1st code informed search by using A* algorithm

```
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': {},
   'Chakan': {}  # explicit dead end so no KeyError when expanding
}

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

def a_star_search(start_city, goal_city):
   open_cities = [(estimated_distance[start_city], start_city)]  # (f_score, city)
   actual_cost = {start_city: 0}         # g value
   previous_city = {start_city: None}    # for path reconstruction

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

      # Goal check
      if current_city == goal_city:
         path = []
         while current_city is not None:
            path.append(current_city)
            current_city = previous_city[current_city]
         return path[::-1], actual_cost[goal_city]

      # Explore neighbors
      for neighbor, distance in city_graph[current_city].items():
         new_cost = actual_cost[current_city] + distance              # g'
         total_estimated_cost = new_cost + estimated_distance[neighbor] # f' = g'+h
```

```
            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
if __name__ == "__main__":
    path, total_distance = a_star_search('Pune', 'Mumbai')
    print("Shortest Path:", path)
    print("Total Distance:", total_distance)
```

2nd code

```
from collections import deque

def bfs_shortest_path(maze, start_cell, goal_cell):
    # Get maze dimensions
    total_rows = len(maze)
    total_cols = len(maze[0])

    # 4 possible moves: Right, Left, Down, Up
    move_directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    # Initialize queue for BFS and helper data
    cells_to_visit = deque([start_cell])    # Queue of cells to explore
    visited_cells = set([start_cell])       # Keep track of visited cells
    came_from = {start_cell: None}          # Store parent cell to rebuild path

    # Start exploring
    while cells_to_visit:
        current_row, current_col = cells_to_visit.popleft()

        #  Check if goal is reached
        if (current_row, current_col) == goal_cell:
            path = []
            cell = goal_cell
            while cell:
                path.append(cell)
                cell = came_from[cell]
```

```
            path.reverse()  # from start to goal
            return path

        # 🔁 Explore all 4 possible moves
        for row_change, col_change in move_directions:
            next_row = current_row + row_change
            next_col = current_col + col_change

            # Check if the next cell is valid (inside maze & not a wall)
            if 0 <= next_row < total_rows and 0 <= next_col < total_cols:
                if maze[next_row][next_col] == 0 and (next_row, next_col) not in visited_cells:
                    visited_cells.add((next_row, next_col))
                    came_from[(next_row, next_col)] = (current_row, current_col)
                    cells_to_visit.append((next_row, next_col))

    # If no path found
    return None


# 🧱 Maze layout: 0 = free space, 1 = wall
maze_grid = [
    [0, 0, 1, 0],
    [1, 0, 1, 0],
    [0, 0, 0, 0],
    [0, 1, 1, 0]
]

# 🏁 Start and goal cells
start_position = (0, 0)
goal_position = (3, 3)

# 🔍 Find the shortest path
shortest_path = bfs_shortest_path(maze_grid, start_position, goal_position)

# 🎯 Display result
print("Shortest Path from Start to Goal:")
print(shortest_path)
```

3. Implement a Depth-First Search (DFS) algorithm to traverse a tree or graph representing a game
map. The goal is to explore all possible paths from the starting node to the target location,
marking visited nodes to avoid cycles, and visualize the order of traversal.

```
# 🎯 Depth-First Search (DFS) Traversal
# Author: Shravani Farkade

# Step 1️⃣ Create a simple game map (graph)
game_map = {
    'Start': ['Forest', 'River'],
    'Forest': ['Cave', 'Mountain'],
    'River': ['Village'],
    'Cave': [],
    'Mountain': [],
    'Village': ['Castle'],
    'Castle': []
}

# Step 2️⃣ DFS Function (Recursive)
def dfs_traverse(current_location, visited_locations):
    print(f"Visiting: {current_location}")   # Show current location
    visited_locations.add(current_location)  # Mark as visited

    # Explore connected locations
    for next_place in game_map[current_location]:
        if next_place not in visited_locations:
            dfs_traverse(next_place, visited_locations)

# Step 3️⃣ Call DFS starting from 'Start'
visited = set()
print("🗺️ Game Map DFS Traversal Order:\n")
dfs_traverse('Start', visited)
print("\n✅ All reachable locations have been explored.")
```

4.The A* algorithm for a maze-based game environment.

```
import heapq   # Used for priority queue (select smallest f = g + h)

def a_star_shortest_path(maze, start_cell, goal_cell):
    # Step 1: Get maze dimensions
    total_rows = len(maze)
    total_cols = len(maze[0])

    # Step 2: Define 4 possible moves (Right, Left, Down, Up)
    move_directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    # Step 3: Initialize helper data
    open_cells = [(0, start_cell)]   # (f_score, cell)
    g_cost = {start_cell: 0}         # g(n): actual cost from start
    came_from = {start_cell: None}   # to reconstruct path
    visited_cells = set()            # optional (for clarity)

    # Step 4: Define the heuristic (Manhattan distance)
    def heuristic(current, goal):
        """Estimated distance (Manhattan)."""
        return abs(current[0] - goal[0]) + abs(current[1] - goal[1])

    # Step 5: Start exploring
    while open_cells:
        f_score, current_cell = heapq.heappop(open_cells)

        # Goal check ✅
        if current_cell == goal_cell:
            path = []
            while current_cell:
                path.append(current_cell)
                current_cell = came_from[current_cell]
            path.reverse()
            return path, g_cost[goal_cell]

        # Skip already processed cells
        if current_cell in visited_cells:
            continue
        visited_cells.add(current_cell)

        # Step 6: Explore all 4 neighbors
        for row_change, col_change in move_directions:
            next_row = current_cell[0] + row_change
            next_col = current_cell[1] + col_change
```

```
            # Step 7️⃣ Check if neighbor is valid (inside grid & not wall)
            if 0 <= next_row < total_rows and 0 <= next_col < total_cols:
                if maze[next_row][next_col] == 0:
                    neighbor = (next_row, next_col)
                    new_cost = g_cost[current_cell] + 1   # g' = g + step
                    total_estimated = new_cost + heuristic(neighbor, goal_cell)  # f = g + h

                    # Step 8️⃣ If new better path found
                    if neighbor not in g_cost or new_cost < g_cost[neighbor]:
                        g_cost[neighbor] = new_cost
                        came_from[neighbor] = current_cell
                        heapq.heappush(open_cells, (total_estimated, neighbor))

    # Step 9️⃣ If no path found ❌
    return None, float('inf')


# 🧱 Step 🔟: Maze layout (0 = free space, 1 = wall)
maze_grid = [
    [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]
]

# 🏁 Step 1️⃣1️⃣ Start and Goal positions
start_position = (0, 0)
goal_position = (4, 4)

# 🚀 Step 1️⃣2️⃣ Run the A* Algorithm
path, total_cost = a_star_shortest_path(maze_grid, start_position, goal_position)

# 🎯 Step 1️⃣3️⃣ Display Results
if path:
    print("✅ Shortest Path Found (A*):")
    print(path)
    print(f"➡️ Total Steps: {len(path) - 1}")
else:
    print("❌ No Path Found.")
```

5. Implementation of 8 puzzles game.

```
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. Implementation of Tic-Tac-Toe game.

```
# 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. Implementation of Tower of Hanoi game.

```
# Tower of Hanoi Program

def tower_of_hanoi(n, source, auxiliary, destination):
    if n == 1:
        print(f"Move disk 1 from {source} to {destination}")
        return

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

    # Move nth disk
    print(f"Move disk {n} from {source} to {destination}")
```

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


print(" Tower of Hanoi Solution ")
num_disks = int(input("Enter the number of disks: "))
print(f"\nSteps to move {num_disks} disks from A → C:\n")

tower_of_hanoi(num_disks, 'A', 'B', 'C')

print(f"\nTotal moves required: {2 ** num_disks - 1}")
print("\nProgram completed successfully!")
```

8. Implementation of Water jug problems.

```
from collections import deque

# Step 1: Define jug capacities and goal
jugA_capacity = 4
jugB_capacity = 3
target = 2  # Goal: measure exactly 2 liters

# Step 2: BFS function to find solution
def water_jug_bfs():
    visited_states = set()  # store visited states to avoid repetition
    queue = deque([((0, 0), [])])  # ((jugA, jugB), path_taken)

    while queue:
        (jugA, jugB), path = queue.popleft()  # current state + path history

        # ✅ Goal check
        if jugA == target or jugB == target:
            path.append((jugA, jugB))
            print("\n✅ Solution found!")
            print("Path to reach goal:")
            for step in path:
                print(step)
```

```
            print(f"\nTotal Steps: {len(path)-1}")
            return

        # Skip already visited states
        if (jugA, jugB) in visited_states:
            continue

        visited_states.add((jugA, jugB))
        path.append((jugA, jugB))

        # Step 3: Generate all possible next moves
        next_states = [
            (jugA_capacity, jugB),  # Fill Jug A
            (jugA, jugB_capacity),  # Fill Jug B
            (0, jugB),              # Empty Jug A
            (jugA, 0),              # Empty Jug B
            # Pour A → B
            (jugA - min(jugA, jugB_capacity - jugB), jugB + min(jugA, jugB_capacity - jugB)),
            # Pour B → A
            (jugA + min(jugB, jugA_capacity - jugA), jugB - min(jugB, jugA_capacity - jugA))
        ]

        # Step 4: Add next valid states to queue
        for next_state in next_states:
            if next_state not in visited_states:
                queue.append((next_state, path.copy()))

    print("❌ No solution found.")

# Step 5: Run the BFS function
water_jug_bfs()
```

9. Uber Ride Price Prediction using PCA and EDA: Dataset can be change(iris dataset)

- **Perform Exploratory Data Analysis (EDA) on Uber ride data**
- **Use Principal Component Analysis (PCA) to reduce dimensionality**
- **Compare the performance of models with and without PCA**

10. Uber Ride Price Prediction using PCA and EDA: Dataset can be change(iris dataset)

- **Perform Exploratory Data Analysis (EDA) on Uber ride data**
- **Use Principal Component Analysis (PCA) to reduce dimensionality**

● **Evaluate models using metrics like R2, RMSE, MAE**

```
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 mean_squared_error, mean_absolute_error, r2_score

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

# ------------------------------------------------------------
# Step 2: Data Cleaning
# ------------------------------------------------------------
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 and extract features
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())
```

```
# ------------------------------------------------------------
# Step 3: EDA
# ------------------------------------------------------------
plt.figure(figsize=(10, 6))
sns.heatmap(df.corr(numeric_only=True), 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']

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

# ------------------------------------------------------------
# Step 5: Linear Regression without PCA
# ------------------------------------------------------------
lin_no_pca = LinearRegression()
lin_no_pca.fit(X_train, y_train)
y_pred_no_pca = lin_no_pca.predict(X_test)

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 + Linear Regression
# ------------------------------------------------------
# Scale data
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s = scaler.transform(X_test)

# Apply PCA
pca = PCA(n_components=3, random_state=42)
X_train_pca = pca.fit_transform(X_train_s)
X_test_pca = pca.transform(X_test_s)

# Train Linear Regression on PCA data
lin_pca = LinearRegression()
lin_pca.fit(X_train_pca, y_train)
y_pred_pca = lin_pca.predict(X_test_pca)

# Evaluation
r2_pca = r2_score(y_test, y_pred_pca)
rmse_pca = np.sqrt(mean_squared_error(y_test, y_pred_pca))
mae_pca = mean_absolute_error(y_test, 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 7: PCA Visualization
# ------------------------------------------------------
print("\n🧩 Explained Variance Ratio by PCA:")
print(pca.explained_variance_ratio_)

plt.figure(figsize=(7, 5))
sns.scatterplot(x=X_train_pca[:, 0], y=X_train_pca[:, 1],
                hue=y_train, 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 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("→ Performed full EDA, scaling, PCA, and regression using sklearn.")
print("→ PCA reduced data to 3 key components (dimensionality reduction).")
print("→ Compared Linear Regression performance with and without PCA.")
print("→ R², RMSE, and MAE show how PCA affects prediction accuracy.")
print("→ PCA simplifies data and removes redundancy, but accuracy may vary by dataset.")
```

11. Implement a Linear Regression model to predict house prices from area, bedrooms, and location features. Apply K-Fold Cross-Validation to validate the model.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import KFold, cross_validate
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LinearRegression
from sklearn.metrics import make_scorer, r2_score, mean_absolute_error, mean_squared_error

# Load
df = pd.read_csv("/mnt/data/Your_Dataset.csv")
# features & target
numeric_features = ['Area', 'Bedrooms']
categorical_features = ['Location']
y = df['Price'].values
X = df[numeric_features + categorical_features]

# Preprocess: One-Hot encode Location
preprocess = ColumnTransformer(
    transformers=[
        ('cat', OneHotEncoder(handle_unknown='ignore', drop='first'), categorical_features),
        ('num', 'passthrough', numeric_features)
    ]
)

# Pipeline: preprocessing + Linear Regression
model = Pipeline(steps=[
    ('prep', preprocess),
    ('lin', LinearRegression())
])

# K-Fold CV with proper metrics
k = 5
cv = KFold(n_splits=k, shuffle=True, random_state=42)

scoring = {
```

```
    'r2': make_scorer(r2_score),
    'mae': make_scorer(mean_absolute_error, greater_is_better=False),   # negative
    'rmse': make_scorer(lambda yt, yp: np.sqrt(mean_squared_error(yt, yp)),
greater_is_better=False)
}

cv_results = cross_validate(model, X, y, cv=cv, scoring=scoring, return_train_score=False)

# Convert negatives back to positive for reporting
r2_scores = cv_results['test_r2']
mae_scores = -cv_results['test_mae']
rmse_scores = -cv_results['test_rmse']

print("\n Linear Regression with One-Hot + 5-Fold CV (sklearn)")
print("-------------------------------------------------------")
print(f"R² per fold  : {np.round(r2_scores, 4)}")
print(f"Avg R²       : {np.mean(r2_scores):.4f}")
print(f"Avg RMSE     : {np.mean(rmse_scores):,.2f}")
print(f"Avg MAE      : {np.mean(mae_scores):,.2f}")

# Train final model on full data for deployment/prediction
model.fit(X, y)

# Safe example prediction (category may be unseen)
example_df = pd.DataFrame([{'Area': 2500, 'Bedrooms': 3, 'Location': 'Urban'}])
pred_price = model.predict(example_df)[0]
print(f"\n Predicted price for 2500 sqft, 3-bedroom Urban house: ₹{pred_price:,.2f}")
```

12. Build a Linear Regression model from scratch to predict students' final exam scores based on their study hours. Implement all computations manually (without using built-in regression

libraries) — including parameter estimation, prediction, and model evaluation using Mean Squared Error (MSE) and R2 Score.

```
# ------------------------------------------------------
# Program 12
# Linear Regression from Scratch
# Predict Students' Final Exam Scores Based on Study Hours
# ------------------------------------------------------

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

# ------------------------------------------------------
# Step 1: Load Dataset
# ------------------------------------------------------
df = pd.read_csv("student_exam_scores_12_13.csv")  # ✅ Correct: CSV file
print("✅ Dataset Loaded Successfully!\n")
print(df.head())

# ------------------------------------------------------
# Step 2: Select Required Columns
# ------------------------------------------------------
X = df['hours_studied'].values
y = df['exam_score'].values
n = len(X)

# ------------------------------------------------------
# Step 3: Compute Parameters Manually
# ------------------------------------------------------
mean_x = np.mean(X)
mean_y = np.mean(y)

# slope (m) and intercept (b)
numerator = np.sum((X - mean_x) * (y - mean_y))
denominator = np.sum((X - mean_x) ** 2)
m = numerator / denominator
b = mean_y - m * mean_x

print(f"\n📈 Equation of Line: y = {m:.4f}x + {b:.4f}")

# ------------------------------------------------------
```

```
# Step 4: Make Predictions
# ------------------------------------------------------------
y_pred = m * X + b

# ------------------------------------------------------------
# Step 5: Evaluate Model
# ------------------------------------------------------------
mse = np.mean((y - y_pred) ** 2)

ss_total = np.sum((y - mean_y) ** 2)
ss_res = np.sum((y - y_pred) ** 2)
r2 = 1 - (ss_res / ss_total)

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

# ------------------------------------------------------------
# Step 6: Predict Example Value
# ------------------------------------------------------------
study_hours = 8
predicted_score = m * study_hours + b
print(f"\n🎯 Predicted Exam Score for {study_hours} Study Hours: {predicted_score:.2f}")

# ------------------------------------------------------------
# Step 7: Visualization
# ------------------------------------------------------------
plt.figure(figsize=(8,5))
plt.scatter(X, y, color='blue', label='Actual Data')
plt.plot(X, y_pred, color='red', label='Regression Line')
plt.title("Study Hours vs Exam Score (Linear Regression from Scratch)")
plt.xlabel("Study Hours")
plt.ylabel("Exam Score")
plt.legend()
plt.grid(True)
plt.show()

# ------------------------------------------------------------
# Step 8: Interpretation
# ------------------------------------------------------------
print("\n🧠 Interpretation:")
print("→ Model built manually without sklearn using slope & intercept formulas.")
print("→ MSE shows average squared prediction error.")
print("→ R² Score indicates model accuracy (closer to 1 = better).")
```

```
print("→ The model successfully predicts students' exam performance based on study hours.")
```

13. Build a Linear Regression model to predict students' exam scores using study hours, attendance, and internal marks. Validate model accuracy using K-Fold Cross-Validation.

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

# ------------------------------------------------------------
# Step 1: Load Dataset
# ------------------------------------------------------------
df = pd.read_csv("student_exam_scores_12_13.csv")
print("✅ Dataset Loaded Successfully!\n")
print(df.head())
print("\n📋 Columns in dataset:", list(df.columns))

# ------------------------------------------------------------
# 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)
# Including 'sleep_hours' as an extra feature for better performance
X = df[['hours_studied', 'sleep_hours', 'attendance_percent', 'Internal_marks']].values
y = df['exam_score'].values

# ------------------------------------------------------------
# Step 3: Initialize Model and K-Fold
# ------------------------------------------------------------
model = LinearRegression()
```

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

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

# ---------------------------------------------------------
# Step 4: Apply K-Fold Cross Validation
# ---------------------------------------------------------
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 the model
    model.fit(X_train, y_train)

    # Predict on test fold
    y_pred = model.predict(X_test)

    # Evaluate performance
    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 - sklearn)")
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}")

# ---------------------------------------------------------
# Step 6: Train Final Model on Full Dataset
# ---------------------------------------------------------
model.fit(X, y)
print("\n✅ Final Model Coefficients:")
print(f"Intercept (θ₀): {model.intercept_:.2f}")
print(f"Coefficients (θ₁, θ₂, θ₃, θ₄): {model.coef_}")

# ---------------------------------------------------------
```

```
# Step 7: Predict Example Student's Exam Score
# ------------------------------------------------------------
example = np.array([[8, 7, 85, 75]])  # 8h study, 7h sleep, 85% attendance, 75 internal
predicted_score = model.predict(example)[0]
print(f"\n📈 Predicted Exam Score for student (8h, 7h sleep, 85%, 75 marks):
{predicted_score:.2f}")

# ------------------------------------------------------------
# Step 8: Visualization - R² Scores per Fold
# ------------------------------------------------------------
plt.figure(figsize=(7, 4))
sns.barplot(x=np.arange(1, k + 1), y=r2_scores, color="skyblue")
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("→ Linear Regression (sklearn) used to predict students' exam scores based on 4
features:")
print("   study hours, sleep hours, attendance, and internal marks.")
print("→ 5-Fold Cross Validation ensures consistent model evaluation.")
print("→ R² Score shows how well input features explain exam scores.")
print("→ RMSE & MAE represent average prediction error (in marks).")
print("→ A higher R² and lower RMSE/MAE indicate good model performance.")
```

14. Develop a Linear Regression model to estimate IT professionals' salaries based on experience, education, and skills. Evaluate performance using 5-Fold Cross-Validation.

```
# ------------------------------------------------------------
# Program 14
```

```
# Linear Regression using sklearn + 5-Fold Cross Validation
# Predict IT Professionals' Salaries based on Experience, Education, and Skills
# ------------------------------------------------------------

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

# ------------------------------------------------------------
# Step 1: Load Dataset
# ------------------------------------------------------------
df = pd.read_excel("Salary Data 14.xlsx")  # Ensure this file is in your working directory
print("✅ Dataset Loaded Successfully!\n")
print(df.head())

# ------------------------------------------------------------
# Step 2: Handle Missing Values
# ------------------------------------------------------------
print("\nMissing values before cleaning:\n", df.isnull().sum())

# Fill numeric columns (Age, Experience, Salary) with mean
numeric_cols = ['Age', 'Years of Experience', 'Salary']
for col in numeric_cols:
    if col in df.columns:
        df[col] = df[col].fillna(df[col].mean())

# Fill categorical columns (Gender, Education Level, Job Title) with mode
categorical_cols = ['Gender', 'Education Level', 'Job Title']
for col in categorical_cols:
    if col in df.columns:
        df[col] = df[col].fillna(df[col].mode()[0])

print("\n✅ Missing values handled successfully!\n")
print(df.isnull().sum())

# ------------------------------------------------------------
# Step 3: Encode Categorical Features
# ------------------------------------------------------------
le_gender = LabelEncoder()
```

```
le_edu = LabelEncoder()
le_job = LabelEncoder()

df['Gender_Encoded'] = le_gender.fit_transform(df['Gender'])
df['Education_Encoded'] = le_edu.fit_transform(df['Education Level'])
df['Job_Encoded'] = le_job.fit_transform(df['Job Title'])

# ------------------------------------------------------------
# Step 4: Prepare Feature Matrix (X) and Target (y)
# ------------------------------------------------------------
X = df[['Age', 'Years of Experience', 'Gender_Encoded',
        'Education_Encoded', 'Job_Encoded']].values
y = df['Salary'].values

# ------------------------------------------------------------
# Step 5: Initialize Model and K-Fold
# ------------------------------------------------------------
model = LinearRegression()
k = 5
kf = KFold(n_splits=k, shuffle=True, random_state=42)

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

# ------------------------------------------------------------
# Step 6: Apply 5-Fold Cross Validation
# ------------------------------------------------------------
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
    model.fit(X_train, y_train)

    # Predict
    y_pred = model.predict(X_test)

    # 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 7: Print Model Evaluation
# ------------------------------------------------------------
```

```
print("\n📊 Linear Regression Model Evaluation (5-Fold Cross Validation - sklearn)")
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}")

# ------------------------------------------------------
# Step 8: Train Final Model on Full Dataset
# ------------------------------------------------------
model.fit(X, y)
print("\n✅ Final Model Trained Successfully!")
print(f"Intercept (θ₀): {model.intercept_:.2f}")
print(f"Coefficients (θ₁...θ₅): {model.coef_}")

# ------------------------------------------------------
# Step 9: Example Prediction
# ------------------------------------------------------
example = np.array([[30, 5,
                     le_gender.transform(['Male'])[0],
                     le_edu.transform(["Bachelor's"])[0],
                     le_job.transform(['Software Engineer'])[0]]])

predicted_salary = model.predict(example)[0]
print(f"\n💰 Predicted Salary for 30-year-old Male Software Engineer (5 yrs exp, Bachelor's):
₹{predicted_salary:,.2f}")

# ------------------------------------------------------
# Step 10: Data Visualization
# ------------------------------------------------------
# (1) Experience vs Salary by Education
plt.figure(figsize=(8, 5))
sns.scatterplot(x=df['Years of Experience'], y=df['Salary'],
                hue=df['Education Level'], palette='coolwarm')
plt.title("Years of Experience vs Salary by Education Level")
plt.xlabel("Years of Experience")
plt.ylabel("Salary (₹)")
plt.show()

# (2) Average Salary by Education Level
plt.figure(figsize=(8, 5))
sns.barplot(x='Education Level', y='Salary', data=df, color='skyblue')
plt.title("Average Salary by Education Level")
plt.ylabel("Average Salary (₹)")
```

```
plt.show()

# (3) Correlation Heatmap
plt.figure(figsize=(7, 4))
sns.heatmap(df[['Age', 'Years of Experience', 'Salary']].corr(),
            annot=True, cmap='magma', fmt=".2f")
plt.title("Feature Correlation Heatmap")
plt.show()

# ------------------------------------------------------
# Step 11: Interpretation
# ------------------------------------------------------
print("\n🧠 Interpretation:")
print("→ Linear Regression (sklearn) used to estimate IT professionals' salaries.")
print("→ Features: Age, Experience, Gender, Education, Job Title.")
print("→ 5-Fold Cross Validation ensures stable and reliable model evaluation.")
print("→ R² shows how much of salary variation is explained by input features.")
print("→ RMSE & MAE represent average prediction errors (in ₹).")
print("→ Higher R² and lower RMSE/MAE indicate better model fit.")
print("→ Model can help HR teams estimate salaries for different profiles.")
```

15. Create a Linear Regression model to forecast monthly sales using ad spend, discounts, and customer footfall. Use 5-Fold Cross-Validation to assess prediction accuracy.

```
# ------------------------------------------------------
# Program 15
# Linear Regression using sklearn + 5-Fold Cross Validation
# Forecast Monthly Revenue using Ad Spend, Discount Applied, and Units Sold
# ------------------------------------------------------

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

```
# ------------------------------------------------------------
# Step 1: Load Dataset
# ------------------------------------------------------------
df = pd.read_csv("15 ad spends.csv")  # Replace with your file path
print("✅ Dataset Loaded Successfully!\n")
print(df.head())

# ------------------------------------------------------------
# Step 2: Handle Missing Values
# ------------------------------------------------------------
print("\nMissing values before cleaning:\n", df.isnull().sum())

# Fill numeric columns with mean
numeric_cols = ['Ad_Spend', 'Discount_Applied', 'Units_Sold', 'Revenue']
for col in numeric_cols:
    if col in df.columns:
        df[col] = df[col].fillna(df[col].mean())

print("\n✅ Missing values handled successfully!\n")

# ------------------------------------------------------------
# Step 3: Define Feature Matrix (X) and Target (y)
# ------------------------------------------------------------
X = df[['Ad_Spend', 'Discount_Applied', 'Units_Sold']].values
y = df['Revenue'].values

# ------------------------------------------------------------
# Step 4: Initialize Model and K-Fold
# ------------------------------------------------------------
model = LinearRegression()
k = 5
kf = KFold(n_splits=k, shuffle=True, random_state=42)

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

# ------------------------------------------------------------
# Step 5: Apply 5-Fold Cross Validation
# ------------------------------------------------------------
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
```

```
    model.fit(X_train, y_train)

    # Predict
    y_pred = model.predict(X_test)

    # 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 6: Display Model Evaluation
# ------------------------------------------------------------
print("\n📊 Linear Regression Model Evaluation (5-Fold Cross Validation - sklearn)")
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}")

# ------------------------------------------------------------
# Step 7: Train Final Model on Full Dataset
# ------------------------------------------------------------
model.fit(X, y)
print("\n✅ Final Model Trained Successfully!")
print(f"Intercept (θ₀): {model.intercept_:.2f}")
print(f"Coefficients (θ₁, θ₂, θ₃): {model.coef_}")

# ------------------------------------------------------------
# Step 8: Predict Example Month's Revenue
# ------------------------------------------------------------
# Example: Ad Spend = ₹75,000, Discount = 0.15 (15%), Units Sold = 130
example = np.array([[75000, 0.15, 130]])
predicted_revenue = model.predict(example)[0]
print(f"\n💰 Predicted Monthly Revenue (Ad Spend=₹75,000, Discount=15%, Units Sold=130):
₹{predicted_revenue:,.2f}")

# ------------------------------------------------------------
# Step 9: Visualization
# ------------------------------------------------------------
# (1) Ad Spend vs Revenue
plt.figure(figsize=(7, 4))
sns.scatterplot(x='Ad_Spend', y='Revenue', data=df, color='skyblue')
plt.title("Ad Spend vs Revenue")
```

```
plt.xlabel("Ad Spend (₹)")
plt.ylabel("Revenue (₹)")
plt.show()

# (2) Discount vs Revenue
plt.figure(figsize=(7, 4))
sns.scatterplot(x='Discount_Applied', y='Revenue', data=df, color='orange')
plt.title("Discount vs Revenue")
plt.xlabel("Discount Applied (%)")
plt.ylabel("Revenue (₹)")
plt.show()

# (3) Units Sold vs Revenue
plt.figure(figsize=(7, 4))
sns.scatterplot(x='Units_Sold', y='Revenue', data=df, color='green')
plt.title("Units Sold vs Revenue")
plt.xlabel("Units Sold")
plt.ylabel("Revenue (₹)")
plt.show()

# (4) Correlation Heatmap
plt.figure(figsize=(7, 4))
sns.heatmap(df[['Ad_Spend', 'Discount_Applied', 'Units_Sold', 'Revenue']].corr(),
            annot=True, cmap='magma', fmt=".2f")
plt.title("Feature Correlation Heatmap")
plt.show()

# ------------------------------------------------------------
# Step 10: Interpretation
# ------------------------------------------------------------
print("\n🧠 Interpretation:")
print("→ Linear Regression (sklearn) used to forecast monthly revenue.")
print("→ Features: Ad Spend, Discount Applied, and Units Sold.")
print("→ 5-Fold Cross Validation ensures reliable accuracy measurement.")
print("→ R² shows how much of revenue variation is explained by the inputs.")
print("→ RMSE & MAE indicate average prediction error (in ₹).")
print("→ Ad Spend and Units Sold positively affect revenue; excessive discounts reduce it.")
print("→ Model helps businesses optimize marketing and pricing strategies.")
```

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.

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.

```
# ------------------------------------------------------
# Naïve Bayes - Email Spam Detection (using sklearn)
# Program 16
# ------------------------------------------------------

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

# ------------------------------------------------------
# Step 1: Load Dataset
# ------------------------------------------------------
data = pd.read_csv("AIML/emails_16_17_18_19.csv")
print("✅ Dataset Loaded Successfully!\n")
print(data.head())

# ------------------------------------------------------
# Step 2: Prepare Data
# ------------------------------------------------------
# Assume the dataset has features as 0/1 word counts and "Prediction" as target
X = data.drop(columns=["Email No.", "Prediction"]).values
y = data["Prediction"].values   # 0 = Not Spam, 1 = Spam

# Class balance visualization
plt.figure(figsize=(6, 4))
sns.countplot(x="Prediction", data=data, palette='coolwarm')
plt.title("Class Distribution (Spam vs Not Spam)")
```

```
plt.xlabel("Class")
plt.ylabel("Count")
plt.show()

# ---------------------------------------------------------
# Step 3: 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 4: Train Naïve Bayes Model (sklearn)
# ---------------------------------------------------------
model = MultinomialNB()       # suitable for word-count or frequency features
model.fit(X_train, y_train)

# ---------------------------------------------------------
# Step 5: Make Predictions
# ---------------------------------------------------------
y_pred = model.predict(X_test)

# ---------------------------------------------------------
# Step 6: Evaluate Model
# ---------------------------------------------------------
acc = accuracy_score(y_test, y_pred)
cm = confusion_matrix(y_test, y_pred)
print(f"\n📈 Accuracy: {acc:.4f}\n")

# Confusion Matrix Heatmap
plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues",
            xticklabels=["Not Spam", "Spam"],
            yticklabels=["Not Spam", "Spam"])
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.title("Confusion Matrix - Naïve Bayes")
plt.show()

# Detailed classification metrics
print("\n📊 Classification Report:")
print(classification_report(y_test, y_pred, target_names=["Not Spam", "Spam"]))
```

18. Implementation an Email Spam Detection model using a Support Vector Machine (SVM) for binary classification, where emails are categorized as Normal (Not Spam) or Abnormal (Spam). Apply oversampling or undersampling techniques to handle class imbalance and analyze model performance using appropriate evaluation metrics.

```
# ------------------------------------------------------------
# Program 18
# Email Spam Detection with SVM + SMOTE (sklearn only)
# ------------------------------------------------------------

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 (
    accuracy_score, confusion_matrix, ConfusionMatrixDisplay,
    classification_report, roc_auc_score, roc_curve, precision_recall_fscore_support
)
from imblearn.over_sampling import SMOTE
import matplotlib.pyplot as plt
import seaborn as sns

# ------------------------------------------------------------
# Step 1: Load preprocessed dataset (numeric features)
# ------------------------------------------------------------
data = pd.read_csv("emails_16_17_18_19.csv")       # update path if needed
X = data.drop(columns=["Email No.", "Prediction"])
y = data["Prediction"]                        # 0 = Normal, 1 = Spam

# Class balance (before)
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 (stratified to keep class ratio)
# ---------------------------------------------------------
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42
)

# ---------------------------------------------------------
# Step 3: Oversample only the TRAIN set (SMOTE)
# ---------------------------------------------------------
smote = SMOTE(random_state=42)
X_train_res, y_train_res = smote.fit_resample(X_train, y_train)

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: Scale features (sparse-friendly)
# ---------------------------------------------------------
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
#   - linear kernel often works well for text-like data
#   - you can try kernel='rbf' and tune C, gamma if needed
# ---------------------------------------------------------
svm_model = SVC(kernel='linear', probability=False, random_state=42)
svm_model.fit(X_train_scaled, y_train_res)

# ---------------------------------------------------------
# Step 6: Predict
# ---------------------------------------------------------
y_pred = svm_model.predict(X_test_scaled)
y_scores = svm_model.decision_function(X_test_scaled)  # for ROC-AUC

# ---------------------------------------------------------
# Step 7: Evaluate
```

```
# ----------------------------------------------------------
acc = accuracy_score(y_test, y_pred)
prec, rec, f1, _ = precision_recall_fscore_support(y_test, y_pred, average='binary', pos_label=1)
auc = roc_auc_score(y_test, y_scores)

print(f"\n✅ SVM + SMOTE Results")
print(f"Accuracy : {acc:.4f}")
print(f"Precision: {prec:.4f}")
print(f"Recall   : {rec:.4f}")
print(f"F1-score : {f1:.4f}")
print(f"ROC-AUC  : {auc:.4f}\n")

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

# Detailed report
print("Classification Report:\n",
      classification_report(y_test, y_pred, target_names=["Normal", "Spam"]))

# ROC Curve
fpr, tpr, _ = roc_curve(y_test, y_scores)
plt.figure(figsize=(6,4))
plt.plot(fpr, tpr, label=f"ROC curve (AUC = {auc:.3f})")
plt.plot([0,1],[0,1],'--', label="Chance")
plt.xlabel("False Positive Rate"); plt.ylabel("True Positive Rate")
plt.title("ROC Curve - SVM (with SMOTE)")
plt.legend(); plt.grid(True); plt.show()
```

19. Implement an Email Spam Detection model from scratch using the Support Vector Machine (SVM) algorithm for binary classification, where emails are labeled as Normal (Not Spam) or Abnormal (Spam). Analyze model performance using appropriate evaluation metrics.

```
# ---------------------------------------------------------
# Program 19
# Email Spam Detection with SVM (sklearn metrics, no "from scratch")
# ---------------------------------------------------------

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 (
    accuracy_score, precision_score, recall_score, f1_score,
    confusion_matrix, ConfusionMatrixDisplay, classification_report,
    roc_curve, roc_auc_score
)
import matplotlib.pyplot as plt
import seaborn as sns

# ---------------------- STEP 1: LOAD DATA ----------------------
data = pd.read_csv("emails_16_17_18_19.csv")  # update path if needed
X = data.drop(columns=["Email No.", "Prediction"])
y = data["Prediction"]  # 0 = Normal, 1 = Spam

# (Optional) Class distribution quick check
plt.figure(figsize=(6, 4))
sns.countplot(x=y, palette="coolwarm")
plt.xticks([0, 1], ["Normal (Not Spam)", "Abnormal (Spam)"])
plt.title("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, stratify=y, random_state=42
)

# ---------------------- STEP 3: FEATURE SCALING ----------------------
# with_mean=False is safe for sparse/text-like matrices
scaler = StandardScaler(with_mean=False)
X_train_s = scaler.fit_transform(X_train)
X_test_s  = scaler.transform(X_test)

# ---------------------- STEP 4: TRAIN SVM MODEL ----------------------
# linear kernel works well for text-like features
svm_model = SVC(kernel='linear', probability=False, random_state=42)
```

```
svm_model.fit(X_train_s, y_train)

# ---------------------- STEP 5: PREDICT ----------------------
y_pred = svm_model.predict(X_test_s)
y_scores = svm_model.decision_function(X_test_s)  # for ROC-AUC

# ---------------------- STEP 6: EVALUATE (sklearn metrics) ----------------------
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)
auc  = roc_auc_score(y_test, y_scores)

print("\n✅ SVM Results (sklearn metrics)")
print(f"Accuracy : {acc:.4f}")
print(f"Precision: {prec:.4f}")
print(f"Recall   : {rec:.4f}")
print(f"F1-score : {f1:.4f}")
print(f"ROC-AUC  : {auc:.4f}\n")

print("Classification Report:\n",
      classification_report(y_test, y_pred, target_names=["Normal", "Spam"]))

# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
ConfusionMatrixDisplay(cm, display_labels=["Normal", "Spam"]).plot(cmap="Blues")
plt.title("Confusion Matrix - SVM"); plt.show()

# ROC Curve
fpr, tpr, _ = roc_curve(y_test, y_scores)
plt.figure(figsize=(6,4))
plt.plot(fpr, tpr, label=f"AUC = {auc:.3f}")
plt.plot([0,1],[0,1],'--', label="Chance")
plt.xlabel("False Positive Rate"); plt.ylabel("True Positive Rate")
plt.title("ROC Curve - SVM")
plt.legend(); plt.grid(True); plt.show()
```

20. Implement an SVM model from scratch with a Polynomial Kernel to predict student performance

(Pass/Fail) using the Student Performance Dataset based on features like study time, absences,
and internal scores. Assess the model performance using precision, recall, and F1-score.

```
# ------------------------------------------------------------
# Program 20
# SVM with Polynomial Kernel - From Scratch (Concept Implementation)
# Predict Student Performance (Pass/Fail)
# ------------------------------------------------------------

import numpy as np
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

# ---------------------- Step 1: Load Data ----------------------
df = pd.read_csv("student_performance_dataset_20.csv")  # <-- update your path
print("✅ Dataset Loaded Successfully!\n")
print(df.head())

# Encode target labels (Pass=1, Fail=0)
df["Pass_Fail"] = df["Pass_Fail"].map({"Pass": 1, "Fail": 0})

# Select features
X = df[["Study_Hours_per_Week", "Attendance_Rate", "Internal_Scores"]].values
y = df["Pass_Fail"].values

# ---------------------- Step 2: Split & Scale ----------------------
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, stratify=y, random_state=42
)

scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s = scaler.transform(X_test)

# ---------------------- Step 3: Implement Polynomial Kernel (From Scratch) ----------------------
def polynomial_kernel(X1, X2, degree=3, coef0=1):
    """
    Compute polynomial kernel manually: (X1 · X2^T + coef0)^degree
```

```
    """
    return (np.dot(X1, X2.T) + coef0) ** degree

# Check kernel output
print("\nSample Polynomial Kernel Matrix (first 3 samples):\n")
print(polynomial_kernel(X_train_s[:3], X_train_s[:3]))

# ---------------------- Step 4: Train SVM (using custom kernel) ----------------------
svm_poly = SVC(kernel='poly', degree=3, coef0=1, C=1.0, random_state=42)
svm_poly.fit(X_train_s, y_train)
y_pred = svm_poly.predict(X_test_s)

# ---------------------- Step 5: Manual Metric Calculation ----------------------
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
accuracy = (TP + TN) / (TP + TN + FP + FN)

# ---------------------- Step 6: Display Results ----------------------
print("\n📊 SVM with Polynomial Kernel (From Scratch Metrics)")
print(f"True Positives : {TP}")
print(f"True Negatives : {TN}")
print(f"False Positives: {FP}")
print(f"False Negatives: {FN}\n")
print(f"Accuracy : {accuracy * 100:.2f}%")
print(f"Precision: {precision:.4f}")
print(f"Recall   : {recall:.4f}")
print(f"F1-Score : {f1:.4f}")

# ---------------------- Step 7: Confusion Matrix Visualization ----------------------
import seaborn as sns
cm = np.array([[TN, FP], [FN, TP]])
plt.figure(figsize=(5, 4))
sns.heatmap(cm, annot=True, fmt="d", cmap="magma",
            xticklabels=["Predicted Fail", "Predicted Pass"],
            yticklabels=["Actual Fail", "Actual Pass"])
plt.title("Confusion Matrix - SVM (Polynomial Kernel)")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
```

```
plt.show()

# ---------------------- Step 8: Interpretation ----------------------
print("\n🧠 Interpretation:")
print("→ Polynomial Kernel computed manually as (X·Xᵀ + c)ᵈ.")
print("→ Model trained using SVM with degree=3 polynomial kernel.")
print("→ Metrics (Precision, Recall, F1) computed manually to verify performance.")
print("→ The model successfully distinguishes passing and failing students.")
```

21. Develop an SVM classifier from scratch using a Polynomial Kernel on the Breast Cancer Wisconsin Dataset to distinguish between benign and malignant tumors. Evaluate the classifier using a confusion matrix and ROC curve to analyze diagnostic accuracy.

```
# ------------------------------------------------------------
# Program 21
# Breast Cancer Classification using Polynomial Kernel SVM
# Evaluated with Confusion Matrix and ROC Curve
# ------------------------------------------------------------

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 (
    accuracy_score, precision_score, recall_score, f1_score,
    confusion_matrix, ConfusionMatrixDisplay, roc_curve, roc_auc_score, classification_report
)
import matplotlib.pyplot as plt
import seaborn as sns

# ---------------------- Step 1: Load and Preprocess Data ----------------------
df = pd.read_csv("Breast Cancer Wisconsin (Diagnostic)_21 (1).csv")  # Update path if needed
print("✅ Dataset Loaded Successfully!\n")
```

```
print(df.head())

# Drop unnecessary column(s)
if 'id' in df.columns:
    df = df.drop(columns=['id'])

# Encode target: M → 1 (Malignant), B → 0 (Benign)
df['diagnosis'] = np.where(df['diagnosis'] == 'M', 1, 0)

# Split features and labels
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, stratify=y, random_state=42
)

# ---------- Step 3: Feature Scaling ----------
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s = scaler.transform(X_test)

# ---------- Step 4: Train SVM (Polynomial Kernel) ----------
model = SVC(kernel='poly', degree=2, coef0=1, C=5.0, probability=True, random_state=42)
model.fit(X_train_s, y_train)

# ---------- Step 5: Predictions ----------
y_pred = model.predict(X_test_s)
y_prob = model.predict_proba(X_test_s)[:, 1]  # For ROC curve

# ---------- Step 6: Evaluate Model ----------
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)
auc  = roc_auc_score(y_test, y_prob)

print("\n📊 Model Evaluation (Polynomial SVM - sklearn metrics)")
print(f"Accuracy : {acc*100:.2f}%")
print(f"Precision: {prec:.4f}")
print(f"Recall   : {rec:.4f}")
print(f"F1-Score : {f1:.4f}")
print(f"AUC Score: {auc:.4f}\n")
```

```
print("Classification Report:\n",
      classification_report(y_test, y_pred, target_names=["Benign", "Malignant"]))

# ---------------------- Step 7: Confusion Matrix ----------------------
cm = confusion_matrix(y_test, y_pred)
ConfusionMatrixDisplay(cm, display_labels=["Benign", "Malignant"]).plot(cmap="coolwarm")
plt.title("Confusion Matrix - Polynomial SVM (Breast Cancer)")
plt.show()

# ---------------------- Step 8: ROC Curve ----------------------
fpr, tpr, _ = roc_curve(y_test, y_prob)
plt.figure(figsize=(6, 5))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC Curve (AUC = {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 (Breast Cancer Classification)')
plt.legend(loc='lower right')
plt.grid(True)
plt.show()

# ---------------------- Step 9: Interpretation ----------------------
print("\n🧠 Interpretation:")
print("→ Polynomial SVM (degree=2) used to classify tumors as Benign or Malignant.")
print("→ Features standardized using StandardScaler for better optimization.")
print("→ Model evaluated with Accuracy, Precision, Recall, F1-score, and AUC.")
print("→ ROC Curve shows diagnostic accuracy and model's ability to separate classes.")
```