import matplotlib

import matplotlib.pyplot as plt

from sklearn import datasets

import numpy as np

from numpy.linalg import norm

from sklearn.cluster import KMeans

Iniciation K-Moyennes++

def initPlusPlus(X,K):

    N,p = np.shape(X)

    C = np.zeros((p,K))

    generator = np.random.default\_rng()

    index = np.random.choice(N, 1,replace = False)

    liste\_index = [index]

    C[:,0] = X[index,:]

    X = np.delete(X,index,0)

    #print("k=0 C[k]=",C[:,0],"index=",index)

    k=1

    while k < K:

        # calcul des distances

        NN = X.shape[0]

        dist = np.zeros(NN)

        for i in range(NN):

            # Para cada ponto em X, calcular a menor distância até qualquer centroide já escolhido

            dist[i] = np.min([np.linalg.norm(X[i, :] - C[:, j])\*\*2 for j in range(k)])

        # calcul des probabilités

        dist\_sum = np.sum(dist)

        if dist\_sum == 0:

            dist\_sum = 1  # Evitar divisão por zero caso todas as distâncias sejam muito pequenas

        probs = dist / dist\_sum  # Probabilidade proporcional à distância

        # tirage aléatoire selon proba

        cumulative\_probs = np.cumsum(probs)

        r = np.random.rand()  # Gerar um número aleatório para o sorteio

        index\_next = np.where(cumulative\_probs >= r)[0][0]  # Selecionar o próximo ponto

        C[:, k] = X[index\_next, :]

        #print(f"k={k} C[k]={C[:, k]} index={index\_next}")

        # Remover o ponto escolhido de X

        X = np.delete(X, index\_next, 0)

        k += 1

    return C

Iniciation K-Medoids++

def initMedoidsPlusPlus(X, K):

    N, p = np.shape(X)

    C = np.zeros((K, p))  # Initialize C to store medoids as rows now (K, p)

    # Step 1: Randomly select the first medoid from the data points

    index = np.random.choice(N, 1, replace=False)

    C[0, :] = X[index, :].squeeze()  # Fix the shape mismatch by assigning correctly

    selected\_indices = [index[0]]

    X\_remaining = np.delete(X, index, 0)  # Remove the selected medoid from X

    k = 1

    while k < K:

        # Step 2: Calculate the minimum distance from each point in X\_remaining to the already selected medoids

        NN = X\_remaining.shape[0]

        dist = np.zeros(NN)

        for i in range(NN):

            dist[i] = np.min([np.linalg.norm(X\_remaining[i, :] - C[j, :])\*\*2 for j in range(k)])  # Use correct shape

        # Step 3: Calculate the probabilities proportional to the squared distances

        dist\_sum = np.sum(dist)

        if dist\_sum == 0:

            dist\_sum = 1  # Avoid division by zero

        probs = dist / dist\_sum

        # Step 4: Select the next medoid based on the computed probabilities

        cumulative\_probs = np.cumsum(probs)

        r = np.random.rand()  # Generate a random number for probabilistic selection

        index\_next = np.where(cumulative\_probs >= r)[0][0]

        # Add the selected medoid to the C matrix

        C[k, :] = X\_remaining[index\_next, :].squeeze()  # Ensure correct shape for the medoid

        # Remove the selected point from X\_remaining and update selected\_indices

        selected\_indices.append(index\_next)

        X\_remaining = np.delete(X\_remaining, index\_next, 0)

        k += 1

    return C

K-Moyennes

def my\_kmeans(X,K,init,Visualisation=False,Seuil=0.001,Max\_iterations = 100):

    N,p = np.shape(X)

    iteration = 0

    Dist=np.zeros((K,N))

    J=np.zeros(Max\_iterations+1)

    J[0] = 10000000

    # Initialisation des clusters

    Index\_init = np.random.choice(N, K,replace = False)

    C = np.zeros((p,K))

    if init == "plusplus":

        C = initPlusPlus(X, K)

    else:

        for k in range(K):

            C[:,k] = X[Index\_init[k],:].T

    while iteration < Max\_iterations:

        iteration +=1

        #################################################################

        # E step : estimation des données manquantes

        for k in range(K):

            Dist[k, :] = np.linalg.norm(X - C[:, k].T, axis=1)

        # affectation des données aux clusters les plus proches

        y = np.argmin(Dist, axis=0)

        #################################################################

        # M Step : calcul des meilleurs centres

        for k in range(K):

            C[:, k] = np.mean(X[y == k], axis=0)

        #################################################################

        # test du critère d'arrêt l'évolution du critère est inférieure

        # au Seuil en pour ceent

        J[iteration] = np.sum([np.linalg.norm(X[y == k] - C[:, k].T)\*\*2 for k in range(K)])/N

        #J[iteration] += np.sum(np.min(Dist[y:1],axis=0))/N

        # Vérifier si le changement dans J est inférieur au seuil

        if abs(J[iteration] - J[iteration - 1]) / J[iteration - 1] < Seuil:

            break

    return C, y, J[1:iteration]

K-Médoïdes

def my\_kmedoides(X, K, init, Visualisation=False, Seuil=0.001, Max\_iterations=100):

    N, p = np.shape(X)

    iteration = 0

    Normes = np.zeros((K, N))

    J = np.zeros(Max\_iterations+1)

    # Initialisation des clusters

    Index\_init = np.random.choice(N, K, replace=False)

    C = np.zeros((K, p))

    if init == "plusplus":

        C = initMedoidsPlusPlus(X, K)  # Use the Medoids++ initialization

    else:

        for k in range(K):

            C[k, :] = X[Index\_init[k], :]

    while iteration < Max\_iterations:

        #################################################################

        # Affectation des données aux médoïdes les plus proches

        for k in range(K):

            # Calculer la distance de Manhattan entre chaque point et le médoïde k

            Normes[k, :] = np.sum(np.abs(X - C[k, :]), axis=1)  # Correct broadcasting for distance calculation

        # Affecter chaque point au médoïde le plus proche (avec la distance minimale)

        y = np.argmin(Normes, axis=0)  # Assigner chaque point au médoïde ayant la plus faible distance

        #################################################################

        # Calcul des meilleurs médoïdes

        for k in range(K):

            points\_in\_cluster = X[y == k]  # Points assignés au médoïde k

            if len(points\_in\_cluster) > 0:

                # Calculer la somme des distances pour chaque point dans le cluster

                distances = np.sum(np.abs(points\_in\_cluster[:, np.newaxis, :] - points\_in\_cluster[np.newaxis, :, :]), axis=2)

                # Trouver le point qui minimise la somme des distances dans le cluster

                medoid\_index = np.argmin(np.sum(distances, axis=1))

                C[k, :] = points\_in\_cluster[medoid\_index]  # Le nouveau médoïde est celui qui minimise les distances

        #################################################################

        # Test du critère d'arrêt

        J[iteration] = np.sum([np.linalg.norm(X[y == k] - C[k, :].T)\*\*2 for k in range(K)])/N

        # Vérifier si le changement dans J est inférieur au seuil (critère de convergence)

        if iteration > 0 and abs(J[iteration] - J[iteration - 1]) / J[iteration - 1] < Seuil:

            break  # Arrêter si le critère est respecté

        iteration += 1  # Passer à l'itération suivante

    return C, y, J[:iteration+1]

K-Means ScikitLearn

def kmeans\_sklearn(X, K, max\_iter=100, tol=0.0001, random\_state=None):

    # Initialize the KMeans model with the specified number of clusters and parameters

    kmeans = KMeans(n\_clusters=K, max\_iter=max\_iter, tol=tol, random\_state=random\_state)

    # Fit the model to the data

    kmeans.fit(X)

    # Retrieve the centroids, labels, and inertia (sum of squared distances)

    centroids = kmeans.cluster\_centers\_

    labels = kmeans.labels\_

    inertia = kmeans.inertia\_

    return centroids, labels, inertia

Resultat Réel

colors =['r','b','g','c','m','o']

n\_colors = 6

#########################################################

#''' K means '''

iris = datasets.load\_iris()

X = iris.data[:, :2]  # we only take the first two features.

y = iris.target

K = 3

N,p = np.shape(X)

fig = plt.figure(2, figsize=(8, 6))

plt.clf()

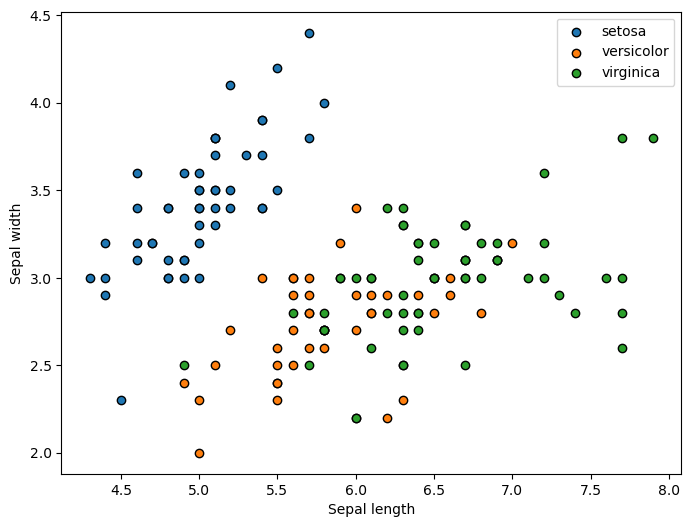
plt.scatter(X[0:50, 0], X[0:50, 1], cmap=plt.cm.Set1,edgecolor='k',label=iris.target\_names[0])

plt.scatter(X[50:100, 0], X[50:100, 1], cmap=plt.cm.Set1,edgecolor='k',label=iris.target\_names[1])

plt.scatter(X[100:150, 0], X[100:150, 1], cmap=plt.cm.Set1,edgecolor='k',label=iris.target\_names[2])

plt.xlabel('Sepal length')

plt.ylabel('Sepal width')

plt.legend(scatterpoints=1)

Resultat Calculé

# Run k-means and k-medoids clustering

Cluster, y2, Critere = my\_kmeans(iris.data, K, "", Visualisation=False)

ClusterPlus, yPlus, CriterePlus = my\_kmeans(iris.data, K, "plusplus", Visualisation=False)

ClusterMedoid, yMedoid, CritereMedoid = my\_kmedoides(X, K, "")

ClusterMedoidPlus, yMedoidPlus, CritereMedoidPlus = my\_kmedoides(X, K, "plusplus")

ClusterSklearn, ySklearn, CritereSklearn = kmeans\_sklearn(X, K)

# Create subplots for side-by-side comparison

fig, axs = plt.subplots(3, 2, figsize=(18, 10))

# Plot first graph (K-means)

for k in range(K):

    axs[0,0].plot(X[y2 == k, 0], X[y2 == k, 1], colors[k % len(colors)] + 'o')

axs[0,0].plot(Cluster[0, :], Cluster[1, :], 'kx')

axs[0,0].set\_title(f'K-means ({K} clusters)')

# Plot second graph (K-means++ initialization)

for k in range(K):

    axs[0,1].plot(X[yPlus == k, 0], X[yPlus == k, 1], colors[k % len(colors)] + 'o')

axs[0,1].plot(ClusterPlus[0, :], ClusterPlus[1, :], 'kx')

axs[0,1].set\_title(f'K-means++ ({K} clusters)')

# Plot third graph (K-medoids)

for k in range(K):

    axs[1,0].plot(X[yMedoid == k, 0], X[yMedoid == k, 1], colors[k % len(colors)] + 'o')

axs[1,0].plot(ClusterMedoid[:, 0], ClusterMedoid[:, 1], 'kx')

axs[1,0].set\_title(f'K-medoids ({K} clusters)')

# Plot fourth graph (K-medoids)

for k in range(K):

    axs[1,1].plot(X[yMedoidPlus == k, 0], X[yMedoidPlus == k, 1], colors[k % len(colors)] + 'o')

axs[1,1].plot(ClusterMedoidPlus[:, 0], ClusterMedoidPlus[:, 1], 'kx')

axs[1,1].set\_title(f'K-medoids++ ({K} clusters)')

# Plot Scikit-learn KMeans

for k in range(K):

    axs[2, 0].plot(X[ySklearn == k, 0], X[ySklearn == k, 1], colors[k % len(colors)] + 'o')

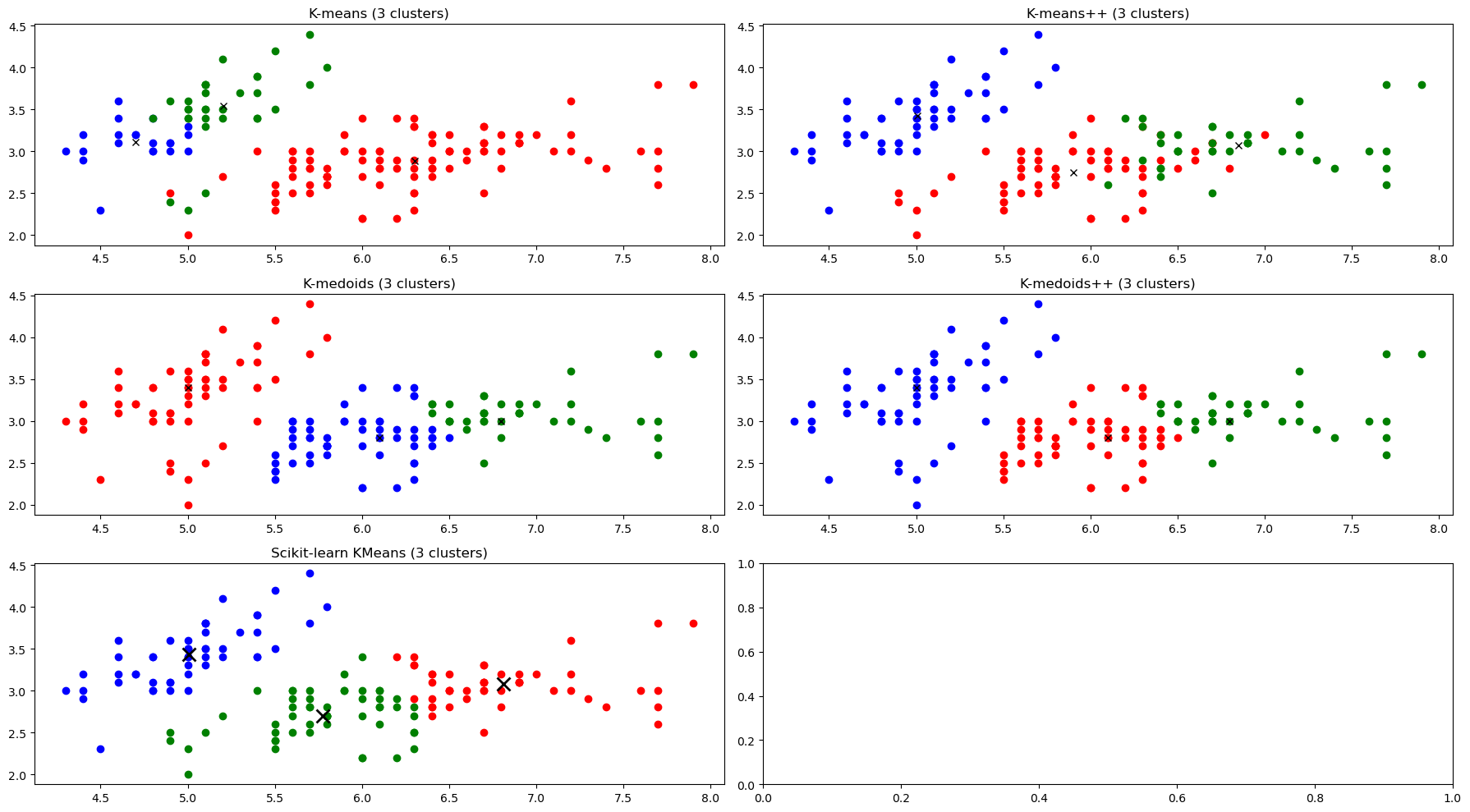
axs[2, 0].plot(ClusterSklearn[:, 0], ClusterSklearn[:, 1], 'kx', markersize=12, markeredgewidth=2)

axs[2, 0].set\_title(f'Scikit-learn KMeans ({K} clusters)')

# Display the figure

plt.tight\_layout()

plt.show()



**Conclusions**

Les trois algorithmes — K-means, K-means++ et K-medoids — produisent des résultats similaires aux clusters officiels et à l'implémentation scikit-learn. Cependant, K-means présente une variation significative lorsqu'il est exécuté plusieurs fois, en raison de sa forte dépendance à l'aléatoire pour l'initialisation des centroids.

Ce problème est atténué dans K-means++, où l'initialisation est plus systématique, avec tous les points sauf le premier sélectionnés selon une règle probabiliste qui améliore la cohérence. K-medoids, quant à lui, montre des résultats stables, car il utilise des points réels du jeu de données comme centroids, offrant ainsi une meilleure cohérence et interprétabilité.

Il convient de noter que, dans la visualisation en 2D, certains points d’un cluster peuvent apparaître plus proches du centroid d’un autre cluster. Cela est dû à l'influence d'autres caractéristiques non représentées dans la projection en 2D, qui jouent un rôle important dans la classification.

fig = plt.figure(figsize=(8, 6))

plt.plot(CritereMedoidPlus, 'o-', color='maroon')

plt.plot(CritereMedoid, 'o-', color='r')

plt.plot(CriterePlus, 'o-', color='navy')

plt.plot(Critere, 'o-', color='b')

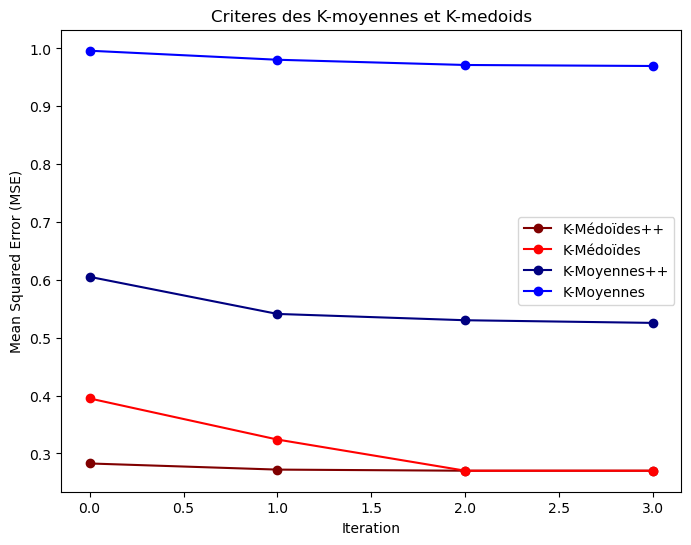
plt.legend(['K-Médoïdes++', 'K-Médoïdes', 'K-Moyennes++', 'K-Moyennes'])

plt.xlabel('Iteration')

plt.ylabel('Mean Squared Error (MSE)')

plt.title('Criteres des K-moyennes et K-medoids')

plt.show()



**Conclusions**

Bien que le nombre d'itérations varie en fonction des appels de fonction (en raison de l'initialisation aléatoire), K-means++ se révèle généralement plus efficace que le K-means classique, grâce à son initialisation plus stratégique. K-medoids, tout en affichant le MSE le plus bas dans l'ensemble, peut également être affecté par l'aléa de l'initialisation.

Resultats pour Cancer Dataset

# Load the Breast Cancer dataset

breast\_cancer = datasets.load\_breast\_cancer()

Xbreast = breast\_cancer.data#[:, :2]  # We take the first two features for visualization purposes

ybreast = breast\_cancer.target

Kbreast = 2  # Number of clusters

Nbreast, pbreast = np.shape(X)

colors = ['r', 'g', 'b', 'y', 'c', 'm']

# Plot the data (First two features)

fig = plt.figure(2, figsize=(8, 6))

plt.clf()

plt.scatter(Xbreast[ybreast == 0, 0], Xbreast[ybreast == 0, 1], color='r', edgecolor='k', label=breast\_cancer.target\_names[0])

plt.scatter(Xbreast[ybreast == 1, 0], Xbreast[ybreast == 1, 1], color='b', edgecolor='k', label=breast\_cancer.target\_names[1])

plt.xlabel(breast\_cancer.feature\_names[0])

plt.ylabel(breast\_cancer.feature\_names[1])

plt.legend(scatterpoints=1)

plt.title("Breast Cancer Dataset")

plt.show()

# Run k-means and k-medoids clustering

Cluster\_b, y2\_b, Critere\_b = my\_kmeans(breast\_cancer.data, Kbreast, "", Visualisation=False)

ClusterPlus\_b, yPlus\_b, CriterePlus\_b = my\_kmeans(breast\_cancer.data, Kbreast, "plusplus", Visualisation=False)

ClusterMedoid\_b, yMedoid\_b, CritereMedoid\_b = my\_kmedoides(Xbreast, Kbreast, "")

ClusterMedoidPlus\_b, yMedoidPlus\_b, CritereMedoidPlus\_b = my\_kmedoides(Xbreast, Kbreast, "plusplus")

ClusterSklearn\_b, ySklearn\_b, Inertia\_b = kmeans\_sklearn(Xbreast, Kbreast)

# Create subplots for side-by-side comparison

fig, axs = plt.subplots(3, 2, figsize=(18, 10))

# Plot first graph (K-means)

for k in range(Kbreast):

    axs[0,0].plot(Xbreast[y2\_b == k, 0], Xbreast[y2\_b == k, 1], colors[k % len(colors)] + 'o')

axs[0,0].plot(Cluster\_b[0, :], Cluster\_b[1, :], 'kx')

axs[0,0].set\_title(f'K-means ({Kbreast} clusters)')

# Plot second graph (K-means++ initialization)

for k in range(Kbreast):

    axs[0,1].plot(Xbreast[yPlus\_b == k, 0], Xbreast[yPlus\_b == k, 1], colors[k % len(colors)] + 'o')

axs[0,1].plot(ClusterPlus\_b[0, :], ClusterPlus\_b[1, :], 'kx')

axs[0,1].set\_title(f'K-means++ ({Kbreast} clusters)')

# Plot third graph (K-medoids)

for k in range(Kbreast):

    axs[1,0].plot(Xbreast[yMedoid\_b == k, 0], Xbreast[yMedoid\_b == k, 1], colors[k % len(colors)] + 'o')

axs[1,0].plot(ClusterMedoid\_b[:, 0], ClusterMedoid\_b[:, 1], 'kx')

axs[1,0].set\_title(f'K-medoids ({Kbreast} clusters)')

# Plot fourth graph (K-medoids)

for k in range(Kbreast):

    axs[1,1].plot(Xbreast[yMedoidPlus\_b == k, 0], Xbreast[yMedoidPlus\_b == k, 1], colors[k % len(colors)] + 'o')

axs[1,1].plot(ClusterMedoidPlus\_b[:, 0], ClusterMedoidPlus\_b[:, 1], 'kx')

axs[1,1].set\_title(f'K-medoids++ ({Kbreast} clusters)')

# Plot Scikit-learn KMeans

for k in range(Kbreast):

    axs[2, 0].plot(Xbreast[ySklearn\_b == k, 0], Xbreast[ySklearn\_b == k, 1], colors[k % len(colors)] + 'o')

axs[2, 0].plot(ClusterSklearn\_b[:, 0], ClusterSklearn\_b[:, 1], 'kx', markersize=12, markeredgewidth=2)

axs[2, 0].set\_title(f'Scikit-learn KMeans ({Kbreast} clusters)')

# Display the figure

plt.tight\_layout()

plt.show()

# Plot the criterion (MSE or within-cluster sum of squares) over iterations

fig = plt.figure(figsize=(8, 6))

plt.plot(CritereMedoidPlus\_b, 'o-', color='maroon')

plt.plot(CritereMedoid\_b, 'o-', color='r')

plt.plot(CriterePlus\_b, 'o-', color='navy')

plt.plot(Critere\_b, 'o-', color='b')

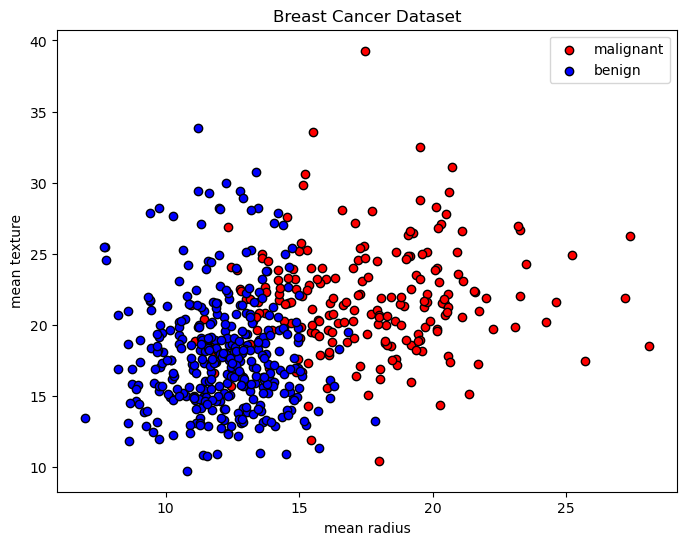
plt.legend(['K-Médoïdes++', 'K-Médoïdes', 'K-Moyennes++', 'K-Moyennes'])

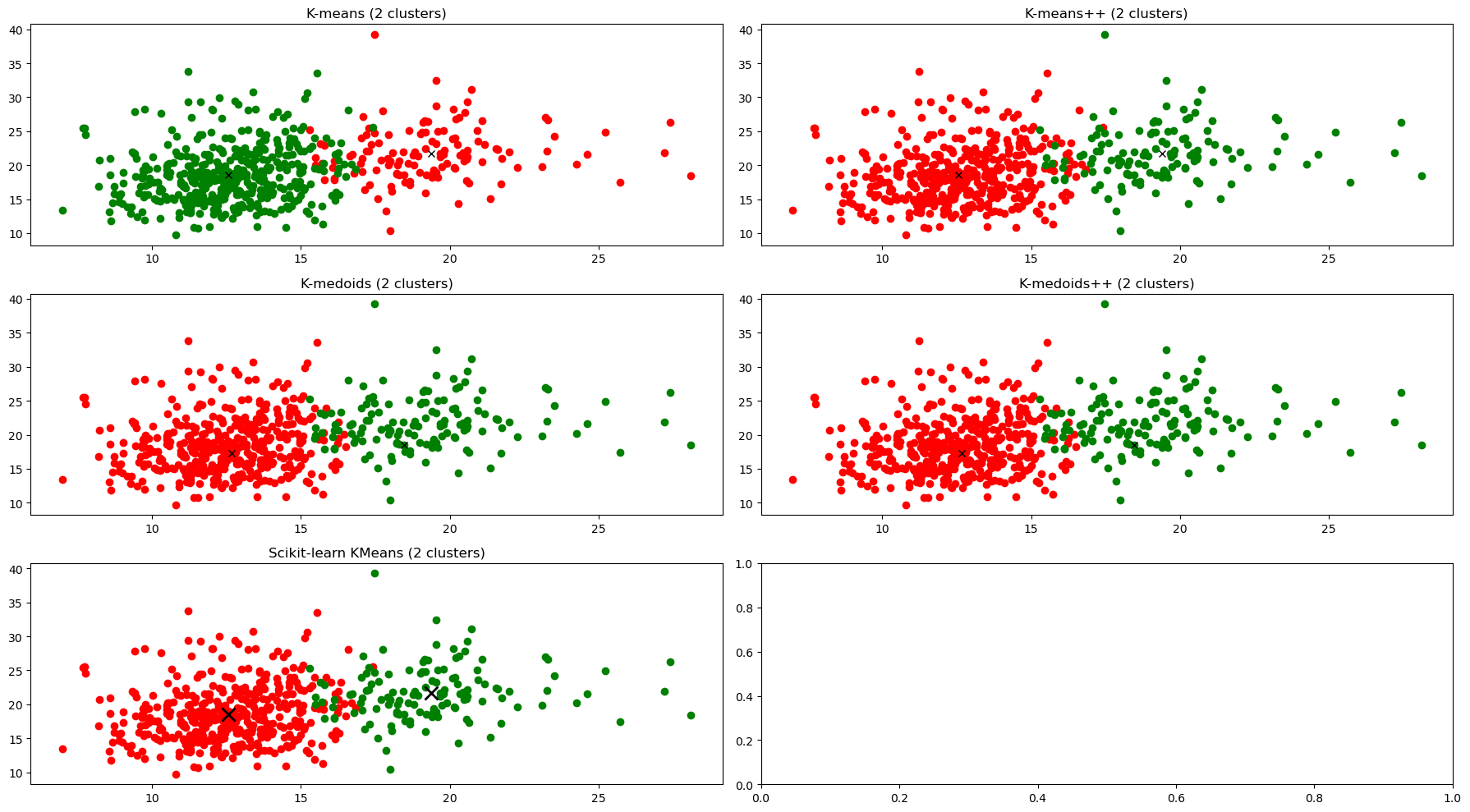
plt.xlabel('Iteration')

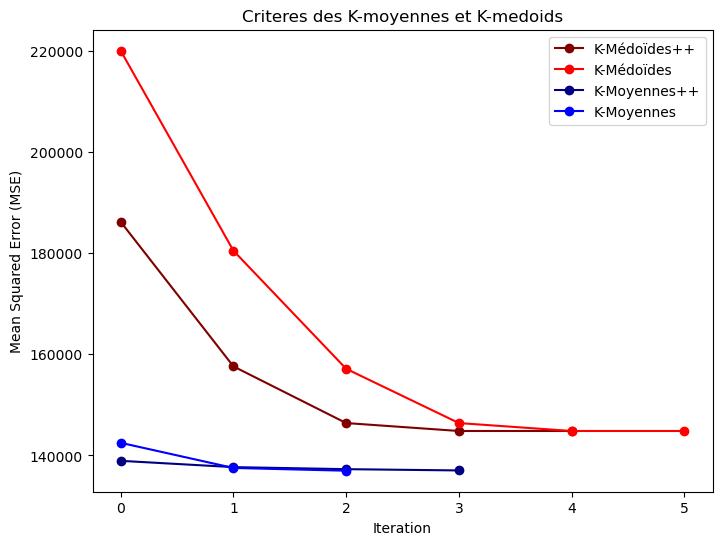
plt.ylabel('Mean Squared Error (MSE)')

plt.title('Criteres des K-moyennes et K-medoids')

plt.show()







**Conclusions**

Les résultats obtenus avec le jeu de données Breast Cancer sont cohérents avec ceux du jeu de données Iris, démontrant ainsi la robustesse et l’applicabilité de ces méthodes de classification pour différents ensembles de données.