1.

```
def aStarAlgo(start_node, end_node):
    open_set = set([start_node])
    closed_set = set()

    g = {}
    parents = {}

    g[start_node] = 0
    parents[start_node] = start_node

    while open_set:
        n = None
        # Find node in open set with lowest f(n) = g(n) +
h(n)
        for v in open_set:
            if n is None or g[v] + heuristic(v) < g[n] +
heuristic(n):
                n = v

        if n is None:
            print("Path does not exist")
            return None

        if n == end_node:
            path = []

            while parents[n] != n:
                path.append(n)
                n = parents[n]

            path.append(start_node)
```

```
            path.reverse()
            print('Path found: {}'.format(path))
            return path

        for (m, weight) in get_neighbors(n):
            if m not in open_set and m not in closed_set:
                open_set.add(m)
                parents[m] = n
                g[m] = g[n] + weight
            else:
                if g[m] > g[n] + weight:
                    g[m] = g[n] + weight
                    parents[m] = n

                    if m in closed_set:
                        closed_set.remove(m)
                        open_set.add(m)

        open_set.remove(n)
        closed_set.add(n)

    print("Path does not exist")
    return None

def get_neighbors(v):
    if v in Graph_node:
        return Graph_node[v]
    else:
        return []

def heuristic(n):
    heuristic_dist = {
        'A': 11, 'B': 6, 'C': 99, 'D': 1, 'E': 7, 'G': 0
```

```
    }
    return heuristic_dist.get(n, float('inf'))

Graph_node = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1), ('G', 9)],
    'C': [],
    'E': [('D', 6)],
    'D': [('G', 1)],
    'G': []
}

# Run the algorithm
aStarAlgo('A', 'G')
```

2.

```
from collections import deque

def bfs_shortest_path(maze, start, goal):
    rows = len(maze)
    cols = len(maze[0])

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

    queue = deque([start])
    visited = set([start])

    # To reconstruct path
    parent = {start: None}
```

```
    while queue:
        x, y = queue.popleft()

        # If goal is found, reconstruct path
        if (x, y) == goal:
            path = []
            curr = goal
            while curr:
                path.append(curr)
                curr = parent[curr]
            path.reverse()
            return path  # Shortest path

        # Explore neighbors
        for dx, dy in directions:
            nx, ny = x + dx, y + dy

            # Check bounds and obstacles
            if 0 <= nx < rows and 0 <= ny < cols:
                if maze[nx][ny] == 0 and (nx, ny) not in
visited:
                    visited.add((nx, ny))
                    parent[(nx, ny)] = (x, y)
                    queue.append((nx, ny))

    return None  # No path found


maze = [
    [0, 0, 1, 0],
    [1, 0, 1, 0],
    [0, 0, 0, 0],
    [0, 1, 1, 0]
```

```
]

start = (0, 0)
goal = (3, 3)

path = bfs_shortest_path(maze, start, goal)
print("Shortest Path:", path)
```

3.

```
# Depth First Search (DFS) in Python

def dfs(graph, start, target, visited=None):
    if visited is None:
        visited = set()  # To avoid revisiting nodes
(prevent cycles)

    visited.add(start)  # Mark current node as visited
    print(start, end=" ")  # Visualize traversal order

    # If the target is found, we stop
    if start == target:
        return True

    # Explore neighbors
    for neighbor in graph[start]:
        if neighbor not in visited:
            if dfs(graph, neighbor, target, visited):  #
Recursively call DFS
                return True

    return False
```

```
# Example Graph (Game Map)
game_map = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

# Starting and Target nodes
start_node = 'A'
target_node = 'F'

print("DFS Traversal Order:")
dfs(game_map, start_node, target_node)
```

4.

```
import heapq

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

def astar(maze, start, goal):
    rows, cols = len(maze), len(maze[0])
```

```
    # Priority queue (min-heap) for A*
    open_set = []
    heapq.heappush(open_set, (0, start))

    # Store cost from start to current node
    g_cost = {start: 0}

    # Parent dictionary to reconstruct path
    parent = {start: None}

    # Movement directions (up, down, left, right)
    directions = [(0,1), (0,-1), (1,0), (-1,0)]

    while open_set:
        _, current = heapq.heappop(open_set)

        if current == goal:
            # Reconstruct shortest path
            path = []
            while current:
                path.append(current)
                current = parent[current]
            return path[::-1]

        x, y = current

        for dx, dy in directions:
            nx, ny = x + dx, y + dy

            # Check boundaries and obstacles
            if 0 <= nx < rows and 0 <= ny < cols and
maze[nx][ny] == 0:
```

```
                new_cost = g_cost[current] + 1  # Cost of
moving to neighbor

                if (nx, ny) not in g_cost or new_cost <
g_cost[(nx, ny)]:
                    g_cost[(nx, ny)] = new_cost
                    f_cost = new_cost + heuristic((nx, ny),
goal)
                    parent[(nx, ny)] = current
                    heapq.heappush(open_set, (f_cost, (nx,
ny)))

    return None  # No path found

# Example Maze (0 = free, 1 = wall)
maze = [
    [0, 0, 0, 1, 0],
    [1, 1, 0, 1, 0],
    [0, 0, 0, 0, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 1, 0]
]

start = (0, 0)
goal = (4, 4)

path = astar(maze, start, goal)

print("Shortest Path using A*:", path)
```

5.

```
from collections import deque

# Function to print the puzzle state
def print_puzzle(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()

# Function to get all possible moves from the current state
def get_neighbors(state):
    neighbors = []
    index = state.index('0')  # '0' represents the blank space
    moves = {
        0: [1, 3],
        1: [0, 2, 4],
        2: [1, 5],
        3: [0, 4, 6],
        4: [1, 3, 5, 7],
        5: [2, 4, 8],
        6: [3, 7],
        7: [4, 6, 8],
        8: [5, 7]
    }

    for move in moves[index]:
        new_state = list(state)
        new_state[index], new_state[move] = new_state[move], new_state[index]
        neighbors.append(''.join(new_state))
    return neighbors
```

```
# BFS algorithm to find the solution
def bfs(start, goal):
    visited = set()
    queue = deque([(start, [start])])  # store (state, path)

    while queue:
        state, path = queue.popleft()
        if state == goal:
            return path  # found the solution
        if state in visited:
            continue
        visited.add(state)
        for neighbor in get_neighbors(state):
            if neighbor not in visited:
                queue.append((neighbor, path + [neighbor]))
    return None

# -------- Main Code --------
start = "123405678"   # 0 represents blank space
goal = "123456780"

print("Initial State:")
print_puzzle(start)
print("Goal State:")
print_puzzle(goal)

solution = bfs(start, goal)

if solution:
    print("\nSteps to solve:")
    for step in solution:
        print_puzzle(step)
```

```
else:
    print("No solution found!")
```

6.

```
# Simple Tic-Tac-Toe Game

# Create the board
board = [' ' for _ in range(9)]

# Function to display the board
def print_board():
    print(f"{board[0]} | {board[1]} | {board[2]}")
    print("--+---+--")
    print(f"{board[3]} | {board[4]} | {board[5]}")
    print("--+---+--")
    print(f"{board[6]} | {board[7]} | {board[8]}")
    print()

# Function to check for a win
def check_winner(player):
    win_pos =
[(0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4
,6)]
    for a,b,c in win_pos:
        if board[a] == board[b] == board[c] == player:
            return True
    return False

# Main game loop
def play_game():
    current = 'X'
```

```
    for turn in range(9):
        print_board()
        move = int(input(f"Player {current}, enter position
(1-9): ")) - 1
        if board[move] != ' ':
            print("Invalid move! Try again.")
            continue
        board[move] = current

        if check_winner(current):
            print_board()
            print(f"Player {current} wins!")
            return
        current = 'O' if current == 'X' else 'X'
    print_board()
    print("It's a draw!")

# Run the game
play_game()
```

7.

```
# Tower of Hanoi Problem using Recursion

def tower_of_hanoi(n, source, helper, destination):
    if n == 1:
        print(f"Move disk 1 from {source} → {destination}")
        return
    tower_of_hanoi(n-1, source, destination, helper)
    print(f"Move disk {n} from {source} → {destination}")
    tower_of_hanoi(n-1, helper, source, destination)
```

```
# -------- Main Code --------
n = int(input("Enter number of disks: "))
print("\nSteps to solve Tower of Hanoi:\n")
tower_of_hanoi(n, 'A', 'B', 'C')
```

8.

```
from collections import deque
from math import gcd
from typing import List, Tuple, Optional, Dict

State = Tuple[int, int]

class WaterJugProblem:
    def __init__(self, capA: int, capB: int, target: int,
start: State = (0, 0)):
        self.capA = capA
        self.capB = capB
        self.target = target
        self.start = start

    def is_goal(self, s: State) -> bool:
        # Target is considered met if either jug has exactly
target liters
        return s[0] == self.target or s[1] == self.target

    def solvable(self) -> bool:
        # Classic condition: target <= total capacity and
target % gcd(capA, capB) == 0
```

```
        if self.target < 0 or self.target > max(self.capA,
self.capB) and self.target > (self.capA + self.capB):
            return False
        if self.target == 0:
            return True
        return self.target % gcd(self.capA, self.capB) == 0
and self.target <= (self.capA + self.capB)

    def successors(self, s: State) -> List[Tuple[State,
str]]:
        x, y = s
        A, B = self.capA, self.capB
        nxt = []

        # 1) Fill A
        if x < A:
            nxt.append(((A, y), f"Fill A to {A}"))

        # 2) Fill B
        if y < B:
            nxt.append(((x, B), f"Fill B to {B}"))

        # 3) Empty A
        if x > 0:
            nxt.append(((0, y), "Empty A"))

        # 4) Empty B
        if y > 0:
            nxt.append(((x, 0), "Empty B"))

        # 5) Pour A -> B
        if x > 0 and y < B:
            pour = min(x, B - y)
```

```
            nxt.append(((x - pour, y + pour), f"Pour {pour}
from A->B"))

        # 6) Pour B -> A
        if y > 0 and x < A:
            pour = min(y, A - x)
            nxt.append(((x + pour, y - pour), f"Pour {pour}
from B->A"))

        return nxt

    def bfs(self) -> Optional[List[Tuple[State, str]]]:
        """Return path as list of (state,
action_taken_to_reach_it). First step has action 'Start'."""
        if not self.solvable():
            return None

        q = deque([self.start])
        parent: Dict[State, State] = {self.start: self.start}
        act: Dict[State, str] = {self.start: "Start"}
        visited = {self.start}

        while q:
            s = q.popleft()
            if self.is_goal(s):
                # Reconstruct path
                path: List[Tuple[State, str]] = []
                cur = s
                while True:
                    path.append((cur, act[cur]))
                    if cur == parent[cur]:
                        break
                    cur = parent[cur]
```

```
                path.reverse()
                return path

            for ns, a in self.successors(s):
                if ns not in visited:
                    visited.add(ns)
                    parent[ns] = s
                    act[ns] = a
                    q.append(ns)

        return None


def print_solution(path: List[Tuple[State, str]]):
    print("Solution (state: (A, B)):")
    for i, (st, action) in enumerate(path):
        prefix = f"Step {i:02d}:"
        print(f"{prefix:<9} {action:<18} -> {st}")

if __name__ == "__main__":
    # === Example 1 ===
    # Problem: Jug A=4L, Jug B=3L, target=2L (classic)
    # problem = WaterJugProblem(capA=4, capB=3, target=2,
start=(0,0))
    # path = problem.bfs()
    # if path is None:
    #     print("No solution exists.")
    # else:
    #     print_solution(path)

    # === Example 2 (change caps/target easily) ===
    problem2 = WaterJugProblem(capA=5, capB=7, target=6)
    path2 = problem2.bfs()
```

```
    if path2: print_solution(path2)
```

9.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score, mean_squared_error


df = pd.read_csv("uber - uber.csv")
# Drop unnamed index column if it exists
if 'Unnamed: 0' in df.columns:
    df = df.drop('Unnamed: 0', axis=1)


print("Shape:", df.shape)
print("Columns:", df.columns.tolist())
print("Missing Values:\n", df.isnull().sum())
print(df.describe())


# Basic EDA visualization
plt.figure(figsize=(10, 5))
sns.histplot(df['fare_amount'], bins=50, kde=True, color='skyblue')
plt.title('Distribution of Fare Amounts')
plt.xlabel('Fare Amount')
plt.ylabel('Frequency')
plt.show()


# Correlation heatmap
plt.figure(figsize=(10,8))
corr_matrix = df.select_dtypes(include=['float64', 'int64']).corr()
sns.heatmap(corr_matrix, cmap="coolwarm", annot=True, fmt='.2f', square=True,
linewidths=0.5)
plt.title("Feature Correlation Heatmap")
```

```
plt.tight_layout()
plt.show()


# ----- Data Preprocessing -----
df = df.dropna()
df_num = df.select_dtypes(include=['float64', 'int64'])
target = [c for c in df_num.columns if 'price' in c.lower() or 'fare' in c.lower()]
y = df_num[target[0]]
X = df_num.drop(columns=[target[0]])


# ----- Train/Test Split -----
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


# ----- Without PCA -----
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s = scaler.transform(X_test)
lr = LinearRegression().fit(X_train_s, y_train)
pred = lr.predict(X_test_s)


r2_no_pca = r2_score(y_test, pred)
rmse_no_pca = mean_squared_error(y_test, pred)
print("\nWithout PCA -> R2:", r2_no_pca, "RMSE:", rmse_no_pca)


# ----- With PCA -----
pca = PCA(n_components=0.95)
X_train_pca = pca.fit_transform(X_train_s)
X_test_pca = pca.transform(X_test_s)
lr_pca = LinearRegression().fit(X_train_pca, y_train)
pred_pca = lr_pca.predict(X_test_pca)


r2_pca = r2_score(y_test, pred_pca)
rmse_pca = mean_squared_error(y_test, pred_pca)
print("With PCA -> R2:", r2_pca, "RMSE:", rmse_pca)
print("No. of PCA Components:", pca.n_components_)


# ----- Visual Comparisons -----
```

```
# Explained Variance Plot
plt.figure(figsize=(8,5))
plt.plot(np.cumsum(pca.explained_variance_ratio_), marker='o')
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('PCA Explained Variance')
plt.grid(True)
plt.show()

# R2 & RMSE Comparison
plt.figure(figsize=(6,4))
metrics = pd.DataFrame({
    'Metric': ['R2 Score', 'RMSE'],
    'Without PCA': [r2_no_pca, rmse_no_pca],
    'With PCA': [r2_pca, rmse_pca]
})
metrics.set_index('Metric').plot(kind='bar', figsize=(6,4), color=['steelblue','orange'])
plt.title('Model Performance: With vs Without PCA')
plt.ylabel('Value')
plt.grid(True)
plt.show()

# Actual vs Predicted (Optional for better understanding)
plt.figure(figsize=(8,5))
sns.scatterplot(x=y_test, y=pred, label='Without PCA', alpha=0.5)
sns.scatterplot(x=y_test, y=pred_pca, label='With PCA', alpha=0.5)
plt.xlabel("Actual Fare")
plt.ylabel("Predicted Fare")
plt.title("Actual vs Predicted Comparison")
plt.legend()
plt.show()
```

10.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.linear_model import LinearRegression
```

```
from sklearn.metrics import r2_score, mean_squared_error, mean_absolute_error


# --- LOAD DATA ---
df = pd.read_csv("uber - uber.csv")   # change path if needed
# Drop unnamed index column if it exists
if 'Unnamed: 0' in df.columns:
    df = df.drop('Unnamed: 0', axis=1)

# --- EXPLORATORY DATA ANALYSIS (EDA) ---
print("Dataset Shape:", df.shape)
print("\nColumns:", df.columns.tolist())
print("\nMissing Values:\n", df.isnull().sum())
print("\nSummary Statistics:\n", df.describe())
# Distribution of target variable
plt.figure(figsize=(8,5))
sns.histplot(df['fare_amount'], bins=50, kde=True, color='skyblue')
plt.title("Distribution of Fare Amounts")
plt.xlabel("Fare Amount")
plt.ylabel("Frequency")
plt.show()


# Correlation heatmap
plt.figure(figsize=(10,8))
corr_matrix = df.select_dtypes(include=['float64', 'int64']).corr()
sns.heatmap(corr_matrix, cmap='coolwarm', annot=True, fmt='.2f', square=True, linewidths=0.5)
plt.title("Feature Correlation Heatmap")
plt.tight_layout()
plt.show()
# --- DATA CLEANING ---
df = df.dropna()
df_num = df.select_dtypes(include=['float64', 'int64'])

# Separate features and target
target = [c for c in df_num.columns if 'fare' in c.lower() or 'price' in c.lower()]
y = df_num[target[0]]
X = df_num.drop(columns=[target[0]])
# --- TRAIN/TEST SPLIT ---
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# --- SCALING ---
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
```

```
X_test_scaled = scaler.transform(X_test)

# --- MODEL WITHOUT PCA ---
lr = LinearRegression()
lr.fit(X_train_scaled, y_train)
pred = lr.predict(X_test_scaled)

r2_no_pca = r2_score(y_test, pred)
rmse_no_pca = mean_squared_error(y_test, pred)
mae_no_pca = mean_absolute_error(y_test, pred)

print("\n----- MODEL WITHOUT PCA -----")
print("R²:", r2_no_pca)
print("RMSE:", rmse_no_pca)
print("MAE:", mae_no_pca)
# --- MODEL WITH PCA ---
pca = PCA(n_components=0.95)
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)

lr_pca = LinearRegression()
lr_pca.fit(X_train_pca, y_train)
pred_pca = lr_pca.predict(X_test_pca)

r2_pca = r2_score(y_test, pred_pca)
rmse_pca = mean_squared_error(y_test, pred_pca)
mae_pca = mean_absolute_error(y_test, pred_pca)

print("\n----- MODEL WITH PCA -----")
print("R²:", r2_pca)
print("RMSE:", rmse_pca)
print("MAE:", mae_pca)
print("No. of PCA Components:", pca.n_components_)

# --- VISUAL COMPARISONS ---

#Explained Variance by PCA
plt.figure(figsize=(8,5))
plt.plot(np.cumsum(pca.explained_variance_ratio_), marker='o', color='purple')
plt.xlabel('Number of Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('PCA Explained Variance')
plt.grid(True)
plt.show()
```

```
#Comparison of Metrics
metrics = pd.DataFrame({
    'Metric': ['R²', 'RMSE', 'MAE'],
    'Without PCA': [r2_no_pca, rmse_no_pca, mae_no_pca],
    'With PCA': [r2_pca, rmse_pca, mae_pca]
}).set_index('Metric')

metrics.plot(kind='bar', figsize=(7,5), color=['#00BFC4', '#F8766D'])
plt.title("Model Performance: With vs Without PCA")
plt.ylabel("Metric Value")
plt.grid(True)
plt.show()

#Actual vs Predicted Comparison
plt.figure(figsize=(8,5))
sns.scatterplot(x=y_test, y=pred, label='Without PCA', alpha=0.5)
sns.scatterplot(x=y_test, y=pred_pca, label='With PCA', alpha=0.5)
plt.xlabel("Actual Fare")
plt.ylabel("Predicted Fare")
plt.title("Actual vs Predicted (With & Without PCA)")
plt.legend()
plt.show()
```

11.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import KFold, cross_val_score
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
import matplotlib.pyplot as plt
df = pd.read_csv("Houses.csv")

# Drop rows with missing target values
df = df.dropna(subset=["Price"])

df = pd.get_dummies(df, columns=['Location'], drop_first=True)

X = df.drop(columns=['Price'])
y = df['Price']
```

```
model = LinearRegression()
kf = KFold(n_splits=5, shuffle=True, random_state=42)
# Evaluate with Cross-Validation

r2_scores = cross_val_score(model, X, y, cv=kf, scoring='r2')
mse_scores = -cross_val_score(model, X, y, cv=kf, scoring='neg_mean_squared_error')
mae_scores = -cross_val_score(model, X, y, cv=kf, scoring='neg_mean_absolute_error')
rmse_scores = np.sqrt(mse_scores)
# Train final model on full dataset
model.fit(X, y)
y_pred = model.predict(X)

print("\n===== Final Model (Full Data) =====")
print(f"R² Score:  {r2_score(y, y_pred):.4f}")
print(f"MSE:       {mean_squared_error(y, y_pred):.2f}")
print(f"RMSE:      {np.sqrt(mean_squared_error(y, y_pred)):.2f}")
print(f"MAE:       {mean_absolute_error(y, y_pred):.2f}")

print("===== Cross Validation Results (5-Fold) =====")
print(f"R² per fold:   {np.round(r2_scores, 4)}")
print(f"MSE per fold:  {np.round(mse_scores, 2)}")
print(f"RMSE per fold: {np.round(rmse_scores, 2)}")
print(f"MAE per fold:  {np.round(mae_scores, 2)}")

print("\n===== Average Metrics =====")
print(f"Average R²:   {r2_scores.mean():.4f}")
print(f"Average MSE:  {mse_scores.mean():.2f}")
print(f"Average RMSE: {rmse_scores.mean():.2f}")
print(f"Average MAE:  {mae_scores.mean():.2f}")
#Visualization: Actual vs Predicted
plt.scatter(y, y_pred, alpha=0.6, color='teal')
plt.xlabel("Actual Prices")
plt.ylabel("Predicted Prices")
plt.title("Actual vs Predicted House Prices (Linear Regression)")
plt.grid(True, linestyle="--", alpha=0.6)
plt.show()
```

12.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Create a simple synthetic dataset (size only for simplicity)
```

```
data = {
    'Study_Hours': [1, 2, 3, 4, 5, 6, 7, 8],
    'Exam_Score': [35, 40, 50, 55, 60, 70, 75, 80]
}

df = pd.DataFrame(data)
df
print(df.describe())

# Plot Size vs Price
plt.scatter(df['Study_Hours'], df['Exam_Score'])
plt.xlabel('Study_Hours')
plt.ylabel('Exam_Score')
plt.title('Study Hours vs Exam Score')
plt.show()
X = df['Study_Hours'].values
y = df['Exam_Score'].values
X = df['Study_Hours'].values
y = df['Exam_Score'].values

# Normalize features for faster convergence
X = (X - np.mean(X)) / np.std(X)

# Hyperparameters
alpha = 0.01   # learning rate
epochs = 1000  # number of iterations

m = 0  # slope
c = 0  # intercept
n = len(X)

cost_history = []

for i in range(epochs):
    # Predicted values
    y_pred = m * X + c

    # Compute error
    error = y_pred - y

    # Cost function (Mean Squared Error)
    cost = (1/(2*n)) * np.sum(error**2)
    cost_history.append(cost)
```

```
    # Compute gradients
    t0 = (1/n) * np.sum(error)
    t1 = (1/n) * np.sum(error * X)


    # Update parameters
    c = c - alpha * t0
    m = m - alpha * t1

print("Trained parameters:")
print(f"m (slope) = {m}")
print(f"c (intercept) = {c}")
plt.plot(range(epochs), cost_history)
plt.xlabel('Iterations')
plt.ylabel('Cost J')
plt.title('Cost Function during Training')
plt.show()
# Predicted prices
y_pred = m * X + c

# Plot actual vs predicted
plt.scatter(X, y, color='blue', label='Actual')
plt.plot(X, y_pred, color='red', label='Predicted')
plt.xlabel('Normalized Size')
plt.ylabel('Exam_Score')
plt.title('Actual vs Predicted Exam_Score')
plt.legend()
plt.show()

# Mean Squared Error and R2 Score
from sklearn.metrics import mean_squared_error, r2_score

mse = mean_squared_error(y, y_pred)
r2 = r2_score(y, y_pred)
print("MSE:", mse)
print("R² Score:", r2)
```

13.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import KFold, cross_val_score
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
```

```
import seaborn as sns

data = pd.read_csv('Student_Marks.csv')
print(data.head(), "\n")

df = data[['hours_studied', 'attendance_percent', 'previous_scores', 'exam_score']].dropna()

X = df[['hours_studied', 'attendance_percent', 'previous_scores']]
y = df['exam_score']

model = LinearRegression()
kf = KFold(n_splits=5, shuffle=True, random_state=42)
cv_scores = cross_val_score(model, X, y, cv=kf, scoring='r2')

print("Cross-Validation R² Scores:", cv_scores)
print(f"\nMean R² Score: {np.mean(cv_scores):.4f}")
print(f"Standard Deviation: {np.std(cv_scores):.4f}")
model.fit(X, y)
coeff_df = pd.DataFrame(model.coef_, X.columns, columns=['Coefficient'])
print("\nRegression Coefficients:")
print(coeff_df)
print(f"\nIntercept: {model.intercept_:.2f}")

y_pred = model.predict(X)
plt.figure(figsize=(7,5))
plt.scatter(y, y_pred, alpha=0.6, color='teal')
plt.xlabel("Actual Exam Scores")
plt.ylabel("Predicted Exam Scores")
plt.title("Actual vs Predicted Exam Scores")
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--', lw=2)
plt.show()
plt.figure(figsize=(6,5))
sns.heatmap(df.corr(), annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Feature Correlation Heatmap')
plt.show()
from sklearn.metrics import mean_squared_error, mean_absolute_error

mse = mean_squared_error(y, y_pred)
mae = mean_absolute_error(y, y_pred)
r2 = np.mean(cv_scores)

print(f"\nModel Performance Metrics:")
print(f"Mean Absolute Error (MAE): {mae:.4f}")
print(f"Mean Squared Error (MSE): {mse:.4f}")
```

```
print(f"R² Score (5-Fold Mean): {r2:.4f}")
```

14.

```
# ===== Linear Regression Model for Salary Prediction =====
import pandas as pd
import numpy as np
from sklearn.model_selection import KFold, cross_val_score
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score
import matplotlib.pyplot as plt
import seaborn as sns
#Load your data
df = pd.read_csv("Salary Data.csv")
df = df.dropna(subset=["Salary"])

# Split features/target
num_cols = ["Years of Experience", "Age"]
cat_cols = ["Gender", "Education Level", "Job Title"]

# One-hot encode categorical columns (simple + easy)
df = pd.get_dummies(df, columns=cat_cols, drop_first=True)
df.head()
df.columns
X = df.drop("Salary", axis=1)
y = df["Salary"]


# Initialize model and 5-Fold CV
model = LinearRegression()
kf = KFold(n_splits=5, shuffle=True, random_state=42)

# Evaluate using 5-Fold Cross-Validation
r2_scores = cross_val_score(model, X, y, cv=kf, scoring="r2")
mse_scores = -cross_val_score(model, X, y, cv=kf, scoring="neg_mean_squared_error")
mae_scores = -cross_val_score(model, X, y, cv=kf, scoring="neg_mean_absolute_error")

rmse_scores = np.sqrt(mse_scores)

#Print all metrics nicely
print("===== Cross Validation Results (5-Fold) =====")
print(f"R² per fold:   {np.round(r2_scores, 4)}")
```

```
print(f"MSE per fold:  {np.round(mse_scores, 2)}")
print(f"RMSE per fold: {np.round(rmse_scores, 2)}")
print(f"MAE per fold:  {np.round(mae_scores, 2)}")

print("\n===== Average Metrics =====")
print(f"Average R²:   {r2_scores.mean():.4f}")
print(f"Average MSE:  {mse_scores.mean():.2f}")
print(f"Average RMSE: {rmse_scores.mean():.2f}")
print(f"Average MAE:  {mae_scores.mean():.2f}")

# Fit final model on all data and print final accuracy on same data
model.fit(X, y)
y_pred = model.predict(X)

print("\n===== Final Model (Full Data) =====")
print(f"R² Score:  {r2_score(y, y_pred):.4f}")
print(f"MSE:       {mean_squared_error(y, y_pred):.2f}")
print(f"RMSE:      {np.sqrt(mean_squared_error(y, y_pred)):.2f}")
print(f"MAE:       {mean_absolute_error(y, y_pred):.2f}")

plt.figure(figsize=(7,5))
plt.scatter(y, y_pred, alpha=0.6, color='teal')
plt.xlabel("Actual Exam Scores")
plt.ylabel("Predicted Exam Scores")
plt.title("Actual vs Predicted Exam Scores")
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--', lw=2)
plt.show()
plt.figure(figsize=(6,5))
# corr_mat=df['Age','Years of Experience', 'Education Level', 'Salary'].corr()
sns.heatmap(df[["Age", "Years of Experience", "Education Level_Master's", "Education
Level_PhD", "Job Title_Accountant", "Gender_Male", "Salary"]].corr(), annot=True,
cmap='coolwarm', fmt='.2f')
plt.title('Feature Correlation Heatmap')
plt.show()
```

15.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import KFold, cross_val_score
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, make_scorer
```

```
df = pd.read_csv("ad_spends.csv")

# --- Minimal monthly aggregation ---
df['Transaction_Date'] = pd.to_datetime(df['Transaction_Date'])
df['Month'] = df['Transaction_Date'].dt.to_period('M').dt.to_timestamp()

monthly = df.groupby('Month').agg(
    Monthly_Sales=('Revenue', 'sum'),
    Ad_Spend=('Ad_Spend', 'sum'),
    Discount=('Discount_Applied', 'mean'),
    Customer_Footfall=('Clicks', 'sum')
).reset_index()

# --- Model: Linear Regression ---
X = monthly[['Ad_Spend', 'Discount', 'Customer_Footfall']]
y = monthly['Monthly_Sales']
model = LinearRegression()

def rmse(y_true, y_pred):
    return np.sqrt(mean_squared_error(y_true, y_pred))
rmse_scorer = make_scorer(rmse, greater_is_better=False)

kfold = KFold(n_splits=5, shuffle=True, random_state=42)
neg_rmse_scores = cross_val_score(model, X, y, scoring=rmse_scorer, cv=kfold)
r2_scores = cross_val_score(model, X, y, scoring='r2', cv=kfold)

print('Fold RMSE:', np.round(-neg_rmse_scores, 3))
print('Fold R²:', np.round(r2_scores, 3))
print('Average RMSE:', (-neg_rmse_scores).mean())
print('Average R²:', r2_scores.mean())

# Fit on all months
model.fit(X, y)
print('Model trained on monthly aggregates!')

df.head()
monthly.head()
import matplotlib.pyplot as plt
import seaborn as sns

# ----- Correlation Matrix -----
plt.figure(figsize=(6,4))
#corr = monthly[["Monthly_Sales", "Ad_Spend", "Discount", "Customer_Footfall"]].corr()
sns.heatmap(monthly.corr(), annot=True, cmap="coolwarm", fmt=".2f")
```

```
plt.title("Correlation Matrix")
plt.show()



# ----- Actual vs Predicted Chart -----
y_pred = model.predict(X)

plt.figure(figsize=(8,5))
plt.plot(monthly["Month"], y, label="Actual Sales", marker='o')
plt.plot(monthly["Month"], y_pred, label="Predicted Sales", marker='x')
plt.title("Actual vs Predicted Monthly Sales")
plt.xlabel("Month")
plt.ylabel("Sales")
plt.legend()
plt.grid(True)
plt.show()
```

16.

```
# Apply the Naïve Bayes algorithm to a real-world classification problem such as email spam detection, sentiment analysis, or disease diagnosis.
# Train and test the model, then evaluate its performance using a Confusion Matrix and related metrics such as accuracy, precision, recall, and F1-score.

import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import (
    confusion_matrix, accuracy_score, precision_score, recall_score, f1_score,
classification_report
)

CSV_PATH = "emails.csv"
def load_csv_robust(path):
    try:
        return pd.read_csv(path)
    except Exception as e:
        print("[info] Fast CSV parse failed, falling back (engine='python', onbadlines='skip').")
        # If there are malformed rows, skip them
        df = pd.read_csv(
            path,
```

```
            engine="python",
            onbadlines="skip",     # skip offending lines that have too many/few columns
            sep=",",               # enforce comma delimiter
            quotechar='"',         # handle quoted fields
        )
        return df

def main():
    # 1) Load
    df = load_csv_robust(CSV_PATH)
    print(f"[info] Loaded shape: {df.shape}")

    # 2) Clean columns
    # Common in this dataset: an ID-like column such as 'Email No.'—drop if present.
    drop_cols = ["Prediction", "Email No."]
    feature_cols = [c for c in df.columns if c not in drop_cols]

    # Keep only numeric features (coerce anything weird)
    X = df[feature_cols].apply(pd.to_numeric, errors="coerce").fillna(0)

    # Labels
    y = pd.to_numeric(df["Prediction"], errors="coerce")
    # Drop rows where label is missing after coercion
    mask = y.notna()
    X = X.loc[mask].reset_index(drop=True)
    y = y.loc[mask].astype(int).reset_index(drop=True)

    # 3) Split
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.25, random_state=42, stratify=y
    )

    # 4) Train Naïve Bayes (Gaussian works well on count-like features too)
    model = GaussianNB()
    model.fit(X_train, y_train)

    # 5) Predict & metrics
    y_pred = model.predict(X_test)

    cm = confusion_matrix(y_test, y_pred)
    acc = accuracy_score(y_test, y_pred)
    prec = precision_score(y_test, y_pred, pos_label=1)
    rec  = recall_score(y_test, y_pred,  pos_label=1)
    f1   = f1_score(y_test, y_pred,      pos_label=1)
```

```
    print("\n=== Performance Metrics ===")
    print(f"Accuracy : {acc:.4f}")
    print(f"Precision: {prec:.4f}")
    print(f"Recall   : {rec:.4f}")
    print(f"F1-score : {f1:.4f}\n")

    print("=== Classification Report ===")
    print(classification_report(y_test, y_pred, target_names=["ham","spam"]))

    # 6) Confusion Matrix
    plt.figure(figsize=(5, 4))
    plt.imshow(cm, interpolation="nearest")
    plt.title("Confusion Matrix - Naïve Bayes (Email Spam)")
    plt.xlabel("Predicted label")
    plt.ylabel("True label")
    plt.xticks([0, 1], ["ham", "spam"], rotation=45, ha="right")
    plt.yticks([0, 1], ["ham", "spam"])
    for i in range(cm.shape[0]):
        for j in range(cm.shape[1]):
            plt.text(j, i, str(cm[i, j]), ha="center", va="center")
    plt.tight_layout()
    plt.show()

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

17.

```
# Implement the Naïve Bayes algorithm from scratch to solve a real-world classification problem
# such as email spam detection, sentiment analysis, or disease diagnosis.
import numpy as np, pandas as pd, matplotlib.pyplot as plt
# === Cell 2 — Load, clean, and split (short & error-free) ===
import numpy as np, pandas as pd, csv

# Load safely (works on any pandas version)
try:
    df = pd.read_csv("emails.csv")
except Exception:
    print("[Info] Standard read failed, using safe parser...")
    with open("emails.csv", "r", encoding="utf-8", errors="ignore") as f:
        rows = [r for r in csv.reader(f) if len(r) > 1]
    df = pd.DataFrame(rows[1:], columns=rows[0])
```

```
# Prepare features & labels
y = pd.to_numeric(df["Prediction"], errors="coerce").fillna(0).astype(int)
X = df.drop(columns=["Prediction", "Email No."], errors="ignore") \
       .apply(pd.to_numeric, errors="coerce").fillna(0)

# Stratified 75/25 split (no sklearn)
rng = np.random.default_rng(42)
idx0, idx1 = np.where(y==0)[0], np.where(y==1)[0]
rng.shuffle(idx0); rng.shuffle(idx1)
cut0, cut1 = int(0.75*len(idx0)), int(0.75*len(idx1))
train_idx = np.concatenate([idx0[:cut0], idx1[:cut1]])
test_idx  = np.concatenate([idx0[cut0:], idx1[cut1:]])
rng.shuffle(train_idx); rng.shuffle(test_idx)

X_train, y_train = X.iloc[train_idx].values, y.iloc[train_idx].values
X_test,  y_test  = X.iloc[test_idx].values,  y.iloc[test_idx].values

print("Data ready:", X_train.shape, X_test.shape)

# === Cell 3 — Naive Bayes (from scratch), metrics, confusion matrix ===
import numpy as np, pandas as pd, matplotlib.pyplot as plt

# Multinomial NB with Laplace smoothing
alpha = 1.0
classes = np.array([0, 1])
V = X_train.shape[1]

# Counts per class
cls_counts = np.array([np.sum(y_train == c) for c in classes], dtype=float)
cls_log_prior = np.log(cls_counts / cls_counts.sum())
cls_feat_sum = np.vstack([X_train[y_train == c].sum(axis=0) for c in classes])  # (2, V)

# Likelihoods: P(w|c) = (count(w,c)+alpha) / (sum_w count(w,c) + alpha*V)
smoothed = cls_feat_sum + alpha
denom = cls_feat_sum.sum(axis=1, keepdims=True) + alpha * V
log_prob = np.log(smoothed / denom)  # shape (2, V)

# Predict on test set
log_joint = X_test @ log_prob.T + cls_log_prior     # (n, 2)
y_pred = classes[np.argmax(log_joint, axis=1)]

# Confusion matrix
cm = np.zeros((2, 2), dtype=int)
```

```
for t, p in zip(y_test, y_pred):
    cm[int(t), int(p)] += 1

# Metrics with zero-division safety
total = cm.sum()
acc = (cm[0,0] + cm[1,1]) / total if total else 0.0
prec_denom = (cm[0,1] + cm[1,1])
rec_denom  = (cm[1,0] + cm[1,1])
prec = cm[1,1] / prec_denom if prec_denom else 0.0
rec  = cm[1,1] / rec_denom  if rec_denom  else 0.0
f1   = (2*prec*rec)/(prec+rec) if (prec+rec) else 0.0

print(f"Accuracy={acc:.4f}  Precision={prec:.4f}  Recall={rec:.4f}  F1={f1:.4f}")
display(pd.DataFrame(cm, index=["True_Ham","True_Spam"],
columns=["Pred_Ham","Pred_Spam"]))

# Plot confusion matrix
plt.figure(figsize=(5,4))
plt.imshow(cm, interpolation="nearest")
plt.title("Confusion Matrix — Naive Bayes (from scratch)")
plt.xlabel("Predicted"); plt.ylabel("True")
plt.xticks([0,1], ["Ham","Spam"]); plt.yticks([0,1], ["Ham","Spam"])
for i in range(2):
    for j in range(2):
        plt.text(j, i, str(cm[i,j]), ha="center", va="center")
plt.tight_layout(); plt.show()
```

18.

```
import pandas as pd, numpy as np, csv, matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MaxAbsScaler
from sklearn.svm import LinearSVC
from sklearn.metrics import (
    confusion_matrix, accuracy_score, precision_score, recall_score, f1_score, roc_auc_score
)

# Robust load (works on all pandas versions)
try:
    df = pd.read_csv("emails.csv")
except Exception:
    with open("emails.csv","r",encoding="utf-8",errors="ignore") as f:
```

```
        rows=[r for r in csv.reader(f) if len(r)>1]
    df=pd.DataFrame(rows[1:],columns=rows[0])

y = pd.to_numeric(df["Prediction"], errors="coerce").fillna(0).astype(int)
X = df.drop(columns=["Prediction","Email No."], errors="ignore") \
       .apply(pd.to_numeric, errors="coerce").fillna(0)

# Split 75/25 stratified
X_train,X_test,y_train,y_test = train_test_split(
    X.values, y.values, test_size=0.25, random_state=42, stratify=y)
print("Before:", dict(zip(*np.unique(y_train, return_counts=True))))

# Simple random oversampling (manual, no imblearn needed)
maj, minc = np.bincount(y_train).argmax(), np.bincount(y_train).argmin()
maj_idx, min_idx = np.where(y_train==maj)[0], np.where(y_train==minc)[0]
reps = int(np.ceil(len(maj_idx)/len(min_idx)))
min_aug = np.tile(min_idx, reps)[:len(maj_idx)]
idx_new = np.concatenate([maj_idx, min_aug])
np.random.default_rng(42).shuffle(idx_new)

X_train_rs, y_train_rs = X_train[idx_new], y_train[idx_new]
print("After :", dict(zip(*np.unique(y_train_rs, return_counts=True))))

# Scale and train
scaler = MaxAbsScaler()
X_train_rs = scaler.fit_transform(X_train_rs)
X_test_sc  = scaler.transform(X_test)

clf = LinearSVC(C=1.0, random_state=42)
clf.fit(X_train_rs, y_train_rs)

# Predict + metrics
y_pred = clf.predict(X_test_sc)
scores = clf.decision_function(X_test_sc)
cm = confusion_matrix(y_test, y_pred)
acc = accuracy_score(y_test, y_pred)
prec = precision_score(y_test, y_pred)
rec  = recall_score(y_test, y_pred)
f1   = f1_score(y_test, y_pred)
auc  = roc_auc_score(y_test, scores)

print(f"Acc={acc:.3f} Prec={prec:.3f} Rec={rec:.3f} F1={f1:.3f} AUC={auc:.3f}")
print(pd.DataFrame(cm, index=["True_Ham","True_Spam"],
columns=["Pred_Ham","Pred_Spam"]))
```

```
plt.imshow(cm, cmap="Blues"); plt.title("Confusion Matrix - SVM"); plt.xlabel("Pred");
plt.ylabel("True")
for i in range(2):
    for j in range(2): plt.text(j,i,cm[i,j],ha="center",va="center")
plt.show()
```

19.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
data = pd.read_csv("emails.csv")
# Separate features and target
X = data.drop(columns=["Email No.", "Prediction"])
y = data["Prediction"]
# Visualize class distribution
plt.figure(figsize=(6, 4))
sns.countplot(x=y, palette="coolwarm")
plt.xticks([0, 1], ["Normal (Not Spam)", "Abnormal (Spam)"])
plt.title("Class Distribution (Without SMOTE)")
plt.xlabel("Email Type")
plt.ylabel("Count")
plt.show()
# ---------------------- STEP 2: TRAIN-TEST SPLIT ----------------------
X_train, X_test, y_train, y_test = train_test_split(
X, y, test_size=0.2, random_state=42
)
# ---------------------- STEP 3: FEATURE SCALING ----------------------
scaler = StandardScaler(with_mean=False)
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
# ---------------------- STEP 4: TRAIN SVM MODEL ----------------------
svm_model = SVC(kernel='linear', random_state=42)
svm_model.fit(X_train_scaled, y_train)
# ---------------------- STEP 5: MAKE PREDICTIONS ----------------------
y_pred = svm_model.predict(X_test_scaled)
```

```
# ---------------------- STEP 6: MANUAL METRIC CALCULATION ----------------------

# Compute confusion matrix components
TP = np.sum((y_pred == 1) & (y_test == 1))
TN = np.sum((y_pred == 0) & (y_test == 0))
FP = np.sum((y_pred == 1) & (y_test == 0))
FN = np.sum((y_pred == 0) & (y_test == 1))

# Calculate metrics manually
precision = TP / (TP + FP) if (TP + FP) != 0 else 0
recall = TP / (TP + FN) if (TP + FN) != 0 else 0
f1 = (2 * precision * recall) / (precision + recall) if (precision + recall) != 0 else 0
accuracy = (TP + TN) / (TP + TN + FP + FN)
# ---------------------- STEP 7: DISPLAY RESULTS ----------------------
print("      Manual Evaluation Metrics (SVM Email Spam Detection):\n")
print(f"True Positives (TP): {TP}")
print(f"True Negatives (TN): {TN}")
print(f"False Positives (FP): {FP}")
print(f"False Negatives (FN): {FN}\n")
print(f"   Accuracy:  {accuracy * 100:.2f}%")
print(f"       Precision: {precision:.2f}")
print(f"    Recall:    {recall:.2f}")
print(f"          F1-Score:  {f1:.2f}")

# ---------------------- STEP 8: CONFUSION MATRIX VISUALIZATION ----------------------
cm = np.array([[TN, FP],
               [FN, TP]])

plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Predicted Normal', 'Predicted Spam'],
            yticklabels=['Actual Normal', 'Actual Spam'])
plt.title("Confusion Matrix - SVM Email Spam Detection (Manual Metrics)")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()
```

20.

```
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 SVC

df = pd.read_csv("StudentPerformance.csv")
X = df[['Study_Hours_per_Week', 'Attendance_Rate', 'Internal_Scores']].values
y = np.where(df['Final_Exam_Score'] >= 50, 1, 0)

# Check if both classes exist
unique_classes = np.unique(y)
print(f"Classes in y: {unique_classes}")

# If only one class, use median instead of fixed threshold
if len(unique_classes) == 1:
    y = np.where(df['Final_Exam_Score'] >= df['Final_Exam_Score'].median(), 1, 0)

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 = SVC(kernel='rbf', C=1.0, gamma='scale', random_state=42)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

TP = np.sum((y_pred == 1) & (y_test == 1))
TN = np.sum((y_pred == 0) & (y_test == 0))
FP = np.sum((y_pred == 1) & (y_test == 0))
FN = np.sum((y_pred == 0) & (y_test == 1))

precision = TP / (TP + FP) if (TP + FP) != 0 else 0
recall = TP / (TP + FN) if (TP + FN) != 0 else 0
f1 = (2 * precision * recall) / (precision + recall) if (precision + recall) != 0 else 0
print("\n===== Manual Evaluation Metrics =====")
print(f"True Positives (TP): {TP}")
print(f"True Negatives (TN): {TN}")
print(f"False Positives (FP): {FP}")
print(f"False Negatives (FN): {FN}\n")
print(f"Precision: {precision:.2f}")
print(f"Recall:    {recall:.2f}")
print(f"F1-Score:  {f1:.2f}")
```

21.

```
# Step 1: Import 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 SVC
from sklearn.metrics import confusion_matrix, classification_report,
roc_curve, auc
import matplotlib.pyplot as plt

# Step 2: Load dataset
df = pd.read_csv("data.csv")

# Step 3: Drop unused or missing columns
df = df.dropna(axis=1)  # remove columns with NaN
df = df.drop(columns=['id'])  # drop ID column if present

# Step 4: Encode target variable
df['target'] = df['diagnosis'].map({'M': 1, 'B': 0})
df = df.drop(columns=['diagnosis'])

# Step 5: Split features and labels
X = df.drop(columns=['target'])
y = df['target']

# Step 6: Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Step 7: Feature scaling
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

```
# Step 8: Train SVM with polynomial kernel
model = SVC(kernel='poly', degree=3, C=1.0, probability=True,
random_state=42)
model.fit(X_train, y_train)

# Step 9: Predictions
y_pred = model.predict(X_test)
y_prob = model.predict_proba(X_test)[:, 1]

# Step 10: Evaluation
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))

print("\nClassification Report:")
print(classification_report(y_test, y_pred))

# Step 11: ROC Curve
fpr, tpr, thresholds = roc_curve(y_test, y_prob)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(6, 5))
plt.plot(fpr, tpr, label=f'ROC Curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], 'r--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('SVM (Polynomial Kernel) - ROC Curve')
plt.legend()
plt.show()
```