NP

AP 1

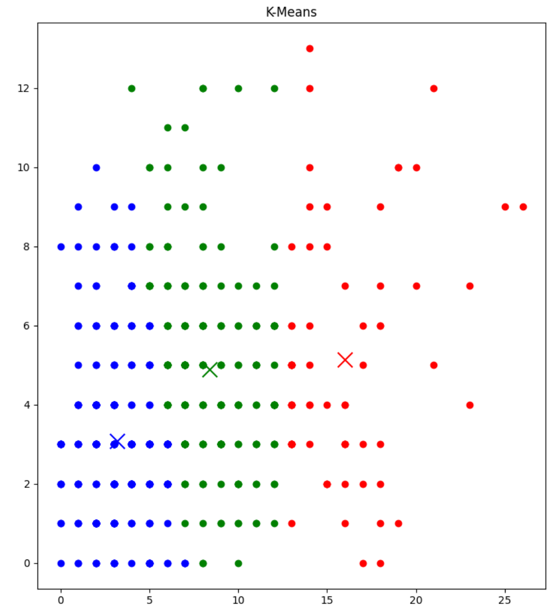
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Load data
df = pd.read_csv('./content/Players.csv').drop(['Rank', 'Player', 'Earnings'],
axis=1)
data = df.to_numpy()

# Euclidean distance function
def norm2_distance(a, b):
    return np.sqrt(np.sum((a - b) ** 2))

# K-Means Initialization
def initialize_centroids(data, k):
    np.random.seed(42)
    random_indices = np.random.permutation(data.shape[0])
    centroids = data[random_indices[:k]]
    return centroids

# K-Medoids Initialization
def initialize_medoids(data, k):
    np.random.seed(42)
    random_indices = np.random.permutation(data.shape[0])
    return data[random_indices[:k]]

# Assign clusters for K-Means
def assign_clusters_centroids(data, centroids):
    clusters = []
    for point in data:
        distances = [norm2_distance(point, centroid) for centroid in centroids]
        cluster = np.argmin(distances)
        clusters.append(cluster)
    return np.array(clusters)

# Assign clusters for K-Medoids
def assign_clusters_medoids(data, medoids):
    clusters = []
```

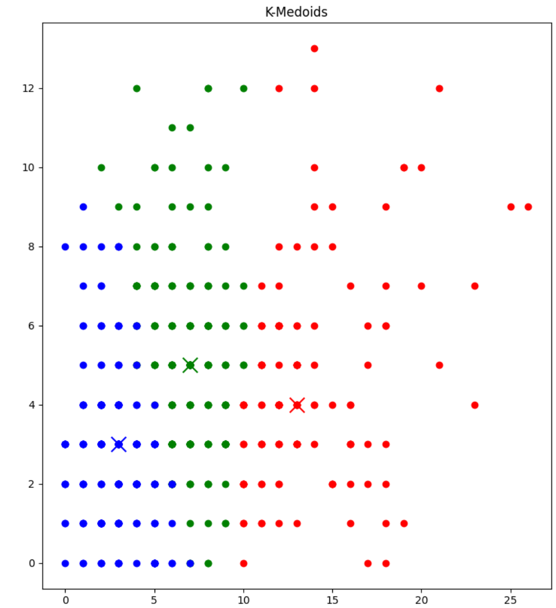
```
    for point in data:
        distances = [norm2_distance(point, medoid) for medoid in medoids]
        cluster = np.argmin(distances)
        clusters.append(cluster)
    return np.array(clusters)

# Update centroids for K-Means
def update_centroids(data, clusters, k):
    new_centroids = np.zeros((k, data.shape[1]))
    for i in range(k):
        points_in_cluster = data[clusters == i]
        if points_in_cluster.shape[0] > 0:
            new_centroids[i] = np.mean(points_in_cluster, axis=0)
    return new_centroids

# Update medoids for K-Medoids
def update_medoids(data, clusters, k):
    new_medoids = np.zeros((k, data.shape[1]))
    for i in range(k):
        points_in_cluster = data[clusters == i]
        if points_in_cluster.shape[0] > 0:
            medoid = min(points_in_cluster, key=lambda point:
np.sum([norm2_distance(point, other) for other in points_in_cluster]))
            new_medoids[i] = medoid
    return new_medoids

# K-Means Algorithm
def kmeans(data, k=3, max_iterations=100):
    centroids = initialize_centroids(data, k)

    for _ in range(max_iterations):
        clusters = assign_clusters_centroids(data, centroids)
        new_centroids = update_centroids(data, clusters, k)

        if np.all(centroids == new_centroids):
            break
        centroids = new_centroids

    return centroids, clusters

# K-Medoids Algorithm
def kmedoids(data, k=3, max_iterations=100):
    medoids = initialize_medoids(data, k)
```

```python
    for _ in range(max_iterations):
        clusters = assign_clusters_medoids(data, medoids)
        new_medoids = update_medoids(data, clusters, k)

        if np.all(medoids == new_medoids):
            break
        medoids = new_medoids

    return medoids, clusters

# Inertia Calculation (sum of squared distances for centroids/medoids)
def calculate_inertia(data, centers, clusters):
    inertia = 0
    for i, center in enumerate(centers):
        points_in_cluster = data[clusters == i]
        inertia += np.sum((points_in_cluster - center) ** 2)
    return inertia

# Frobenius norm for clusters
def frobenius_norm(data, centers, clusters):
    total_frobenius = 0
    for i, center in enumerate(centers):
        points_in_cluster = data[clusters == i]
        distances = points_in_cluster - center
        total_frobenius += np.linalg.norm(distances, ord='fro')
    return total_frobenius

# Run K-Means and K-Medoids
centroids, kmeans_clusters = kmeans(data, k=3)
medoids, kmedoids_clusters = kmedoids(data, k=3)

# Calculate inertia for both methods
kmeans_inertia = calculate_inertia(data, centroids, kmeans_clusters)
kmedoids_inertia = calculate_inertia(data, medoids, kmedoids_clusters)

# Calculate Frobenius norm for both methods
kmeans_frobenius = frobenius_norm(data, centroids, kmeans_clusters)
kmedoids_frobenius = frobenius_norm(data, medoids, kmedoids_clusters)

# Print results
print(f"K-Means Inertia: {kmeans_inertia}")
```

```python
print(f"K-Medoids Inertia: {kmedoids_inertia}")
print(f"K-Means Frobenius Norm: {kmeans_frobenius}")
print(f"K-Medoids Frobenius Norm: {kmedoids_frobenius}")

# Plot clustering results
colors = ['r', 'g', 'b']

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 6))

# K-Means Clusters
for i in range(3):
    points = data[kmeans_clusters == i]
    ax1.scatter(points[:, 0], points[:, 1], c=colors[i])
for i, centroid in enumerate(centroids):
    ax1.scatter(centroid[0], centroid[1], c=colors[i], marker='x', s=200)
ax1.set_title('K-Means')

# K-Medoids Clusters
for i in range(3):
    points = data[kmedoids_clusters == i]
    ax2.scatter(points[:, 0], points[:, 1], c=colors[i])
for i, medoid in enumerate(medoids):
    ax2.scatter(medoid[0], medoid[1], c=colors[i], marker='x', s=200)
ax2.set_title('K-Medoids')

plt.show()
```

Output:






```
K-Means Inertia: 5981.267700500336
K-Medoids Inertia: 6143.0
K-Means Frobenius Norm: 133.16333836621902
K-Medoids Frobenius Norm: 134.7190655373426
```

Inertia is a measure of how tight or compact the clusters are. Lower inertia indicates that the data points are closer to their respective cluster centers, which suggests better-defined clusters. We can clearly see that K_means is the better algorithm in this case, since it has a lower score than k-medoids inertia, so tighter clusters.

I considered both vector norms and matrix norms, frobenius Norm gives a different perspective on the spread or shape of the clusters, the lower score means tighter clusters, and as we can see k_means has slightly lower score, thus tighter clusters.

Overall **k_means method is better** that k_medoids.

Real world applications

Input data: valorant players rank

Norms: vector norm L2 and matrix norm frobenius

As you can see above, i ran multiple tests above, compared k_means and k_medoids algorithm based on inertia, also implemented both matrix and vector norms, and clustered the dataset above.