TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA Công Nghệ Thông Tin

ĐỀ THI VÀ BÀI LÀM

Tên học phần: **Trí tuệ nhân tạo**
Mã học phần: Hình thức thi: *Tự luận có giám sát*
Đề số: **Đ0001** Thời gian làm bài: 75 phút *(không kể thời gian chép/phát đề)*
Được sử dụng tài liệu khi làm bài.

Họ tên:…Nguyễn Hữu Khoa…….**Lớp**:…22T_DT5……..**MSSV**:…102220237……..

Sinh viên làm bài trực tiếp trên tệp này, lưu tệp với định dạng MSSV_HọTên.pdf và nộp bài thông qua MSTeam:

Câu 1 (*3 điểm*): Trong các lâu đài cổ người ta thường xây dựng các đường hầm bí mật để thoát hiểm trong các trường hợp khẩn cấp. Các đường hầm chỉ có thể vào từ một cửa vào duy nhất tại phòng Trung tâm và thoát ra ở rất nhiều cửa ra. Các cửa ra đều nằm ở rìa lâu đài, do vậy, nếu thoát ra được rìa lâu đài thì coi như đã thoát hiểm. Để ngụy trang, người ta cho đào nhiều nhánh hầm cụt và cửa vào giả. Ngoài ra, để tăng khả năng thoát hiểm, người ta còn xây dựng các đường hầm giao nhau tại một số vị trí. Để nghiệm thu công trình, chủ lâu đài cần kiểm tra xem từ phòng trung tâm có thể thoát hiểm qua hệ thống đường hầm hay không. Hãy sử dụng thuật toán **A*** với hàm f(x) là tổng chi phí được định nghĩa f(x)=g(x)+h(x) (trong đó g(x)-chi phí từ điểm xuất phát đến ô hiện tại, h(x)-hàm ước lượng khoảng cách còn lại đến cửa ra, sử dụng khoảng cách Manhattan) để giúp chủ lâu đài kiểm tra hệ thống trên.

Biết rằng lâu đài là một hình vuông được chia lưới ô vuông gồm n dòng, n cột. Trên đồ hoạ, ô ở dòng i cột j được ghi số 1 nếu có đường hầm, số 0 nếu không có (ô ở góc trên trái có toạ độ (0,0)). 2 ô chỉ có thể thông nhau nếu chúng có chung cạnh.

Dữ liệu nhập vào từ tập tin văn bản "A_in.csv" gồm:

- Dòng đầu chứa 3 số nguyên dương n <20, D và C (trong đó D, C là dòng và cột của phòng trung tâm).

- n dòng tiếp theo, mỗi dòng chứa n số là các số ở các vị trí tương ứng trên họa đồ.

Kết quả tìm được ghi ra tập tin văn bản "**A_out.csv**". Dòng đầu chứa số m là số ô phải đi qua, nếu không thoát được thì m = -1. Trong trường hợp thoát được, m dòng tiếp theo: mỗi dòng chứa 2 số là số hiệu dòng cột của các ô phải đi qua theo đúng trình tự của một cách thoát hiểm.

Ví dụ: A_in.csv	Tệp A_out.csv
4 2 1 **0 1 1 0** **1 0 0 1** **1 1 1 1** **0 1 1 0**	**2** **2 1** **2 0**

a) Xác định hàm h(x)

Trả lời: Minh hoạ giải thích hàm

Hàm h(x) - heuristic:

- Sử dụng khoảng cách Manhattan đến rìa lâu đài gần nhất ($|x1-x2|+|y1-y2|$)
- h(x) = min(x, y, n-1-x, n-1-y) với (x,y) là tọa độ hiện tại
- Đây là ước lượng khoảng cách tối thiểu đến bất kỳ cửa ra nào ở rìa lâu đài

Trả lời: Dán code hàm h(x)

```
def manhattan_distance(current, goal):
    """
    Calculate Manhattan distance to the nearest edge of the castle
    This is our h(x) heuristic function
    """
    x, y = current
    n = goal  # goal is the size of the maze (to reach any edge)
    # Distance to nearest edge (top, bottom, left, or right)
    return min(x, y, n - 1 - x, n - 1 - y)
```

b) Viết chương trình hoàn thiện cho bài toán trên

Trả lời: Dán code vào bên dưới

```
import csv
from heapq import heappush, heappop
import numpy as np

def manhattan_distance(current, goal):
    """
    Calculate Manhattan distance to the nearest edge of the castle
    This is our h(x) heuristic function
    """
    x, y = current
    n = goal  # goal is the size of the maze (to reach any edge)
    # Distance to nearest edge (top, bottom, left, or right)
    return min(x, y, n - 1 - x, n - 1 - y)

def is_edge(pos, n):
    """Check if position is at the edge of the castle"""
    x, y = pos
    return x == 0 or x == n - 1 or y == 0 or y == n - 1

def get_neighbors(pos, maze, n):
    """Get valid neighboring positions"""
    x, y = pos
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # right, down, left, up
    neighbors = []
    for dx, dy in directions:
        new_x, new_y = x + dx, y + dy
        if (0 <= new_x < n and 0 <= new_y < n and
            maze[new_x][new_y] == 1):  # Check if cell has a tunnel
```

```
            neighbors.append((new_x, new_y))
    return neighbors

def astar_escape(maze, start, n):
    """
    A* algorithm implementation for castle escape
    f(x) = g(x) + h(x) where:
    - g(x) is the actual cost from start to current position
    - h(x) is the Manhattan distance to nearest edge
    """
    frontier = []
    heappush(frontier, (0, start))  # (f(x), position)
    came_from = {start: None}
    g_score = {start: 0}

    while frontier:
        current = heappop(frontier)[1]

        # Check if we've reached an edge
        if is_edge(current, n):
            # Reconstruct path
            path = []
            while current is not None:
                path.append(current)
                current = came_from[current]
            return path[::-1]  # Reverse path to get start-to-goal order

        for next_pos in get_neighbors(current, maze, n):
            # g(x) is just the number of steps taken
            tentative_g_score = g_score[current] + 1

            if next_pos not in g_score or tentative_g_score < g_score[next_pos]:
                came_from[next_pos] = current
                g_score[next_pos] = tentative_g_score
                # f(x) = g(x) + h(x)
                f_score = tentative_g_score + manhattan_distance(next_pos, n)
                heappush(frontier, (f_score, next_pos))

    return None  # No escape path found

def read_input(filename):
    """Read maze from CSV file"""
    with open(filename, 'r') as file:
        reader = csv.reader(file)
        # Read first line: n, D, C
        n, start_row, start_col = map(int, next(reader))

        # Read maze
        maze = []
        for _ in range(n):
            row = list(map(int, next(reader)))
            maze.append(row)

    return np.array(maze), n, (start_row, start_col)
```

```python
def write_output(filename, path):
    """Write solution to CSV file"""
    with open(filename, 'w', newline='') as file:
        writer = csv.writer(file)
        if path is None:
            writer.writerow([-1])  # No solution found
        else:
            writer.writerow([len(path)])  # Number of cells in path
            for row, col in path:
                writer.writerow([row, col])  # Write coordinates

def solve_castle_escape(input_file, output_file):
    # Read input
    maze, n, start = read_input(input_file)

    # Check if start position has a tunnel
    if maze[start[0]][start[1]] != 1:
        path = None
    else:
        # Find escape path using A*
        path = astar_escape(maze, start, n)

    # Write output
    write_output(output_file, path)
    return path

# Run the solution
if __name__ == "__main__":
    input_file = "A_in.csv"
    output_file = "A_out.csv"
    path = solve_castle_escape(input_file, output_file)

    if path is None:
        print("Không tìm thấy đường thoát hiểm")
    else:
        print(f"Tìm thấy đường thoát với {len(path)} bước")
        print("Đường đi:", path)
```

 # Trả lời: Giải thích chương trình
Giải thích về thuật toán và cách xác định hàm h(x):

1. Hàm f(x) = g(x) + h(x):

   - g(x): số bước đã đi từ điểm xuất phát đến vị trí hiện tại
   - h(x): khoảng cách Manhattan đến rìa gần nhất
   - Đảm bảo tìm được đường đi ngắn nhất đến cửa ra

2Các điểm chính của giải thuật:

   - Bắt đầu từ phòng trung tâm (D,C)
   - Chỉ di chuyển qua các ô có giá trị 1 (có đường hầm)
   - Di chuyển theo 4 hướng (lên, xuống, trái, phải)
   - Kết thúc khi đến được rìa lâu đài

- Nếu không tìm được đường đi, trả về -1

3. Định dạng input/output:
   a. Input: Ma trận n×n từ file CSV, với tọa độ phòng trung tâm
   b. Output: Số bước đi và tọa độ các ô theo thứ tự đi

Chương trình sẽ tìm đường đi ngắn nhất từ phòng trung tâm ra rìa lâu đài, sử dụng A* với heuristic là khoảng cách Manhattan đến rìa gần nhất.

c) Kết quả thực thi trên tệp "A_out.csv

 # Trả lời: Dán kết quả vào bên dưới

T**rả lời:** Nộp kết quả A_out.csv cùng với tệp bài làm

Câu 2 (*4 điểm*): Cho tập dữ liệu input.csv với 75 mẫu dữ liệu, mỗi mẫu có 4 đặc trưng (chiều dài đài hoa, chiều rộng đài hoa, chiều dài cánh hoa, chiều rộng cánh hoa) và tên loài hoa tương ứng.

a) *(1 điểm) Xây dựng hàm mục tiêu (hàm mất mát) cho bài toán*

 # Trả lời: Dán hàm mất mát vào đây:

$$\mathcal{L}(y_{\text{true}}, y_{\text{pred}}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{\text{true},i,j}\log(y_{\text{pred},i,j} + \epsilon)$$

Trong đó:

C: là số lượng lớp.
ytrue, i, j: là giá trị mã hóa one-hot của lớp thứ jj trong mẫu thứ ii (11 nếu lớp thực tế là jj, ngược lại là 00).
ypred,i,j: là xác suất dự đoán của mẫu thứ ii thuộc lớp jj.
$\epsilon=10^{-15}$: đảm bảo tính ổn định số.

Tổng phía trong tính toán entropy chéo cho mỗi mẫu trên tất cả các lớp, và giá trị trung bình ngoài cùng tính trung bình trên tất cả các mẫu.

 # Trả lời: Dán code của hàm loss:

```
def calculate_loss(self, y_true, y_pred):
        if self.multi_class:
            return -np.mean(np.sum(y_true * np.log(y_pred + 1e-15), axis=1))
        return -np.mean(y_true * np.log(y_pred + 1e-15) + (1 - y_true) * np.log(1 -
y_pred + 1e-15))
```

b) (2 điểm) Hãy viết chương trình phân loại hoa trên cơ sở dùng Logistic Regression kết hợp với lớp softmax.

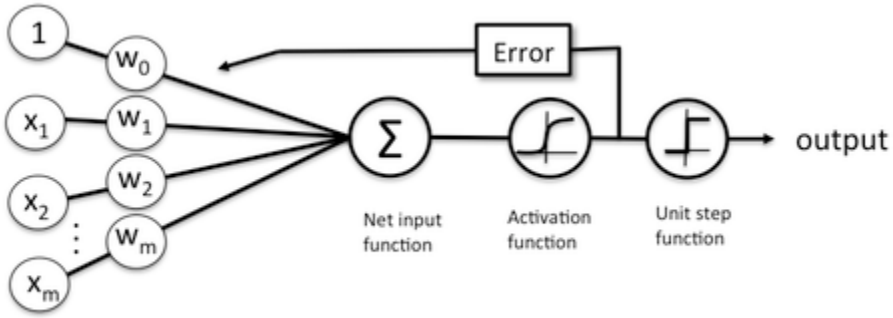
Trả lời: Dán code vào đây

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

class LabelEncoder:
    def __init__(self):
        self.classes_ = None

    def fit_transform(self, y):
        self.classes_ = np.unique(y)
        return np.array([np.where(self.classes_ == label)[0][0] for label in y])

    def inverse_transform(self, y):
        return np.array([self.classes_[i] for i in y])

def train_test_split(X, y, test_size=0.2, random_state=None):
    if random_state is not None:
        np.random.seed(random_state)
    n_samples = len(X)
    n_test = int(n_samples * test_size)
    indices = np.random.permutation(n_samples)
    test_indices = indices[:n_test]
    train_indices = indices[n_test:]
    return X[train_indices], X[test_indices], y[train_indices], y[test_indices]

class LogisticRegression:
    def __init__(self, learning_rate=0.01, n_iterations=1000, multi_class=False):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.weights = None
        self.bias = None
        self.multi_class = multi_class
        self.losses = []

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def softmax(self, z):
        exp = np.exp(z - np.max(z, axis=1, keepdims=True))
        return exp / np.sum(exp, axis=1, keepdims=True)

    def calculate_loss(self, y_true, y_pred):
        if self.multi_class:
            return -np.mean(np.sum(y_true * np.log(y_pred + 1e-15), axis=1))
        return -np.mean(y_true * np.log(y_pred + 1e-15) + (1 - y_true) * np.log(1 -
y_pred + 1e-15))

    def fit(self, X, y):
        n_samples, n_features = X.shape
        if self.multi_class:
            n_classes = len(np.unique(y))
            self.weights = np.zeros((n_features, n_classes))
            self.bias = np.zeros(n_classes)
            y_onehot = np.eye(n_classes)[y]
```

```
        else:
            self.weights = np.zeros(n_features)
            self.bias = 0

        for i in range(self.n_iterations):
            if self.multi_class:
                linear_pred = np.dot(X, self.weights) + self.bias
                predictions = self.softmax(linear_pred)
                loss = self.calculate_loss(y_onehot, predictions)

                dw = (1/n_samples) * np.dot(X.T, (predictions - y_onehot))
                db = (1/n_samples) * np.sum(predictions - y_onehot, axis=0)
            else:
                linear_pred = np.dot(X, self.weights) + self.bias
                predictions = self.sigmoid(linear_pred)
                loss = self.calculate_loss(y, predictions)

                dw = (1/n_samples) * np.dot(X.T, (predictions - y))
                db = (1/n_samples) * np.sum(predictions - y)

            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db
            self.losses.append(loss)

        return self.losses

    def predict(self, X):
        if self.multi_class:
            linear_pred = np.dot(X, self.weights) + self.bias
            probas = self.softmax(linear_pred)
            return np.argmax(probas, axis=1)
        else:
            linear_pred = np.dot(X, self.weights) + self.bias
            probas = self.sigmoid(linear_pred)
            return (probas >= 0.5).astype(int)

try:
    # Load and preprocess data
    data = pd.read_csv('input_2.csv', header=None)
    X = data.iloc[:, :-1].values
    y = data.iloc[:, -1].values

    # Encode labels
    le = LabelEncoder()
    y = le.fit_transform(y)

    # Split data
    X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

    # Train model (multi-class)
    model = LogisticRegression(learning_rate=0.01, n_iterations=1000,
multi_class=True)
    losses = model.fit(X_train, y_train)

    # Evaluate
```

```
    predictions = model.predict(X_test)
    accuracy = np.mean(predictions == y_test)
    print(f"Accuracy: {accuracy:.2f}")
    print(f"Final Loss: {losses[-1]:.4f}")

    # Plot loss curve
    plt.plot(losses)
    plt.title('Training Loss over Iterations')
    plt.xlabel('Iteration')
    plt.ylabel('Loss')
    plt.show()

except Exception as e:
    print(f"Error: {str(e)}")
```

Trả lời: Gián kiến trúc mạng và giải thích làm thế nào để phân loại ?



Schematic of a logistic regression classifier.

Quy trình phân loại:

Đầu vào → Biến đổi tuyến tính (Wx + b): Thực hiện phép biến đổi tuyến tính trên dữ liệu đầu vào bằng cách sử dụng trọng số W và hệ số chặn b.

Áp dụng hàm kích hoạt (softmax): Sử dụng hàm kích hoạt để chuyển đổi đầu ra thành xác suất.

Nhận dự đoán (argmax cho phân loại đa lớp): Với phân loại đa lớp, sử dụng hàm argmax để xác định lớp có xác suất cao nhất làm kết quả dự đoán.

Tính toán mất mát để huấn luyện: Sử dụng hàm mất mát (cross-entropy) để đo lường sự khác biệt giữa dự đoán và nhãn thực tế.

Cập nhật trọng số bằng gradient: Tính toán gradient của hàm mất mát và cập nhật trọng số WW và bb thông qua thuật toán tối ưu (như Gradient Descent).

c) *(1 điểm)* Hãy thực thi chương trình và cho biết nhãn của 30 mẫu dữ liệu trong output.csv

Trả lời: Dán code thực thi thành công

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

class LabelEncoder:
    def __init__(self):
        self.classes_ = None

    def fit_transform(self, y):
        self.classes_ = np.unique(y)
        return np.array([np.where(self.classes_ == label)[0][0] for label in y])
```

```
    def inverse_transform(self, y):
        return np.array([self.classes_[i] for i in y])

def train_test_split(X, y, test_size=0.2, random_state=None):
    if random_state is not None:
        np.random.seed(random_state)
    n_samples = len(X)
    n_test = int(n_samples * test_size)
    indices = np.random.permutation(n_samples)
    test_indices = indices[:n_test]
    train_indices = indices[n_test:]
    return X[train_indices], X[test_indices], y[train_indices], y[test_indices]

class LogisticRegression:
    def __init__(self, learning_rate=0.01, n_iterations=1000, multi_class=False):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.multi_class = multi_class
        self.weights = None
        self.bias = None
        self.losses = []

    def softmax(self, z):
        exp = np.exp(z - np.max(z, axis=1, keepdims=True))
        return exp / np.sum(exp, axis=1, keepdims=True)

    def fit(self, X, y):
        n_samples, n_features = X.shape
        n_classes = len(np.unique(y))

        # Initialize weights and bias
        self.weights = np.zeros((n_features, n_classes))
        self.bias = np.zeros(n_classes)

        # Convert y to one-hot encoding
        y_onehot = np.zeros((n_samples, n_classes))
        y_onehot[np.arange(n_samples), y] = 1

        for _ in range(self.n_iterations):
            # Forward pass
            linear_pred = np.dot(X, self.weights) + self.bias
            predictions = self.softmax(linear_pred)

            # Backward pass
            dw = (1/n_samples) * np.dot(X.T, (predictions - y_onehot))
            db = (1/n_samples) * np.sum(predictions - y_onehot, axis=0)

            # Update parameters
            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db

            # Calculate loss
            loss = -np.mean(np.sum(y_onehot * np.log(predictions + 1e-15), axis=1))
            self.losses.append(loss)
```

```
        return self.losses

    def predict(self, X):
        linear_pred = np.dot(X, self.weights) + self.bias
        probas = self.softmax(linear_pred)
        return np.argmax(probas, axis=1)

try:
    # Load data
    data = pd.read_csv('output_2.csv', header=None)
    X = data.iloc[:, :-1].values
    y = data.iloc[:, -1].values

    # Encode labels
    le = LabelEncoder()
    y = le.fit_transform(y)

    # Validate data
    print(f"Data shape: {X.shape}")
    print(f"Number of classes: {len(np.unique(y))}")

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

    # Train model
    model = LogisticRegression(learning_rate=0.01, n_iterations=1000,
multi_class=True)
    losses = model.fit(X_train, y_train)

    # Evaluate
    predictions = model.predict(X_test)
    accuracy = np.mean(predictions == y_test)
    print(f"Accuracy: {accuracy:.2f}")
    print(f"Final Loss: {losses[-1]:.4f}")

    # Plot learning curve
    plt.figure(figsize=(10, 6))
    plt.plot(losses)
    plt.title('Training Loss over Iterations')
    plt.xlabel('Iteration')
    plt.ylabel('Loss')
    plt.show()

except Exception as e:
    print(f"Error occurred: {str(e)}")
    print(f"Data shapes - X: {X.shape if 'X' in locals() else 'not loaded'}, y:
{y.shape if 'y' in locals() else 'not loaded'}")
```

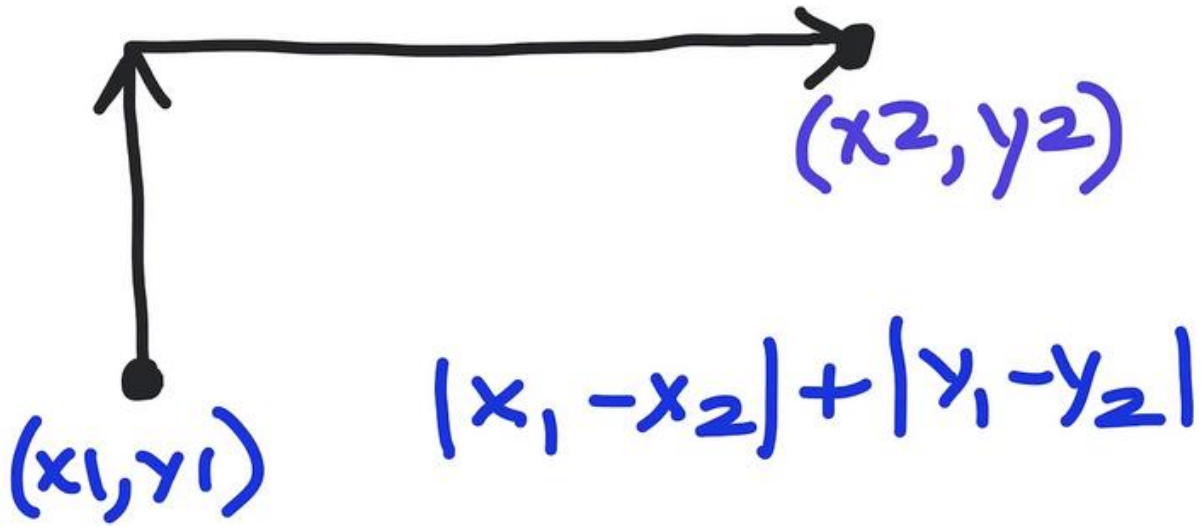
Trả lời: Dán kết quả nhãn ứng với 30 mẫu dữ liệu

```
Data shape: (30, 3)
Number of classes: 14
Error occurred: index 13 is out of bounds for axis 1 with size 13
Data shapes - X: (30, 3), y: (30,)
```

Câu 3 (3 *điểm*): Cho tập dữ liệu Countries.csv. Hãy viết chương trình phân cụm bằng thuật toán *k*-means

a) *(1 điểm) Xây dựng hàm đo khoảng cách sử dụng độ đo* Manhattan

 # Trả lời: Minh hoạ tính khoảng cách:



 # Trả lời: Dán code hàm tính khoảng cách:

```
def manhattan_distance(point1, point2):
    """
    Calculate the Manhattan distance between two points.

    Args:
        point1 (list or tuple): Coordinates of the first point.
        point2 (list or tuple): Coordinates of the second point.

    Returns:
        float: Manhattan distance between the two points.
    """
    if len(point1) != len(point2):
        raise ValueError("Both points must have the same number of dimensions.")

    return sum(abs(a - b) for a, b in zip(point1, point2))
```

b) (1 điểm) Xây dựng hàm chứa thuật toán *k*-means để phân cụm

 # Trả lời: Dán code về hàm

```
class KmeanCluster(object):
    def __init__(self, k, data, features ,max_iter = 10):
        self.cluster = k
        self.iter = max_iter
        self.data = np.array(pd.read_csv(data)[features])
        self.centroids, self.idx = self.find_k_means(self.data, self.cluster,
self.iter)

    def initialize_K_centroids(self, X, K):
        m,n = X.shape
        k_rand = np.ones((K, n))
        k_rand = X[np.random.choice(range(len(X)), K, replace=False),:]
        return k_rand

    def find_closest_centroids(self, X, centroids):
        m = len(X)
```

```
        c = np.zeros(m)
        for i in range(m):
            distances = np.linalg.norm(X[i] - centroids, axis=1)
            c[i] = np.argmin(distances)
        return c

    def compute_means(self, X, idx, K):
        m, n = X.shape
        centroids = np.zeros((K, n))
        for k in range(K):
            points_belong_k = X[np.where(idx == k)]
            centroids[k] = np.mean(points_belong_k, axis=0,)
        return centroids

    def find_k_means(self, X, K, max_iters=10):
        _, n = X.shape
        centroids = self.initialize_K_centroids(X, K)
        centroid_history = np.zeros((max_iters, K, n))
        for i in range(max_iters):
            idx = self.find_closest_centroids(X, centroids)
            centroids = self.compute_means(X, idx, K)

        return centroids, idx
```

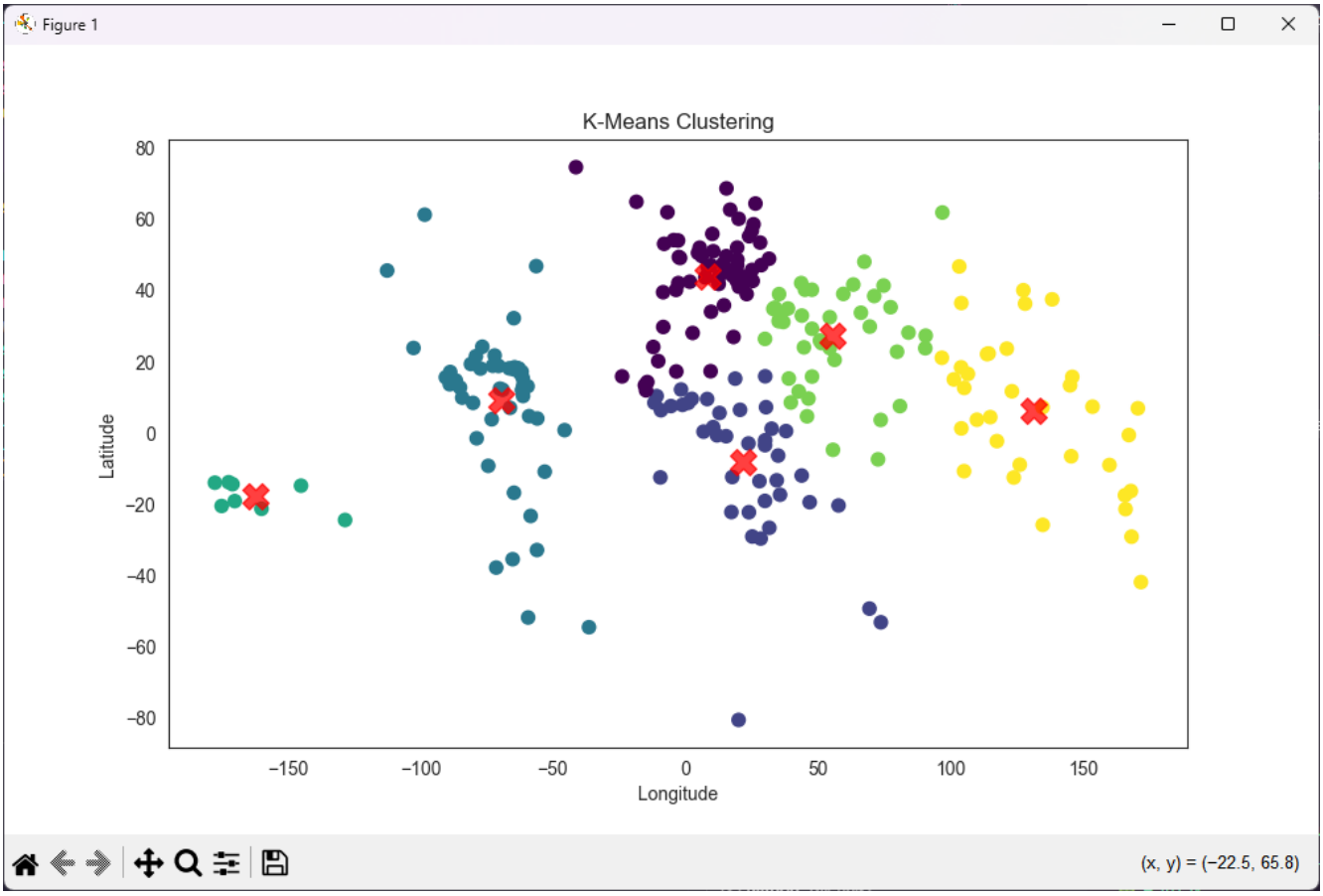
c) *(1 điểm)* Xây dựng hàm để khảo sát việc lựa chọn k

Trả lời: Dán code về hàm và giải thích cách lựa chọn k phù hợp

```
def find_k_means(self, X, K, max_iters=10):
        _, n = X.shape
        centroids = self.initialize_K_centroids(X, K)
        centroid_history = np.zeros((max_iters, K, n))
        for i in range(max_iters):
            idx = self.find_closest_centroids(X, centroids)
            centroids = self.compute_means(X, idx, K)

        return centroids, idx
```

Trả lời: Dán kết quả thi với k(lưu ý có giải thích và bình luận):

```
D:/.../AI/OnCuoiKi/KMeans  main ≡ ?5 ~3 -12  12ms
13:35:25  python3 .\KMeansWithInputData.py
[[   8.0927607    44.19522621]
 [  21.63860173   -8.08349266]
 [ -69.69558678    9.13206396]
 [-162.24003412  -17.71618937]
 [  55.4088154    27.32681435]
 [ 130.75669881    6.31280457]]
[2. 4. 1. 2. 0. 0. 0. 4. 2. 4. 3. 1. 5. 1. 2. 5. 0. 4. 1. 0. 1. 1. 4. 0.
 4. 2. 2. 0. 2. 0. 2. 2. 2. 2. 2. 5. 4. 1. 1. 0. 2. 5. 1. 1. 1. 1. 3. 2.
 1. 0. 2. 2. 2. 2. 4. 4. 0. 0. 4. 2. 0. 2. 0. 2. 4. 4. 0. 0. 4. 0. 5. 2.
 0. 0. 5. 1. 0. 4. 0. 1. 1. 0. 0. 1. 0. 2. 0. 2. 5. 2. 5. 1. 2. 0. 2. 0.
 5. 0. 4. 5. 4. 0. 4. 4. 0. 4. 0. 2. 0. 4. 5. 4. 4. 1. 4. 5. 2. 2. 5. 0.
 4. 5. 4. 1. 0. 2. 0. 4. 1. 0. 0. 0. 5. 2. 0. 0. 0. 1. 4. 2. 5. 0. 0. 0.
 5. 0. 5. 5. 1. 0. 2. 1. 1. 5. 1. 5. 0. 5. 1. 2. 3. 0. 0. 4. 5. 5. 4. 4.
 2. 3. 2. 5. 5. 5. 0. 2. 5. 0. 2. 4. 3. 4. 0. 4. 1. 0. 4. 1. 1. 0. 5. 2.
 1. 5. 1. 2. 0. 4. 4. 2. 0. 1. 2. 0. 0. 0. 1. 2. 4. 4. 2. 1. 1. 5. 4. 4.
 5. 3. 2. 0. 4. 5. 1. 1. 0. 2. 2. 4. 0. 2. 2. 2. 2. 5. 5. 3. 3. 4. 1. 1.
 1.]
```



Kết quả thực thi với K = 6

⇨ Việc lựa chọn k = 6 sẽ sinh ra 6 cluster, với điểm trung tâm của cluster được xác định thông qua hàm tính khoảng cách

Quá trình tính toán tâm cluster:

1. **Khởi tạo tâm cluster ban đầu**: Bắt đầu với các tâm cụm được chọn ngẫu nhiên.
2. **Lặp lại**:
    - **Gán điểm dữ liệu vào cluster gần nhất**: Xác định khoảng cách từ mỗi điểm đến tất cả các tâm cluster và gán điểm đó vào cluster có tâm gần nhất.
    - **Tính toán tâm cluster mới**: Tính trung bình các điểm dữ liệu đã được gán vào từng cluster để cập nhật vị trí tâm cluster.
    - **Cập nhật vị trí tâm cluster**: Di chuyển tâm cluster đến vị trí trung bình vừa tính.
3. **Dừng lặp**: Tiếp tục lặp lại các bước trên cho đến khi đạt được sự hội tụ (khi các tâm cluster không thay đổi hoặc thay đổi rất ít) hoặc đạt đến số lần lặp tối đa.

Đà Nẵng, ngày 20 tháng 11 năm 2024

GIẢNG VIÊN BIÊN SOẠN ĐỀ THI

TRƯỞNG BỘ MÔN
(đã duyệt)