AIML Problem Statements

1. Implement the Informed Search algorithm for real-life problems.

4. Develop a pathfinding solution using the A* algorithm for a maze-based game environment. The agent must find the most cost-efficient route from the start position to the goal, considering movement costs and a suitable heuristic function (e.g., Manhattan distance) to guide the search efficiently.

```
from queue import PriorityQueue

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

start = (0, 0)

goal = (3, 4)

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

def a_star(start, goal):
    pq = PriorityQueue()
    pq.put((0, start, [start]))   # (priority, current_node, path)
    visited = set()


    while not pq.empty():
        cost, node, path = pq.get()
        if node == goal:
            return path
        x, y = node
        for dx, dy in [(1,0), (-1,0), (0,1), (0,-1)]:
            nx, ny = x + dx, y + dy
```

```
            if 0 <= nx < len(maze) and 0 <= ny < len(maze[0]) and maze[nx][ny] == 0 and (nx,ny) not in visited:
                g = len(path)
                h = heuristic((nx, ny), goal)
                f = g + h
                pq.put((f, (nx, ny), path + [(nx, ny)]))
                visited.add(node)

    return None
path = a_star(start, goal)
if path:
    print("Shortest Path Found:")
    print(path)
    print(f"Total Steps: {len(path)-1}")
else:
    print("No Path Found.")
```

Output:

Shortest Path Found:

[(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (3, 2), (3, 3), (3, 4)]

Total Steps: 7

2. Design an algorithm using Breadth-First Search (BFS) to find the shortest path from a start node to a goal node in a maze represented as a grid graph. The maze contains obstacles (walls) and free cells. Implement BFS to ensure that the first found path is the optimal one in terms of the number of steps

```
from queue import Queue
maze = [
    [0, 1, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [1, 1, 0, 0, 0]
```

```
]
start = (0, 0)
goal = (3, 4)
def bfs(start, goal):
    q = Queue()
    q.put((start, [start]))
    visited = set()

    while not q.empty():
        node, path = q.get()

        if node == goal:
            return path

        x,y = node
        for dx, dy in  [(1,0), (-1,0), (0,1), (0,-1)]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < len(maze) and 0 <= ny < len(maze[0]) and maze[nx][ny] == 0 and
(nx,ny) not in visited:
                q.put(((nx, ny), path + [(nx, ny)]))
                visited.add((nx, ny))

    return None
path = bfs(start, goal)
if path:
    print("Shortest Path Found (BFS):")
    print(path)
    print(f"Total Steps: {len(path) - 1}")
else:
```

```
    print(" No Path Found.")
```

o/p:

Shortest Path Found (BFS):

[(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (3, 2), (3, 3), (3, 4)]

Total Steps: 7

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.

```
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['G'],
    'F': [],
    'G': []
}
def dfs(start, goal, graph):
    stack = [(start, [start])]
    visited = set()

    while stack:
        node, path = stack.pop()
        if node == goal:
            print("Goal Found:", path)
            return
        if node not in visited:
            visited.add(node)
            for neighbor in reversed(graph[node]):
```

```
            stack.append((neighbor, path + [neighbor]))
    print("Goal Not Found ")
dfs('A', 'G', graph)
```

A->B->E->G

5. Implementation of 8 puzzles game.

```
p = ["1","2","3","4","5","6","7","8"," "]
g = ["1","2","3","4","5","6","7","8"," "]

# Display Board
def show():
    print(p[0], p[1], p[2])
    print(p[3], p[4], p[5])
    print(p[6], p[7], p[8])

# Game Loop
while p != g:
    show()
    move = input("Move (W=Up A=Left S=Down D=Right): ").upper()
    i = p.index(" ")     # blank position

    if move == "W" and i > 2:
        p[i], p[i-3] = p[i-3], p[i]
    elif move == "S" and i < 6:
        p[i], p[i+3] = p[i+3], p[i]
    elif move == "A" and i % 3 != 0:
        p[i], p[i-1] = p[i-1], p[i]
    elif move == "D" and i % 3 != 2:
        p[i], p[i+1] = p[i+1], p[i]
```

```
    else:
        print("Invalid Move")

print("Solved!")
```

6. Implementation of Tic-Tac-Toe game.

```
board = [" "] * 9
player = "X"
def show():
    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]}")

def check():
    wins = [(0,1,2),(3,4,5),(6,7,8),
            (0,3,6),(1,4,7),(2,5,8),
            (0,4,8),(2,4,6)]
    return any(board[a]==board[b]==board[c]!=" " for a,b,c in wins)

for _ in range(9):
    show()
    pos = int(input("Enter 0-8: "))
    if board[pos] != " ":
        print("Invalid"); break
    board[pos] = player
    if check():
        show(); print(player, "wins!"); break
    player = "O" if player=="X" else "X"
```

```
else:
    show(); print("Draw!")
```

7. Implementation of Tower of Hanoi game.

```
def tower_of_hanoi(n, source, auxiliary, destination):
  if n == 1:
    print(f" move disk 1 from {source}->{destination}")
    return 1
  moves = 0
  moves += tower_of_hanoi(n-1,source,destination,auxiliary)
  print(f" moves disk {n} from {source}->{destination}")
  moves+=1
  moves += tower_of_hanoi(n-1,auxiliary,source,destination)
  return moves

n = int(input("enter number of disks: "))
print("\n Steps to solve tower_of_hanoi")
total_moves = tower_of_hanoi(n,  "A","B","C")
print(f"\n✅ Total moves required: {total_moves}")
```

8. Implementation of water jug problem.

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 R^2, RMSE, MAE

```
import pandas as pd
```

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

# Load Excel file
df=pd.read_csv("uber_9 _ 10.xls - uber_9 _ 10.xls.csv")
# Show basic info
print(df.head())
print(df.info())

# Drop rows with missing values
df = df.dropna()

# Convert pickup_datetime to datetime
df['pickup_datetime'] = pd.to_datetime(df['pickup_datetime'], errors='coerce')

# Extract useful time features
df['hour'] = df['pickup_datetime'].dt.hour
df['day'] = df['pickup_datetime'].dt.day
df['month'] = df['pickup_datetime'].dt.month

# Define features and target
X = df.drop(columns=['fare_amount'])
y = df['fare_amount']
```

```
# ---------------- Visualization Part Added ----------------
# Pairplot
sns.pairplot(pd.concat([X, y.rename('fare_amount')], axis=1))
plt.show()

# Heatmap
plt.figure(figsize=(10,6))
sns.heatmap(X.corr(), annot=True, cmap='coolwarm')
plt.title("Feature Correlation Heatmap")
plt.show()
# ----------------------------Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Standardize
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Linear Regression without PCA
model = LinearRegression()
model.fit(X_train_scaled, y_train)
y_pred = model.predict(X_test_scaled)

print("\nWithout PCA:")
print("R2:", r2_score(y_test, y_pred))
print("RMSE:", np.sqrt(mean_squared_error(y_test, y_pred)))
print("MAE:", mean_absolute_error(y_test, y_pred))
```

```
# PCA
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)

model_pca = LinearRegression()
model_pca.fit(X_train_pca, y_train)
y_pred_pca = model_pca.predict(X_test_pca)

print("\nWith PCA:")
print("R2:", r2_score(y_test, y_pred_pca))
print("RMSE:", np.sqrt(mean_squared_error(y_test, y_pred_pca)))
print("MAE:", mean_absolute_error(y_test, y_pred_pca))
```

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 R^2 Score.

```
import numpy as np
import pandas as pd

df = pd.read_csv("C:/Users/sarth/Downloads/student_exam_scores_12_13.csv")
df = df.dropna(subset=["hours_studied", "exam_score"])

X = df["hours_studied"].values.astype(float)
y = df["exam_score"].values.astype(float)

if X.size < 2:
    raise ValueError("Need at least 2 data points to fit linear regression.")

X_mean = X.mean()
```

```
y_mean = y.mean()

num = np.sum((X - X_mean) * (y - y_mean))
den = np.sum((X - X_mean)**2)
if den == 0:
    raise ValueError("Cannot compute slope — all X values are identical.")
b1 = num / den
b0 = y_mean - b1 * X_mean

y_pred = b0 + b1 * X

mse = np.mean((y - y_pred)**2)
r2 = 1 - (np.sum((y - y_pred)*2) / np.sum((y - y_mean)*2))

print(f"Intercept (b0): {b0:.4f}")
print(f"Slope (b1): {b1:.4f}")
print(f"MSE: {mse:.4f}")
print(f"R2: {r2:.4f}")
print("First 5 actual vs predicted:")
for actual, pred in zip(y[:5], y_pred[:5]):
    print(f"  actual={actual:.2f}  pred={pred:.2f}")
```

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.

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.

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

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.

```python
import pandas as pd
from sklearn.model_selection import KFold, cross_val_score
from sklearn.linear_model import LinearRegression

# Example dataset
data = {
    'area': [1000, 1500, 2000, 2500, 3000],
    'bedrooms': [2, 3, 3, 4, 4],
    'location': [1, 2, 2, 3, 3],  # Encoded location
    'price': [200000, 250000, 300000, 350000, 400000]
}

df = pd.DataFrame(data)

# Features & target
X = df[['area', 'bedrooms', 'location']]
y = df['price']

# Model
model = LinearRegression()

# K-Fold Cross Validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(model, X, y, cv=kf, scoring='r2')

print("R² Scores from K-Fold:", scores)
print("Mean R²:", scores.mean())
```

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.

```
import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.naive_bayes import MultinomialNB

from sklearn.metrics import confusion_matrix, accuracy_score, precision_score, recall_score,
f1_score


df = pd.read_csv("C:/Users/ /Downloads/emails_16_17_18_19 (1).csv")

df = df.select_dtypes(include=["number"])

df = df.dropna()


X = df.drop(columns=["Prediction"])

y = df["Prediction"]


X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.3, random_state=42,
stratify=y)


model = MultinomialNB()

model.fit(X_train, y_train)

y_pred = model.predict(X_test)


cm = confusion_matrix(y_test, y_pred)

accuracy = accuracy_score(y_test, y_pred)

precision = precision_score(y_test, y_pred)

recall = recall_score(y_test, y_pred)

f1 = f1_score(y_test, y_pred)
```

```
print("Confusion Matrix:\n", cm)
print(f"Accuracy: {accuracy:.4f}")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1-Score: {f1:.4f}")
```