if s == g:
        return

    global visited
    while(1):

        pos = find_pos(curr_state)
        new = up(curr_state,pos)

        if new != curr_state:
            if new == g:
                print ("Goal State found !! The intermediate States are :")
                print (visited + [g])
                return
            else:
                if new not in visited:
                    enqueue(new)



        new = down(curr_state,pos)

        if new != curr_state:
            if new == g:
                print ("Goal State found !! The intermediate States are :")
                print (visited + [g])
```

```
        return
    else:
        if new not in visited:
            enqueue(new)

new = right(curr_state,pos)

if new != curr_state:
    if new == g:
        print ("Goal State found !! The intermediate States are :")
        print (visited + [g])
        return
    else:
        if new not in visited:
            enqueue(new)

new = left(curr_state,pos)

if new != curr_state:
    if new == g:
        print ("Goal State found !! The intermediate States are :")
        print (visited + [g])
        return
    else:
        if new not in visited:
            enqueue(new)

if len(q) > 0:
    curr_state = dequeue(g)
```

```
        else:
            print ("not found")
            return



def main():
    s = [[2,8,3],[1,5,4],[7,6,0]]
    g = [[1,2,7],[8,0,5],[3,4,6]]
    global q
    global visited
    q = q + [s]
    visited = visited + [s]
    search(s,g)

if __name__ == "__main__":
    main()
```

8 puzzle , heuristic , A*

```
import sys
import copy

q = []
visited = []

def compare(s,g):
    if s==g:
        return(1)
    else:
```

```
        return(0)

def find_pos(s):

    for i in range(len(s)):
        for j in range(len(s[0])):
            if s[i][j] == 0:
                return([i,j])



def up(s,pos):

    i = pos[0]
    j = pos[1]

    if i > 0:
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i-1][j]
        temp[i-1][j] = 0
        return (temp)
    else:
        return (s)



def down(s,pos):

    i = pos[0]
    j = pos[1]
```

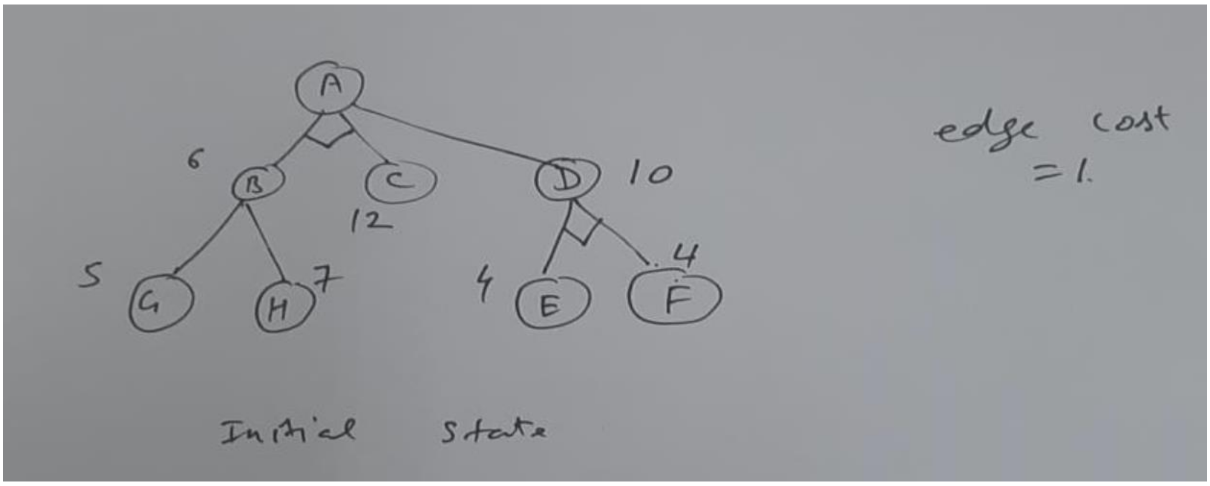
```
    if i < 2:
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i+1][j]
        temp[i+1][j] = 0
        return (temp)
    else:
        return (s)


def right(s,pos):

    i = pos[0]
    j = pos[1]

    if j < 2:
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i][j+1]
        temp[i][j+1] = 0
        return (temp)
    else:
        return (s)


def left(s,pos):

    i = pos[0]
    j = pos[1]

    if j > 0:
```

```
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i][j-1]
        temp[i][j-1] = 0
        return (temp)
    else:
        return (s)

def enqueue(s,val):
    global q
    q = q + [(val,s)]



def g_val(s,curr_state):
    d = 0
    for i in range(3):
        for j in range(3):
            if s[i][j] != curr_state[i][j]:
                d += 1
    return d

def h_val(g,curr_state):


    d = 0
    for i in range(3):
        for j in range(3):
            if curr_state[i][j] != g[i][j]:
                d += 1
    return d
```

```
def heuristic_val(s,curr_state,g):
    heur_val = g_val(s,curr_state) + h_val(g,curr_state)
    return heur_val

def dequeue(g,s):
    global q
    global visited
    q.sort()
    elem = q[0][1]
    visited = visited + [q[0][1]]
    del q[0]
    return (elem)

def search(s,g):
    curr_state = copy.deepcopy(s)
    if s == g:
        return
    global visited
    while(1):
        pos = find_pos(curr_state)
        new = up(curr_state,pos)
        if new != curr_state:
            if new == g:
                print ("found!! The intermediate states are:")
                print (visited + [g])
                return
            else:
                if new not in visited:
```

```
            enqueue(new,heuristic_val(s,new,g))

    new = down(curr_state,pos)
    if new != curr_state:
        if new == g:
            print ("found!! The intermediate states are:")
            print (visited + [g])
            return
        else:
            if new not in visited:
                enqueue(new,heuristic_val(s,new,g))

    new = right(curr_state,pos)
    if new != curr_state:
        if new == g:
            print ("found!! The intermediate states are:")
            print (visited + [g])
            return
        else:
            if new not in visited:
                enqueue(new,heuristic_val(s,new,g))

    new = left(curr_state,pos)
    if new != curr_state:
        if new == g:
            print ("found!! The intermediate states are:")
            print (visited + [g])
            return
        else:
```

```
            if new not in visited:
                enqueue(new,heuristic_val(s,new,g))

        if len(q) > 0:
            curr_state = dequeue(g,s)
        else:
            print ("not found")
            return



def main():

    s = [[2,0,3],[1,8,4],[7,6,5]]
    g = [[1,2,3],[8,0,4],[7,6,5]]
    global q
    global visited
    q = q
    visited = visited + [s]

    search(s,g)

if __name__ == "__main__":
    main()
```



AO*

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

node_cost = {'B': 6, 'C': 12, 'D': 10, 'E': 4, 'F': 4, 'G': 5, 'H': 7}

edge_cost = 1
head_node = 'A'

def find_min_cost(nodes):
    return min((sum(node_cost[node] for node in path) + edge_cost * (len(path) - 1), path) for path in nodes)

def traverse():
    curr = [head_node]
    total_cost = 0
```

```
    while curr:
        min_cost, next_nodes = find_min_cost(curr)
        if next_nodes:
            total_cost += min_cost
            curr = next_nodes
        else:
            print(f"Total moves: {total_cost}")
            return total_cost

if __name__ == "__main__":
    traverse()
```

0/1 knapsack

```
import random

# Define items
items = {
    'A': {'weight': 2, 'value': 3},
    'B': {'weight': 3, 'value': 5},
    'C': {'weight': 4, 'value': 7},
    'D': {'weight': 5, 'value': 9}
}

# Define constants
population_size = 4
max_capacity = 9
mutation_order = ['C', 'A', 'D', 'B']
```

```
# Initialize population
population = [
    [1, 1, 1, 1],
    [1, 0, 0, 0],
    [1, 0, 1, 0],
    [1, 0, 0, 1]
]

# Fitness function
def fitness(chromosome):
    total_weight = sum(items[item]['weight'] for i, item in enumerate(items) if chromosome[i])
    total_value = sum(items[item]['value'] for i, item in enumerate(items) if chromosome[i])

    if total_weight > max_capacity:
        return 0

    return total_value

# Selection
def selection(population):
    sorted_population = sorted(population, key=fitness, reverse=True)
    return sorted_population[:2]

# Crossover
def crossover(parent1, parent2):
    crossover_point = len(parent1) // 2
    child = parent1[:crossover_point] + parent2[crossover_point:]
    return child

# Mutation
```

```python
def mutation(child):
    for gene in mutation_order:
        index = list(items.keys()).index(gene)
        child[index] = 1 - child[index]
    return child


# Main Genetic Algorithm
for iteration in range(4):
    print(f"Iteration {iteration + 1}:")

    # Selection
    selected_population = selection(population)

    # Crossover and Mutation
    offspring1 = crossover(selected_population[0], selected_population[1])
    offspring1 = mutation(offspring1)

    # Update population
    population = selected_population + [offspring1]

    # Display current best solution
    best_chromosome = max(population, key=fitness)
    print(f"Best chromosome: {best_chromosome}, Fitness: {fitness(best_chromosome)}")

# Final best solution
best_chromosome = max(population, key=fitness)
print("\nFinal Best Solution:")
print(f"Chromosome: {best_chromosome}")
print(f"Total Weight: {sum(items[item]['weight'] for i, item in enumerate(items) if
best_chromosome[i])}")
print(f"Total Value: {fitness(best_chromosome)}")
```

KNN

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsRegressor

# Step 1: Generate synthetic data
data = {
    'Experience': [5, 8, 3, 10, 6, 4, 7],
    'Written_Test_Score': [8, 7, 6, 9, 7, 5, 8],
    'Interview_Score': [10, 6, 8, 9, 7, 6, 9],
    'Salary': [70000, 90000, 60000, 100000, 80000, 65000, 85000]
}

# Step 2: Convert data to DataFrame and save it to CSV
df = pd.DataFrame(data)
df.to_csv('salary_dataset.csv', index=False)

# Step 3: Load the dataset and split into features and target variable
dataset = pd.read_csv('salary_dataset.csv')
X = dataset.iloc[:, :-1].values  # Features (Experience, Written_Test_Score, Interview_Score)
y = dataset.iloc[:, -1].values   # Target variable (Salary)

# Split the dataset 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)

# Feature scaling
```

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Step 4: Train the KNN model
k_values = [3, 5, 7]  # Different values of K
for k in k_values:
    knn_model = KNeighborsRegressor(n_neighbors=k)
    knn_model.fit(X_train, y_train)

    # Predict salaries for the candidates
    candidate_a = [[5, 8, 10]]  # Candidate (a) with 5 Yrs experience, 8 written test score, 10 interview score
    candidate_b = [[8, 7, 6]]    # Candidate (b) with 8 Yrs experience, 7 written test score, 6 interview score

    predicted_salary_a = knn_model.predict(scaler.transform(candidate_a))
    predicted_salary_b = knn_model.predict(scaler.transform(candidate_b))

    print(f"For K = {k}:")
    print(f"Predicted salary for candidate (a): ${predicted_salary_a[0]}")
    print(f"Predicted salary for candidate (b): ${predicted_salary_b[0]}")
    print()
```

NAIVE BAYES

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

```
import numpy as np

from sklearn.neighbors import KNeighborsRegressor

from sklearn.metrics import mean_squared_error, r2_score

from sklearn.naive_bayes import GaussianNB


# Load your dataset from the CSV file (replace 'data.csv' with your actual filename)

data = pd.read_csv('dataset.csv')


# Separate features (X) and target variable (y)

X = data[['Graduation_Percentage','Experience','Written_Score','Interview_Score']]

y = data['Selection']




bayesian = GaussianNB()  # You can adjust the value of K here


# Train the model

bayesian.fit(X, y)


# Function to predict salary for a new candidate

def predict_salary(Graduation_Percentage,Experience, Written_Score, Interview_Score,
bayes_model):

 # Create a new data point for the candidate

  new_candidate = [[Graduation_Percentage,Experience, Written_Score, Interview_Score]]

 # Predict salary using the KNN model

  predicted_salary = bayes_model.predict(new_candidate)

  return predicted_salary[0]


# Predicting salaries for candidates (a) and (b)
```

```
candidate_a_salary = predict_salary(90,5, 8, 10, bayesian)
candidate_b_salary = predict_salary(75,8, 7, 6, bayesian)

print(f"Predicted Salary for Candidate (a): {candidate_a_salary:.2f}")
print(f"Predicted Salary for Candidate (b): {candidate_b_salary:.2f}")
```

Decision Tree

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.naive_bayes import GaussianNB
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier


iris = load_iris()
X = iris.data  # Features
y = iris.target  # Target labels

# Split data into training and testing sets (e.g., 70% train, 30% test)
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


# Create the decision tree classifier object
```

```
clf = DecisionTreeClassifier(criterion="gini", max_depth=3)

# Train the model on the training data
clf.fit(X_train, y_train)
# Make predictions on the test data


y_pred = clf.predict(X_test)
print(y_pred)
# Evaluate model performance (e.g., accuracy)
from sklearn.metrics import accuracy_score

accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```