1.

```
import heapq   # For priority queue (to get smallest f-value first)

# Graph: Cities and distances between them
city_graph = {
    'Pune': {'Lonavala': 66, 'Talegaon': 30},
    'Talegaon': {'Lonavala': 22, 'Chakan': 19},
    'Lonavala': {'Khopoli': 13},
    'Khopoli': {'Panvel': 33},
    'Panvel': {'Vashi': 20},
    'Vashi': {'Mumbai': 23},
    'Mumbai': {}
}

# Heuristic: Estimated distance (in km) from each city to Mumbai
estimated_distance = {
    'Pune': 120, 'Talegaon': 100, 'Lonavala': 85,
    'Khopoli': 70, 'Panvel': 45, 'Vashi': 25, 'Mumbai': 0,'Chakan':110
}

def a_star_search(start_city, goal_city):
    open_cities = [(0, start_city)]      # (f_score, city_name)
    actual_cost = {start_city: 0}       # Cost from start city
    previous_city = {start_city: None}   # To track the final path

    while open_cities:
        f_score, current_city = heapq.heappop(open_cities)

        # If destination is reached
        if current_city == goal_city:
            path = []
            while current_city:
                path.append(current_city)
                current_city = previous_city[current_city]
            return path[::-1], actual_cost[goal_city]

        # Explore neighboring cities
        for neighbor, distance in city_graph[current_city].items():
            new_cost = actual_cost[current_city] + distance
            total_estimated_cost = new_cost + estimated_distance[neighbor]

            # If this route is better, update details
            if neighbor not in actual_cost or new_cost < actual_cost[neighbor]:
                actual_cost[neighbor] = new_cost
                previous_city[neighbor] = current_city
```

```
            heapq.heappush(open_cities, (total_estimated_cost, neighbor))

    return None, float('inf')

# Example run
path, total_distance = a_star_search('Pune', 'Mumbai')
print("Shortest Path:", path)
print("Total Distance:", total_distance)
```

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 = []
            while goal:
                path.append(goal)
```

```
                goal = parent[goal]
            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) Traversal of a Game Map
# Author: Shravani Farkade

# ---------------------- STEP 1: REPRESENT GAME MAP ----------------------
# Each node represents a location, and edges represent possible paths.
game_map = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
```

```
    'D': [],
    'E': ['F'],
    'F': []
}

# ---------------------- STEP 3: DFS USING RECURSION ----------------------
visited = []   # To mark visited nodes
order = []     # To store traversal order

def dfs(node):
    """Recursive DFS traversal to explore all paths in the graph."""
    if node not in visited:
        print(f"Visited: {node}")
        visited.append(node)
        order.append(node)
        # Explore all neighboring nodes
        for neighbor in game_map[node]:
            dfs(neighbor)

# ---------------------- STEP 4: DFS TARGET SEARCH ----------------------
def dfs_find_target(node, target):
    """DFS traversal that stops when target node is found."""
    if node not in visited:
        print(f"Visited: {node}")
        visited.append(node)
        if node == target:
            print(f"🎯 Target '{target}' found!")
            return True
        for neighbor in game_map[node]:
            if dfs_find_target(neighbor, target):
                return True
    return False

# ---------------------- STEP 5: RUN BOTH VERSIONS ----------------------
print("\n================ DFS Recursive Traversal ================")
start_node = 'A'
dfs(start_node)
print("\n✅ DFS Traversal Order:")
print(" → ".join(order))

print("\n================ DFS Target Search ================")
visited = []
target = 'F'
found = dfs_find_target(start_node, target)
if not found:
    print(f"❌ Target '{target}' not found in the map.")
```

```
# --------------------- STEP 6: COMPLEXITY INFO ---------------------
print("\n 🧮 Time Complexity: O(V + E)")
print(" 🧮 Space Complexity: O(V)")
```

4.

```
import heapq   # For priority queue (to get smallest f-value first)

# Step 1: Maze Grid (0 = Free path, 1 = Wall)
maze = [
  [0, 1, 0, 0, 0],
  [0, 1, 0, 1, 0],
  [0, 0, 0, 1, 0],
  [0, 1, 0, 0, 0],
  [0, 0, 0, 1, 0]
]

start = (0, 0)  # Starting cell
goal = (4, 4)   # Goal cell

# Step 2: Heuristic Function (Manhattan Distance)
def heuristic(a, b):
  """Returns estimated distance between current cell and goal."""
  return abs(a[0] - b[0]) + abs(a[1] - b[1])

# Step 3: Get Valid Neighboring Cells (4 directions)
def get_neighbors(cell):
  directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]  # Down, Up, Right, Left
  neighbors = []
  for dr, dc in directions:
    r, c = cell[0] + dr, cell[1] + dc
    if 0 <= r < len(maze) and 0 <= c < len(maze[0]) and maze[r][c] == 0:
      neighbors.append((r, c))
  return neighbors

# Step 4: A* Search Algorithm
def a_star_search(start, goal):
  open_cells = [(0, start)]       # (f_score, cell)
  actual_cost = {start: 0}        # g(n): cost from start
  previous_cell = {start: None}   # For reconstructing the path

  while open_cells:
```

```
        f_score, current = heapq.heappop(open_cells)

        # ✅ Goal Check
        if current == goal:
            path = []
            while current:
                path.append(current)
                current = previous_cell[current]
            return path[::-1], actual_cost[goal]

        # Explore all valid neighbors
        for neighbor in get_neighbors(current):
            new_cost = actual_cost[current] + 1  # cost per step
            total_estimated = new_cost + heuristic(neighbor, goal)

            # If better path found, update values
            if neighbor not in actual_cost or new_cost < actual_cost[neighbor]:
                actual_cost[neighbor] = new_cost
                previous_cell[neighbor] = current
                heapq.heappush(open_cells, (total_estimated, neighbor))

    return None, float('inf')

# Step 5: Run the Algorithm
path, total_cost = a_star_search(start, goal)

# Step 6: Display Results
if path:
    print("✅ Shortest Path Found:")
    print(path)
    print(f"🧭 Total Steps: {len(path) - 1}")
else:
    print("❌ No Path Found.")
```

5.

```
from collections import deque

# Print the puzzle
```

```python
def print_board(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()

# Get all valid next states
def get_neighbors(state):
    neighbors = []
    zero = state.index(0)

    moves = {
        "Up": -3,
        "Down": 3,
        "Left": -1,
        "Right": 1
    }

    for move, diff in moves.items():
        new_pos = zero + diff

        # invalid left/right jumps
        if move == "Left" and zero % 3 == 0:
            continue
        if move == "Right" and zero % 3 == 2:
            continue
        if new_pos < 0 or new_pos > 8:
            continue

        new_state = state[:]
        new_state[zero], new_state[new_pos] = new_state[new_pos], new_state[zero]

        neighbors.append((new_state, move))

    return neighbors

# BFS search
def solve_puzzle(start, goal):
    queue = deque()
    queue.append((start, []))
    visited = set()
    visited.add(tuple(start))
    while queue:
        state, path = queue.popleft()

        if state == goal:
            return path
        for new_state, move in get_neighbors(state):
```

```
        t = tuple(new_state)
        if t not in visited:
            visited.add(t)
            queue.append((new_state, path + [move]))
    return None

# MAIN
start = [1, 2, 3,
         4, 0, 5,
         6, 7, 8]

goal = [1, 2, 3,
        4, 5, 6,
        7, 8, 0]

print("Start State:")
print_board(start)

solution = solve_puzzle(start, goal)

if solution:
    print("Solved in", len(solution), "moves")
    print("Moves:", solution)
else:
    print("No solution found")
```

6.

```
# 🎮 Tic-Tac-Toe Game (2 Player)
# Author: Shravani Farkade

# Step 1: Initialize the board
board = [' ' for _ in range(9)]

# Step 2: Display the board
def print_board():
    print()
    print(board[0] + ' | ' + board[1] + ' | ' + board[2])
    print('--+---+--')
    print(board[3] + ' | ' + board[4] + ' | ' + board[5])
    print('--+---+--')
    print(board[6] + ' | ' + board[7] + ' | ' + board[8])
```

```
    print()

# Step 3: Check for a winner
def check_winner(player):
    win_conditions = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],  # Rows
        [0, 3, 6], [1, 4, 7], [2, 5, 8],  # Columns
        [0, 4, 8], [2, 4, 6]              # Diagonals
    ]
    for combo in win_conditions:
        if all(board[i] == player for i in combo):
            return True
    return False

# Step 4: Check if the board is full (Draw)
def is_full():
    return ' ' not in board

# Step 5: Main game loop
def play_game():
    current_player = 'X'

    while True:
        print_board()
        print(f"Player {current_player}'s turn.")
        move = int(input("Enter your move (1-9): ")) - 1

        if board[move] != ' ':
            print("❌ That spot is already taken! Try again.")
            continue

        board[move] = current_player

        if check_winner(current_player):
            print_board()
            print(f"🎉 Player {current_player} wins!")
            break

        if is_full():
            print_board()
            print("🤝 It's a draw!")
            break

        # Switch player
        current_player = 'O' if current_player == 'X' else 'X'

# Step 6: Start the game
```

```
play_game()
```

7.

```
# 🏰 Tower of Hanoi - Recursive Implementation
# Author: Shravani Farkade

def tower_of_hanoi(n, source, auxiliary, destination):
    # Base Case: Only one disk
    if n == 1:
        print(f"Move disk 1 from {source} → {destination}")
        return

    # Step 1: Move n-1 disks from source to auxiliary
    tower_of_hanoi(n - 1, source, destination, auxiliary)

    # Step 2: Move the largest disk from source to destination
    print(f"Move disk {n} from {source} → {destination}")

    # Step 3: Move n-1 disks from auxiliary to destination
    tower_of_hanoi(n - 1, auxiliary, source, destination)

# Step 4: Take number of disks as input
n = int(input("Enter number of disks: "))

print("\n 📦 Tower of Hanoi Solution Steps:")
tower_of_hanoi(n, 'A', 'B', 'C')

print(f"\n ✅ Total Moves Required: {2**n - 1}")
```

8.

```
# 💧 Water Jug Problem using BFS
# Author: Shravani Farkade

from collections import deque

# Step 1: Define capacities
jugA, jugB = 4, 3  # capacities of the jugs
target = 2         # goal: measure exactly 2 liters

# Step 2: BFS to find solution
def water_jug_bfs():
    visited = set()            # store visited states
    queue = deque([(0, 0)])      # initial state (both jugs empty)

    while queue:
        a, b = queue.popleft()

        # If goal is reached
        if a == target or b == target:
            print(f"✅ Solution found: ({a}, {b})")
            return

        # Skip already visited states
        if (a, b) in visited:
            continue
        visited.add((a, b))

        # Print current state
        print(f"Visited: ({a}, {b})")

        # Step 3: Generate all possible next states
        possible_states = [
            (jugA, b),      # Fill Jug A
            (a, jugB),      # Fill Jug B
            (0, b),         # Empty Jug A
            (a, 0),         # Empty Jug B
            (a - min(a, jugB - b), b + min(a, jugB - b)),  # Pour A → B
            (a + min(b, jugA - a), b - min(b, jugA - a))   # Pour B → A
        ]

        for state in possible_states:
            if state not in visited:
```

```
            queue.append(state)

    print("❌ No solution found.")

# Step 4: Run the BFS function
water_jug_bfs()
```

9. and 10

```
# -----------------------------------------------------------
# UBER RIDE PRICE PREDICTION USING PCA (sklearn) +
# LINEAR REGRESSION (From Scratch)
# Author: Shravani Farkade
# Dataset: uber.csv
# -----------------------------------------------------------

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.metrics import mean_squared_error, mean_absolute_error, r2_score

df = pd.read_csv("/mnt/data/uber - uber.csv")
print("✅ Dataset Loaded Successfully!\n")
print(df.head())

cols_to_drop = ['Unnamed: 0', 'key']
df = df.drop(columns=[c for c in cols_to_drop if c in df.columns], errors='ignore')

# Drop missing values
df = df.dropna()

# Convert datetime column
if 'pickup_datetime' in df.columns:
    df['pickup_datetime'] = pd.to_datetime(df['pickup_datetime'], errors='coerce')
    df['hour'] = df['pickup_datetime'].dt.hour
    df['day'] = df['pickup_datetime'].dt.day
    df['month'] = df['pickup_datetime'].dt.month
```

```
    df['year'] = df['pickup_datetime'].dt.year
    df = df.drop(columns=['pickup_datetime'])

print("\n 🧹 Cleaned Dataset Info:")
print(df.info())

print("\n 📊 Basic Statistics:")
print(df.describe())

# Correlation heatmap
plt.figure(figsize=(10, 6))
sns.heatmap(df.corr(), annot=True, cmap='coolwarm')
plt.title("Feature Correlation Heatmap (Uber Dataset)")
plt.show()

# Fare distribution
plt.figure(figsize=(8, 5))
sns.histplot(df['fare_amount'], bins=40, kde=True, color='skyblue')
plt.title("Distribution of Fare Amounts")
plt.xlabel("Fare Amount ($)")
plt.ylabel("Frequency")
plt.show()

# Passenger count distribution
plt.figure(figsize=(7, 4))
sns.countplot(x='passenger_count', data=df, palette='viridis')
plt.title("Passenger Count Distribution")
plt.show()

# ---------------------------------------------------------
# Step 4: Train-Test Split
# ---------------------------------------------------------
X = df.drop(columns=['fare_amount'])
y = df['fare_amount'].values.reshape(-1, 1)

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

# ---------------------------------------------------------
# Step 5: Linear Regression (From Scratch)
# ---------------------------------------------------------
def linear_regression_train(X_train, y_train):
    X_b = np.c_[np.ones((X_train.shape[0], 1)), X_train]  # Add intercept
    theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y_train)
    return theta_best
```

```
def linear_regression_predict(X_test, theta):
    X_b = np.c_[np.ones((X_test.shape[0], 1)), X_test]
    return X_b.dot(theta)

# Train model without PCA
theta_no_pca = linear_regression_train(X_train, y_train)
y_pred_no_pca = linear_regression_predict(X_test, theta_no_pca)

r2_no_pca = r2_score(y_test, y_pred_no_pca)
rmse_no_pca = np.sqrt(mean_squared_error(y_test, y_pred_no_pca))
mae_no_pca = mean_absolute_error(y_test, y_pred_no_pca)

print("\n📈 Model Performance (Without PCA):")
print(f"R² Score : {r2_no_pca:.4f}")
print(f"RMSE     : {rmse_no_pca:.4f}")
print(f"MAE      : {mae_no_pca:.4f}")

# -----------------------------------------------------------
# Step 6: PCA using sklearn
# -----------------------------------------------------------
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

pca = PCA(n_components=3)
X_pca = pca.fit_transform(X_scaled)

print("\n🔢 Explained Variance Ratio by PCA:")
print(pca.explained_variance_ratio_)

# PCA Visualization (first 2 components)
plt.figure(figsize=(7, 5))
sns.scatterplot(x=X_pca[:, 0], y=X_pca[:, 1], hue=y.flatten(), palette='coolwarm',
alpha=0.6)
plt.title("PCA Visualization (First 2 Components)")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.show()

# -----------------------------------------------------------
# Step 7: Linear Regression on PCA-transformed Data (From Scratch)
# -----------------------------------------------------------
X_train_pca, X_test_pca, y_train_pca, y_test_pca = train_test_split(
    X_pca, y, test_size=0.2, random_state=42
)

theta_pca = linear_regression_train(X_train_pca, y_train_pca)
y_pred_pca = linear_regression_predict(X_test_pca, theta_pca)
```

```
r2_pca = r2_score(y_test_pca, y_pred_pca)
rmse_pca = np.sqrt(mean_squared_error(y_test_pca, y_pred_pca))
mae_pca = mean_absolute_error(y_test_pca, y_pred_pca)

print("\n 📊 Model Performance (With PCA):")
print(f"R² Score : {r2_pca:.4f}")
print(f"RMSE     : {rmse_pca:.4f}")
print(f"MAE      : {mae_pca:.4f}")

# ---------------------------------------------------------
# Step 8: Comparison Summary
# ---------------------------------------------------------
comparison = pd.DataFrame({
  "Model": ["Without PCA", "With PCA"],
  "R² Score": [r2_no_pca, r2_pca],
  "RMSE": [rmse_no_pca, rmse_pca],
  "MAE": [mae_no_pca, mae_pca]
})

print("\n 🔍 Performance Comparison:")
print(comparison)

plt.figure(figsize=(8, 5))
sns.barplot(
  data=comparison.melt(id_vars='Model', var_name='Metric', value_name='Value'),
  x='Metric', y='Value', hue='Model', palette='viridis'
)
plt.title("Uber Ride Price Prediction: Model Comparison (With vs Without PCA)")
plt.show()

# ---------------------------------------------------------
# Step 9: Interpretation
# ---------------------------------------------------------
print("\n 📘 Interpretation:")
print("→ Linear Regression implemented manually using Normal Equation.")
print("→ PCA (from sklearn) reduced dimensionality to 3 components.")
print("→ Models evaluated using R², RMSE, and MAE for fair comparison.")
print("→ Visualization shows feature correlations and PCA projections.")
print("→ PCA helps simplify data and reduce redundancy, but may not always improve accuracy.")
```

11

```
# ---------------------------------------------------------
# HOUSE PRICE PREDICTION USING LINEAR REGRESSION (From Scratch) + K-Fold CV
# Author: Shravani Farkade
# ---------------------------------------------------------

import pandas as pd
import numpy as np
from sklearn.model_selection import KFold
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

df = pd.read_csv("/mnt/data/Housing Dataset.csv")  # Adjust path if needed
print("✅ Dataset Loaded Successfully!")

df = df[['price', 'sqft_living', 'bedrooms', 'city']]

le = LabelEncoder()
df['city_encoded'] = le.fit_transform(df['city'])

X = df[['sqft_living', 'bedrooms', 'city_encoded']].values
y = df['price'].values


def linear_regression_train(X_train, y_train):
  # Add bias (intercept) term
  X_b = np.c_[np.ones((X_train.shape[0], 1)), X_train]
  # Normal Equation: θ = (XᵀX)^(-1) Xᵀy
  theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y_train)
  return theta_best

def linear_regression_predict(X_test, theta):
  X_b = np.c_[np.ones((X_test.shape[0], 1)), X_test]
  return X_b.dot(theta)

# ---------------------------------------------------------
# Step 4: Apply 5-Fold Cross Validation
# ---------------------------------------------------------
k = 5
kf = KFold(n_splits=k, shuffle=True, random_state=42)

r2_scores = []
rmse_scores = []
mae_scores = []

for train_index, test_index in kf.split(X):
```

```
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # Train model from scratch
    theta = linear_regression_train(X_train, y_train)

    # Predict
    y_pred = linear_regression_predict(X_test, theta)

    # Evaluate
    r2_scores.append(r2_score(y_test, y_pred))
    rmse_scores.append(np.sqrt(mean_squared_error(y_test, y_pred)))
    mae_scores.append(mean_absolute_error(y_test, y_pred))

print("\n 📊 Linear Regression Model Evaluation (K-Fold Cross Validation - From Scratch)")
print(f"Number of folds: {k}")
print("-" * 60)
print(f"R² Scores for each fold: {np.round(r2_scores, 4)}")
print(f"Average R² Score: {np.mean(r2_scores):.4f}")
print("-" * 60)
print(f"Average RMSE: {np.mean(rmse_scores):.2f}")
print(f"Average MAE: {np.mean(mae_scores):.2f}")

# ----------------------------------------------------------
# Step 6: Train Final Model on Entire Dataset
# ----------------------------------------------------------
theta_final = linear_regression_train(X, y)
print("\n ✅ Final Model trained on full dataset successfully!")

# ----------------------------------------------------------
# Step 7: Predict Example House Price
# ----------------------------------------------------------
example = np.array([[2000, 3, 10]])  # Example house input
predicted_price = linear_regression_predict(example, theta_final)
print(f"\n 🏠 Predicted price for a 2000 sqft, 3-bedroom house (city code 10): ${predicted_price[0]:,.2f}")

print("\n 📘 Interpretation:")
print("→ Linear Regression implemented manually using Normal Equation.")
print("→ Model evaluated with 5-Fold Cross Validation.")
print("→ Add more features (bathrooms, year built, etc.) to improve accuracy.")
```

12.

```
# ---------------------------------------------------------
# Linear Regression from Scratch
# Predict Exam Score based on Study Hours
# Author: Shravani Farkade
# ---------------------------------------------------------

import pandas as pd
import numpy as np

# Step 1: Load dataset
df = pd.read_csv("/content/sample_data/StudentPerformance.csv")

# Select relevant columns
X = df['Hours_Studied'].values
y = df['Exam_Score'].values

# Step 2: Calculate mean values
x_mean = np.mean(X)
y_mean = np.mean(y)

# Step 3: Compute slope (b1) and intercept (b0)
b1 = np.sum((X - x_mean) * (y - y_mean)) / np.sum((X - x_mean)**2)
b0 = y_mean - b1 * x_mean

print(f"📘 Model Parameters: Intercept (b0) = {b0:.4f}, Slope (b1) = {b1:.4f}")

# Step 4: Predict scores
y_pred = b0 + b1 * X

# Step 5: Model evaluation
mse = np.mean((y - y_pred)**2)
r2 = 1 - (np.sum((y - y_pred)**2) / np.sum((y - y_mean)**2))

print("\n📊 Model Evaluation:")
print(f"Mean Squared Error (MSE): {mse:.4f}")
print(f"R² Score: {r2:.4f}")

# Step 6: Predict for a given study hour
hours = 8
predicted_score = b0 + b1 * hours
print(f"\n🧮 Predicted Exam Score for {hours} hours of study: {predicted_score:.2f}")
```

13.

```
# --------------------------------------------------------
# MULTIPLE LINEAR REGRESSION (From Scratch) + K-Fold CV
# Author: Shravani Farkade
# --------------------------------------------------------

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score

# --------------------------------------------------------
# Step 1: Load Dataset
# --------------------------------------------------------
df = pd.read_csv("/content/sample_data/Student_Marks.csv")
print("✅ Dataset Loaded Successfully!\n")
print(df.head())

# --------------------------------------------------------
# Step 2: Data Preprocessing
# --------------------------------------------------------
# Drop ID column if present
if 'student_id' in df.columns:
   df = df.drop(columns=['student_id'])

# Define features (X) and target (y)
X = df[['hours_studied', 'attendance_percent', 'previous_scores']].values
y = df['exam_score'].values.reshape(-1, 1)

# --------------------------------------------------------
# Step 3: Linear Regression Functions (From Scratch)
# --------------------------------------------------------
def linear_regression_train(X_train, y_train):
   """Train Linear Regression using the Normal Equation"""
   X_b = np.c_[np.ones((X_train.shape[0], 1)), X_train]      # Add intercept
   theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y_train)
   return theta_best

def linear_regression_predict(X_test, theta):
   """Predict using learned parameters"""
   X_b = np.c_[np.ones((X_test.shape[0], 1)), X_test]       # Add intercept
   return X_b.dot(theta)
```

```
# --------------------------------------------------------
# Step 4: Apply K-Fold Cross Validation
# --------------------------------------------------------
k = 5
kf = KFold(n_splits=k, shuffle=True, random_state=42)

r2_scores = []
rmse_scores = []
mae_scores = []

for train_index, test_index in kf.split(X):
   X_train, X_test = X[train_index], X[test_index]
   y_train, y_test = y[train_index], y[test_index]

   theta = linear_regression_train(X_train, y_train)
   y_pred = linear_regression_predict(X_test, theta)

   r2_scores.append(r2_score(y_test, y_pred))
   rmse_scores.append(np.sqrt(mean_squared_error(y_test, y_pred)))
   mae_scores.append(mean_absolute_error(y_test, y_pred))

# --------------------------------------------------------
# Step 5: Display Evaluation Results
# --------------------------------------------------------
print("\n📊 Model Evaluation (5-Fold Cross Validation - From Scratch)")
print("-" * 65)
print(f"R² Scores per Fold: {np.round(r2_scores, 4)}")
print(f"Average R² Score : {np.mean(r2_scores):.4f}")
print(f"Average RMSE     : {np.mean(rmse_scores):.2f}")
print(f"Average MAE      : {np.mean(mae_scores):.2f}")

# --------------------------------------------------------
# Step 6: Train Final Model on Full Dataset
# --------------------------------------------------------
theta_final = linear_regression_train(X, y)
print("\n✅ Final Model Coefficients (θ):")
print(theta_final)

# --------------------------------------------------------
# Step 7: Predict Example Student's Exam Score
# --------------------------------------------------------
example = np.array([[1, 8, 85, 75]])      # 1 = intercept term
predicted_score = float(example.dot(theta_final))

print(f"\n🧮 Predicted Exam Score for student (8h, 85%, prev 75):
{predicted_score:.2f}")
```

```
# --------------------------------------------------------
# Step 8: Visualization
# --------------------------------------------------------
sns.set(style="whitegrid", font_scale=1.1)


# R² Score Bar Plot
plt.figure(figsize=(7, 4))
sns.barplot(x=np.arange(1, k + 1), y=r2_scores, palette="viridis")
plt.title("R² Scores Across 5 Folds (K-Fold Cross Validation)")
plt.xlabel("Fold Number")
plt.ylabel("R² Score")
plt.ylim(0, 1)
for i, v in enumerate(r2_scores):
  plt.text(i, v + 0.02, f"{v:.2f}", ha='center', fontsize=10)
plt.show()

# --------------------------------------------------------
# Step 9: Interpretation
# --------------------------------------------------------
print("\n 📘 Interpretation:")
print("→ Model trained using Normal Equation (no sklearn LinearRegression).")
print("→ Features: study hours, attendance, and previous internal scores.")
print("→ R² shows how much variance in exam scores is explained by inputs.")
print("→ RMSE and MAE represent average prediction error.")
print("→ Randomly scattered residuals indicate a good linear fit.")
```

14.

```
# --------------------------------------------------------
# IT SALARY PREDICTION USING LINEAR REGRESSION (From Scratch) + 5-Fold CV
# Author: Shravani Farkade
# --------------------------------------------------------

import pandas as pd
import numpy as np
from sklearn.model_selection import KFold
from sklearn.preprocessing import LabelEncoder
```

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

df = pd.read_csv("/mnt/data/IT_Salaries.csv")
print("✅ Dataset Loaded Successfully!")
print(df.head())

df = df[['salary_usd', 'years_experience', 'education_required', 'required_skills']]

le_edu = LabelEncoder()
le_skill = LabelEncoder()
df['education_encoded'] = le_edu.fit_transform(df['education_required'])
df['skills_encoded'] = le_skill.fit_transform(df['required_skills'])

X = df[['years_experience', 'education_encoded', 'skills_encoded']].values
y = df['salary_usd'].values.reshape(-1, 1)   # reshaped for consistency

def linear_regression_train(X_train, y_train):
  X_b = np.c_[np.ones((X_train.shape[0], 1)), X_train]  # Add intercept
  theta_best = np.linalg.inv(X_b.T.dot(X_b)).dot(X_b.T).dot(y_train)
  return theta_best

def linear_regression_predict(X_test, theta):
  X_b = np.c_[np.ones((X_test.shape[0], 1)), X_test]
  return X_b.dot(theta)

k = 5
kf = KFold(n_splits=k, shuffle=True, random_state=42)

r2_scores, rmse_scores, mae_scores = [], [], []

for train_index, test_index in kf.split(X):
  X_train, X_test = X[train_index], X[test_index]
  y_train, y_test = y[train_index], y[test_index]

  # Train model
  theta = linear_regression_train(X_train, y_train)
  y_pred = linear_regression_predict(X_test, theta)

  # Evaluate
  r2_scores.append(r2_score(y_test, y_pred))
  rmse_scores.append(np.sqrt(mean_squared_error(y_test, y_pred)))
  mae_scores.append(mean_absolute_error(y_test, y_pred))

# ---------------------------------------------------------
# Step 5: Display Results
# ---------------------------------------------------------
```

```
print("\n 📊 Linear Regression Model Evaluation (5-Fold Cross Validation - From
Scratch)")
print("-" * 70)
print(f"R² Scores per Fold: {np.round(r2_scores, 4)}")
print(f"Average R² Score: {np.mean(r2_scores):.4f}")
print(f"Average RMSE: {np.mean(rmse_scores):.2f}")
print(f"Average MAE: {np.mean(mae_scores):.2f}")

theta_final = linear_regression_train(X, y)
print("\n ✅ Final Model Trained Successfully!")
print("Model Coefficients (θ):")
print(theta_final)

example = np.array([[5, le_edu.transform(['Bachelor'])[0], le_skill.transform(['Python,
SQL'])[0]]])
predicted_salary = linear_regression_predict(example, theta_final)[0, 0]

print(f"\n 💼 Predicted Salary for 5 yrs exp (Bachelor, Python+SQL):
${predicted_salary:,.2f}")

# ---------------------------------------------------------
# Step 8: Interpretation
# ---------------------------------------------------------
print("\n 📘 Interpretation:")
print("→ Linear Regression implemented manually using Normal Equation.")
print("→ Model predicts IT professionals' salaries based on experience, education, and
skills.")
print("→ Evaluated using 5-Fold Cross Validation for reliability.")
print("→ Results show how much each factor influences salary.")
```

16.

```
# 📧 Naïve Bayes From Scratch - Simple Version
# Real-world problem: Email Spam Detection

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sns
```

```python
# Step 1: Load dataset
data = pd.read_csv("/content/emails.csv")

# Step 2: Prepare data
X = data.drop(columns=["Email No.", "Prediction"]).values
y = data["Prediction"].values

plt.figure(figsize=(6,4))
sns.countplot(x="Prediction",data=data,palette='coolwarm')
plt.xticks=([0,1],["Not Spam","Spam"])
plt.title("Classes vs Count")
plt.xlabel("Classes")
plt.ylabel("Count")
plt.show()

# Split into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Step 3: Implement Naive Bayes from scratch
class NaiveBayes:
  def fit(self, X, y):
    self.classes = np.unique(y)
    self.class_priors = {}
    self.feature_probs = {}
    n_features = X.shape[1]

    for c in self.classes:
      X_c = X[y == c]
      # Prior probability P(C)
      self.class_priors[c] = X_c.shape[0] / X.shape[0]
      # Likelihood P(x_i | C) using Laplace smoothing
      self.feature_probs[c] = (X_c.sum(axis=0) + 1) / (X_c.sum() + n_features)

  def predict(self, X):
    y_pred = []
    for x in X:
      posteriors = []
      for c in self.classes:
        # log(P(C)) + Σ log(P(x_i|C))
        prior = np.log(self.class_priors[c])
        likelihood = np.sum(x * np.log(self.feature_probs[c]))
        posterior = prior + likelihood
        posteriors.append(posterior)
      y_pred.append(self.classes[np.argmax(posteriors)])
    return np.array(y_pred)

# Step 4: Train and test model
```

```
model = NaiveBayes()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

# Step 5: Evaluate results
acc = accuracy_score(y_test, y_pred)
cm = confusion_matrix(y_test, y_pred)

print("📊 Naïve Bayes From Scratch - Email Spam Detection")
print("Accuracy:", round(acc, 4))
print("Confusion Matrix:\n", cm)
```

18.

```
# 📧 Email Spam Detection using Support Vector Machine (SVM) + SMOTE
Oversampling
# Author: Shravani Farkade

# ---------------------- IMPORT LIBRARIES ----------------------
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.metrics import (
   classification_report, confusion_matrix, ConfusionMatrixDisplay, accuracy_score)
from imblearn.over_sampling import SMOTE
import matplotlib.pyplot as plt
import seaborn as sns

# ---------------------- STEP 1: LOAD DATA ----------------------
data = pd.read_csv("/content/emails.csv")

# Separate features and target
X = data.drop(columns=["Email No.", "Prediction"])
y = data["Prediction"]

# Visualize class distribution before oversampling
plt.figure(figsize=(6, 4))
sns.countplot(x=y, palette="coolwarm")
plt.xticks([0, 1], ["Normal (Not Spam)", "Abnormal (Spam)"])
plt.title("Before SMOTE - Class Distribution")
```

```
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: APPLY SMOTE FOR OVERSAMPLING ----------------------
smote = SMOTE(random_state=42)
X_train_res, y_train_res = smote.fit_resample(X_train, y_train)

# Visualize class distribution after SMOTE
plt.figure(figsize=(6, 4))
sns.countplot(x=y_train_res, palette="coolwarm")
plt.xticks([0, 1], ["Normal (Not Spam)", "Abnormal (Spam)"])
plt.title("After SMOTE - Balanced Class Distribution")
plt.xlabel("Email Type")
plt.ylabel("Count")
plt.show()

# ---------------------- STEP 4: FEATURE SCALING ----------------------
scaler = StandardScaler(with_mean=False)
X_train_scaled = scaler.fit_transform(X_train_res)
X_test_scaled = scaler.transform(X_test)

# ---------------------- STEP 5: TRAIN SVM MODEL ----------------------
svm_model = SVC(kernel='linear', random_state=42)
svm_model.fit(X_train_scaled, y_train_res)

# ---------------------- STEP 6: MAKE PREDICTIONS ----------------------
y_pred = svm_model.predict(X_test_scaled)

# ---------------------- STEP 7: EVALUATE PERFORMANCE ----------------------
# Accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"✅ Model Accuracy: {accuracy * 100:.2f}%\n")

# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=["Normal",
"Spam"])
disp.plot(cmap="Blues")
plt.title("Confusion Matrix - SVM Email Spam Detection (with SMOTE)")
plt.show()
```

```
# Classification Report
print("📊 SVM Email Spam Detection (with SMOTE) - Classification Report")
print(classification_report(y_test, y_pred, target_names=["Normal", "Spam"]))
```

19.

```
# 📨 Email Spam Detection using Support Vector Machine (SVM)
# Author: Shravani Farkade

# ---------------------- IMPORT LIBRARIES ----------------------
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

# ---------------------- STEP 1: LOAD DATA ----------------------
data = pd.read_csv("/content/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.

```
# 🎓 SVM with Polynomial Kernel (sklearn)
# + Manual Metric Calculation (Precision, Recall, F1)
# Author: Shravani Farkade

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

# ------------------------------
# Step 1: Load & preprocess data
# ------------------------------
df = pd.read_csv("/mnt/data/StudentPerformance.csv")

# Select key numeric features
X = df[['Hours_Studied', 'Attendance', 'Previous_Scores']].values
y = np.where(df['Exam_Score'] >= 50, 1, 0)  # Pass=1, Fail=0

# Train-test split
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 2: Train SVM with Polynomial Kernel
# ------------------------------
model = SVC(kernel='poly', degree=3, coef0=1, C=1.0)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

# ------------------------------
# Step 3: 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

# Print results
print("📊 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.

```
# 🎗️ Breast Cancer Classification using Polynomial SVM
# Manual Metrics + Manual AUC
# Author: Shravani Farkade

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 roc_curve
import matplotlib.pyplot as plt

# ---------------------- STEP 1: LOAD AND PREPROCESS DATA ----------------------
df = pd.read_csv("/content/Breast Cancer Wisconsin (Diagnostic)_21.csv")
```

```
# Drop 'id' column and encode target
df = df.drop(columns=['id'])
df['diagnosis'] = np.where(df['diagnosis'] == 'M', 1, 0)

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

# ---------------------- 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, stratify=y
)

# ---------------------- STEP 3: FEATURE SCALING ----------------------
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# ---------------------- STEP 4: TRAIN POLYNOMIAL SVM ----------------------
# Enable probability=True to use predict_proba()
model = SVC(kernel='poly', degree=2, coef0=1, C=5.0, probability=True,
random_state=42)
model.fit(X_train, y_train)

# ---------------------- STEP 5: PREDICTION ----------------------
y_pred = model.predict(X_test)
y_prob = model.predict_proba(X_test)[:, 1]  # ✅ Works now

# ---------------------- STEP 6: MANUAL METRICS ----------------------
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))

accuracy = (TP + TN) / (TP + TN + FP + FN)
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

# ---------------------- STEP 7: DISPLAY CONFUSION MATRIX ----------------------
cm = pd.DataFrame(
    np.array([[TP, FN],
              [FP, TN]]),
    columns=['Predicted: Positive', 'Predicted: Negative'],
    index=['Actual: Positive', 'Actual: Negative']
)
```

```
print("\n 📊 Confusion Matrix (Manual):")
print(cm)

# ---------------------- STEP 8: PRINT MANUAL METRICS ----------------------
print("\n ✅ Manual Evaluation Metrics:")
print(f"Accuracy:  {accuracy * 100:.2f}%")
print(f"Precision: {precision:.2f}")
print(f"Recall:    {recall:.2f}")
print(f"F1-Score:  {f1:.2f}")

# ---------------------- STEP 9: MANUAL AUC CALCULATION ----------------------
fpr, tpr, _ = roc_curve(y_test, y_prob)
manual_auc = np.trapz(tpr, fpr)  # Trapezoidal integration

print(f"\n 💎 Manual AUC (Area Under ROC Curve): {manual_auc:.2f}")

# ---------------------- STEP 10: PLOT ROC CURVE ----------------------
plt.figure(figsize=(6, 5))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC Curve (Manual AUC =
{manual_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - Polynomial SVM (Manual Metrics)')
plt.legend(loc='lower right')
plt.show()
```

```
# 📧 Naïve Bayes From Scratch - Sentiment Analysis Version
# Author: Shravani Farkade

# ---------------------- IMPORT LIBRARIES ----------------------
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.feature_extraction.text import CountVectorizer
import matplotlib.pyplot as plt
import seaborn as sns
import re

# ---------------------- STEP 1: LOAD DATA ----------------------
data = pd.read_csv("/mnt/data/sentiment_analysis_16_17.csv")

# Select relevant columns
```

```
texts = data["text"]
labels = data["sentiment"]

# ---------------------- STEP 2: VISUALIZE CLASS DISTRIBUTION ----------------------
plt.figure(figsize=(6,4))
sns.countplot(x=labels, palette='coolwarm')
plt.title("Sentiment Class Distribution")
plt.xlabel("Sentiment")
plt.ylabel("Count")
plt.show()

# ---------------------- STEP 3: TEXT CLEANING FUNCTION ----------------------
def clean_text(text):
    text = text.lower()
    text = re.sub(r'[^a-z\s]', '', text)
    return text

data["clean_text"] = texts.apply(clean_text)

# ---------------------- STEP 4: FEATURE EXTRACTION (BAG OF WORDS) ----------------------
vectorizer = CountVectorizer(stop_words='english')
X = vectorizer.fit_transform(data["clean_text"]).toarray()
y = labels.values

# ---------------------- STEP 5: SPLIT INTO TRAIN AND TEST ----------------------
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# ---------------------- STEP 6: IMPLEMENT NAIVE BAYES ----------------------
class NaiveBayes:
    def fit(self, X, y):
        self.classes = np.unique(y)
        self.class_priors = {}
        self.feature_probs = {}
        n_features = X.shape[1]

        for c in self.classes:
            X_c = X[y == c]
            # Prior P(C)
            self.class_priors[c] = X_c.shape[0] / X.shape[0]
            # Likelihood P(x_i|C) using Laplace smoothing
            self.feature_probs[c] = (X_c.sum(axis=0) + 1) / (X_c.sum() + n_features)

    def predict(self, X):
        y_pred = []
        for x in X:
```

```
            posteriors = []
            for c in self.classes:
                prior = np.log(self.class_priors[c])
                likelihood = np.sum(x * np.log(self.feature_probs[c]))
                posterior = prior + likelihood
                posteriors.append(posterior)
            y_pred.append(self.classes[np.argmax(posteriors)])
        return np.array(y_pred)

# ---------------------- STEP 7: TRAIN AND PREDICT ----------------------
model = NaiveBayes()
model.fit(X_train, y_train)
y_pred = model.predict(X_test)

# ---------------------- STEP 8: EVALUATE ----------------------
acc = accuracy_score(y_test, y_pred)
cm = confusion_matrix(y_test, y_pred)

print("📊 Naïve Bayes From Scratch - Sentiment Analysis")
print(f"✅ Accuracy: {acc * 100:.2f}%")
print("Confusion Matrix:\n", cm)

# Visualize confusion matrix
plt.figure(figsize=(5,4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Purples',
            xticklabels=np.unique(y), yticklabels=np.unique(y))
plt.title("Confusion Matrix - Naïve Bayes (Sentiment Analysis)")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```