2025-05-13 15:14

Tags:

# Data\_Mining\_Lab6\_Report

**Họ và Tên**: Mai Phong Đăng **MSSV**: 22280008 **Lớp**: 22KDL

## Thuật Toán K-means và K-meadian

### **K-means**

* Đọc và Khám phá dữ liệu:

import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
from matplotlib.colors import ListedColormap  
#reading the dataset  
blobs = pd.read\_csv('data.csv')  
colnames = list(blobs.columns[1:-1])  
print(blobs.head())

* trực quan hóa:

customcmap = ListedColormap(['crimson', 'mediumblue', 'darkmagenta'])  
  
fig,ax = plt.subplots(figsize=(8, 6))  
plt.scatter(x=blobs['x'], y=blobs['y'], s=150,  
 c=blobs['cluster'].astype('category'),  
 cmap=customcmap)  
ax.set\_xlabel(r'x', fontsize=14)  
ax.set\_ylabel(r'y', fontsize=14)  
plt.xticks(fontsize=12)  
plt.yticks(fontsize=12)  
plt.show()

* Bước 1: Khởi tạo tâm cụm ngẫu nhiên

customcmap = ListedColormap(['crimson', 'mediumblue', 'darkmagenta'])  
fig,ax = plt.subplots(figsize=(8, 6))  
plt.scatter(x=blobs['x'], y=blobs['y'], s=150,  
 c=blobs['cluster'].astype('category'),  
 cmap=customcmap)  
ax.set\_xlabel(r'x', fontsize=14)  
ax.set\_ylabel(r'y', fontsize=14)  
plt.xticks(fontsize=12)  
plt.yticks(fontsize=12)  
plt.show()

* Bước 2: tính toán khoảng cách Euclidean

def rsserr(a,b):  
 '''  
 Calculate the root sum squared error  
 a and b are numpy arrays  
 '''  
 return (np.sum(np.square(a-b)))

* Bước 3: Gán điểm dữ liệu cho cụm gần nhất:

def rsserr(a,b):  
 '''  
 Calculate the root sum squared error  
 a and b are numpy arrays  
 '''  
 return (np.sum(np.square(a-b)))

* Bước 4: Cập nhật tâm cụm

centroids = df.groupby('centroid').agg('mean').loc[:, colnames].reset\_index(drop = True)

* Bước 5: hoàn thiện thuật toán

def kmeans(dset, k=2, tol=1e-4):  
 # Work in a copy   
 working\_dset = dset.copy()  
 # Define some variables to hold the error, the  
 # stopping signal and a counter for the iterations  
 err = []  
 goahead = True  
 j = 0  
  
 # Step 2: Inititate clusters by defining centroids  
 centroids = initiate\_centroids(k, working\_dset)  
  
 while(goahead):  
 # Step 3 and 4: Assign centroids and calc error  
 working\_dset['centroid'], j\_err = centroid\_assignation(working\_dset, centroids)  
 err.append(sum(j\_err))  
 # Step 5: update centoids position  
 centroids = working\_dset.groupby('centroid').agg('mean').reset\_index(drop = True)  
  
 # Step 6: Restart iteration  
 if j > 0:  
 # Is the error less than a tolerance (1e-4))  
 if err[j-1]-err[j] < tol:  
 goahead = False  
 j+=1  
 return working\_dset['centroid'], j\_err, centroids

* Tìm số cụm tối ưu bằng Elbow plot:

err\_total = []  
n = 10  
df\_elbow = blobs[['x', 'y']]  
for i in range(n):  
 \_, my\_err, \_ = kmeans(df\_elbow, i+1)  
 err\_total.append(sum(my\_err))  
fig, ax = plt.subplots(figsize=(8, 6))  
plt.plot(range(1, n+1), err\_total, linewidth=3, marker = 'o')  
ax.set\_xlabel(r'Number of clusters', fontsize=14)  
ax.set\_ylabel(r'Total error', fontsize=14)  
plt.xticks(fontsize=12)  
plt.yticks(fontsize=12)  
plt.show()

### **K-Medians**

* K-Medians là một biến thể của K-Means nhưng sử dụng khoảng cách Manhattan thay vì Euclidean và chọn medoid (điểm thực tế trong dữ liệu) thay vì centroids.
* *Bước 1: Khởi tạo Medoids*

def initiate\_medoids(k, dset):  
 '''  
 Select k data points as medoids   
 k: number of medoids  
 dset: pandas dataframe  
 '''  
 medoids = dset.sample(k)  
 return medoids

* *Bước 2: Tính khoảng cách Manhattan*

def kmed\_rsserr(point1, point2):  
 """Tính khoảng cách Manhattan giữa hai điểm."""  
 return np.sum(np.abs(point1 - point2))

* *Bước 3: Gán điểm dữ liệu cho Medoids gần nhất*

def assign\_to\_medoids(dset, medoids\_df):  
 """  
 Gán mỗi điểm trong dset cho medoid gần nhất dựa trên khoảng cách Manhattan.  
 """  
 num\_obs = dset.shape[0]  
 num\_medoids = medoids\_df.shape[0]  
   
 assignments = np.zeros(num\_obs, dtype=int)  
 assignment\_errors = np.zeros(num\_obs)  
  
 # Chuyển sang numpy arrays để tăng tốc độ  
 dset\_values = dset.values  
 medoids\_values = medoids\_df.values  
  
 for i in range(num\_obs):  
 current\_point = dset\_values[i, :]  
 distances\_to\_medoids = np.zeros(num\_medoids)  
 for j in range(num\_medoids):  
 distances\_to\_medoids[j] = kmed\_rsserr(current\_point, medoids\_values[j, :])  
   
 assignments[i] = np.argmin(distances\_to\_medoids)  
 assignment\_errors[i] = np.min(distances\_to\_medoids)  
   
 return assignments, assignment\_errors

* *Bước 4: Cập nhật medoids*
* Cập nhật medoids bằng cách tính giá trị trung vị cho từng thành phần (chiều) riêng biệt.

def update\_medoids(dset, assignments, k):  
 """  
 dset: DataFrame chứa dữ liệu (chỉ các cột đặc trưng).  
 assignments: Mảng chứa chỉ số cụm được gán cho mỗi điểm.  
 k: Số lượng cụm (medoids).  
 """  
 new\_medoids\_list = []  
 feature\_cols = dset.columns.tolist()  
   
 for i in range(k): # Với mỗi cụm i  
 # Lấy các điểm thuộc cụm i  
 cluster\_points\_df = dset[assignments == i]  
   
 if cluster\_points\_df.empty:  
 # Nếu cụm rỗng, chọn một điểm ngẫu nhiên  
 print(f"Cảnh báo: Cụm {i} rỗng. Chọn medoid ngẫu nhiên.")  
 random\_point = dset.sample(1)  
 new\_medoids\_list.append(random\_point.iloc[0])  
 continue  
   
 # Tạo một dict để lưu các giá trị trung vị cho từng thành phần  
 medoid\_components = {}  
   
 # Tính giá trị trung vị cho từng thành phần (chiều)  
 for feature in feature\_cols:  
 medoid\_components[feature] = cluster\_points\_df[feature].median()  
   
 # Tạo medoid mới từ các giá trị trung vị đã tính  
 new\_medoid = pd.Series(medoid\_components)  
 new\_medoids\_list.append(new\_medoid)  
   
 # Tạo DataFrame từ danh sách các medoid mới  
 new\_medoids\_df = pd.DataFrame(new\_medoids\_list)  
   
 # Đảm bảo các cột được giữ lại và đúng thứ tự  
 if not new\_medoids\_df.empty:  
 new\_medoids\_df = new\_medoids\_df[feature\_cols]  
   
 return new\_medoids\_df.reset\_index(drop=True)

* Bước 5: hoàn thiện thuật toán

def kmedians(dset\_features, k, tol=1e-4, max\_iter=100):  
 """  
 Thuật toán K-Medians.  
 """  
 # Khởi tạo medoids  
 current\_medoids = initiate\_centroids(k, dset\_features)   
   
 error\_history = []  
   
 for iteration in range(max\_iter):  
 # Gán các điểm cho medoid gần nhất  
 assignments, assignment\_errors = assign\_to\_medoids(dset\_features, current\_medoids)  
   
 total\_error = np.sum(assignment\_errors)  
 error\_history.append(total\_error)  
   
 # Cập nhật medoids  
 new\_medoids = update\_medoids(dset\_features, assignments, k)  
  
 if new\_medoids.empty or new\_medoids.shape[0] < k :  
 print(f"Lỗi ở vòng lặp {iteration}: Không đủ medoid được tạo ra. Dừng thuật toán.")  
 break  
   
 # Kiểm tra sự hội tụ  
 if current\_medoids.equals(new\_medoids):  
 print(f"Hội tụ ở vòng lặp {iteration+1} do medoids không thay đổi.")  
 break  
   
 current\_medoids = new\_medoids  
   
 if iteration > 0:  
 if abs(error\_history[iteration-1] - error\_history[iteration]) < tol:  
 print(f"Hội tụ ở vòng lặp {iteration+1} do thay đổi lỗi nhỏ hơn dung sai.")  
 break  
   
 if iteration == max\_iter - 1:  
 print("Đạt số vòng lặp tối đa.")  
   
 final\_assignments, final\_errors = assign\_to\_medoids(dset\_features, current\_medoids)  
 return final\_assignments, final\_errors, current\_medoids, error\_history

## Kết quả và so sánh

Cả K-Means và K-Medians đều có khả năng phân cụm tốt cho tập dữ liệu này, tuy nhiên có một số khác biệt cơ bản:

1. **Độ đo khoảng cách**:
   * K-Means sử dụng khoảng cách Euclidean (bình phương)
   * K-Medians sử dụng khoảng cách Manhattan (tổng giá trị tuyệt đối)
2. **Cập nhật tâm cụm**:
   * K-Means sử dụng trung bình các điểm (centroids có thể không phải là điểm thực tế trong dữ liệu)
   * K-Medians sử dụng điểm thực tế trong dữ liệu (medoid)
3. **Độ nhạy với outlier**:
   * K-Means nhạy hơn với các điểm ngoại lai do sử dụng bình phương khoảng cách
   * K-Medians ít nhạy cảm hơn với các điểm ngoại lai Trực quan kết quả cho thấy cả hai thuật toán đều có thể phân tách 3 nhóm dữ liệu một cách hiệu quả, với vài khác biệt nhỏ về biên của các cụm.

# References